Conventional Natural Language Processing in the NWO Priority Programme on Language and Speech Technology

January 1996 Deliverables

Gertjan van Noord
Mark-Jan Nederhof
Rob Koeling
Gosse Bouma

Vakgroep Alfa-informatica & BCN
Rijksuniversiteit Groningen
{vannoord,markjan,koeling,gosse}@let.rug.nl

March 8, 1996

Contents

Chapter 1

Introduction

This document contains the deliverables after one year of work by the Conventional NLP group of the Priority Programme on Language and Speech Technology, organised by the Dutch Research Organisation (NWO). For an overview of the goals of this priority programme we refer to Boves et al. (1995).

In this document three parts are identified. The first part defines the grammatical formalism that is used for developing the grammar and the parsers. It defines the interface between the parsers and the grammar. The second part documents the current state of the grammar; it explains some of the techniques that are used and some of the design choices that have been made. Finally it presents a work plan for grammar development in 1996. The third part consists of the documentation of the parsers that have been implemented. It discusses the parser generator that is used to construct parsers on the basis of the grammar, based on static evaluation techniques. Furthermore a number of different parser-drivers are being discussed. Most work until now has been directed towards an implementation of a bottom-up Earley parser on the one hand, and a head-corner parser on the other hand.

The modules that have been implemented are all integrated in two ways. Firstly we have constructed a specialized version of the Hdrug system for developing the parsers and grammars. Hdrug provides many useful tools for visualizing parse trees, feature structures, grammar rules, lexical entries, comparisons of different parsers with respect to speed and memory usage, etc. Secondly the modules are all integrated in a SICStus runtime system. This can be seen as a first version of the minimal module (deliverable for March 1997).

Although the work we report on here is the result of a collaboration of the four authors mentioned on the title page we indicate the responsible authors for each chapter. Chapter 2 is written by Mark-Jan Nederhof and Gertjan van Noord; chapter 3 is written by Gosse Bouma, Gertjan van Noord and Rob Koeling; chapter 4 is written by Rob Koeling and Gosse Bouma; chapter 5 is written by Gertjan van Noord; chapters 6 and 7 are written by Mark-Jan Nederhof. Chapters 8, 9 and 10 are written by Gertjan van Noord.

Part I

Grammar Formalism

Chapter 2

The Formalism for the OVIS2 Grammar

2.1 Motivations

In this chapter we define the formalism that we use for the OVIS2 Grammar. This formalism is equivalent to Definite Clause Grammars. We have chosen for DCG because:

- DCG's provide for a balance between *computational efficiency* on the one hand and *linguistic expressiveness* on the other.
- DCG's are a (simple) member of the class of declarative and constraint-based grammar formalisms. Such formalisms are widely used in linguistic descriptions for NLP.
- DCG's are straightforwardly related to context-free grammars. Almost all parsing technology is developed for CFG; porting this technology to DCG is usually possible (although there are many non-trivial problems as well).

If we consider the formalism for OVIS-Grammar in more detail the following decisions can be formulated:

- Semantic constraints can be separated from syntactic constraints. This is motivated by the desire to experiment with parsing strategies in which semantic constraints are delayed, in order to obtain *packing* in the case of ambiguities.
- External Prolog calls (in ordinary DCG these are introduced in right-hand sides using curly brackets) are not allowed. This is motivated by our desire to experiment with parsing strategies that differ in the order of processing of right-hand sides.
- Rules can easily be mapped to their 'context-free skeleton' (by taking the functor symbol). This implies that we don't allow the underspecification of categories in rules. This is motivated by our desire to experiment with parsing strategies in which part of the work is achieved on the basis of the context-free skeleton of the grammar. [1]
- Rules are given with an identifier. Such rule identifiers have a number of possible uses (debugging, grammar filtering, grammar documentation).
- For each rule the grammar writer can declare which daughter of the rule should be regarded as the head. This allows experiments with head-driven parsing strategies.

[1] Thus we stick to the 'official' notion of a definite clause: there cannot be meta-variables ranging over predicate names.

2.2 Grammar rules

The previous motivations lead to the following proposal regarding the 'abstract' syntax of grammar rules. To simplify the presentation we will define this abstract syntax on top of the Prolog syntax. Grammar rules are defined as a ternary predicate, `grammar_rule/3`.

The first argument of this predicate is a ground Prolog term indicating the name of the rule. We will refer to this argument as the 'rule identifier'. [2]

The second argument of the rule is the mother category. Categories are Prolog terms, but not variables.

The third argument of the rule is a list of categories. Note that we require that the length of the list is given, and that each of the categories appearing in the list is not a variable.

An example of a grammar rule is provided:

```
grammar_rule(vp_vpnp,
        vp(Subj,Agr,Sem),
                 [v(Subj,Agr,trans,l(Arg,Sem)),
                  np(_,Arg)
                 ]).
```

Note that this definition implies that terminal symbols cannot be introduced in rules directly. It is assumed that lexical lookup is defined seperately (cf. section 2.3).

2.2.1 Head declarations

For each grammar rule there can be a declaration indicating which daughter of the rule is the *head*. Such a declaration is a unit clause for the relation `head/2` where the first argument is a rule identifier, and the second argument is an integer n indicating that the nth daughter of the corresponding rule should be regarded as the head.

Clearly we require that n should not be larger than the number of daughters of the given rule. If no appropriate declaration is available for a given rule then the first daughter will be the head of that rule.

2.3 Lexicon

The lexicon is defined by the relation `lex/3`. As an example, the lexical entry 'sleeps' could be encoded as:

```
lex(sleep8,
    v(np,agr(3,sg),intrans,l(X,sleep(X))),
    sleeps
    ).
```

The third argument is the terminal symbol introduced by this lexical category. The second argument is the category (a non-variable term). The first argument is the rule identifier (unique and ground). In cases where a lexical entry introduces a sequence of terminal symbols the third argument is also allowed to be a (non-empty) list of atoms.

```
lexical_entry: lex(#id,#cat,#word_or_words)
word_or_words: #word |
               #words
word: ``Prolog atom''
words: [#word|#words0]

words0: [] |
        [#word|#words0]
```

[2] Note that even though the rule identifier is ground, it can be a complex term. Rule identifiers need not be unique.

2.4 Top Category

The top category for the grammar (or *start symbol*) is defined by the unary predicate `top_category`. Its argument is an arbitrary Prolog term, but not a variable.

2.5 Filters to Identify Semantic Constraints

In order to allow the separation of syntactic constraints on the one hand and semantic constraints on the other hand, we define the following type of declaration. The idea is that those parts of a category are identified which contain semantic information.

For example, the following declarations imply that the third argument of a vp and the fourth argument of a v should be ignored during syntactic processing.

```
filter_sem([Sem],vp(_,_,Sem)).
filter_sem([Sem],v(_,_,_,Sem)).
```

The meaning of such declarations can be clarified by applying the previous constraint to the following mini-grammar:

```
lex(sleep8,
    v(np,agr(3,sg),intrans,l(X,sleep(X))),
    sleeps
    ).

grammar_rule(vp_vpnp,
    vp(Subj,Agr,Sem),
        [v(Subj,Agr,trans,l(Arg,Sem)),
         np(_,Arg)
        ]).
```

As a first step of a possible compilation, this grammar can be re-written into the following format, in which the semantic constraints are defined separately; these can thus easily be subject to delayed evaluation:

```
lex(sleep8,
    v(np,agr(3,sg),intrans,Filtered),
    sleeps
    ) :-
        Filtered = l(X,sleep(X)).

grammar_rule(vp_vpnp,
        vp(Subj,Agr,Filtered),
                 [v(Subj,Agr,trans,l(Arg,Sem)),
                  np(_,Arg)
                 ]) :-
        Filtered = Sem.
```

In this example we are thus assuming that it is relevant to filter the semantic constraints associated with the left-hand-side categories of rules (and lexical entries).

The disadvantage of this encoding may be that it requires redundant matching of the syntactic constraints in order for the variables in the semantic constraint to be properly related to their position in the category. I.e. suppose you have a local parse tree constructed by the syntactic part of the rule, there is no easy way to apply the semantic constraints of that same rule without matching the syntactic part of the rule with that local tree.

As a further compilation step, therefore, we can compile separate syntactic and semantic parts of rules on the basis of the semantic filter declarations. Thus, a variant of the `lex` predicate is compiled with four arguments rather than three. There are now two arguments indicating the category of the lexical entry. One category contains syntactic information, the second category contains essentially semantic information.

Similarly, rules are compiled into a predicate with arity 4 in which the extra argument represents the semantic category of the lhs. This results in the following for the previous example:

```
lex(sleep8,
        v(np,agr(3,sg),intrans,_),
        sleeps,
        v(_,_,_,l(X,sleep(X)))
        ).

rule(vp_vpnp,
        vp(Subj,Agr,_),
                 [v(Subj,Agr,trans,l(Arg,Sem)),
                  np(_,Arg)
                 ],
        vp(_,_,Sem)).
```

Whatever the most appropriate use of semantic filter declarations may turn out to be, we define the syntax of such semantic filter declarations now as follows:

```
semantic_filter: sem_filter(#vars,``Prolog term'')
vars: [] |
      [#var|#vars]
var: ``A Prolog variable''
```

We require that each variable in the list of variables is unique in that list, and occurs exactly once in the second argument of `sem_filter`. Finally there is a global constraint on a grammar that for each mother category (and lexical category) there is exactly one matching semantic filter declaration.

Note that we have also tried to detect automatically which constraints could be usefully delayed. This is discussed in chapter 6.

The advantages over the use of this type of declaration over other possible formats are

- Applications where we do not want to separate syntactic and semantic constraints are not complicated in any way.
- The identification of constraints to be delayed is essentially a form of procedural information. It is thus good to define this separately from the grammar rules themselves (which are assumed to be totally declarative).
- The technique equally works for lexical entries and rules.

2.6 Feature-structures and feature constraints

The use of first-order terms as our primary datastructures to encode linguistic information may seem ill-adviced, given the fact that almost all work in computational grammar writing uses 'feature-structures' of some sort. For this reason we provide a package (as a SICStus Prolog module) that will allow the use of feature-structure constraints, whereas the resulting data-structures will still be first-order terms.

2.6.1 The feature library

The feature library provides extensive possibilities to compile feature equations into Prolog terms, and to view such compiled Prolog terms as feature-structures. The motivation for such an approach may be that you want feature structures for readability on the one hand, but Prolog terms and Prolog unification of such terms for effiency reasons internally. The package is heavily influenced by the work of Chris Mellish (Mellish, 1988).

Types

Before feature structures can be compiled into terms, a number of type declarations need to be specified. The declarations that need to be defined are `top/1`, `type/3` and `at/1`. These three definitions define a type hierarchy. This hierarchy has the shape of a tree. The `top/1` definition defines the daughter nodes of the root of the tree. This root is always called 'top'.

Attributes can be attached to a single type in the type hierarchy. If a type is associated with an attribute then this attribute is inherited by all of its subtypes. The top node of the type hierarchy can be seen as a variable. You can not specify any attributes for this type. The `type/3` predicate defines for a given type (first argument) a list of subtypes (second argument) and a list of attributes (third argument).

The `at/1` definitions define terminals of the tree that do not introduce attributes. It is an abbreviation of a `type/3` definition in which the second and third argument are both the empty list.

As an example, consider the following type tree definition:

```
top([boolean,sign,cat]).
 type(boolean,[+,-],[]).
  at(+).
  at(-).
 type(sign,[],[cat,phon,sem]).
 type(cat,[noun,verb],[agr]).
 type(noun,[],[pro]).
 type(verb,[],[aux,inv,subj]).
```

This piece of code defines the type-tree given in figure 2.1.

The meaning of such a type tree can be understood as follows. The class of objects is divided in three mutually exclusive subclasses, called boolean, sign and cat. Objects of type boolean can be further subdivided into classes + or -. Objects of type sign can be further specified for a `cat`, `phon` or `sem` attribute.

The meaning of this type tree can also be understood by looking at the way in which objects of a certain type are represented as Prolog terms. This is illustrated in figure 2.2.

Equational constraints

If the type definition is compiled, then the following predicates can be used: <=>/2, =>/2, ==>/2. The first predicate equates two *paths*, the second predicate assigns a type to a path, and the third predicate assigns an arbitrary Prolog term to a path.

A path is a Prolog term followed by a sequence of attributes, seperated by a colon (:). Therefore, given the previous example of a type tree, we can have the following equational constraint:

```
X:cat => noun.
    X = sign(_H,cat(noun(_G,_F),_E,_D),_C,_B,_A)
Y:cat:agr <=> Y:cat:subj:cat:agr.
    Y = sign(_O,cat(verb(_N,_M,_L,sign(_K,cat(_J,_E,_I),
                                 _H,_G,_F)),_E,_D),_C,_B,_A)
Z:phon ==> [jan,kust,marie].
   sign(_D,_C,[jan,kust,marie],_B,_A)
```

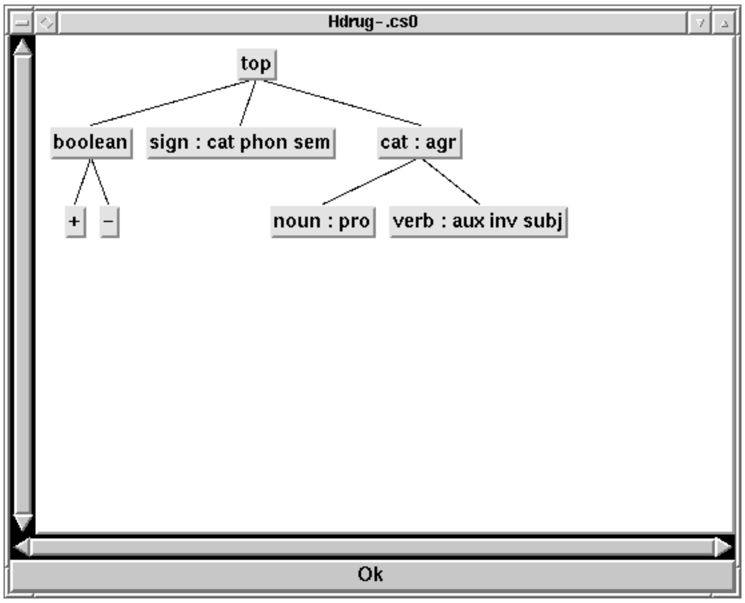


Figure 2.1: A simple type tree. Some types are associated with attributes.

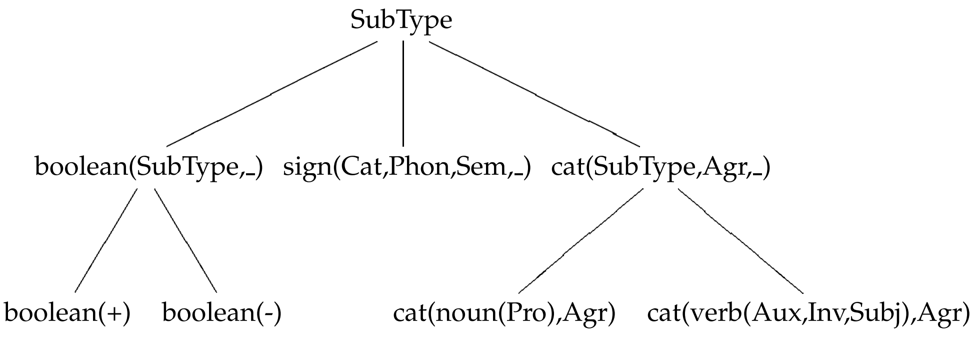


Figure 2.2: The type tree: all types are replaced with their encoding as Prolog terms.

Viewing Prolog Terms representing Feature Structures

Note that a couple of predicates are available to view Prolog terms as feature structures. For example, the predicate `tk_fs/1` from the feature library can be used to view a term in a Tk window. For example, you may try the conjunction:

```
Y:cat:agr <=> Y:cat:subj:cat:agr, tk_fs(fs(Y)).
```

The query

```
Y:cat:agr <=> Y:cat:subj:cat:agr, latex_fs(fs(Y)).
```

produces:

$$\left[\begin{array}{ll} \text{SIGN} & \\ \textit{cat}: & \left[\begin{array}{ll} \text{VERB} & \\ \textit{agr}: \mathrm{A} & \\ \textit{subj}: & \left[\begin{array}{ll} \text{SIGN} & \\ \textit{cat}: & \left[\begin{array}{l} \text{CAT} \\ \textit{agr}: \mathrm{A} \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

But if you insist on ordinary output, try:

```
tty_fs(X).
```

This produces:

```
 {sign}
 |cat {verb}
 |    |agr <A>
 |    |subj {sign}
 |    |     |cat {cat}
 |    |     |    |agr <A>.
```

Lists

You can add (ordinary Prolog) lists to your type tree by the following definition in which you should give atomic names for the attributes `HeadAtt` and `TailAtt` which you want to use as attributes to point to the head and to the tail of a list, e.g.:

```
list_type(hd,tl).
```

This will allow the use of attributes `hd` and `tl` for referring to parts of lists. Furthermore, lists of typed objects will be shown appropriately. For example:

```
[-user].
| list_type(hd,tl).
| {user consulted, 20 msec 48 bytes}
^D
yes
| ?- X:tl:hd:cat => verb.

X = [_A,sign(_K,cat(verb(_J,_I,_H,_G),_F,_E),_D,_C,_B)|_L] ?

yes
| ?- X:tl:hd:cat => verb, latex_fs(fs(X)).
[1] 6835

X = [_A,sign(_K,cat(verb(_J,_I,_H,_G),_F,_E),_D,_C,_B)|_L] ?
```

produces:

$$\langle \_, \left[\begin{array}{l} \text{SIGN} \\ \textit{cat} : \text{VERB} \end{array} \right] | \_ \rangle$$

Extensionality

Direct subtypes of type 'top' are represented using an extra variable position. This is to make sure that objects are only identical if they have been unified. For some types, such as 'boolean', this does not make much sense. Types that you want to consider as 'extensional' in this way are to be declared with the predicate `extensional/1`.

The following example illustrates the difference. Without the extensional predicate we have:

```
X:inv => -, X:aux => -, tty_fs(X).
 {verb}
 |aux <B> {-}
 |inv <B>.
```

After declaring that boolean and '-' be extensional types (and recompiling the type tree), we get:

```
X:inv => -, X:aux => -, tty_fs(X).
 {verb}
 |aux {-}
 |inv {-}.
```

The difference is that the system does not show explicitly that the values of aux and inv are the same in the second example. This is redundant information because objects of extensional types always are the same if they have the same information content.

Unify_except

The library provides the predicates `unify_except/3`,`unify_except_l/3` and `overwrite/4`. The first argument takes two feature terms and a path. The first and second argument are unified *except for the value at the path*.

As an example (assuming the simple type system given above), we may have:

```
| ?- unify_except(X,Y,cat:agr).

X = sign(_G,cat(_F,_E,_D),_C,_B,_A),
Y = sign(_G,cat(_F,_H,_D),_C,_B,_A) ?
```

The predicate `unify_except_l` is similar, except it takes a list of paths rather than a single path as its third argument. Finally, the predicate `overwrite/4` can be understood by looking at its definition:

```
overwrite(FS,FS2,Path,Type) :-
        unify_except(FS2,FS,Path),
        FS2:Path => Type.
```

Find_type

The meta-logical predicates `find_type/2` and `find_type/3` can be used to get the most specific type of a feature term. The first argument is the feature term, the second argument is a list of most specific types (for simple usage just consider the first element of this list). The optional third argument is a list of attributes that are appropriate for this type. For example:

```
| ?- X:agr <=> X:subj:agr, find_type(X,[Y|_]).

X = cat(verb(_G,_F,_E,cat(_D,_B,_C)),_B,_A),
Y = verb ?
```

It is clear that `find_type/2,3` is a meta-logical predicate by looking at the following example, where the conjuncts are swapped:

```
| ?- find_type(X,[Y|_]), X:agr <=> X:subj:agr.

X = cat(verb(_G,_F,_E,cat(_D,_B,_C)),_B,_A),
Y = top ? ;
```

Disjunction and Negation over Atomic Values

A special mechanism is provided for atomic values to allow for disjunction and negation over such atomic values. These atomic values are not declared in the type-system as shown above, but rather they are introduced by the predicate `boolean_type/2`. The first argument of this predicate is an identifier, the second argument of this predicate is a list of lists that is understood as a set product. For example, agreement features could be defined as:

```
boolean_type(agr,[[1,2,3],[sg,pl],[mas,fem,neut]])
```

So valid and fully specified values for agreement consist of an element from each of the three lists. The syntax for type-assignment is extended to include disjunction (';'), conjunction ('&') and negation ('~') of types. For example, to express that `X` has either singular masculine or not-second person agreement, we simply state:

```
X => ( sg & mas ; ~2 ).
```

The following example illustrates the use of this package:

```
| ?- [-user].
| boolean_type(agr,[[1,2,3],[sg,pl],[mas,fem,neut]]).
| {user consulted, 10 msec 368 bytes}

yes
| ?- compile_boolean_types.

yes
| ?- X => ( sg & mas ; ~2 ).

X = agr(0,_L,_K,_J,_I,_H,_G,_G,_G,_G,_G,_F,_F,_E,_D,_C,_B,_A,1) ?
```

The example shows how complex terms are created for such boolean types. This is useful because disjunction and negation can be handled by ordinary unification in this way (Mellish, 1988). Luckily the pretty printing routines will turn such complex turns back into something more readible:

```
| ?- X: agr => (sg & mas ; ~2 & neut), latex_fs(fs(X)).
```

produces:

$$\begin{bmatrix} \text{CAT} \\ \mathit{agr} : 1\&\text{NEUT};\text{SG}\&\text{MAS};3\&\text{NEUT} \end{bmatrix}$$

2.6.2 Grammar rules with feature-constraints

It may seem that the use of feature-constraints contradicts the requirement that the right-hand side of a rule should not contain Prolog constraints. However note that there is a difference here between *the way in which we define a rule*, and *the actual rule as a result of this definition*. Thus we view such feature-constraints as constraints of the former kind. Simple relations to encode macros or templates fall in the same category. Another way to think about this is to assume that all Prolog calls are evaluated during compilation time.

Given the definition of the feature-constraints => and <=> in `library(feature)`, a typical grammar rule may be defined as follows:

```
grammar_rule(1,S,[Np,Vp]) :-
     S:cat => s,
    np(Np),
    vp(Vp),
    subj_agreement(Vp,Np).

np(Np) :-
    Np:cat => n,
    Np:lex => -.

vp(Vp) :-
    Vp:cat => v,
    Vp:lex => -.

subj_agreement(Vp,Np) :-
    Vp:subj <=> Np.
```

As far as the other modules of the system are concerned, this grammar rule will be defined as something like:

```
grammar_rule(1,s,[n(Agr,-),v(-,n(Agr,-))]).
```

2.7 The Grammar for Grammars

This section summarizes the previous sections by giving a somewhat more precise characterisation of the grammar format we use. We assume the grammar is a (Sicstus) Prolog program defining the predicates: `grammar_rule/3`, `lex/3` and `top_category/1`, and possibly also `sem_filter/2`. The grammar may contain comments according to the regular syntax of Prolog, and definitions of other predicates beside the four mentioned above; these will of course be ignored.

For describing the allowed grammars, we use the following three meta rules, which apply for `X` ranging over any nonterminal.

```
X-sequence : #empty |
             X X-sequence

X-series : #empty |
           X-series-proper

X-series-proper : X |
                  X , X-series-proper
```

```
X-list : [ X-series ]

X-list-proper : [ X-series-proper ]

#empty :
```

(The `X`-lists describe the normal notation for lists in Prolog.)

Below we give the basic syntax for grammars. We disregard comments, layout, and possible use of "syntactic sugar", such as pre-defined operators.[3] We assume that the regular syntax for (Sicstus) Prolog applies in these cases.

```
#grammar : #declaration-sequence

#declaration : #top_cat_declaration |
               #rule_declaration |
               #lex_declaration |
               #sem_filter_declaration |
               #head_declaration

#top_cat_declaration : top_category( #category ).

#rule_declaration : grammar_rule( #id, #lhs, #rhs ).

#lex_declaration : lex( #id, #lhs, #words ).

#sem_filter_declaration : sem_filter( #variable-list, #feature ).

#head_declaration : head( #id, #integer ).
```

It is reasonable to assume that the grammar contains at least one top-category declaration, at least one grammar rule and at least one lexical entry. This is not required however for proper functioning of the parser generator.

Top categories also function as an interface between the parser and the semantic module of the OVIS system: values may be passed through the variables in the top categories. We refer to such values as *semantic values*.

The semantic-filter declarations indicate that computation of certain subfeatures (those in `#feature` where a variable in `#variable-list` occurs) can be omitted for what the computation of parses is concerned. These subfeatures cannot lead to rejection of a parse, but are needed only after a parse has been found, for the computation of the semantic value of that parse. See also section 6.4.2.

```
#id : #term
```

It is reasonable to assume that id's are unique identifiers for either grammar rules or lexical entries (but not both at the same time). No such uniqueness is required however for proper functioning of the parser generator. It is also not required that id's are ground terms, although this is a reasonable assumption. Variables that possibly occur in the id's will not be considered bound to any other variable in a declaration of a lexical entry or grammar rule, even if the names are identical.

```
#lhs : #category

#rhs : #member-list
```

[3] Definitions of operators in the grammar file are discouraged.

```
#member : #category

#category : #atom |
            #functor( #subcategory-series-proper )

#subcategory : #variable |
               #atom |
               #functor( #subcategory-series-proper )

#words : #word |
         #word-list-proper

#word : #atom

#term : #variable |
        #atom |
        #functor( #term-series-proper )

#feature : #variable |
           #atom |
           #functor( #feature-series-proper )
```

The syntax of terms and features is very similar to the one of categories (with the difference that categories cannot be variables, at least not officially[4]). However, we use the word "category" exclusively for the objects occurring as top category, or as lhs or member in grammar rules or lexical entries. We use the word "feature" for (sub)categories in which some variables may have been instantiated; these play a role in the dynamic semantics of grammars. We use the word "term" for all other Prolog terms which may occur during parsing or parser generation.

```
#variable : ``Sicstus_prolog_variable''

#atom : ``Sicstus_prolog_atom''

#functor : ``Sicstus_prolog_functor''
```

At the moment, the lexical entries and the grammar rules are combined in one file, which has extension `.g`. We envisage that in the future, the lexical entries will be separated from the grammar rules and the top-category declarations, in a file with extension `.lex`. We refer to the collection of all such entries as the *lexicon*.

[4]If categories are variables, then they are replaced by proper categories in all possible ways by the parser generator.

Part II

The OVIS Grammar

Chapter 3

The Ovis Grammar

3.1 Introduction

This chapter documents the grammar development efforts carried out for the *Grammar-based NLP* module during the first twelve months of the OVIS project.

The OVIS grammar module contains the grammatical knowledge required to analyse a word graph and to determine what the meaning of the utterance corresponding to the word graph is.

Ideally, to construct such a grammar one would take a general-purpose computational grammar of Dutch, and adapt it to the current domain and application. Adaptation would consist mainly of extending the lexical coverage and construction of the semantics-application interface. Unfortunately, however, we do not know of any computational grammar for Dutch that could easily be adapted to the present task (which, among others, requires processing of spoken language, extensive coverage of locative and temporal expressions, and the construction of fine-grained semantic representations).

The OVIS grammar is being developed specially for the purposes of this project. On the one hand this has the advantage that the grammar can be tailored to the specific requirements of the present project. On the other hand, we want to adopt general solutions as much as possible, as this increases the chances that the grammar can be used in other domains as well. Thus, in designing the grammar we seek a balance between short-term goals (a grammar which covers utterances typical for the OVIS-domain and is reasonably robust and efficient) and long-term goals (a grammar which covers the major constructions of Dutch in a general way).

The grammar currently covers some of the more common verbal subcategorization types (intransitives, transitives, verbs selecting a PP, and modal and auxiliary verbs), NP-syntax (including pre- and postnominal modification, with the exception of relative clauses), PP-syntax, the distribution of VP-modifiers, various clausal types (declaratives, yes/no and WH-questions, and subordinate clauses), all temporal expressions and locative phrases relevant to the domain, and various typical spoken language constructs.

Below, we describe the grammar formalism, the grammar fragment that has been implemented so far, and a workplan, based on limited experience with the test data that have been collected so far, for the next twelve months.

3.2 Linguistic Issues and Grammar Architecture

From a linguistic perspective, the OVIS-grammar can be characterized as a constraint-based grammar, which makes heavy use of lexical information. The design of the grammar is inspired to a certain extent by HPSG (Pollard and Sag, 1994).

3.2.1 Signs

In unification-based grammar formalisms, linguistic information is represented by means of typed feature-structures. Each word or phrase in the grammar is associated with such a feature-structure, in which syntactic and semantic information is bundled. Within HPSG, such feature-structures are called *signs*, a terminology which we will follow here.

At present, the grammar makes use of 15 different types of signs, where each type roughly corresponds to a different category in traditional linguistic terminology. For each type of sign, a number of features are defined. For example, for the type NP, the features AGR, NFORM, CASE, and SEM are defined. These features are used to encode the agreement properties of an NP, (morphological) form (i.e. is it a normal NP or temporal, locative, or numeral expresssion), case and semantics respectively. A more detailed presentation of these features follows in the sections below.

There are a number of features which occur in most types of sign, and which play a special role in the grammar. The feature SC (SUBCATEGORIZATION) (present on signs of type *v, sbar, det, a ,n* and *p*), for instance, is a feature whose value is a list of signs. It represents the subcategorization properties of a given sign. As will be explained below, it is used to implement rules which perform a kind of functor-argument application (as in Categorial Grammar).

The feature SLASH is present on *v, s, ques* and *sbar*. Its value is a list of signs. It is used to implement a (restricted) version of the account of nonlocal dependencies proposed in Pollard and Sag (1994) and Sag (1995). The value of SLASH is the list of signs which are 'missing' from a given constituent. The 'missing' element is typically connected to a preposed element in a topicalization sentence or WH-question. At the moment, these are the only unbounded dependency constructions dealt with in the grammar. The same mechanism can also be used to account for relative clauses.

The feature VSLASH is similar to SLASH in that it records the presence of a missing element, a verb in this case. It is used to implement an account of Dutch main clauses, based on the idea that main clauses are structurally similar to subordinate clauses, except for the fact that the finite verb occurs as first or second constituent within the clause and the clause final position where finite verbs occur in subordinate clauses is occupied by an *empty* verbal sign (i.e. an element which is not visible in the phonological or orhtographic representation of the sentence).

The feature SEM is present on all signs. It is used to encode the semantics of a word or phrase, encoded as a so-called *quasi logical form* (Alshawi, 1992). The feature MOD is present on the types *a, pp, p, adv, sbar* and *modifier*. It is used to account for the semantics of modifiers. Its value is a list of *quasi-logical forms*. In the sections below on syntax, we only give an informal impression of the semantics. The details of the semantic construction rules and principles are dealt with in section 4.

We conclude this section with a note on implementation. The various types for signs correspond roughly with different linguistic categories. In the actual implementation, however, there is no type *sign* with subtypes *n, det, v, etc*. Although introducing a type *sign* would be conceptually cleaner, we have refrained from doing so for reasons of efficiency. A type *sign* with various subtypes introduces Prolog terms which are all headed by the functor `sign`. One of the arguments of this term is used to encode subtype information. If *n, v, det, etc*. are not subtypes of *sign*, however, they introduce prolog terms headed by a functor `n, v, det`, *etc*. It turns out to be the case that such a representation has great benefits for parsing efficiency. As efficiency is a factor which cannot be neglected within the OVIS project, we have adopted the latter encoding.

3.2.2 Syntax: Principles, Structures, and Rules

An important restriction imposed by the grammar-parser interface is that rules must specify the category of their mother and daughters. That is, each rule must specify the type of sign of its mother and daughters. A consequence of this requirement is that general rule-schemata, as used in Categorial Grammar and HPSG cannot be used in the OVIS grammar. A rule which specifies that a head daughter may combine with a complement daughter, if this complement unifies with the first element on SC of the head (i.e. a version of the categorial rule for functor-argument

```
hd_comp_struct(Head,Complements,Mother) :-
        hd_struct(Head,Complements,[],Head,Mother).

hd_mod_struct(Head,Modifier,Mother) :-
        hd_struct(Head,[],[],Modifier,Mother),
        Head:sem <=> HeadSem,
        Modifier:mod <=> [HeadSem].

hd_struct(Head,Complements,Subject,SemanticHead,Mother) :-
        head_feature_principle(Head,Mother),
        valence_principle(Head,Complements,Subject,Mother),
        filler_principle(Head,[],Mother),
        SemanticHead:sem <=> Mother:sem.

head_feature_principle(Head,Mother) :-
        unify_ifdef(Head,Mother,vform),
        unify_ifdef(Head,Mother,agr),
        unify_ifdef(Head,Mother,case),
        unify_ifdef(Head,Mother,mod),
        unify_ifdef(Head,Mother,pform),
        unify_ifdef(Head,Mother,aform),
        unify_ifdef(Head,Mother,vslash).

valence_principle(Head,Complements,Subject,Mother) :-
        ifdef(sc,Head,HeadSc,[]),
        ifdef(sc,Mother,MotherSc,[]),
        append(Complements,MotherSc,HeadSc),
        ifdef(subj,Head,HeadSubj,[]),
        ifdef(subj,Mother,MotherSubj,[]),
        append(Subject,MotherSubj,HeadSubj).
```

Figure 3.1: Structures and Principles

application) cannot be implemented directly, as it leaves the categories of the daughters and mother unspecified. Nevertheless, generalizations of this type do play a role in the grammar. We adopt an architecture for grammar rules similar to that of HPSG, in which individual rules are classified in various *structures*, which are in turn defined in terms of general *principles*.

Following the practice of HPSG, we assume that rules normally introduce a structure in which one of the daughters can be identified as the *head*. The head of a rule is the daughter whose category or type is (usually) identical to that of mother. The head daughter either subcategorizes for the other (complement) daughters or else is modified by the other (modifier) daughters.

The two most common structures are the *head-complement* and *head-modifier* structure.[1] In Figure 3.1 we list the definitions for these structures and the principles they refer to, except for the *filler* principle, which is presented in the section on topicalization.

Head-complement and *head-modifier* structures are instances of a *headed* structure. The definition of *headed* structures refers to the HEAD-FEATURE, VALENCE, and FILLER principle, and furthermore fixes the semantic head of a phrase. Note that the definition of `hd-struct` has a number of parameters. The idea is that a headed structure will generally consist of a head daughter, and furthermore of zero or more complement daughters, possibly a subject daughter,[2] and possibly

[1]Other structures are the *main-clause* and *head-filler* structure. These are discussed in the sections on main-clause syntax and topicalization. See also appendix A.

[2]The current grammar does not introduce subjects as part of a headed structure however, but only introduces subjects

```
rule(np_det_n, NP, [Det, N ]) :-
     NP=> np, Det=> det, N=>n,
     NP:nform => norm,
     hd_comp_struct(Det,[N],NP).
```

```
rule(n_adj_n, N1, [ AdjP, N0 ]) :-
     N1 => n, AdjP => a, N0 => n,
     AdjP:agr <=> N0:agr,
     hd_mod_struct(N0,AdjP,N1).
```

Figure 3.2: **NP Rules**

$$\texttt{np\_det\_n:} \quad {}_{np}\begin{bmatrix} \text{AGR} & \boxed{1} \\ \text{NFORM} & \textit{norm} \\ \text{SEM} & \boxed{2} \end{bmatrix} \longrightarrow {}_{det}\begin{bmatrix} \text{SC} & \left\langle \boxed{3} \right\rangle \\ \text{AGR} & \boxed{1} \\ \text{SEM} & \boxed{2} \end{bmatrix} \; \boxed{3}\,\text{n}$$

$$\texttt{n\_adj\_n:} \quad {}_{n}\begin{bmatrix} \text{SC} & \boxed{1} \\ \text{AGR} & \boxed{2} \\ \text{SEM} & \boxed{3} \end{bmatrix} \longrightarrow {}_{a}\begin{bmatrix} \text{SC} & \left\langle \right\rangle \\ \text{AGR} & \boxed{2} \\ \text{SEM} & \boxed{3} \\ \text{MOD} & \left\langle \boxed{4} \right\rangle \end{bmatrix} \; {}_{n}\begin{bmatrix} \text{SC} & \boxed{1} \\ \text{AGR} & \boxed{2} \\ \text{SEM} & \boxed{4} \end{bmatrix}$$

Figure 3.3: **Attribute value matrices for NP Rules**

a modifier. *Head-complement* and *Head-modifier* structures differ from each other only in that the first introduces complements, but no modifiers, whereas the second introduces no complements, but a modifier, and in the fact that the syntactic head is also the semantic head in *head-complement* structures, but not in *head-modifier* structure. In *head-modifier* structures, the semantic contribution of the head to the meaning of the phrase as a whole is handled by unifying the head semantics with the value of (the first element of) MOD on the modifier.

The HEAD FEATURE PRINCIPLE states for a number of features (the so-called *head-features*) that their value on the head daughter and mother must be unified. As this principle generalizes over various types of sign, its definition requires the predicate `unify_ifdef`. The constraint `unify_ifdef( Head, Mother, agr)`, for instance, requires that if both `Head` and `Mother` are of a type for which the feature AGR is defined, the value of AGR on `Head` and `Mother` should unify. If this feature is undefined on either `Head` or `Mother`, the constraint is satisfied trivially.

The VALENCE PRINCIPLE determines the value of the so-called *valence features* SC and SUBJ. It says that the value of SC (and similarly for SUBJ) on the head daughter of a rule is the concatenation (`append`) of the list of complement daughters and the value of SC on the mother. Another way to put this is that the value of SC on the mother is the value of SC on the head daughter minus the elements on SC that correspond to the complement daughters. Note that the formulation of the VALENCE PRINCIPLE is complicated by the fact that SC (or SUBJ) may sometimes not be defined on the mother. In that case, it is assumed that the value of SC on the head daughter must correspond exactly to the list of complement daughters. The constraint `ifdef(sc,Mother,MotherSc,[])` states that the value of SC on `Mother` unifies with `MotherSc`, if SC is defined for the type of `Mother`. Otherwise, `MotherSc` is assigned the value `[]`.

The structures defined in Figure 3.1 are used in the definition of grammar rules. Two typical grammar rules are shown Figure 3.2. The `np-det-n` rule introduces a *head-complement* structure in which (following the traditional semantic (and categorial grammar) analysis) the determiner is the head, and the noun the complement. The `n-adj-n` rules introduces a *head-modifier* structure where the adjective is the modifier. Note that for a given rule, the types of the mother and daughters must be specified, and furthermore, the number of complements is always specified.

in main clause structures (which are not defined in terms of `hd-struct`). For the sake of generality, however, we have included the possibility of subject daughters.

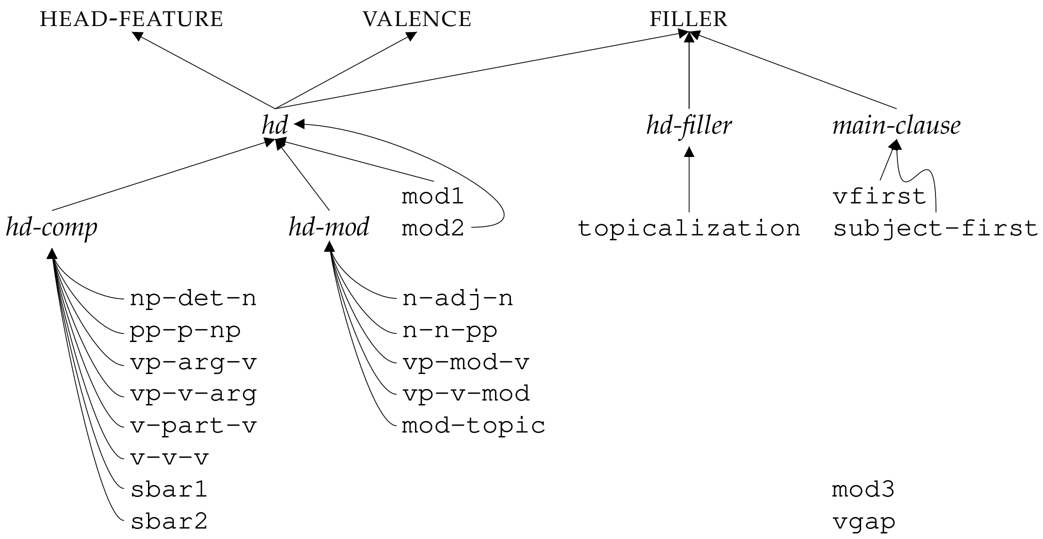


Figure 3.4: The Rule Hierarchy (with PRINCIPLES shown in small case, *structures* in italic, and `rules` in typewriter). Note that the `mod3` rule (a unary rule which transforms temporal NP's into verbal modifiers) and the `vgap` rule (a rule which introduces verbal gaps) are exceptional in that they do not inherit from general principles.

This implies that the constraints in the principles in Figure 3.1 can be reduced to a number of basic constraints on the values of particular features defined for the signs in the rule. Thus, the definition in Figure 3.2 gives rise to two rules which can be depicted in matrix notation as shown Figure 3.3.

An overview of all grammar rules defined in the fragment at the moment, together with the structures and principles from which they inherit, is given in Figure 3.4. The corresponding code is given in Appendix 3.5.

The classification of rules into structures, which are in turn defined in terms of principles, allows us to state complicated rules succinctly and to express a number of generalizations. Nevertheless, it is also clear that the rules could have been more general, if rule *schemata* (in which the type of the daughters, or even the number of daughters is not necessarily specified) had been allowed. Given this restriction, one may even wonder whether the VALENCE PRINCIPLE (and the features SC and SUBJ that come with it) cannot be eliminated in favour of more specific rules. Valence features are particularly important for grammars employing rule schemata, but they are much less crucial for more traditional types of grammar. Although eliminating valence features is not impossible in principle, we believe that the present set-up still has advantages, although these are less apparent than in grammars which make use of rule schemata. Expressing valence information lexically, instead of using more detailed syntactic rules, has the advantage that idiosyncratic subcategorization requirements (such as the restriction that *denken* (*to think*) requires a PP-complement headed by *aan* (*about*), or the fact that *komen* (*to come*) may combine with the particle *aan* (the combination of which means *to arrive*)) need not be stated in the rules. Similarly, all constraints having to do with case marking and agreement can be expressed lexically, as well as the semantic relation between a head and its dependents.

3.2.3 The Lexicon

The lexicon is basically a list of tuples `lex(Word, Sign)`, associating each word in the grammar with a specific sign.

Constraint-based grammars in general, and lexicalist constraint-based grammars in particular, tend to store lots of grammatical information in the lexicon. This is also true for the OVIS

```
v(V) :-                                 intransitive(Pred,Sign) :-
        V:lex => basic,                         iv(Sign),
        V:vslash => [],                         iv_sem(Sign,Pred).
        V => v,
        V:subj <=> [Subj],              transitive(Pred,Sign) :-
        Subj => np,                             tv(Sign),
        Subj:nform => norm.                     tv_sem(Sign,Pred).

iv(V) :-
        v(V),
        V:sc <=> [].

tv(V) :-
        v(V),
        V:sc <=> [Obj],
        Obj:nform => norm,
        Obj => np,
        Obj:case => acc.
```

Figure 3.5: Fragment of the lexical hierarchy

grammar. A lexical entry for a transitive verb, for instance, not only contains information about the morphological form of this verb, but also contains the features SC and SUBJ, whose values are lists of signs, for which quite detailed constraints may be defined. Furthermore, for all lexical signs it is the case that their semantics is represented by means of a feature-structure. This structure can also be quite complex. To avoid massive reduplication of identical information in the lexicon, the use of inheritance is therefore essential.

In Figure 3.5, we illustrate the use of inheritance in the lexicon. All lexical entries for verbs have a number of properties in common, such as the fact that they are of type *v*, and take a (single) (normal, i.e. non-locative or temporal) NP as subject. This is expressed by the template `v(V)`.[3] Intransitive verbs (`iv(V)`) can now be characterized syntactically as verbs which do not subcategorize for any (nonsubject) complements. Transitive verbs (`tv(V)`) subcategorize for an NP with accusative case. The templates `intransitive(Pred,Sign)` and `transitive(Pred,Sign)`, finally, combine the syntactic and semantic properties of intransitive and transitive verbs. The variable `Pred` is used in the semantics to fix the value of the predicate defined by a particular verb.[4] In Figure 3.6 the attribute-value matrices for the templates `iv(V)` and `tv(V)` are given.

The lexicon itself (i.e. the predicate `lex/2`) is defined in terms of `entry/3`, `inflection/5` and `lexical_rules/2` (see Figure 3.7).[5] The definition of `entry(Root,Sign,MorphForms)` defines for each root form what its associated sign is, and what its morphological forms are. For instance, for verbs we must typically distinguish a first person singular form, a second and third person singular form, and a plural form (which is also the form of the infinitive).[6] The predicate `inflection/5` defines how inflected forms are derived. It selects one of the morphological forms of an entry (i.e. `Word` will be instantiated as one of the members of `MorphForms`), and adds the corresponding constraints to `Sign`. This is illustrated for verbal signs in the example, but rules

[3]The information that the subject NP must have nominative case is stated in the rule which introduces subjects. The restriction that subjects must be [NFORM *norm*] must be overwritten for verbs which select a dummy subject (such as weather verbs) and so-called 'raising' verbs which do not impose a semantic (or thematic) restriction on their subject.

[4]By default, the value of `Pred` for a given word is identical to the root form of that word.

[5]The following discussion makes a number of simplifications compared to the actual implementation. These simplifications are without linguistic consequence.

[6]There are a number of additional rules for verbs with exceptional paradigms. At the moment, we do not include participle forms.

$$
\text{iv}\left(\left[\begin{array}{ll}
\text{LEX} & \textit{basic} \\
\text{SC} & \langle\rangle \\
\text{SUBJ} & \left\langle \left[\begin{array}{ll}\text{NFORM} & \textit{norm}\end{array}\right]_{np} \right\rangle \\
\text{VSLASH} & \langle\rangle
\end{array}\right]_{v}\right)
\qquad
\text{tv}\left(\left[\begin{array}{ll}
\text{LEX} & \textit{basic} \\
\text{SC} & \left\langle \left[\begin{array}{ll}\text{NFORM} & \textit{norm} \\ \text{CASE} & \textit{acc}\end{array}\right]_{np} \right\rangle \\
\text{SUBJ} & \left\langle \left[\begin{array}{ll}\text{NFORM} & \textit{norm}\end{array}\right]_{np} \right\rangle \\
\text{VSLASH} & \langle\rangle
\end{array}\right]_{v}\right)
$$

Figure 3.6: Attribute-value matrices for iv and tv

for adjectival and nominal inflection also exist. Lexical rules, finally, can be used to transform the sign associated with a lexical entry. For instance, the account of nonlocal dependencies sketched below makes use of a lexical rule which removes a sign from SC and places it on SLASH. A more detailed account of this lexical rule is given in the section on nonlocal dependencies.

The constraints on `lex` are evaluated at compile time. Thus, we will obtain the lexical entries for *arriveer* and *halen* given in Figure 3.8.

3.3 The Analysis of Specific Constructions

3.3.1 Noun Phrases

In (1) we list the four types which are relevant in the syntax of NP's. We distinguish noun-phrases, nouns, determiners, and adjectives.

(3.1)

$$
\left[\begin{array}{ll}
\text{AGR} & \textit{Agr} \\
\text{CASE} & \textit{Case} \\
\text{NFORM} & \textit{Nform} \\
\text{SEM} & \textit{Qlf}
\end{array}\right]_{np}
$$

$$
\left[\begin{array}{ll}
\text{SC} & \textit{listof(Sign)} \\
\text{AGR} & \textit{Agr} \\
\text{SEM} & \textit{Qlf}
\end{array}\right]_{det}
\qquad
\left[\begin{array}{ll}
\text{SC} & \textit{listof(Sign)} \\
\text{AGR} & \textit{Agr} \\
\text{SEM} & \textit{Qlf} \\
\text{MOD} & \textit{listof(Qlf)}
\end{array}\right]_{a}
\qquad
\left[\begin{array}{ll}
\text{SC} & \textit{listof(Sign)} \\
\text{AGR} & \textit{Agr} \\
\text{SEM} & \textit{Qlf}
\end{array}\right]_{n}
$$

The features AGR (*agreement*), CASE, and NFORM (*noun form*) are used to encode agreement properties (encoded as a boolean combination of *person*, *number*, *determiner* and *definiteness*), the case value and the form of an item. Their possible values are listed in (2). Note that AGR contains the information needed for subject-verb agreement, as well as for NP-internal agreement (between determiner, adjective, and noun). CASE and NFORM are relevant for full NP's only.

(3.2) *Agr* (fir $\vee$ sec $\vee$ thi) $\wedge$ (sg $\vee$ plu) $\wedge$ (de $\vee$ het) $\wedge$ (def(inite) $\vee$ indef(inite))
Case nom(inative) $\vee$ acc(usative)
Nform norm(al) $\vee$ loc(ative) $\vee$ temp(oral) $\vee$ num(eral)

Full NP's do never take complements, so they do not have a feature SC. Adjectives may modify a noun, therefore the feature MOD is defined for type *a*.

The two rules we presented in the previous section (Figures 3.2 and 3.3) are used to form NP's consisting of a determiner and a (possibly complex) noun, and N's consisting of an adjective followed by a (possibly complex) noun. The derivation of the NP *de volgende intercity* is shown in Figure 3.9.

```
lex(Word,Sign) :-
        entry(Root,Sign0,WordForms),
        inflection(Root,Word,Sign0,Sign1,MorphForms),
        lexical_rules(Sign1,Sign).

entry(arriveer,Sign,[arriveer,arriveert,arriveren]):-
        intransitive(arriveren,Sign).
entry(halen,Sign,[haal,haalt,halen]) :-
        transitive(halen,Sign).

inflection(_,Word,Sign,Sign,MorphForms) :-
        Sign => v,
        verbal_morphology(Word,MorphForms,Sign).

verbal_morphology(SgFirstForm,[SgFirstForm,_,_],Sign) :-
        Sign:vform => fin,
        subject_verb_agreement(Sign, sg & fir).
verbal_morphology(SgNonFirstForm,[_,SgNonFirstForm,_],Sign) :-
        Sign:vform => fin,
        subject_verb_agreement(Sign, sg & (sec ; thi)).
verbal_morphology(PluForm,[_,_,PluForm],Sign) :-
        Sign:vform => fin,
        subject_verb_agreement(Sign, pl).
verbal_morphology(Inf,[_,_,Inf],Sign) :-
        Sign:vform => inf.

subject_verb_agreement(Sign,Agr) :-
        Sign:subj:h:agr => Agr.
```

Figure 3.7: The organization of the lexicon

$$\text{lex(arriveert, } {}_{v}\begin{bmatrix} \text{LEX} & \textit{basic} \\ \text{VFORM} & \textit{fin} \\ \text{SC} & \langle\,\rangle \\ \text{SUBJ} & \left\langle {}_{np}\begin{bmatrix} \text{NFORM} & \textit{norm} \\ \text{AGR} & \textit{sg} \wedge \textit{fir} \end{bmatrix} \right\rangle \\ \text{VSLASH} & \langle\,\rangle \end{bmatrix}\text{)} \quad \text{lex(halen, } {}_{v}\begin{bmatrix} \text{LEX} & \textit{basic} \\ \text{VFORM} & \textit{inf} \\ \text{SC} & \left\langle {}_{np}\begin{bmatrix} \text{NFORM} & \textit{norm} \\ \text{CASE} & \textit{acc} \end{bmatrix} \right\rangle \\ \text{SUBJ} & \left\langle {}_{np}\begin{bmatrix} \text{NFORM} & \textit{norm} \end{bmatrix} \right\rangle \\ \text{VSLASH} & \langle\,\rangle \end{bmatrix}\text{)}$$

Figure 3.8: Lexical entries for *arriveer* and *halen*

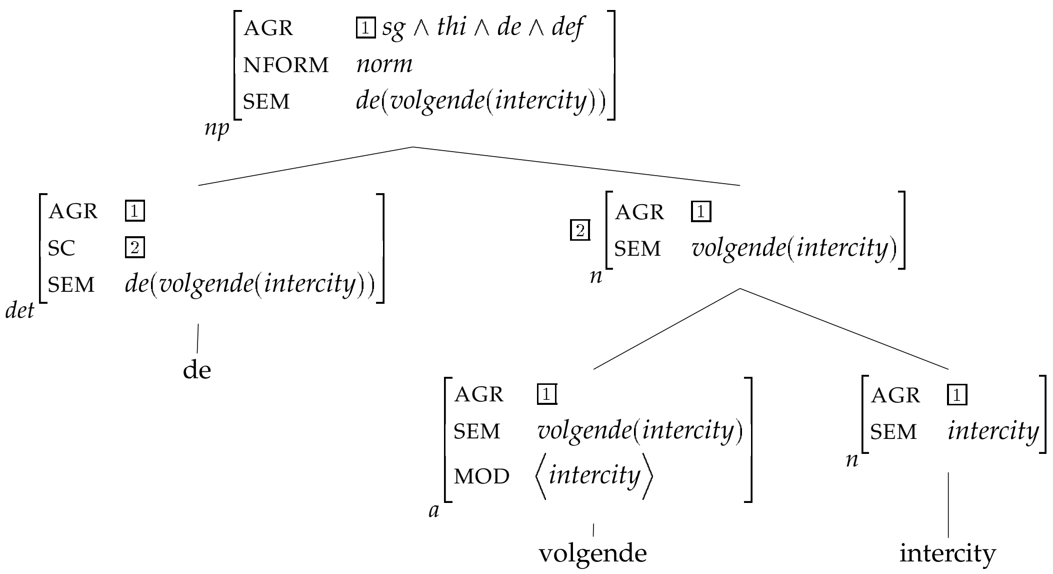


Figure 3.9: de volgende intercity

3.3.2 Prepositional Phrases

Prepositional phrases are of type *pp* and are headed by prepositions, i.e. elements of type *p* (3).

(3.3)

$$\begin{bmatrix} \textsc{pform} & \textit{Pform} \\ \textsc{sem} & \textit{Qlf} \\ \textsc{mod} & \textit{listof(Qlf)} \end{bmatrix}_{pp} \qquad \begin{bmatrix} \textsc{sc} & \textit{listof(Sign)} \\ \textsc{pform} & \textit{Pform} \\ \textsc{sem} & \textit{Qlf} \\ \textsc{mod} & \textit{listof(Qlf)} \end{bmatrix}_{p}$$

Prepositions subcategorize (usually) for an NP, so the value of SC on P will be a list of length one, containing the NP-complement. The feature PFORM takes as value the specific form of the preposition heading the PP (i.e. *van, op, naar, ...*). This information can be used to let a verb select a PP headed by a specific preposition. Full PP's can modify nouns or verb phrases. Therefore, PP has a feature MOD. The feature MOD has to be present on P as well, as the relation between the semantics of the preposition and the element is modifies is encoded as part of the lexical entry of a preposition.

In Figure 3.10 we give the rule which forms PP's and the rule which lets a PP combine as a modifier with a noun. Using these rules, we can derive the phrase *intercity uit Goes* as illustrated in Figure 3.11.

It should be noted that since adjectives precede the nouns they modify and PP's follow them, an expression such as *volgende intercity uit Groningen* will receive two parses. This appears to be a case of spurious ambiguity. There are intensional adjectives, such as *zogenaamde* (*alleged*), which need to be able to take scope over a complex noun, but it seems that modifying PP's never need to take scope over a adjective + noun combination. It is not easy to rule out the latter type of derivation, however, without introducing additional features.

$$\texttt{pp\_p\_np:}\quad {}_{pp}\!\left[\begin{array}{ll}\text{PFORM} & \boxed{1}\\ \text{SEM} & \boxed{2}\\ \text{MOD} & \boxed{3}\end{array}\right] \rightarrow {}_{p}\!\left[\begin{array}{ll}\text{PFORM} & \boxed{1}\\ \text{SC} & \left\langle \boxed{4} \right\rangle\\ \text{SEM} & \boxed{2}\\ \text{MOD} & \boxed{3}\end{array}\right] \boxed{4}\,\mathrm{n}$$

$$\texttt{n\_n\_pp:}\quad {}_{n}\!\left[\begin{array}{ll}\text{SC} & \boxed{1}\\ \text{AGR} & \boxed{2}\\ \text{SEM} & \boxed{3}\end{array}\right] \rightarrow {}_{n}\!\left[\begin{array}{ll}\text{SC} & \boxed{1}\\ \text{AGR} & \boxed{2}\\ \text{SEM} & \boxed{4}\end{array}\right] \; {}_{pp}\!\left[\begin{array}{ll}\text{SEM} & \boxed{3}\\ \text{MOD} & \left\langle \boxed{4} \right\rangle\end{array}\right]$$

Figure 3.10: **PP Rules**

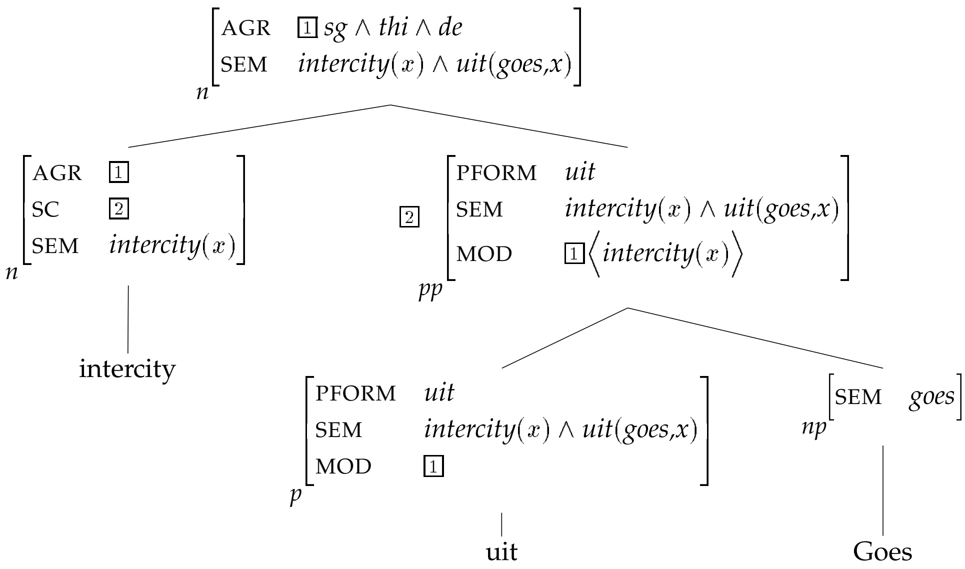


Figure 3.11: intercity uit Goes

3.3.3 Verb Phrases

Both verbs and verb phrases are of type v:

(3.4) $${}_{v}\!\left[\begin{array}{ll}\text{LEX} & \textit{Lex}\\ \text{NULL} & \textit{Null}\\ \text{VFORM} & \textit{Vform}\\ \text{SC} & \textit{listof(Sign)}\\ \text{SUBJ} & \textit{listof(Sign)}\\ \text{SEM} & \textit{Qlf}\\ \text{SLASH} & \textit{listof(Sign)}\\ \text{VSLASH} & \textit{Vslash}\end{array}\right]$$

The features LEX, NULL, and VFORM are specific for v. Their possible values are listed in (5). The feature VFORM is used to distinguish finite, infinitive, *te*-infinitive and past participle verbs (and verb phrases headed by such verbs). The feature LEX is used to distinguish lexical verbs (*ylex*) from verbal phrases that are not lexical (*nlex*). Furthermore, for lexical verbs we distinguish

$$\texttt{vp\_arg\_v:}\quad \begin{bmatrix}\text{LEX} & \textit{nlex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v} \rightarrow \boxed{6}\,\textit{np} \quad \begin{bmatrix}\text{SC} & \left\langle \boxed{6} | \boxed{1} \right\rangle\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v}$$

$$\texttt{vp\_arg\_v:}\quad \begin{bmatrix}\text{LEX} & \textit{nlex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v} \rightarrow \boxed{6}\,\textit{pp} \quad \begin{bmatrix}\text{SC} & \left\langle \boxed{6} | \boxed{1} \right\rangle\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v}$$

$$\texttt{vp\_v\_arg:}\quad \begin{bmatrix}\text{LEX} & \textit{nlex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v} \rightarrow \begin{bmatrix}\text{SC} & \left\langle \boxed{6} | \boxed{1} \right\rangle\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v} \quad \boxed{6}\,\textit{pp}$$

$$\texttt{vp\_v\_arg:}\quad \begin{bmatrix}\text{LEX} & \textit{nlex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v} \rightarrow \begin{bmatrix}\text{SC} & \left\langle \boxed{6} | \boxed{1} \right\rangle\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\end{bmatrix}_{v} \quad \boxed{6} \begin{bmatrix}\text{SC} & \langle\rangle\\ \text{VFORM} & \textit{te}\\ \text{VSLASH} & \langle\rangle\end{bmatrix}_{v}$$

Figure 3.12: VP Complement Rules

between basic and complex lexical units. The latter are combinations of a verb and a separable prefix (*aan+komen*) or combinations of a modal verb and a main verb (*wil vertrekken*). The feature NULL, finally is used to distinguish verbal *traces* (i.e. verbal signs without phonological content) from other verbal signs. The features SUBJ, SLASH, and VSLASH and NULL are discussed in the section below on sentential syntax.

(3.5)	*vform*	fin ∨ inf ∨ te ∨ psp
	lex	ylex ∨ nlex
	ylex	basic ∨ complex
	null	null ∨ nonnull

In Figure 3.12, we list the rules for combining a verb or a verbal projection with one of its complements. As a complement may either precede or follow the verb, two versions of this rule are needed. The predicate `vp_argument/2` specifies which complements may occur as left sister of a verbal head, and which elements can go to the right. Noun phrases can only precede a verbal head (*een kaartje kopen*), PP's may either precede or follow the head (*vanuit Leiden vertrekken, vertrekken vanuit Leiden* , and *te*-infinitives must follow their head (*weigeren naar Groningen te komen*). Note that the type of the complement daughter in these rules is not left unspecified (this would violate the requirements of the parser-grammar interface) but specified indirectly (in the definition of `vp_argument`). The result of combining a verb (or verbal projection) with its complement is a phrase (i.e. the value of LEX on the mother is *nlex*).

The rules in Figure 3.13 specify verbal modification. A modifier can be either an adverb, PP, or temporal NP. At the moment, we allow all of these to either precede or follow the verb (*ik*

$$
\texttt{vp\_mod\_v:}\quad {}_{v}\left[\begin{array}{ll} \text{LEX} & \textit{nlex} \\ \text{SC} & \boxed{1} \\ \text{VFORM} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{VSLASH} & \boxed{4} \end{array}\right] \rightarrow {}_{\textit{modifier}}\left[\begin{array}{ll} \text{MOD} & \left\langle \boxed{5} \right\rangle \end{array}\right] \; {}_{v}\left[\begin{array}{ll} \text{SC} & \boxed{1} \\ \text{VFORM} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{VSLASH} & \boxed{4} \\ \text{SEM} & \boxed{5} \end{array}\right]
$$

$$
\texttt{vp\_v\_mod:}\quad {}_{v}\left[\begin{array}{ll} \text{LEX} & \textit{nlex} \\ \text{SC} & \left\langle\ \right\rangle \\ \text{VFORM} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{VSLASH} & \boxed{4} \end{array}\right] \rightarrow {}_{v}\left[\begin{array}{ll} \text{SC} & \left\langle\ \right\rangle \\ \text{VFORM} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{VSLASH} & \boxed{4} \\ \text{SEM} & \boxed{5} \end{array}\right] \; {}_{\textit{modifier}}\left[\begin{array}{ll} \text{MOD} & \left\langle \boxed{5} \right\rangle \end{array}\right]
$$

$$
\texttt{mod1:}\quad {}_{\textit{modifier}}\left[\begin{array}{ll} \text{MOD} & \boxed{1}\left\langle \_ \right\rangle \end{array}\right] \rightarrow {}_{\textit{adv}}\left[\begin{array}{ll} \text{MOD} & \boxed{1} \end{array}\right]
$$

$$
\texttt{mod2:}\quad {}_{\textit{modifier}}\left[\begin{array}{ll} \text{MOD} & \boxed{1}\left\langle \_ \right\rangle \end{array}\right] \rightarrow {}_{\textit{pp}}\left[\begin{array}{ll} \text{MOD} & \boxed{1} \end{array}\right]
$$

$$
\texttt{mod3:}\ \boxed{1}\ {}_{\textit{modifier}}\left[\begin{array}{ll} \text{MOD} & \left\langle \_ \right\rangle \end{array}\right] \rightarrow {}_{\textit{np}}\left[\begin{array}{ll} \text{NFORM} & \textit{temp} \\ \text{SEM}\ \boxed{2} & \end{array}\right] \quad \text{:-} \quad \texttt{np\_to\_modifier\_sem}(\boxed{1},\boxed{2}).
$$

Figure 3.13: VP Modifier rules

moet morgen in Assen zijn/ in Assen zijn morgen/ morgen zijn in Assen/ tien uur in Assen zijn/?in Assen zijn tien uur). A special type *modifier* (with SEM and MOD as only attributes) in combination with three unary rules is used to introduce the various types of verbal modifier. The fact that adverbs and PP's can be verbal modifiers could also be expressed by defining a predicate `vp_modifier` analogous to the predicate `vp_argument` used in Figure 3.12. The present implementation turns out to be considerably more efficient, however, than an implementation which defines possible modifiers by means of a predicate `vp_modifier`. Note also that temporal NP's have to be transformed into verbal modifiers by means of a lexical rule in any case, as the standard NP-semantics is incompatible with its role as modifier. This 'transformation' is defined by the predicate `np_to_modifier_sem`, which is discussed in the section below on semantics.

A sample derivation is given in Figure 3.14 (the value of the features SLASH and VSLASH is not shown, but is $\langle\ \rangle$ on all verbal signs in this derivation).

It should be noted that we assume that complex verbal expressions have a binary branching structure. Furthermore, we allow modifiers to appear to the left or right of any verbal sign. Note, however, that postverbal modifiers may only be combined with saturated verbal signs (i.e. verbal signs with an empty SC-list). This constraint is added to rule out ambiguities which, in most cases, will be spurious. That is, in cases where an argument precedes a verbal sign and a modifier follows it (as in (6)), only a derivation in which the argument has been combined with the verb before the modifier will be produced (6b). The alternative possibility (6c) seems spurious in most cases. Note also that adding the constraint that postverbal modifiers combine only with saturated verbal signs excludes the possibility of postverbal arguments following a postverbal modifier. If these predictions turn out to be too strict, we may have to remove this constraint. Note also that a certain amount of ambiguity remains. If one modifier precedes a saturated verbal sign, while another follows it (as in (7)), two derivations are possible.

(3.6) a. (dat Rob) de trein mist in Groningen

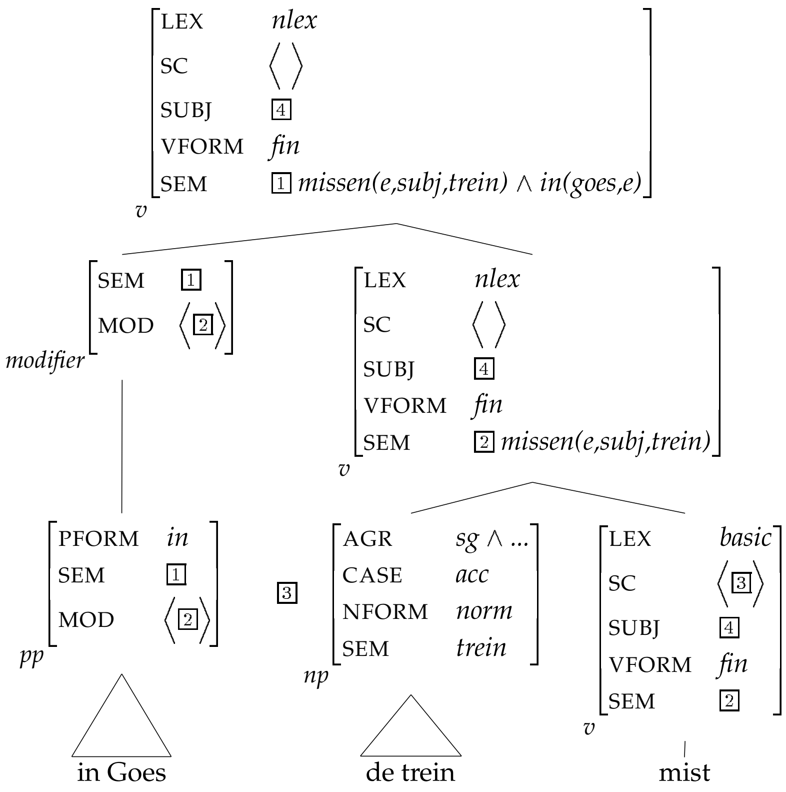


Figure 3.14: (dat Rob) in Goes de trein mist

$$\texttt{v\_part\_v:}\quad {}_{v}\begin{bmatrix}\text{LEX} & \textit{complex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\\ \text{SEM} & \boxed{5}\end{bmatrix} \rightarrow \boxed{6}\,\textit{part}\quad {}_{v}\begin{bmatrix}\text{LEX} & \textit{ylex}\\ \text{SC} & \left\langle \boxed{6} \mid \boxed{1}\right\rangle\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\\ \text{SEM} & \boxed{5}\end{bmatrix}$$

$$\texttt{v\_v\_v:}\quad {}_{v}\begin{bmatrix}\text{LEX} & \textit{complex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\\ \text{SEM} & \boxed{5}\end{bmatrix} \rightarrow {}_{v}\begin{bmatrix}\text{LEX} & \textit{basic}\\ \text{SC} & \left\langle \boxed{6} \mid \boxed{1}\right\rangle\\ \text{VFORM} & \boxed{2}\\ \text{SLASH} & \boxed{3}\\ \text{VSLASH} & \boxed{4}\\ \text{SEM} & \boxed{5}\end{bmatrix}\quad \boxed{6}\ {}_{v}\begin{bmatrix}\text{LEX} & \textit{ylex}\\ \text{VSLASH} & \langle\rangle\end{bmatrix}$$

Figure 3.15: Non-phrasal VP Complement rules

b. (dat Rob) [[[de trein] mist] [in Groningen]]

c. (dat Rob) [[de trein] [mist [in Groningen]]]

(3.7) a. (dat de trein) om tien uur arriveert in Assen

b. (dat de trein) [[[om tien uur] arriveert] [in Assen]]

c. (dat de trein) [[om tien uur] [arriveert [in Assen]]]

Figure 3.15, finally, lists the VP-rules that give rise to 'complex' lexical expressions, instead of phrases. The `v_v_v` rule is used to derive phrases in which a modal verb precedes its infinitival complement (*(dat ik om tien uur) wil vertrekken*). We adopt an analysis of such constructions in which modals *inherit* the arguments on SC of the infinitival verb with which they combine. This is illustrated for the root *wil* in (8).

(3.8)

$$\texttt{wil}\ \mapsto\ {}_{v}\begin{bmatrix}\text{LEX} & \textit{basic}\\ \text{SC} & \left\langle {}_{v}\begin{bmatrix}\text{LEX} & \textit{ylex}\\ \text{SC} & \boxed{1}\\ \text{VFORM} & \textit{inf}\end{bmatrix} \mid \boxed{1}\right\rangle\end{bmatrix}$$

This allows us to derive phrases such as *(dat ik) een kaartje wil kopen* where the finite modal verb combines with the infinitival verb before combining with the object of *kopen* (Figure 3.16). Note that it is essential that the modal verb selects a [LEX *ylex*] argument in this case, as this excludes the derivation of ungrammatical expressions such as *(dat ik) wil een kaartje kopen*. The result of combining a modal with an infinitival verb is [LEX *complex*] (i.e. subsumed by [LEX *ylex*]). This implies that such combinations can be selected by another modal verb (i.e. *(dat ik) een kaartje zou willen kopen*).

The rule `v_part_v` is used to account for constructions such as *(dat ik voor tien uur) aan wil komen*. The prefix (or particle) *aan* of the verb *aankomen* is separated from the root *komen* in this case. As the root *komen* specifies that it selects such a particle on its SC-list, the modal verb inherits this specification. The rule `v_part_v` allows us to combine a verb or verbal complex with a particle. There are two reasons for not using an analogue of the `vp_arg_v`-rule in this case. First,

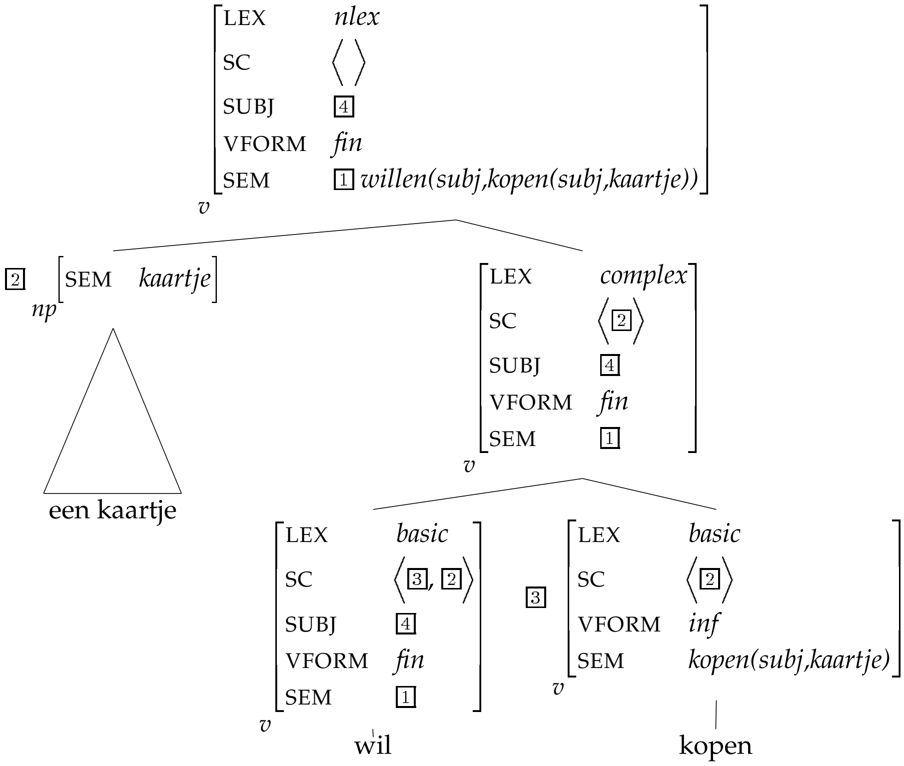


Figure 3.16: (dat ik) een kaartje wil kopen

modifiers may not appear in between a particle and the verbal complex selecting this particle (* *(dat ik) aan om tien uur wil komen*). This is accounted for by requiring that the head in the rule for particles must be [LEX *ylex*] (and combinations of a modifier and a verbal head are always [LEX *nlex*]). Second, particles may appear 'inside' a verb cluster (*(dat ik voor tien uur) zou aan willen komen*). This implies that the result of combining a particle with a verb cluster must be [LEX *ylex*], instead of [LEX *nlex*] as specified on the `vp_arg_v`-rule.

It should be obvious that these two rules, and the limited form of argument inheritance we allow (i.e. structure sharing of SC-lists only, and no concatenation of SC-lists), is not sufficient to account for the full range of verb clustering data in Dutch. For one thing, the grammar as it stands cannot handle 'inverted' word orders (*(dat ik de trein) halen moet*), where the infinitive precedes the modal verb. It is rather straightforward to include rules for inverted word orders. A potentially more problematic omission is the fact that perception verbs (*horen, zien*) and causative *laten*, which also introduce verb clusters (*(dat ik) Rob een kaartje laat kopen*), cannot be accounted for. The reason is that the 'argument inheritance' for such verbs requires concatenation of SC-lists, something which falls outside the scope of the present formalism (see van Noord and Bouma (1995) for a more extensive treatment of Dutch verb clusters in a constraint-based setting).

3.3.4 Subordinate Clauses

Subordinate clauses containing a VP headed by a finite verb are of type *sbar* (the name *sbar* stems from X-bar grammar, where clauses introduced by a complementizer are (barred) projections of S):

(3.9)
$$ {}_{sbar}\begin{bmatrix} \textsc{sc} & \mathit{listof(Sign)} \\ \textsc{sem} & \mathit{Qlf} \\ \textsc{mod} & \mathit{listof(Qlf)} \\ \textsc{slash} & \mathit{listof(Sign)} \end{bmatrix} $$

As finite subordinate clauses are always introduced by a complementizer, we assume that this complementizer is the head of the clause and that it subcategorizes for a subject NP and a (finite) VP. The lexical entry for the complementizer *dat*, for instance, is:

(3.10)
$$ \texttt{dat} \mapsto {}_{sbar}\begin{bmatrix} \textsc{sc} & \left\langle \boxed{1}\ {}_{np}\begin{bmatrix} \textsc{case} & \mathit{nom} \end{bmatrix},\ {}_{v}\begin{bmatrix} \textsc{vform} & \mathit{fin} \\ \textsc{sc} & \langle\, \rangle \\ \textsc{subj} & \langle \boxed{1} \rangle \\ \textsc{sem} & \boxed{2} \\ \textsc{slash} & \boxed{3} \end{bmatrix} \right\rangle \\ \textsc{sem} & \boxed{2} \\ \textsc{mod} & \langle\, \rangle \\ \textsc{slash} & \boxed{3} \end{bmatrix} $$

The complementizer unifies the NP on its SC with the subject of the VP. This implies that the NP is interpreted as subject of the VP. Furthermore, the complementizer has no independent semantics, but simply passes on the semantics of the VP. Since *dat* clauses cannot be modifiers, its MOD feature is empty.

In Figure 3.17, we give the rule constructing subordinate clauses.[7] A sample derivation is given in Figure 3.18.

[7] There is an additional rule, for constructing subordinate clauses with a missing ('extracted') subject . This rule (`sbar2`) could be used in an account of nonlocal dependencies which allows for extraction out of subordinate clauses as well. The rule is of no practical importance at the moment, however, since the feature SLASH can only be shared between signs of type *v* and elements of type *v* on SC of such signs (see the section on nonlocal dependencies below).

$$\texttt{sbar1:}\quad {}_{sbar}\begin{bmatrix} \text{SC} & \langle\, \rangle \\ \text{SLASH} & \boxed{1} \\ \text{SEM} & \boxed{2} \\ \text{MOD} & \boxed{3} \end{bmatrix} \rightarrow {}_{sbar}\begin{bmatrix} \text{SC} & \langle \boxed{4}, \boxed{5} \rangle \\ \text{SLASH} & \boxed{1} \\ \text{SEM} & \boxed{2} \\ \text{MOD} & \boxed{3} \end{bmatrix} \quad \boxed{4}\ {}_{np}\begin{bmatrix} \text{CASE} & nom \end{bmatrix} \quad \boxed{5}\ {}_{v}\begin{bmatrix} \text{SC} & \langle\, \rangle \\ \text{SUBJ} & \langle \boxed{4} \rangle \\ \text{VFORM} & fin \\ \text{SLASH} & \boxed{1} \\ \text{VSLASH} & \langle\, \rangle \end{bmatrix}$$

Figure 3.17: Subordinate clause rule

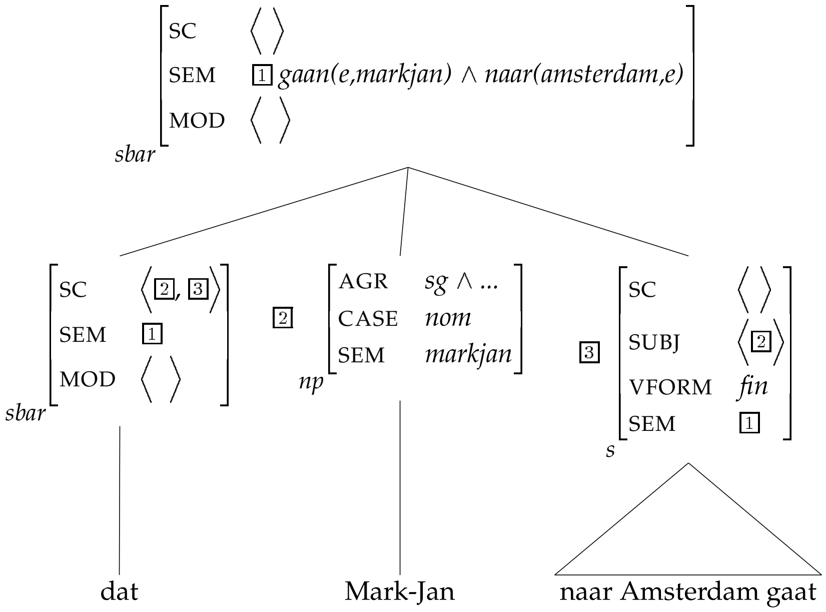


Figure 3.18: dat Mark-Jan naar Amsterdam gaat

$$\texttt{vgap:}\quad {}_{v}\!\left[\begin{array}{ll} \text{LEX} & \textit{basic} \\ \text{NULL} & \textit{null} \\ \text{VFORM} & \textit{fin} \\ \text{SC} & \boxed{1} \\ \text{SUBJ} & \langle\,\rangle \\ \text{SLASH} & \langle\,\rangle \\ \text{VSLASH} & {}_{vslash}\!\left[\begin{array}{ll} \text{VSC} & \boxed{1} \\ \text{VSEM} & \boxed{2} \end{array}\right] \\ \text{SEM} & \boxed{2} \end{array}\right] \rightarrow \epsilon$$

Figure 3.19: Verbal Gap

3.3.5 Main clauses

Main clauses with a finite verb in initial position (i.e. yes/no-questions) are of type *ques* (see (11)). Main clauses in which the finite verb appears in second position (i.e. declarative sentences or WH-questions) are of type *root*.

$$(3.11)\qquad {}_{ques}\!\left[\begin{array}{ll} \text{SUBJ} & \textit{listof(Sign)} \\ \text{SEM} & \textit{Qlf} \\ \text{SLASH} & \textit{listof(Sign)} \end{array}\right] \qquad {}_{root}\!\left[\begin{array}{ll} \text{SEM} & \textit{Qlf} \end{array}\right]$$

Dutch main clauses differ from subordinate clauses in that the finite verb in main clauses appears in first or second position. There is a tradition, both in transformational and non-transformational grammar, to account for this fact by postulating a dependency between the finite verb and the position where finite verbs occur in subordinate clauses. The advantage of postulating such a dependency is that the grammar rules used for subordinate clauses are also applicable in main clauses. In transformational grammar, a depedency of this type can be established by means of a head-movement operation which moves the verb from its final position to a position at the beginning of the sentence.

Within the framework of HPSG (see Netter (1992) and Frank (1994)) we can obtain a similar dependency by postulating a *verbal trace*, i.e. a verbal sign without phonological content, at the end of the clause. Using this verbal trace as the head, we can use the VP rules discussed above to build up a VP as usual.

The rule for introducing such a verbal trace is given in Figure 3.19. Note that the sign for verbal traces differs from that of an ordinary verb in that its subcategorization list in not instantiated, but made reentrant with VSLASH:VSC. Similarly, the semantics of the verbal gap is reentrant with VSLASH:VSEM. Furthermore, a verbal gap is a basic (i.e. non-complex) lexical verb, with no phonological content (i.e. [NULL *null*]). We can also safely assume that verbal traces are finite, as main clauses are always headed by a finite verb. The value of SUBJ is the empty list, as VP's headed by a verbal trace never combine with a subject directly (as will be shown below). Finally, SLASH also can be assumed to be empty.[8]

There are two rules which combine a finite verb with a VP containing a verbal trace, and which also introduce a subject (Figure 3.20). Both are instances of a MAIN-CLAUSE-STRUCT. The *vfirst*-rule introduces phrases of the type *ques*, i.e. instances of verb-first clauses, in which the subject

[8] The lexical rule which moves complements from SC to SLASH does not apply to verbal traces. Instead, it can be applied to the finite verb which 'binds' the trace. Also, if a verbal gap combines with a complement having a non-empty SLASH, the relevant passing on of the SLASH value is handled by the finite verb which binds the trace. This is possible because the SC-list of the verbal trace and the binder will be shared.

$$
{}_{ques}\begin{bmatrix} \text{SLASH} & \boxed{1} \\ \text{SEM} & \boxed{2} \end{bmatrix} \rightarrow
{}_{v}\begin{bmatrix} \text{SC} & \boxed{3} \\ \text{SUBJ} & \left\langle \boxed{4} \right\rangle \\ \text{VFORM} & \textit{fin} \\ \text{LEX} & \textit{basic} \\ \text{SLASH} & \boxed{1} \\ \text{VSLASH} & \left\langle \right\rangle \\ \text{SEM} & \boxed{5} \end{bmatrix}
\;\boxed{4}\,{}_{np}\begin{bmatrix} \text{CASE} & \textit{nom} \end{bmatrix}
\;{}_{v}\begin{bmatrix} \text{SC} & \left\langle \right\rangle \\ \text{SUBJ} & \left\langle \right\rangle \\ \text{VFORM} & \textit{fin} \\ \text{SLASH} & \left\langle \right\rangle \\ \text{VSLASH} & {}_{vslash}\begin{bmatrix} \text{VSC} & \boxed{3} \\ \text{VSEM} & \boxed{5} \end{bmatrix} \\ \text{SEM} & \boxed{2} \end{bmatrix}
$$

$$
{}_{root}\begin{bmatrix} \text{SEM} & \boxed{2} \end{bmatrix} \rightarrow \boxed{4}\,{}_{np}\begin{bmatrix} \text{CASE} & \textit{nom} \end{bmatrix}
\;{}_{v}\begin{bmatrix} \text{SC} & \boxed{3} \\ \text{SUBJ} & \left\langle \boxed{4} \right\rangle \\ \text{VFORM} & \textit{fin} \\ \text{LEX} & \textit{basic} \\ \text{SLASH} & \left\langle \right\rangle \\ \text{VSLASH} & \left\langle \right\rangle \\ \text{SEM} & \boxed{5} \end{bmatrix}
\;{}_{v}\begin{bmatrix} \text{SC} & \left\langle \right\rangle \\ \text{SUBJ} & \left\langle \right\rangle \\ \text{VFORM} & \textit{fin} \\ \text{SLASH} & \left\langle \right\rangle \\ \text{VSLASH} & {}_{vslash}\begin{bmatrix} \text{VSC} & \boxed{3} \\ \text{VSEM} & \boxed{5} \end{bmatrix} \\ \text{SEM} & \boxed{2} \end{bmatrix}
$$

Figure 3.20: Main clause rules

follows the main verb. The *subject-first*-rule introduces phrases of type *root*, in which the subject is first, and the main verb follows the subject. The constraints in MAIN-CLAUSE-STRUCT imply, among others, that the VP must contain a verbal trace, that the SC-information of the main verb is reentrant with VSLASH:VSC of the VP (and thus, indirectly, with the SC-value of the verbal trace), and that the semantics of main verb is shared with the value of VSLASH:VSEM on the VP (and thus, indirectly, with the semantics of the verbal trace). Note also that the VP acts as semantic head of the construction. This is necessary in order to ensure that the effect of verbal modifiers within the VP is properly taken into account.

An example derivation of a subject first main clause is given in Figure 3.21.

3.3.6 Wh-questions and topicalization

In the previous section, we have introduced a rule for verb-initial and subject-initial main clauses. The first phrase in a main clause can also be a (non-subject) complement or a modifier. This is typically the case (non-subject) WH-questions. Sentences with a 'fronted' complement are treated as instances of a non-local dependency construction (where the dependency is mediated through SLASH). In sentences with a fronted modifier, it is always assumed that the first element modifies the remainder of the clause, and thus a local treatment can be given.

Examples of sentences with a fronted complement are given in (12). Note that it is assumed that in sentences such as (12a), where a modal verb occurs with a PP headed by *naar*, this PP is a complement of the modal. In (12b), the fronted element is a complement of the infinitive *halen*, and thus we have a true non-local dependency.

(3.12) a. Naar welk station wilt u?

b. De laatste trein kunt u nog halen.

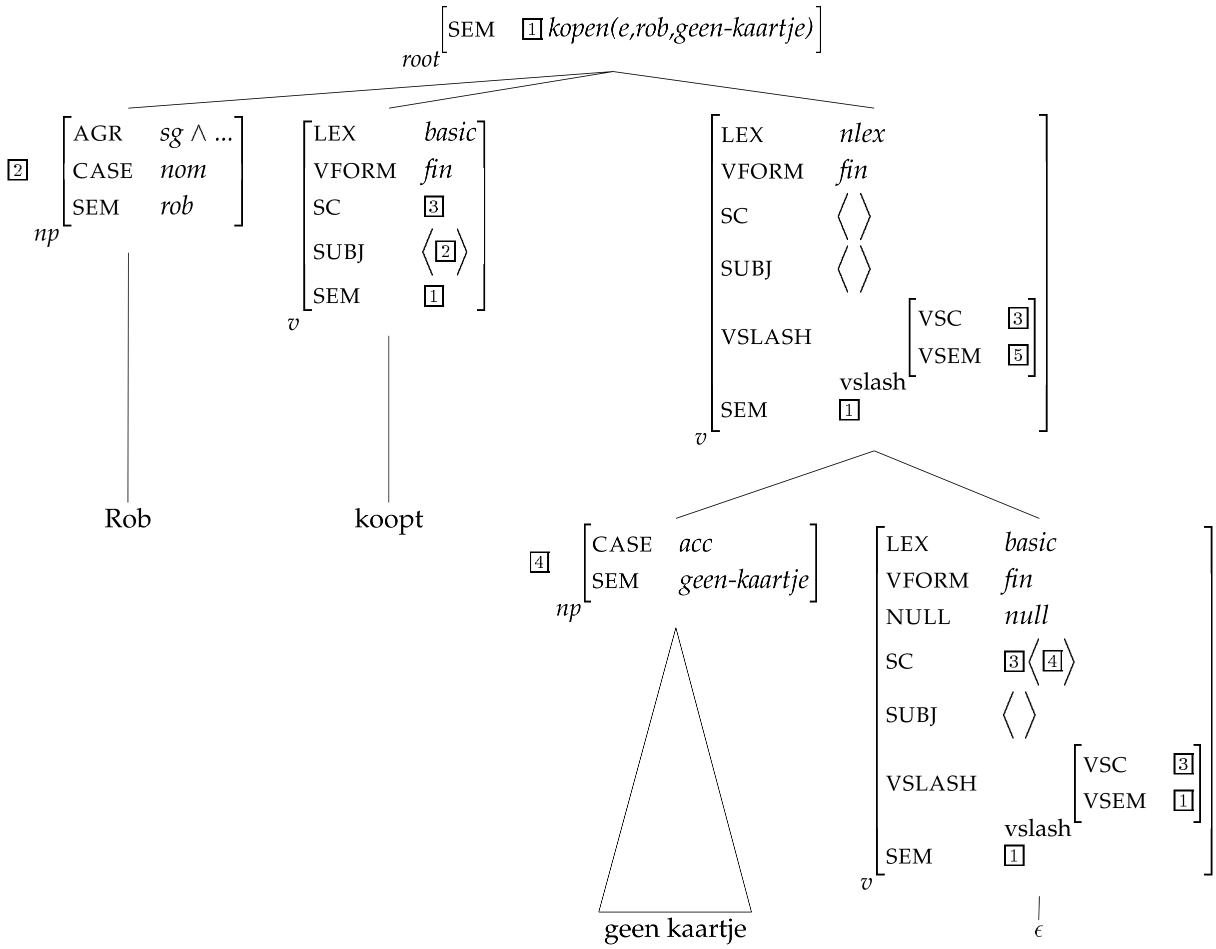


Figure 3.21: Rob koopt geen kaartje

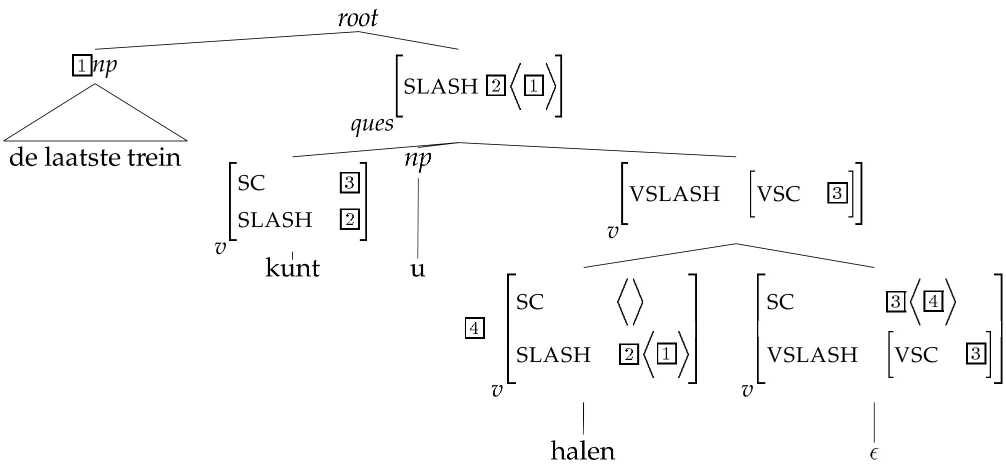


Figure 3.22: De laatste trein kunt u halen

These examples are handled by means of a lexical COMPLEMENT-EXTRACTION rule applicable to verbs, and a syntactic HEAD-FILLER-rule for combining the fronted element with a *ques*-phrase with a non-empty SLASH-value. The COMPLEMENT-EXTRACTION rule can apply in three ways: First, it can take a complement from SC and put it on SLASH (13a). This implies that this complement will not be found locally, but that it will be unified with an element in 'fronted' position. Second, it can make the SLASH value of a verb reentrant with the SLASH value of one of its complements (13b). This implies that true *non-local* dependencies are possible, as the head of a phrase can pass on information about missing elements from one of its dependents. Finally, the rule can apply trivially, in which case the value of SLASH is set to [ ] (13c).

(3.13) a.
$$ {}_{v}\begin{bmatrix} \text{SC} & \langle \boxed{1}\ \text{PP} \rangle \\ \text{SUBJ} & \boxed{2} \\ \text{VFORM} & \boxed{3} \\ \vdots & \end{bmatrix} \rightarrow {}_{v}\begin{bmatrix} \text{SC} & \langle\ \rangle \\ \text{SLASH} & \boxed{1} \\ \text{SUBJ} & \boxed{2} \\ \text{VFORM} & \boxed{3} \\ \vdots & \end{bmatrix} $$

b.
$$ {}_{v}\begin{bmatrix} \text{SC} & \langle v \rangle \\ \text{SUBJ} & \boxed{2} \\ \text{VFORM} & \boxed{3} \\ \vdots & \end{bmatrix} \rightarrow {}_{v}\begin{bmatrix} \text{SC} & \left\langle {}_{v}\begin{bmatrix} \text{SLASH} & \boxed{1} \end{bmatrix} \right\rangle \\ \text{SLASH} & \boxed{1} \\ \text{SUBJ} & \boxed{2} \\ \text{VFORM} & \boxed{3} \\ \vdots & \end{bmatrix} $$

c.
$$ {}_{v}\begin{bmatrix} \text{SLASH} & \_ \\ \text{SC} & \boxed{1} \\ \text{SUBJ} & \boxed{2} \\ \vdots & \end{bmatrix} \rightarrow {}_{v}\begin{bmatrix} \text{SLASH} & \langle\ \rangle \\ \text{SC} & \boxed{1} \\ \text{SUBJ} & \boxed{2} \\ \vdots & \end{bmatrix} $$

An example of a derivation involving SLASH is given in Figure 3.22. The COMPLEMENT EXTRACTION rule has applied to *halen* to produce a verbal sign with an empty SC-list and an *np*

$$_{root}\begin{bmatrix}\text{SEM} & \boxed{1}\end{bmatrix} \rightarrow \boxed{2} \quad _{ques}\begin{bmatrix}\text{SLASH} & \boxed{2} \\ \text{SEM} & \boxed{1}\end{bmatrix}$$

Figure 3.23: Topicalization rule

$$_{root}\begin{bmatrix}\text{SEM} & \boxed{1}\end{bmatrix} \rightarrow \quad _{modifier}\begin{bmatrix}\text{SEM} & \boxed{1} \\ \text{MOD} & \left\langle \boxed{2} \right\rangle\end{bmatrix} \quad _{ques}\begin{bmatrix}\text{SLASH} & \left\langle\; \right\rangle \\ \text{SEM} & \boxed{2}\end{bmatrix}$$

Figure 3.24: Modifier topicalization rule

on SLASH. A verbal trace contains a reentrancy between its SC-list and its VSLASH:VSC-list. As the verbal sign in Figure 3.22 combines with *halen*, the information that *halen* has an *np* on SLASH will therefore also be instantiated on VSLASH:VSC. This information is passed up to the resulting verb phrase. The *complement extraction* rule also applies to the finite verb *kunt*, but in this case it establishes a reentrancy between the SLASH value of the verb on the SC-list of *kunt* and the SLASH-value of *kunt* itself. The VFIRST rule unifies the SC-list of *kunt* with the VSLASH:VSC-list of the verb phrase *halen* ϵ, and thus, SLASH (of the verb on SC of *kunt*, and thus on *kunt* itself) is instantiated as $\langle np \rangle$. This information is passed on to the resulting *ques* phrase, which can then be combined with the initial *np* using the TOPICALIZATION rule in Figure 3.23.

It should be noted that our account of non-local dependencies differs from earlier *slash*-based accounts, such as those in Gazdar et al. (1985) and Pollard and Sag (1994) in that it does not make use of a FOOT FEATURE principle. Instead, we adopt the approach of Sag (1995), who imposes the canonical constraint that the SLASH-value of a head is the set-union of the SLASH-values of its daughters. An EXTRACTION lexical rule can be used to remove an element from SC (COMPS) and to add this element to the set of elements on SLASH. In our implementation, we have made several simplifying assumptions. First, SLASH is not a set, but a list. Second, this list can contain at most one element. This assumption (which has the effect of restricting the number of 'missing' elements from a phrase to at most one) is too restrictive for a highly limited number of cases in English, but appears to be valid for Dutch. Third, instead of imposing a general constraint that SLASH must be the concatenation on the SLASH values of all elements on SC, we allow the COMPLEMENT EXTRACTION rule to unify the value of SLASH with one specific element on SC. We have to make this assumption, as the more general alternative requires the use of *delayed evaluation*, something which we wish to avoid in this grammar, or *difference lists*. While the latter alternative is possible within the present formalism, it also introduces a number of complications which are avoided in the present implementation. The fourth and final simplification is that COMPLEMENT EXTRACTION and SLASH feature passing is only possible for verbs. This is certainly too restrictive, as extraction out of subordinate clauses of type *sbar* (*welke trein zegt Gertjan dat Rob gemist heeft?*) and out of *pp*'s (*Waar gaat deze trein naar toe?*), and a number of other types of phrase is possible as well.

Sentences where the first phrase is a modifier are dealt with without appealing to SLASH. Instead, it is assumed that in sentences such as (14), the fronted elements modify the following *ques* phrase. This requires an additional rule, given in Figure 3.24. Of course, this account rests on the assumption that modifiers of embedded verbs or phrases cannot be fronted, an assumption which is almost certainly false in general (see Hukari and Levine (1994), for instance), but which appears to be rather unproblematic for the kind of data for which the Ovis Grammar is being developed.

(3.14) a. Hoe laat gaat de volgende trein naar Zwolle?

b. Woensdag moet ik om tien uur in Zwolle zijn.

3.4 Special Grammar Rules

The domain which has been selected for the OVIS project (information dialogues concerning public transportation) and the fact that OVIS is about spoken language, imply that it is crucial that a number of grammatical phenomena are described in a robust manner. In particular, temporal expressions, locative expressions (names of cities and stations), and a number of typical spoken language constructions, such as greetings, occur frequently in such dialogues. In this section, we describe how we deal with these phenomena.

The grammar rules discussed below make use of the OVIS grammar formalism, but the rules and lexical entries are not organized according to the linguistic principles discussed above. This is true not only for the syntax, but also for semantics. The reason for dealing with these phenomena by means of a set of more or less *ad hoc* rules and lexical entries is that the constructions discussed below are often extremely idiosyncratic. At the same time, describing the regularities that can be observed does not seem to require the overhead of the grammar architecture we assume for the rest of the grammar. The most economical and robust solution seemed therefore to encapsulate the grammar for these constructions in relatively independent grammar modules.

Temporal Expressions

Temporal expressions occur very frequently in the OVIS domain. It is therefore essential that the grammar provides extensive coverage of these expressions. Some examples of temporal expressions of category NP covered by the grammar are presented in (15). These expressions can be combined with prepositions such as *op (on), om (at), voor (before), na (after)* to form a temporal expression of category PP.

(3.15)	a.	`drie januari`	*January, 3*
	b.	`de derde (januari)`	*the third (of January)*
	c.	`maandag, maandagmiddag`	*Monday, Monday-afternoon*
	d.	`overmorgen`	*the day after tomorrow*
	e.	`drie uur, drie uur vijftien`	*three o'clock, three fifteen*
	f.	`kwart over drie`	*quarter past three*
	g.	`vijftien minuten over (drie)`	*fifteen minutes past (three)*

At the moment, the grammar of temporal expressions is accounted for in a separate module of the grammar. Thus, the general principles which hold for other parts of the grammar (such as the *head-feature principle* and the *slash principle*) need not apply within this module. We believe that such an approach is motivated by the fact that the grammar of temporal expressions contains many idiosyncrasies (such as the fact that *drie uur* combines a singular noun with a plural numeral, or the fact that *vijf over drie* combines two numerals and a preposition) which seem to be limited to this type of expression only. Note that this does not imply that no principled account of the syntax of temporal expressions could be given (see Flickinger (1995) for instance). However, as general principles of the grammar do seem to play only a minor role in the grammar of temporal expressions, we are satisfied with providing a few simple, but idiosyncratic, rules and lexical entries to account for these expressions.

Type-system

Temporal expressions are of type `temp`. To minimalize over-generation, three sub-types (at this moment) are defined: `temp_time`, `temp_date` and `temp_time_coord`. In table 3.1 we give an overview of the prepositions that may subcategorize for temporal NP's and to what types they are restricted.

In the module `times.gram` three top-rules are specified: one for each temporal sub-type. In these rules the type of the analysed (temporal) expression is set to NP. The feature NFORM

Preposition	*Type*
op	temp_date
om,rond,tegen	temp_time
tussen	temp_time_coord
na,voor,van	temp

Table 3.1: Selection restrictions for temporal expressions

receives the value of the temporal sub-type.

We lack a rule for complex temporal expressions like *Maandag drie januari om half vier* (Monday January 3 at three o'clock). These expressions are already analysed by the grammar. They receive an analysis in which the three temporal expressions are all modifiers themselves (e.g. v(mod,v(mod,v(mod,v_{gap})))) and are interpreted in the right way. Adding a rule for complex temporal expressions rule to the times grammar would only increase the number of ambiguities, without increasing the coverage.

In tables 3.2 and 3.3 we give an overview of the existing rules in the times module. The general scheme for these rules is:

```
time_rule(rule_name,Semantics,
   [
     list_of_daughters
   ]).
```

All time expressions can be preceded by *ongeveer* (about), but this word does not contribute to the semantics of the temporal expression.

Rule name	*Example*	*Semantics*
`time_date_0`	*Maandag*	monday
`time_date_1`	*Zeven januari*	date(month(january,7))
`time_date_2`	*De zevende*	date(date(7))
`time_date_3`	*De zevende januari*	date(month(january,7))

Table 3.2: The rules for date expressions

3.4.1 Locative Expressions

The grammar contains rules to recognize locative expressions involving city names, station names, and suffixes of stations names such as *centraal, hoofdstation*. Most of these are dealt with in the lexicon. The lexicon contains about 900 entries for this purpose.

3.4.2 Spoken Language Constructs

A typical property of spoken language is that utterances will often start with introductory remarks such as *hallo, goedemorgen, met Jan Jansen*, etc. Similarly, utterances often are terminated with remarks such as *tot ziens, bedankt, uitstekend, dan weet ik genoeg*, etc. Other phenomena include dialogue markers (which are more meaningful) such as *inderdaad, ik wil naar assen*, and *nou, ik moet er om tien uur zijn*.

Rule name	*Example*	*Semantics*
time_u	*Elf uur*	time(hour(11)))
time_of	*Een uur of elf*	time(hour(11))
time_of2	*Een uur of tien elf*	time(tussen(hour(10),hour(11)))
time_um	*Elf uur tien*	time(past(hour(11),minute(10)))
time_k	*Kwart voor (/over) zeven*	time(before(hour(7),minutes(15)))
time_h	*Half acht*	time(before(hour(7),minutes(30)))
time_m	*Drie (minuten) over acht*	time(past(hour(8),minute(3)))
time_mh	*Drie (minuten) over half acht*	time(before(hour(8),minute(27)))
time_tussenxeny	*Tussen twee (uur) en drie (uur)*	time_coord(tussen(hour(2),hour(3)))
time_uur_en	*tegen tweeen*	time(hour(2))
time_x_honderd_uur	*Achttienhonderd uur*	time(hour(18))

Table 3.3: The rules for time expressions

We have implemented a number of rules and lexical entries to deal with these constructs. However, it is expected that a closer study of the corpus may reveal that additional rules are needed or even that the current attempt should be revised completely.

3.5 Appendix: Grammar Code

3.5.1 Principles and Structures

```
hd_comp_struct(Head,Complements,Mother) :-
        hd_struct(Head,Complements,[],Head,Mother).

hd_mod_struct(Head,Modifier,Mother) :-
        hd_struct(Head,[],[],Modifier,Mother),
        Head:sem <=> HeadSem,
        Modifier:mod <=> [HeadSem].

hd_struct(Head,Complements,Subject,SemanticHead,Mother) :-
        head_feature_principle(Head,Mother),
        valence_principle(Head,Complements,Subject,Mother),
        SemanticHead:sem <=> Mother:sem,
        filler_principle(Head,[],Mother).

main_clause_struct(Head,Subj,VP,Mother) :-
        Head:subj <=> [Subj],
        Head:sem <=> VP:vslash:vsem,
        Head:sc  <=> VP:vslash:vsc,
        Head:vform => fin,
        Head:lex => basic,
        Head:vslash => [],
        filler_principle(Head,[],Mother),
        Subj => np,
        Subj:case => nom,
        VP:sem <=> Mother:sem,
        VP:vform => fin,
        VP:slash => [],
        VP:sc    => [],
```

```
        VP:subj  => [].

hd_filler_struct(Head,Slash,Mother) :-
        filler_principle(Head,Slash,Mother),
        Head:sem <=> Mother:sem.

head_feature_principle(Head,Mother) :-
        unify_ifdef(Head,Mother,vform),
        unify_ifdef(Head,Mother,agr),
        unify_ifdef(Head,Mother,case),
        unify_ifdef(Head,Mother,mod),
        unify_ifdef(Head,Mother,pform),
        unify_ifdef(Head,Mother,aform),
        unify_ifdef(Head,Mother,vslash).

valence_principle(Head,Complements,Subject,Mother) :-
        ifdef(sc,Head,HeadSc,[]),
        ifdef(sc,Mother,MotherSc,[]),
        append(Complements,MotherSc,HeadSc),
        ifdef(subj,Head,HeadSubj,[]),
        ifdef(subj,Mother,MotherSubj,[]),
        append(Subject,MotherSubj,HeadSubj).

filler_principle(Head,Slash,Mother) :-
        ifdef(slash,Head,HeadSlash,[]),
        ifdef(slash,Mother,MotherSlash,[]),
        append(Slash,MotherSlash,HeadSlash).
```

3.5.2 Rules

```
rule(np_det_n, NP,  [ Det, N ] ) :-        % np rules
        NP=>np, Det=>det, N=>n,
        NP:nform => norm,
        hd_comp_struct(Det,[N],NP).

rule(n_adj_n, N1,   [ AdjP, N0 ] ) :-
        N1 => n, AdjP => a, N0 => n,
        AdjP:agr <=> N0:agr,
        hd_mod_struct(N0,AdjP,N1).

rule(n_n_pp, N1,   [ N0, PP ] ) :-
        N1 => n, N0 => n, PP => pp,
        hd_mod_struct(N0,PP,N1).

rule(pp_p_np, PP,   [ P, NP ] ) :-        % pp rules
        PP => pp, P => p, NP => np,
        hd_comp_struct(P,[NP],PP).

rule(vp_arg_v,VP, [Arg, V] ) :-           % vp argument rules
        VP => v, V => v,
        VP:lex => nlex,
        vp_argument(left,Arg),
        hd_comp_struct(V,[Arg],VP).
```

```
rule(vp_v_arg,VP, [V, Inf] ) :-
        VP => v, V => v,
        VP:lex => nlex,
        vp_argument(right,Inf),
        hd_comp_struct(V,[Inf],VP).

vp_argument(left,Arg) :-                  % definition of vp_argument
        Arg => np.

vp_argument(_,Arg) :-
        Arg => pp.

vp_argument(right,Arg) :-
        Arg => v,
        Arg:vform => te,
        Arg:sc => [],
        Arg:vslash => [].

rule(vp_mod_v, VP, [ Mod, V ] ) :-       % vp modifier rules
        VP => v, Mod => modifier, V => v,
        VP:lex => nlex,
        hd_mod_struct(V,Mod,VP).

rule(vp_v_mod, VP, [ V, Mod ] ) :-
        VP => v, V => v, Mod => modifier,
        VP:lex => nlex,
        V:sc => [],
        hd_mod_struct(V,Mod,VP).

rule(mod1, Mod, [ Adv ] ) :-             % unary modifier rules
        Mod => modifier, Adv => adv,
        Adv:mod <=> [_],
        hd_struct(Adv,[],[],Adv,Mod).

rule(mod2, Mod, [ Pp ] ) :-
        Mod => modifier, Pp => pp,
        Pp:mod <=> [_],
        hd_struct(Pp,[],[],Pp,Mod).

rule(mod3, Mod, [ Np ] ) :-
        Mod => modifier, Np  => np,
        Np:nform => temp,
        Np:sem  <=> NpSem,
        np_to_modifier_sem(Mod,NpSem).

rule(v_part_v,VP, [Part, Inf]) :-       % non-phrasal vp argument rules
        VP => v, Part => part, Inf => v,
        VP:lex => complex,
        Inf:lex => ylex,
        hd_comp_struct(Inf,[Part],VP).

rule(v_v_v,VP, [V, Inf] ) :-
        VP => v, V => v, Inf => v,
```

```
        Inf:vslash => [],
        VP:lex => complex,
        V:lex => basic,
        Inf:lex => ylex,
        hd_comp_struct(V,[Inf],VP).

rule(sbar1, Sbar, [ Comp, NP, VP ]) :-   % subordinate clause rules
        Sbar => sbar, Comp => sbar, NP => np, VP => v,
        Sbar:sc <=> [],
        VP:vslash => [],
        VP:vform => fin,
        hd_comp_struct(Comp,[NP,VP],Sbar),

rule(sbar2, Sbar, [ Comp, VP]) :-
        Sbar => sbar, Comp => sbar, VP => v,
        Sbar:sc <=> [],
        VP:vslash => [],
        VP:vform => fin,
        hd_comp_struct(Comp,[VP],Sbar),

rule(vgap,V,[]) :-                      % verbal gaps
        V => v,
        V:lex => basic,
        V:null => null,
        V:vform => fin,
        V:vslash:vsc <=> V:sc,
        V:vslash:vsem <=> V:sem,
        V:slash => [],
        V:subj => [].

rule(vfirst,Q,[Vmain,Subj,VP]) :-       % main clause rules
        Q => ques, Vmain => v, Subj => np, VP => v,
        main_clause_struct(Vmain,Subj,VP,Q).

rule(subject_first,Root,[Subj,Vmain,VP]) :-
        Root => root, Subj => np, Vmain => v, VP => v,
        main_clause_struct(Vmain,Subj,VP,Root).

rule(topicalization, Root, [Topic, Ques]) :-
        Root => root,
        ( Topic => np ; Topic => pp ),
        Ques => ques,
        hd_filler_struct(Ques,[Topic],Root).

rule(mod_topic, Root, [Mod, Ques]) :-
        Root => root, Mod => modifier, Ques => ques,
        Ques:slash => [],
        hd_mod_struct(Ques,Mod,Root).
```

Chapter 4

Semantic Representation in Ovis

The output of the grammatical analysis is a semantic, linguistically motivated and domain-independent, representation of the utterance. This representation is translated into one or more domain-specific *updates*, which are passed on to the pragmatic interpretation module and dialogue manager (DM) for further processing. In the previous sections we motivated the choice for an HPSG-style grammar in which we can couple syntactic and semantic processing as tight as we want to. This means that these kinds of grammar formalisms allow us to choose our favourite semantic representation language. Below, we motivate our choice for *quasi logical forms* (QLF's) as semantic representation language and we discuss how these QLF's are translated into updates.

4.1 The semantic representation language

Disambiguation is still a major problem in computational linguistics Although, because of the restricted domain, the number of sources that can introduce ambiguities on linguistic grounds is rather limited in OVIS, we do have to account for highly ambiguous word-graph input. There are many ambiguities that are introduced either on linguistic grounds or by the speech-recognizer that are resolved by the context in which the utterance occurred. We illustrate this with two examples. First, sentences involving for example a negation operator and a quantifier like (4.1) are ambiguous with respect to the scope of these operators. In the context of this domain, however, we know that we do not want to consider the second reading (4.1c).

(4.1) $a.$ `Gaat er niet een latere (trein)?`
$b.$ $not(\exists x latere\_trein(x) \wedge gaat(x))$
$c.$ $\exists x latere\_trein(x) \wedge not(gaat(x))$

The second example involves an ambiguity introduced by the speech-recognizer. After the system asked the user question (4.2a), the speech-recognizer returns a word-graph that contains (among other expressions) (4.2b) and (4.2c). Although both expressions are acceptable according to the grammar, it is clear that we don't want to consider (4.2c) as an answer to (4.2a).

(4.2) $a.$ `System: Wilt u dat ik de verbinding nog eens herhaal?`
$b.$ `User: Nee, dank u`
$c.$ `User: Negen uur`

Predicate logic is often used to represent the meaning of sentences. Due to its long tradition in describing semantics of natural languages it is now a well established and well understood technique. Nevertheless, it often seems helpful to extend the predicate calculus with more advanced techniques, such as *generalised quantifiers* or *discourse markers*.

The main advantage of artificial languages like predicate logic is that they are unambiguous. An ambiguous natural language utterance will therefore correspond to more than one expression in predicate logic, one for each reading of the utterance. The disadvantage of this approach is that for very ambiguous inputs, expensive computations must be carried out to compute all readings. The alternative adopted in *monotonic semantic interpretation* based formalisms (Cooper et al., 1994) is to represent ambiguity by means of underspecification and to postpone the computation of all possible readings (See also (Nerbonne, 1992), (Pinkal, 1995)).

The most important features of the QLF-formalism (the implementation of the monotonic semantics approach that we decided to use in OVIS) are, that it both makes extensive use of underspecification to represent ambiguities and supports the advanced techniques we mentioned before. Representing ambiguity by means of underspecification is particularly important in the present setting, as the grammar must process with highly ambiguous input (i.e. word graphs). Furthermore, it turns out to be the case that the computation of the domain-specific meaning of an utterance can be carried out without resolving all the sources of ambiguity in the QLF for that utterance. For instance, the scope of quantifiers seems to be to a large extent irrelevant for the computation of (domain-specific) *updates*, and thus a representation in which quantifier-scope is undetermined is sufficient for our purposes.

4.1.1 Quasi Logical Form

A QLF is a *description* of one or more predicate logical formulas. While building a QLF for an utterance information is added *monotonically* (Cooper et al., 1994). Whenever information is added, the amount of underspecification will decrease and, consequently, the resulting QLF will describe a smaller set of logical formulas. After grammatical processing of an utterance is completed, several *knowledge sources* (such as an algorithm for generating quantifier scope or for finding referents for anaphoric expressions) may be used to further reduce the amount of underspecification.

In figure 4.1 we give the (simplified) QLF which the current grammar produces for the example sentence *Ik wil om ongeveer vier uur vertrekken* (I want to leave at about four o'clock). Remember that the grammar manipulates feature-structures, so the QLF is represented as a feature-value matrix. Note that the arguments of the predicates *willen* and *vertrekken* are (generalized) quantifiers that are unscoped with respect to each other.

$$
\begin{bmatrix}
\textit{pred} : \text{and} \\
\textit{args} : \left\langle
\begin{bmatrix}
\textit{pred} : \text{willen} \\
\textit{args} : \left\langle
\begin{bmatrix} \textit{index} : \text{A} \\ \textit{res} : \lambda\text{E.event(E)} \\ q : \text{exist} \end{bmatrix},
\begin{bmatrix} \textit{index} : \text{I} \\ \textit{res} : \lambda\text{K.name(K,ik)} \\ q : \text{exist} \end{bmatrix},
\right. \\
\qquad\qquad \left.
\begin{bmatrix}
\textit{pred} : \text{vertrekken} \\
\textit{args} : \left\langle \begin{bmatrix} \textit{index} : \text{A} \\ \textit{res} : \lambda\text{S.event(S)} \\ q : \text{exist} \end{bmatrix}, \text{I} \right\rangle
\end{bmatrix}
\right\rangle
\end{bmatrix},
\begin{bmatrix} \textit{pred} : \text{at} \\ \textit{args} : \langle \text{A}, \text{hour}(4) \rangle \end{bmatrix}
\right\rangle
\end{bmatrix}
$$

Figure 4.1: (Simplified) QLF for the utterance 'Ik wil om ongeveer vier uur vertrekken'

The QLF in Figure 4.1 represents the domain-independent meaning of the utterance. If we want to reuse the present grammar for extensions of the present domain or for other domains, it is essential to have representations that are not based on domain-specific assumptions.

Events

Events are handled in the tradition of Davidson (Davidson, 1967). Verbs introduce a new (existentially quantified) event variable. When modifiers are attached to a verb, the translation of this modifier is conjoined with the translation of the verb. Both conjuncts share the same event-variable. The translation of *De trein vertrekt om vier uur* (The train leaves at four o'clock) will be

(4.3a) [1] instead of (4.3b).

(4.3) *a.* `and(vertrekken( exist(e,event(e)),de(x,trein(x))),`
`at(e,hour(4)))`
b. `vertrekken(de(x,trein(x)), at(hour(4)) )`

Note that sentences like the one in 4.1 are ambiguous with respect to the attachment of the prepositional phrase. The PP *om vier uur* can either modify *willen* or *vertrekken*. In the context of this project, however, we do not want to worry about this ambiguity. Currently we make the (domain specific) assumption that the modal and the vp-complement both refer to the same event.

4.1.2 The implementation of QLF's

The QLF formalism was developed in the framework of the *Core Language Engine* (CLE, (Alshawi, 1992)). Since then, the formalism was used and further developed in projects such as the *Spoken Language Translator* (Agnäs et al., 1994), *Clare* (Alshawi et al., 1992), in the *Fracas*-project (Cooper et al., 1994) and in *Trace & Unification Grammar* (Block, 1994).

Although our implementation of QLF in the OVIS grammar deviates from it in many ways, it was most inspired by Cooper et al. (1994). Because we think that we can mostly translate QLF's in updates directly, it will not be necessary to resolve the QLF's completely and this gives us the opportunity to omit some of the apparatus supplied by the original formalism. It may well be the case that we do need some of these features later on. These can then be added at a later stage.

The type-system

A QLF is either a *qlf-term* or a *qlf-formula*. A *qlf-term* must be one of the following:

- a term index or variable
- a constant term
- an expression of the form: `t_expr(Index,Cat,Restr,Quant)` [2]
  in which `Cat` is a list of feature-value equations, denoting some extra-linguistic information. At this moment it includes only the features `lex` (the lexical entry associated with this term) and `type` (e.g. quantifier, pronoun, etc.); `Restr` is a predicate logical expression (possibly with lambda-abstraction); *Quant* is a generalized quantifier/determiner.

In the original formalism *indices* and *variables* are distinguished. An index is introduced to uniquely identify a `term` expression. At this moment they do the same job in our implementation. If we decide to do more with resolving underspecified information, we will probably need to distinguish between them.

A QLF formula must be one of the following:

- the application of a predicate to arguments:
  `p_form(Predicate,[Arg1,...,Argn])` in which *Predicate* is a higher order predicate (including logical operators like `and`, `not`, etc). Arguments may be formulas or terms.
- a predicate logical formula with lambda-abstraction:
  `v_form(Var,Formula)`. This type does not exist in the original formalism. There, any predicate may also be a lambda abstraction. Here we decided to add this type for reasons of accessibility of variables that are hidden deep down in the structure otherwise.

[1] Note that this is a QLF-style formula in which nothing is said about the scope of the (generalized) quantifiers. After resolving this QLF the modifier with the event variable will be in the scope of the existential quantifier

[2] In chapter 5 of (Cooper et al., 1994) a fifth argument *Referent* is used. With this argument a contextual restriction on the range of quantification can be specified.

Every man
qlf $\mathtt{t\_expr}(\mathtt{i}, \langle \mathtt{lex} = \mathtt{alle}, \mathtt{type} = \mathtt{q}\rangle, \lambda \mathtt{x}.\mathtt{man}(\mathtt{x}), \forall)$
pl $\forall x.(man(x) \rightarrow ....)$

yawns
qlf $\mathtt{v\_form}(\mathtt{x}, \mathtt{s\_form}(\mathtt{Scope}, \mathtt{p\_form}(\mathtt{yawns}(\mathtt{t\_expr}(\mathtt{e}, \langle \mathtt{type} = \mathtt{event}\rangle, \lambda \mathtt{y}.\mathtt{event}(\mathtt{y}), \exists), \mathtt{x}))))$
pl $\lambda x.\exists y.(event(y) \wedge slapen(y, x))$

Every man yawns
qlf $\mathtt{s\_form}(\mathtt{Scope}, \mathtt{p\_form}(\mathtt{slapen}(\mathtt{t\_expr}(\mathtt{e}, ...), \mathtt{t\_expr}(\mathtt{i}, ...))))$
pl $\forall x.(man(x) \rightarrow (\exists y.event(y) \wedge yawns(y, x)))$
$\exists y.(event(y) \wedge \forall x.(man(x) \rightarrow yawns(y, x)))$

Figure 4.2: The relation between an expression in QLF and a predicate logical formula

- a formula with scoping constraints: `s_form(Scope,Formula)` in which *Scope* is either a meta-variable when scoping is underspecified or a list of term-indices occurring in Formula, in the order in which they are scoped, e.g. [+i,+j] if index +i out-scopes index +j. [3]

In chapter 5 of (Cooper et al., 1994) two more formula constructs are introduced. Both of them (expressions of the form `form(Idx,Category, Restriction,Resolution)` and formulas with a re-interpretation (`Formula:{Term1/Term2,Term3/Term4,...}`) are used to incorporate contextual (discourse) information for resolving the QLF's. These are not used in the current implementation.

The type-sytem can best be illustrated with a simple example in which we compare a QLF expression with its corresponding formula in predicate logic. In figure 4.2 the sentence *Every man yawns* is given both a translation in QLF and in Predicate Logic. In the translation of the full sentence we see that nothing is said about the scope of respectively the quantifier of the event associated with the verb and the quantifier of its argument. All the information that is needed to resolve the scoping relation is stored in the arguments (of type `t_expr`) of *yawn*. Resolving the scoping relation would instantiate the variable *Scope* to [i,e] (for every man there exists a moment that he yawns) or to [e,i] (there exists a moment that every man yawns).

Construction of QLF's

During grammatical analysis QLF's are constructed compositionally. Two kinds of structures can be distinguished: *head complement* and *head modifier* structures. Treatment of these two structures is rather straightforward. In *head complement structures* the head daughter is the syntactic as well as the semantic head of the structure. This means that the semantic content of the complement constituents is combined with the semantic content of the head. The value of the SEM feature of the head is passed up to the mother. The composition involves application of `hd_struct/5` (see figure 3.1), in which *Head* is equal to *SemanticHead*.

In *head modifier structures* the modifier is the semantic head. The semantics of the syntactic head of the structure are plugged into the MOD feature of the modifier. In the next section (on the treatment of the semantics of the lexical categories) we will see how the semantics of the modifier is combined with the semantics of the constituent it modifies. The value of the SEM feature of the modifier is passed up to the mother.

```
hd_mod_struct(Head,Modifier,Mother) :-
```

[3] In fact scoping information is currently not used. Until now, we have not resolved unscoped QLF's. Although during grammatical analysis some scoping information is added, this information is not considered while translating the QLF's to updates. We will come back to this issue later on.

```
        hd_struct(Head,[],[],Modifier,Mother),
        Head:sem <=> HeadSem,
        Modifier:mod <=> [HeadSem].
```

Templates

In this section we explain the treatment of the semantics for each lexical category. We will do this by means of feature structures representing templates we have defined for describing the semantic contents. Apart from templates for these categories, we have also defined templates for the QLF-types we have described before. In the first definition (`det_sem/2`) we give both the predicate and the corresponding feature structure, illustrating the way it is implemented in the current grammar.

NP **semantics**

```
det_sem(S,Quantifier) :-
        term_sign(S,_Index,Cat,Restriction,Quantifier),
        S:sc <=> [N],
        N:sem <=> Restriction,
        Cat:lex  <=> Quantifier,
        Cat:type ==> q.
```

$$\begin{bmatrix} \text{DET} \\ sc : \langle \begin{bmatrix} \text{N} \\ sem : \text{Restriction}\end{bmatrix} \rangle \\ sem : \begin{bmatrix} \text{T\_EXPR} \\ cat : \begin{bmatrix} \text{CAT} \\ lex : \text{Quantifier} \\ type : \text{q} \end{bmatrix} \\ res : \text{Restriction} \\ q : \text{Quantifier} \end{bmatrix} \end{bmatrix}$$

Figure 4.3: The determiner semantics template: det_sem/2

In figure 4.3 we see first the predicate `det_sem/2`. All templates (corresponding to predicates *cat*`_sem/2`) that are explained in this section are defined in the same way. The first argument is the *sign* that is defined by this template. The second argument specifies the case-specific information (e.g. in figure 4.3 the quantifier that is involved here). In the definition we use the templates we have defined for the QLF-types. Here we say that the value of the SEM feature of a determiner must be of type *term_expression* by calling the corresponding template `term_sign/5`.

Determiners are treated as heads. They subcategorize for a constituent of type N. The semantics of that noun specifies the restriction of the determiner. In the original QLF formalism the value of feature `q` (quantifier) is first left underspecified. The translation of the determiner might be dependent on contextual information. The determiner *een*, for example, is usually translated by the existential quantifier, but in sentences like *Een intercity stopt niet in Beilen*, it should be translated by a universal quantifier. We expect however, that translating directly will not cause problems in this application. When this will be the case, we can leave the value of the Q feature unspecified (at this level) and add some extra apparatus for resolving the underspecification at a later stage).

$$\begin{bmatrix} \text{NP} & \\ \mathit{sem}: & \begin{bmatrix} \text{T\_EXPR} & \\ \mathit{cat}: & \begin{bmatrix} \text{CAT} \\ \mathit{lex}: \text{PN} \\ \mathit{type}: \text{name} \end{bmatrix} \\ \mathit{res}: & \begin{bmatrix} \text{V\_FORM} & \\ \mathit{var}: \text{F} & \\ \mathit{form}: & \begin{bmatrix} \text{P\_FORM} \\ \mathit{pred}: \text{name} \\ \mathit{arglist}: \langle \text{F}, \text{PN} \rangle \end{bmatrix} \end{bmatrix} \\ q: \text{exist} & \end{bmatrix} \end{bmatrix}$$

Figure 4.4: The proper name semantics template: pn_sem/2

$$\begin{bmatrix} \text{NP} & \\ \mathit{sem}: & \begin{bmatrix} \text{T\_EXPR} & \\ \mathit{cat}: & \begin{bmatrix} \text{CAT} \\ \mathit{lex}: \text{Pron} \\ \mathit{type}: \text{pronoun} \end{bmatrix} \\ \mathit{res}: & \begin{bmatrix} \text{V\_FORM} & \\ \mathit{var}: \text{E} & \\ \mathit{form}: & \begin{bmatrix} \text{P\_FORM} \\ \mathit{pred}: \text{name} \\ \mathit{arglist}: \langle \text{E}, \text{Pron} \rangle \end{bmatrix} \end{bmatrix} \\ q: \text{exist} & \end{bmatrix} \end{bmatrix}$$

Figure 4.5: The pronoun semantics template: pron_sem/2

The templates for *proper names* and *pronouns* are almost identical (figure 4.4 and 4.5). The lexical entry is plugged into the second argument of predicate `name/2`. The first argument is a variable by which the NP can be modified. The translation of *slimme Jan* (smart John) will for example be:

(4.4) $\lambda x.and(slimme(x), name(x, \text{J}an))$

Obviously the two categories differ in lexical type (specified under the CAT feature).

There is little to say about the semantic content of nouns (figure 4.6). The only thing that is special here, is that it is assumed that quantifiers may scope at this point. The value of the `scope` feature is left unspecified here (and in fact we do not resolve the scoping relation at a later stage either).

Adjectives (figure 4.7 are modifiers. They operate on structures whose semantic content is of type `v_form`. The lambda variables of the two formulas are unified and the semantic content of the structure is the conjunction of the logical formula of the adjective and the logical formula of the structure it modifies.

Verbal semantics

The semantic content of the subject with which the *intransitive verb* combines, is stored in the SUBJ feature. This information is plugged into the second position of the argument list of the verb.

$$
\begin{bmatrix}
\textsc{N} & \\
\textit{sem}: & \begin{bmatrix}
\textsc{V\_FORM} & \\
\textit{var}: \mathrm{F} & \\
\textit{form}: & \begin{bmatrix}
\textsc{S\_FORM} & \\
\textit{form}: & \begin{bmatrix}
\textsc{P\_FORM} \\
\textit{pred}: \mathrm{Noun} \\
\textit{arglist}: \langle \mathrm{F} \rangle
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
$$

Figure 4.6: The noun semantics template: noun_sem/2

$$
\begin{bmatrix}
\textit{sem}: & \begin{bmatrix}
\textsc{V\_FORM} & \\
\textit{var}: \mathrm{V} & \\
\textit{form}: & \begin{bmatrix}
\textsc{S\_FORM} & \\
\textit{scope}: \mathrm{W} & \\
\textit{form}: & \begin{bmatrix}
\textsc{P\_FORM} \\
\textit{pred}: \mathrm{and} \\
\textit{arglist}: \left\langle \begin{bmatrix}
\textsc{P\_FORM} \\
\textit{pred}: \mathrm{Adj} \\
\textit{arglist}: \langle \mathrm{V} \rangle
\end{bmatrix}, \mathrm{X} \right\rangle
\end{bmatrix}
\end{bmatrix}
\end{bmatrix} \\
\textit{mod}: & \left\langle \begin{bmatrix}
\textsc{V\_FORM} & \\
\textit{var}: \mathrm{V} & \\
\textit{form}: & \begin{bmatrix}
\textsc{S\_FORM} \\
\textit{scope}: \mathrm{W} \\
\textit{form}: \mathrm{X}
\end{bmatrix}
\end{bmatrix} \right\rangle
\end{bmatrix}
$$

Figure 4.7: The adjective semantics template: adj_sem/2

The verb introduces a new event. The first argument of the argument list of the verb is filled with the term representing the event. The index associated with this event is unified with the variable of V_FORM at the highest level. This might appear to be strange (remember that a V_FORM is a lambda abstraction); it is done because it makes the event variable much easier to access.

The *transitive verb* is defined similar to the intransitive verb. The only extra information we have to take care of here is the second argument of the verb. The second argument can be found on subcat (i.e the third element on the verb's argument list must be unified with the value of the SEM feature of the element on SC).

$$\left[\begin{array}{ll}
\text{V} & \\
\textit{sem}: & \left[\begin{array}{ll}
\text{V_FORM} & \\
\textit{var}: \text{A} & \\
\textit{form}: & \left[\begin{array}{ll}
\text{S_FORM} & \\
\textit{form}: & \left[\begin{array}{ll}
\text{P_FORM} & \\
\textit{pred}: \text{IV} & \\
\textit{arglist}: \langle & \left[\begin{array}{ll}
\text{T_EXPR} & \\
\textit{index}: \text{A} & \\
\textit{cat}: & \left[\begin{array}{l} \text{CAT} \\ \textit{type}: \text{e} \end{array}\right] \\
\textit{res}: & \left[\begin{array}{ll}
\text{V_FORM} & \\
\textit{var}: \text{D} & \\
\textit{form}: & \left[\begin{array}{l} \text{P_FORM} \\ \textit{pred}: \text{event} \\ \textit{arglist}: \langle \text{D} \rangle \end{array}\right]
\end{array}\right] \\
\textit{q}: \text{exist} &
\end{array}\right] , \text{E}\rangle
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\textit{subj}: & \langle \left[\begin{array}{l} \text{NP} \\ \textit{sem}: \text{E} \end{array}\right] \rangle
\end{array}\right]$$

Figure 4.8: The intransitive verb semantics template: iv_sem/2

The subject argument of the *modal verb* is taken from the SEM feature of SUBJ. The index, which uniquely identifies this term, is unified with the value of SEM of the subject argument of the vp-complement. The reason that just the index is passed through to the vp-complement is to prevent undesired readings in case of quantified terms (*Iedere reiziger wil vertrekken* (Every traveller wants to leave) cannot have the reading that every traveller wants that every traveller leaves). The vp-complement argument is unified with the value of SEM of the V on the subcat list. We have mentioned before that we want to prevent the grammar from generating two readings for sentences with a modal verb and some modifier. Currently the modal verb and the vp-complement both refer to the same event. This is done by unifying the event variable of the modal verb with the event variable of the vp-complement.
In Dutch it is quite common to say something like *Ik wil vanmiddag naar Assen* (I want this afternoon to Assen). These sentences receive a domain-specific analysis. We will come back to this issue in the next paragraph.

Finally we discuss the verbal modifiers. The semantics of *adverbs* is handled in the same way as for *adjectives* (see figure 4.7). The feature-structure in figure 4.10 shows the template for the *preposition* as modifier. It subcategorizes for a NP. Its semantics consist of a conjunction of the formula of the constituent it modifies and an extra constraint on this formula. The constraint is headed by the preposition and its arguments are the lambda variable of the formula it constrains and the formula of the NP it subcategorizes for.

Prepositions can also occur as arguments to verbs. For example in sentences like *Jan geeft een boek aan Marie* (John gives a book to Mary) the PP *aan Marie* is the second argument of *geven*. In the current grammar PP's only occur in argument position in special constructs for sentences like *Ik wil vanmiddag naar Assen*. The value of the SEM feature of this preposition is unified with the SEM feature of the NP it subcategorizes for.

$$
\left[\begin{array}{l}
\textsc{v} \\
sc : \left\langle \left[\begin{array}{l}
\textsc{v} \\
sc : \mathrm{B} \\
vform : \textsc{inf} \\
sem : \left[\begin{array}{l} \textsc{v\_form} \\ var : \mathrm{D} \end{array}\right]_{\mathrm{F}_1} \\
subj : \left\langle \left[\begin{array}{l} \textsc{np} \\ sem : \mathrm{I} \end{array}\right] \right\rangle
\end{array}\right] \middle| \mathrm{B} \right\rangle \\
sem : \left[\begin{array}{l}
\textsc{v\_form} \\
var : \mathrm{D} \\
form : \left[\begin{array}{l}
\textsc{s\_form} \\
form : \left[\begin{array}{l}
\textsc{p\_form} \\
pred : \mathrm{ModalVerb} \\
arglist : \left\langle \left[\begin{array}{l}
\textsc{t\_expr} \\
index : \mathrm{D} \\
cat : \left[\begin{array}{l} \textsc{cat} \\ type : \mathrm{e} \end{array}\right] \\
res : \left[\begin{array}{l} \textsc{v\_form} \\ \lambda\, \mathrm{M.event(M)} \end{array}\right] \\
q : \mathrm{exist}
\end{array}\right], \left[\begin{array}{l} \textsc{t\_expr} \\ index : \mathrm{I} \end{array}\right]_{\mathrm{W}_1}, \mathrm{F}_1 \right\rangle
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
subj : \left\langle \left[\begin{array}{l} \textsc{np} \\ sem : \mathrm{W}_1 \end{array}\right] \right\rangle
\end{array}\right]
$$

Figure 4.9: The modal verb semantics template: modal_sem/2

$$\left[\begin{array}{l} \text{P} \\ \mathit{sc} : \langle \left[\begin{array}{l} \text{NP} \\ \mathit{sem} : \text{B} \end{array}\right] \rangle \\ \mathit{sem} : \left[\begin{array}{l} \text{V\_FORM} \\ \mathit{var} : \text{C} \\ \mathit{form} : \left[\begin{array}{l} \text{S\_FORM} \\ \mathit{scope} : \text{D} \\ \mathit{form} : \left[\begin{array}{l} \text{P\_FORM} \\ \mathit{pred} : \text{and} \\ \mathit{arglist} : \langle \text{E}, \left[\begin{array}{l} \text{P\_FORM} \\ \mathit{pred} : \text{Prep} \\ \mathit{arglist} : \langle \text{C}, \text{B} \rangle \end{array}\right] \rangle \end{array}\right] \end{array}\right] \end{array}\right] \\ \mathit{mod} : \langle \left[\begin{array}{l} \text{V\_FORM} \\ \mathit{var} : \text{C} \\ \mathit{form} : \left[\begin{array}{l} \text{S\_FORM} \\ \mathit{scope} : \text{D} \\ \mathit{form} : \text{E} \end{array}\right] \end{array}\right] \rangle \end{array}\right]$$

Figure 4.10: The preposition semantics template: prep_sem/2

Domain specific constructs

Although we have tried to keep the grammar as general as possible, we also had to take care of two other demands: the grammar has to cover this particular domain (which is rather special because of the importance and frequency of temporal and locative expressions) and the system must be able to work real-time (and thus, efficiency may have higher priority then generality sometimes). A few constructions are therefore interpreted in a domain-specific way.

Temporal NP's as modifiers

In section 3.4 we have motivated the need for a separate grammar for temporal expressions. It seems to be the case that not only the temporal expressions have their own internal structure, but also that temporal NP's can act as verbal modifiers. For example the sentences in (4.5) are acceptable in Dutch:

(4.5) *a.* `Ik wil` *`morgen (om drie uur)`* `vertrekken`
b. `Ik wil` *`drie januari`* `vertrekken`
c. `Ik wil er uiterlijk` *`drie uur`* `zijn`

As NP's normally do not have a modifier semantics, we have added a rule that transforms temporal NP's in modifiers (see figure 4.11 for the semantic part of rule *mod3*).

The structure that is modified is specified in the MOD feature. The semantic content of the modifier is constructed as if it was a PP with P_FORM *om* (at). The semantic content of the (temporal) NP daughter is plugged into the second position of the argument list of the preposition.

Translation of prepositions

Some of the (locative) prepositions in the current grammar are translated in a domain-specific predicate. The predicate *van* for example is translated in *origin*, which is indeed the intended interpretation in most cases. It is wrong in sentences like *De trein van drie uur* (The 3 o'clock

```
np_to_modifier_sem(S,Day) :-
        var_scope_form_sign(S,Var,_Scope,Form),
        pred_form(Form,and,[First,Second]),
        S:mod <=> [VerbSem],
        mod_var_scope_form_sign(VerbSem,Var,_Scope,First),
        pred_form(Second,at,[Var,Day]).
```

$$\begin{bmatrix} \text{MODIFIER} \\ \mathit{sem} : \begin{bmatrix} \text{V\_FORM} \\ \mathit{var} : \text{A} \\ \mathit{form} : \begin{bmatrix} \text{S\_FORM} \\ \mathit{scope} : \text{B} \\ \mathit{form} : \begin{bmatrix} \text{P\_FORM} \\ \mathit{pred} : \text{and} \\ \mathit{arglist} : \langle \text{C}, \begin{bmatrix} \text{P\_FORM} \\ \mathit{pred} : \text{at} \\ \mathit{arglist} : \langle \text{A}, \text{D} \rangle \end{bmatrix} \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathit{mod} : \langle \begin{bmatrix} \text{V\_FORM} \\ \mathit{var} : \text{A} \\ \mathit{form} : \begin{bmatrix} \text{S\_FORM} \\ \mathit{scope} : \text{B} \\ \mathit{form} : \text{C} \end{bmatrix} \end{bmatrix} \rangle \end{bmatrix} \longrightarrow \begin{bmatrix} \text{NP} \\ \mathit{nform} : \text{TEMP} \\ \mathit{sem} : \text{D} \end{bmatrix}$$

Figure 4.11: The template np_to_modifier_sem/2 and its application in rule `mod` → `np` (*mod3*)

train). More fine-grained selection restrictions are necessary here to handle this better.

Modals with locative PP's

Some expressions occur so frequently in this domain, that they deserve a special treatment. This gives us the opportunity to gain some efficiency (e.g. avoid ambiguities). In this particular domain, sentences like *Ik wil naar Zwolle* (I want to Zwolle) are quite common. These constructions are translated in a 3-adic predicate (e.g. *willen_naar/3*) in which the third argument position is filled with the semantic content of the locative NP.

Speech-acts and the `top_a` category

The fact that the dialogue in the OVIS system is guided by the computer (i.e. the system asks questions, the user can just answer these), makes that many utterances will be just a denial, a confirmation, a greeting or a combination of these. Most of these expressions are always used in the same way. Therefore, it seems to be the most efficient way to analyse these just by listing them. This is done in the current grammar by adding an extra category `top_a`. The semantic content of elements of this category is just the *speech-act* denoted by these expressions (i.e. `deny`, `confirm` and `greeting` respectively).
It also often happens that an utterance consists of two elements, either both of `top_a` category, or one of `top_a` category and the second one a 'real' sentence (i.e. a sentence that is acceptable according to the grammar). This might change the speech-act. The complete utterance (4.6) for

instance, is a *correction* on what is said by the system.

(4.6) *Nee,* *om drie uur*
(No, at three o'clock)
`deny` `dcl`

At this moment not much is done about recovering the right speech-act of an utterance. What is implemented until now is rather ad-hoc. All sentences that are accepted by the grammar receive the speech-act `dcl` (declarative). No attempt is made yet to recover speech-acts for questions. The speech-act of a combined utterance is retrieved by the predicate `speech_act_matrix/3`, defined as:

```
speech_act_matrix(deny,dcl,corr).
```

The QLF gets its final form by the addition of the speech-act information. The QLF that is passed to the module that produces the *update* is a `p_form` whose predicate is the speech-act and whose argument-list consists of one element, the semantic content of the expression that is accepted by the grammar (i.e. is not defined in case of just an expression of `top_a` category). This is illustrated in figure 4.12.

User: *Nee*

$$\text{QLF}: \begin{bmatrix} \text{START} & \\ \textit{sem}: & \begin{bmatrix} \text{P\_FORM} \\ \textit{pred}: \text{deny} \\ \textit{arglist}: \langle \_ \rangle \end{bmatrix} \end{bmatrix}$$

User: *Nee, om drie uur*

QLF :

$$\begin{bmatrix} \text{START} & \\ \textit{sem}: & \begin{bmatrix} \text{P\_FORM} & \\ \textit{pred}: \text{corr} & \\ \textit{arglist}: \langle & \begin{bmatrix} \text{V\_FORM} & \\ \textit{var}: \text{A} & \\ \textit{form}: & \begin{bmatrix} \text{S\_FORM} & \\ \textit{form}: & \begin{bmatrix} \text{P\_FORM} \\ \textit{pred}: \text{and} \\ \textit{arglist}: \langle \_, \begin{bmatrix} \text{P\_FORM} \\ \textit{pred}: \text{at} \\ \textit{arglist}: \langle \text{A}, \text{hour}(3) \rangle \end{bmatrix} \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix} \rangle \end{bmatrix} \end{bmatrix}$$

Figure 4.12: Integrating speech-act information

4.2 Constructing Updates from QLF's

The pragmatic interpretation module and dialogue manager (DM) expects the linguistic module to produce so called *updates* representing the domain- and context-specific meaning of the user utterance. The DM keeps track of the information provided by the user by maintaining an *information state* or *form*. This form is a hierarchical structure, in which there are slots for the origin and destination of a connection, for the time at which the user wants to arrive or leave, etc. Updates specify the values of these slots. For example, the QLF in Figure 4.1 is translated into the following update:

(4.7) `userwants.travel.origin.time.at.clock-hour.[=4]`

The first part of the update (`userwants...clock-hour`) is a path specifying the slot that is filled, the second part (`4`) is the value.

As the user responds to the system's questions, responses will usually not be complete sentences, but NP's or PP's, or sequences of such constituents. In such cases, the QLF produced by the grammar will be severely underspecified. An update can be constructed, however, by adding the information from the question to the semantic representation of the answer. The response in (4.8b), for instance, can be translated into the update in (4.8c) by taking into account the fact that this was a response to the question in (4.8a). This mechanims has not been implemented yet.

(4.8) *a.* `system:  Naar welk station wilt u vanuit Groningen reizen?`
To which station do you want to travel from Groningen
b. `user:  Amsterdam`
Amsterdam
c. `userwants.travel.destination.place[=Amsterdam]`

4.2.1 The Update Module

The current implementation of the update module is rather ad-hoc. The updates it produces are not exactly in the right form and the computation is not yet as efficient as it could be. It this section we will explain how the translation is set up at this moment and indicate how we think that we can improve it.

The QLF's are produced by the grammar module. The translation of the QLF's in updates is done in an separate module. This means that this module has to use its own type-system and it has to redefine the templates that we used in the grammar module to build up the QLF's. We can however just reuse the type-system of the grammar (it contains everything we need here). An extra file is added in which the templates are defined. This file is compiled using the type-system of the grammar.

Format of Updates

An update consists of three types of information (as defined in (Veldhuijzen van Zanten, 1995)): an element denoting the communicative function of the utterance, a path specifying the slot that is filled in the information state and the value for this slot.

The communicative function is either *assert*, *deny*, *confirm* or *correct* and is represented by an operator that receives the values `=`, `#`, `+` and `!`, respectively. The value is calculated by considering the speech act that is found during grammatical analysis possibly combined with information from inside the QLF (i.e. at this moment just negation. *Ik wil niet naar Haren* (I do not want to go to Haren) is considered to be a *declarative* utterance by the grammar module, but we want to interpret it as a denial.

The module should produce updates like (4.9). To assert the information that is associated with an utterance expressing that the user wants to leave at half past four.

(4.9)
```
userwants.travel.origin.moment.at.time.[ clock-hour=4;
                                         minute=30]
```

This means that the path `userwants.travel.origin.moment.at.time.` denotes the context in which the two updates `clock-hour=4;` and `minute=30` have to be interpreted. At this moment these two updates are represented by a set of three independent updates (4.10).

(4.10)
```
userwants.travel.origin=[];
moment.at.time.clock-hour=4;
moment.at.time.minute=30
```

Construction of Updates

The QLF that is produced by the grammar module contains underspecified information. We have mentioned before that we think that (in this particular application) we can retrieve the information we need from the QLF's without resolving the underspecification. The first thing we do before computing the update is replacing the terms describing the *proper names* and *pronouns* by constants (i.e. values corresponding to their lexical entries). This is done by replacing the entire term by just the value of the second argument of the predicate `name/2` (i.e. the value of its own *Restriction* feature). We assume that the utterances in this context are not ambiguous with respect to the scope of the event quantifiers, these are also removed. These three steps are performed by walking recursively through the QLF, performing one of the above actions when necessary until an atomic QLF is found.

After the simplification step, the speech act as found by the grammar module is retracted. Remember that the final QLF is a `pred_form` whose *predicate* is the speech act and whose sole argument is the QLF that was built during grammatical analysis. The next step is walking recursively through the QLF (using the templates). When an atomic QLF is found it is either ignored (in case of some specific predicates like the modal verbs *moeten* (must), *kunnen* (can) and *zouden* (should)), speech act information is retracted (in case of a `pred_form` with the negation operator (*not*) as predicate) or (in most cases) it is translated into its equivalent in the update language. Rules are defined for all predicates that have equivalents in the update language. This is illustrated by the next rule.

```
qlf_to_update_rule(Pred,[],Update):-
        pred_form(Pred,arriveren,[_Event,_Sub]),
        update(Update,[travel,destination],[]).
```

The updates that result from this recursive translation procedure look as follows: (for the sentence *Ik wil om half vier vertrekken* (I want to leave at half past three):

```
[userwants=[travel.origin]]=[];
[moment.at]=[[time.clock-hour=4],[time.minute=30]]
```

These structures are flattened. This operation results in structures like 4.10.

Chapter 5

Workplan Grammar development 1996

In the original workplan it was stated that in 1995 we should be able to determine on the basis of a detailed study of a corpus of human-machine dialogues which syntactic constructs should be implemented in which order. Due to the delay in which the corpus material became available such a study has not yet been carried out in enough detail to build such a workplan on.

Alternatively we have both looked at examples of human-human dialogues and examples of German human-machine dialogues. These examples have been used as a primary source of inspiration in constructing the core grammar discussed in the previous chapters. Given the attention we paid to both temporal and locative expressions it can be expected that this first version of the grammar is an excellent starting point for further development now that a real corpus is becoming available.

For this reason we now foresee the following packages of work in the remaining months of 1996. Remember that in october 1996 a frozen version of the grammar should be available.

1. Detailed testing / extending of the grammar on the basis of the corpus material that is now becoming available. A number of smaller issues can already be identified where the grammar should be extended:

   - The distinction between *nee* and *niet* in sentences such as:

     (5.1) a. Nee naar Groningen
     b. Niet naar Groningen

   - Temporal expressions involving *vroeg* and *laat* such as:

     (5.2) a. Vroeg in de morgen
     b. Dinsdagochtend vroeg
     c. Morgen wat later in de middag

   - Locative expressions involving *in* as in:

     (5.3) a. Ik moet om half tien in Assen zijn
     b. Ik moet om half tien in Assen aankomen
     c. Ik ben in Assen en ik moet naar Groningen

   - Temporal expressions involving *('t) heel, ('t) half* as in:

     (5.4) a. Zes over ('t) heel
     b. Drie over ('t) half

   It is expected that a careful study of the corpus will give rise to more issues of a similar nature.

2. Spoken language constructions. The current grammar of spoken language constructions (discussed in section 3.4.2) is of a very preliminary nature. It is expected that this module needs a thorough revision.

3. Construction of updates on the basis of QLFs. Currently the module that implements this translation is not up to date with the latest version of the update language.

4. Disambiguation on the basis of the most recent question. Until now we have not implemented this; but we expect that – based on some preliminary experiments – much can be gained from this.

5. Disambiguation on the basis of sortal restrictions. Currently most prepositions are allowed to combine with most noun-phrases; this can clearly be improved in order to rule out combinations like 'om dinsdag'.

6. Evaluation of the use of the QLF formalism as an intermediate step in the translation to updates.

Part III

The OVIS Parsers

Chapter 6

The Parser Generator

In this chapter, the parser generator is discussed that is used to generate the parser for the spoken input to the OVIS system. The input is a grammar describing the language of utterances of the user, together with a mapping to a semantic representation. The output is a collection of tables, that together with a driver constitutes a parser for utterances of the user.

6.1 Introduction

Real-time parsing of spoken language is a formidable task. Two major obstacles have to be overcome:

- Some description has to be written of the language that is accepted, together with a mapping to some semantic value, representing the "meaning" of sentences in that language. This description is called the *grammar*. Writing a grammar for spoken-language information systems is non-trivial, since on the one hand huge numbers of sentences need to be covered, whereas on the other hand, sentences erroneously introduced in the word graphs (Section 7) should *not* be covered.
- The grammar needs to be implemented in a *parser*, that is, in a computer program that performs the recognition of sentences as described by the grammar together with the mapping to the semantic values. By *parser generation* we mean the (automatic) construction of parsers from grammars. For real-time purposes the parsers need to be fast.

The grammar for the OVIS system is described in the previous chapters. It is written in a feature formalism. This formalism has been and still is under development, mostly due to changing views of the grammar writers. To allow work on the parser generator to proceed while the grammatical formalism is still not fixed, we have chosen to take a fixed intermediate formalism that feature formalisms can usually be translated to. This is the formalism of definite-clause grammars (DCGs) (Pereira and Warren, 1980). Grammars in this formalism will be the input for our parser generator.

In general, grammars are written such that they are easy to read, verify, and edit. They are not written with a particular parsing algorithm in mind, and parsers generated straightforwardly from the grammar may often be very inefficient. Our parser generator, `ovispg`, performs complicated analyses of the grammar in order to obtain the data necessary for generating fast parsers that have moderate memory requirements.

The structure of this document is as follows. In Section 6.2 we discuss the functionality of the system. If you are only interested in obtaining parsers, but do not care how they are constructed, that section may be the only one in this chapter that will interest you. What the tables for the parsers look like is explained in Section 6.3. That section is of interest if you intend to change the driver of the parser.

In Section 6.4 we discuss analysis of the grammar, and the actual generation of the parser is described in Section 6.5. Some auxiliary functions are discussed in Section 6.6. Chapter 7 defines one of the parsing algorithms currently under investigation. The appendices give an overview of the files relevant to the parser generator and discuss some implementational details.

6.2 The functionality

The parser generator `ovispg` strictly speaking does not generate a complete parser, but a number of tables that together with a driver constitutes a parser. A number of options are possible, which affect the resulting tables, or which cause auxiliary functions to be activated.

The most elementary function is activated by:

`ovispg` *filename*

where *filename*`.g` is the name of a file containing a grammar, of the form described in Chapter 2. This command creates a file *filename*`.pl` containing tables for the parser, and a file *filename*`.log` containing information derived during analysis of the grammar. Furthermore, if problems occur during the generation process, they are directly written to standard output.

The `ovispg` command can be called with a number of options. Options can be activated or deactivated by means of flags. Flags are indicated by single letters, preceded by a minus symbol or a plus symbol, e.g. `-h` and `+h`. Some flags are followed by an argument, e.g. `-g 5`. Two or more flags without arguments may be grouped together after one minus symbol or one plus symbol, e.g. `-lx` and `+lx`, or they may be given separately, e.g. `-l -x +h`. The flags may precede or follow the *filename*, so

`ovispg -lx -g 5` *filename*
`ovispg -l -x` *filename* `-g 5`
`ovispg -l` *filename* `-x -g 5`

are equivalent.

Usually, a flag beginning with + activates an option, a flag beginning with − deactivates an option. Some options are activated by default, that is, they hold unless conflicting flags are given explicitly.

The flags may be given in any order, and if the flags have independent effects on the options, then the order is insignificant. If however two flags have conflicting effects, then the textually last flag on the command line has precedence over earlier ones. In particular, when two opposing flags occur, e.g. `-h` and `+h`, then the first will be ignored.

For the following options, flags beginning with + or − may be given:

`+/-f` (non-default) The computation of semantic filters is not done automatically, but based on the `sem_filter/2` declarations (see Section 6.4.2).

`+/-h` (default) Heavy analysis is performed that may result in fast parsers, but that takes a long time. (See Sections 6.4.2 and 6.4.3.)

`+/-l` (default) A log file *filename*`.log` is created that records results obtained during analysis of the grammar.

`+/-x` (non-default) Analysis of the lexicon is very expensive. In the case the lexicon has been analysed already by a previous call to `ovispg`, then this option avoids reanalysis, making use of results stored in file *filename*`.xlog` by a previous incarnation of `ovispg`, speeding up the process. Note: the system does not check automatically whether the lexicon has indeed remained unchanged; it does verify however that *filename*`.xlog` is non-empty.

`+/-t` (non-default) Categories in the grammar may be changed, in order to speed up the parser (see Section 6.4.3).

In the following cases, only flags beginning with − are allowed. Here the minus symbol indicates activation rather than deactivation. None of these options hold by default.

`-p` The grammar in *filename*`.g` is overwritten with its pretty-printed form. Note that variables may be renamed. All other functions of `ovispg` are disabled.

`-v` Internal transformations take place on the grammar, just as if parse tables would be constructed, but no actual parse tables are written to file. Instead, the internally transformed grammar is written to *filename*`.trans`. This function allows monitoring of complicated processes within `ovispg`. All other functions of `ovispg` are disabled.

`-g` *number* A number (*number*) of sentences according to the grammar is computed using a random generator. Motivated by the theoretically undecidable nature of this task, this function aborts if the attempt to generate repeatedly fails. The sentences are written to file *filename*`.ex`. All other functions of `ovispg` are disabled. This option automatically invokes `+h`. See also Section 6.5.

`-o` *number* Static analysis is performed in order to find out whether the occur check may be violated in a bottom-up parser, in other words whether some features may be computed that are their own subfeatures. Only derivations up to some depth *number* are considered. The result of the analysis is written to *filename*`.log` and to the standard output. All other functions of `ovispg` are disabled. See also Section 6.6.1.

Furthermore, typing `ovispg` without any arguments or options causes the above information to be summarized and written to standard output.

6.3 The parse tables

The tables that `ovispg` generates satisfy the following syntax:

```
#tables : #declaration-sequence

#declaration : #top_cat_declaration |
               #rule_declaration |
               #lex_declaration |
               #first_member_index |
               #nonfirst_member_index

#top_cat_declaration : top_category( #id, #category ).

#rule_declaration : grammar_rule( #id, #lhs1, #lhs2, #rhs ).

#lex_declaration : lex( #id, #lhs1, #lhs2, #word, #word-list ).

#first_member_index : first_member_index( #id, #id-list ).

#nonfirst_member_index : nonfirst_member_index( #id, #mem_id-list ).

#top_cat_index : top_cat_index( #id, #id-list ).

#mem_id : ( #id, #more_than_one )

#more_than_one : 2 | 3 | 4 | ...
```

Many of the declarations in the tables are similar to the declarations in the grammar. Top categories have been assigned an identifier. The id's have been made unique throughout the top-category declarations, rule declarations and lexical declarations.

The `#lhs1` are copies of the left-hand side in which some subcategories have been truncated. Such left-hand sides can be used in a first pass in the parser that constructs incomplete parses. A second pass using the complete left-hand sides (`#lhs2`) is then needed to construct the full parses. (See also Section 6.4.2.)

The `#word` and the `#word-list` together represent the list of words of a lexical entry. This avoids having two possible syntaxes for lexical entries, as was the case for grammars (Section 2).

The first-member index declarations encode for each rule with id `#id`, which other rules (with id in `#id-list`) are such that the left-hand side of `#id` unifies with the first member of the right-hand side. The non-first-member index declarations encode a similar fact, but for the other members, viz. the i-th member, $i > 1$, in some right-hand side of a rule. The top-cat index declarations encode which top categories the left-hand side of some rule with id `#id` unifies with.

6.4 The grammar analyses

Unless one of the flags `-g`, `-o` or `-p` is given, the parser generator will go through the following phases:

1. A basic well-formedness check is performed on the grammar. Rules, lexical entries and top-category declarations are given unique identifiers.
2. Epsilon rules are eliminated from the grammar (see Section 6.4.1), as well as unreachable parts of the grammar.
3. The direction of flow of features is computed by means of abstract analysis (see Section 6.4.2).
4. A type system of features is computed (see Section 6.4.3).
5. Categories are made more specific based on the type system (see Section 6.4.3).
6. If the previous step has changed the grammar, the direction of flow of features is recomputed.
7. Semantic filters are computed, either based on semantic-filter declarations or on the computation of direction of flow, depending on the option `+/-f` (see Section 6.4.2).

6.4.1 Elimination of epsilon rules

An *epsilon rule* is a rule of which the right-hand side is the empty list. In most cases, grammar writers can avoid the use of epsilon rules, but often the grammar can be made to look more elegant when some epsilon rules are used. Some parsing algorithms are undefined for grammars with epsilon rules, and some are more efficient if epsilon rules need not be considered. Therefore, we have chosen to eliminate epsilon rules as a first step towards construction of parse tables.

This process takes place in three phases:

1. We compute all features that may occur at the root of subparses where at the leaves only epsilon rules are applied, i.e. subparses that parse the empty string. Let us call such features *nullable features*. For some grammars this process may not terminate; there may be infinitely many of such features. This concern is left to the grammar writer.
2. For each rule and each list of nullable features that unify with some members in the right-hand side, a copy of that rule is added to the grammar in which those nullable features have been substituted.

3. Rules that cannot be used in the construction of any subparse where only lexical entries (not epsilon rules) occur at the leaves, is removed from the grammar. In order to ensure termination of this step, not all possible subparses are actually considered. Instead, we simply regard a rule as "useful", if all members in the right-hand side unify with left-hand sides of other useful rules or lexical entries.

After performing these steps, the grammar will describe the same language as before, but all epsilon rules will have been removed.

The results of the second and third steps are reported in the log file. The second step also computes *patterns*, which are lists over the signs − and +, as many as there are members in the original grammar rule. The sign − indicates that the corresponding member has been substituted with a nullable feature. For example, if we have `grammar_rule(i,a,[b,c,d,e])` and `b` and `d` are nullable features, as computed by the first step above, then one may have a transformed rule `grammar_rule(i,a,[c,e])`, which has pattern `[-,+,-,+]`. The pattern is encorporated into the identifier of the new rule.

Together with elimination of epsilon rules, also rules and lexical entries are eliminated that are unreachable from the top categories. We call a rule or lexical entry *reachable* if the left-hand side is unifiable with

1. a top category, or with
2. a member in a reachable rule.

Note that this definition is inductive.

6.4.2 Direction of flow

Consider the following grammar.

```
         lex( i, a(f(X,Y)), [w]                ).
grammar_rule( j, a(g(Z)),   []                 ).
grammar_rule( k, b(h(W)),   [a(W),a(g(W))]     ).
top_category(               b(h(f(x,y)))       ).
```

In a parse constructed using this grammar, one could say that there is flow of the subfeature `f(X,Y)` in `a(f(X,Y))` in lexical entry `i` to the first member in the right-hand side of rule `k`, since the category `a(f(X,Y))` is more specific than the category `a(W)`. Subsequently, one could say that if `W` is instantiated with `f(X,Y)` in the right-hand side, then also flow occurs upwards at variable `W` in the left-hand side `b(h(W))`.

Such flow information can be computed for the entire grammar. This information tells us which features may end up in the semantic value, and which features are not needed for finding correct parses and whose computation may therefore be delayed.

The exact determination of directions of flow is undecidable, but by making some simplifications we may still obtain a useful approximation. Our simplification consists in assuming that flow takes place only with ground features.

We will now make these ideas more explicit. We introduce the notion of *ctype* (category type). If a category is seen as a tree (Figure 6.1), then a ctype can be identified with a path from the root to some node, labelled with an alternating sequence of functors and argument numbers, starting with a functor and ending with an argument number. Two such paths in two different categories are identified with the same ctype if the list of functors and argument numbers along that path is the same. Figure 6.1 gives an example.

The representation that we will use is that ctypes are partial functions from the set of functors to the set of lists of ctypes. Formally, if the set of ctypes is denoted C and the set of functors F, then for each $c \in C$ we have $c : F \mapsto C^*$. We have a "super ctype" c_0 which represents the set of all categories. As a function it is defined on all main functors of categories. Such a functor is mapped

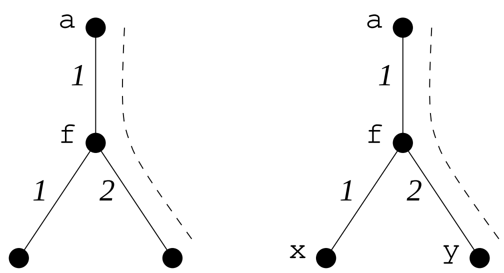


Figure 6.1: Tree representation for terms `a(f(X,Y))` and `a(f(x,y))`. The dashed lines indicate an identical path `a:1:f:2` in both terms, which corresponds with a single ctype.

to a list of ctypes, each of which represents the set of all subcategories in the first argument, the second argument, and so on. [1]

In the running example, we need 9 ctypes: $c_0, \ldots, c_8$, where c_0 is the super ctype. These functions are defined for a restricted collection of functors, as indicated by the following.

$$
\begin{array}{lcl}
c_0(\mathtt{a}) & = & (c_1) \\
c_0(\mathtt{b}) & = & (c_2) \\
c_1(\mathtt{f}) & = & (c_3, c_4) \\
c_1(\mathtt{g}) & = & (c_5) \\
c_2(\mathtt{h}) & = & (c_6) \\
c_6(\mathtt{f}) & = & (c_7, c_8) \\
c_7(\mathtt{x}) & = & () \\
c_8(\mathtt{y}) & = & ()
\end{array}
$$

The ctype c_4 is the one considered in Figure 6.1.

The system of ctypes represents in an abstract way the set if all categories. The next step is to assign sources of flow at all ctypes. A source is one of **above** or **below**. When a subcategory which is not a variable occurs in a left-hand side of a grammar rule, we assign **below** to the corresponding ctype, when this subcategory occurs in the right-hand side, we assign **above** to the corresponding ctype. Similar facts hold for lexical entries and top categories.

In the running example, we derive from the lexical entry with `a(f(X,Y))` in the left-hand side of `i` that **below** can be assigned to c_0 and c_1 since at the corresponding locations in the category, a non-variable subcategory occurs. From the right-hand side of rule `k` we derive that **above** can be assigned to c_0 and c_5. From the top-category declaration, we derive that **above** can be assigned to c_0, c_2, c_6 c_7 and c_8. The other assignments are left to the reader.

After this initial assignment, we take into account *consistent substitution*. Consistent substitution means that multiple occurrences of a variable should have the same value for each instance in a rule. It is the basis for all unification formalisms. Consistent substitution causes flow of feature values from one part of a rule to another.

In the running example, the feature `f(X,Y)` coming from lexical entry `i` comes "from below" and may be passed on through variable `W` in rule `k` to the left-hand side, so that a value comes "from below" there as argument for the functor `h`.

In general, we define a predicate *implies* $\subseteq C \times S \times C \times S$, where $S = \{\mathbf{below}, \mathbf{above}\}$. We have $(c, s, c', s') \in$ *implies* if flow in direction $s \in S$ at ctype c implies flow in direction $s' \in S$ at

[1] For reasons of presentation we ignore the possibility that two functors have the same name but distinct arities.

ctype c'. In the running example, we would have

$$\begin{array}{rcllll} \textit{implies} & = & \{ & (c_1, & \textbf{below}, & c_6, & \textbf{below}), \\ & & & (c_6, & \textbf{above}, & c_1, & \textbf{above}), \\ & & & (c_5, & \textbf{below}, & c_6, & \textbf{below}), \\ & & & (c_6, & \textbf{above}, & c_5, & \textbf{above}), \\ & & & (c_1, & \textbf{below}, & c_5, & \textbf{above}), \\ & & & (c_5, & \textbf{above}, & c_1, & \textbf{below})\} \end{array}$$

The predicate *implies* together with the initial assignments of sources of flow to the ctypes gives rise to more assignments of sources of flow. This process can be summarized as: if we have assigned $s \in S$ to $c \in C$ and if $(c, s, c', s') \in$ *implies*, then assign s' to all sub-ctypes of c', and perform this operation exhaustively. We define the notion of *sub-ctype* inductively as:

- Each ctype is a sub-ctype of itself.
- For all ctypes c and c' and functors f such that $c(f) = (c_1, \ldots, c_k)$ and $c' = c_j$, $1 \leq j \leq k$, each sub-ctypes of c' is a sub-ctype of c.

In the running example, we combine $(c_1, \textbf{below}, c_6, \textbf{below}) \in$ *implies* with the assignment of **below** to c_1 to conclude that **below** should be assigned to c_6 as well as to its sub-ctypes, i.e. to c_7 and c_8.

After this computation, we can derive useful facts about the flow of feature values. For example, if both **below** and **above** have been assigned to a ctype, then we know that two subfeatures may be unified, one coming from "below" and one coming from "above". If only one of **below** and **above** have been assigned to a ctype, then flow of feature values is strictly in one direction, and the corresponding subcategories may be safely replaced by a free variable, without consequences for the computation of correct parses; this may constitute a first pass for the parser, which may be less costly with this simplification of the grammar. However, for the purpose of computing the semantic values for correct parses (the second pass), the original categories need to be used, since the subcategories that were replaced by free variables in the first pass may play a role in the semantic values.

In order to establish which subcategories may play a role in the semantic values, we introduce two imaginary sources of flow: **out-below** and **out-above**. They represent imaginary flow that would originate from the top categories if the variables in these were further instantiated. The assignments of **out-below** and **out-above** are distributed among various ctypes just as **below** and **above**, originating in the assignments of **out-above** to all sub-ctypes of ctypes corresponding with variables in the top categories.

If one of **out-below** or **out-above** is assigned to a ctype then this means that the feature values at that ctype may end up in the semantic value of a derivation.

One application of **out-below** and **out-above** is this: The generated parser may operate more efficiently if it chooses a different representation for features. One prerequisite for such a transformation is that the recognized language should of course be preserved. A second condition is that the semantic values should not be affected. For verifying the latter condition we need to know which features may end up in semantic values.

In the log file, the parser generator may give the following diagnostics for ctypes:

`USELESS:up` There is only flow upwards, that is not used in any way.

`USELESS:down` Analogously.

`UP` There is only flow upwards, that is used exclusively for the computation of the semantic value.

`DOWN` Analogously.

`VARIABLE` There is no flow of feature values, yet the corresponding subcategories may have the function of introducing two occurrences of the same variable in the semantic value.

`DON'T CARE` There is no flow of feature values, yet the corresponding subcategories may have the function of introducing a single occurrence of a variable (a *don't-care*) in the semantic value.

`USELESS` There is no flow of feature values, and no relation to the semantic value can be established.

In order to limit the amount of information in the log file, only those ctypes c are given for which:

- for some sub-ctype of c one of the above diagnostics can be given, and
- c is not a proper-sub-ctype of a ctype where diagnostics can be given. (A *proper-sub-ctype* is a distinct sub-ctype.)

Where flow is in only one direction, or in no direction at all, subcategories may be truncated, in other words, replaced by a free variable. For left-hand sides, this gives rise to two versions, one with the truncated subcategories and one as it occurs in the grammar (c.f. `lhs1` and `lhs2` in Section 6.3).

An alternative to the separation into two passes based on the above analysis is the explicit marking by the grammar writer of ctypes where computation may be delayed. This is done by means of the semantic-filter declarations: based on declarations of the form `sem_filter(#variable-list, #feature)` all ctypes corresponding with occurrences of variables from the variable list in the feature are marked to indicate that they may be delayed until the second pass.

The computation of directions of flow is only performed when the option `+h` is in effect. The results may be subsequently used for the determination of semantic filters, unless `+f` option is in effect, in which case the `sem_filter` declarations from the grammar are used. The `sem_filter` declarations are also used when the flag `-h` has been given.

6.4.3 A type system of feature values

The ctype system explained in the previous section is a good starting point for computation of the system of feature values, i.e. the system of categories and subcategories as they may be instantiated during the parsing process. The main difference between this type system and the ctype system is that consistent substitution of two variables in a rule requires the corresponding types to be the same.

Formally, the set of all types is denoted T, and each type t is a partial function $t : F \mapsto T^*$. Initially, we have one type t for each ctype c, and for each functor f we have $t(f) = (t_1, \ldots, t_k)$ if and only if $c(f) = (c_1, \ldots, c_k)$, where $t_1, \ldots, t_k$ are the types corresponding with ctypes $c_1, \ldots, c_k$. One may define the complete function *type* : $T \mapsto C$. Initially, this function is bijective, later when types are unified it will remain surjective but not injective.

Types are unified due to consistent substitution. For this we use the relation *implies*, which we have explained in Section 6.4.2. We define *bind* $= \{(c, c') \mid (c, s, c', s') \in$ *implies*, some $s, s' \in S\}$. The relation *bind* represents the pairs of ctypes that need to be unified into single types.

We perform *unify*(t, t') for each pair of types t and t' such that $\exists c, c' \in C[(c, c') \in$ *bind*$\wedge$*type*$(c) = t \wedge$ *type*$(c') = t']$.

The computation *unify*(t, t') to unify two types is defined as follows. If t and t' are already the same type, then we do nothing. Otherwise, all references to t' are reset to t, in particular the references in the images of *type* and the images of the types themselves.[2] The type t is then redefined, for all functors f, as below; $t^\dagger$ denotes the type t as it was at the beginning of this procedure.

- $t(f)$ is $t^\dagger(f)$ if $t^\dagger(f)$ is defined.

[2]In the actual implementation, we create an *indirection link* from t' to t. After completion of the type system, the indirection links are removed, in order to allow faster access to the type system. Further note that t and t' may switch roles without consequences for the correctness of the unification algorithm.

- $t(f)$ is $t'(f)$ if $t^\dagger(f)$ is undefined but $t'(f)$ is defined.
- $t(f)$ is undefined if $t^\dagger(f)$ and $t'(f)$ are both undefined.

After changing type t as above, we do for each functor f such that $t^\dagger(f)$ and $t'(f)$ are both defined:

1. Say $t^\dagger(f) = (t_1, \ldots, t_k)$ and $t'(f) = (t'_1, \ldots, t'_k)$.
2. Call *unify*(t_j, t'_j) recursively, for all j such that $1 \le j \le k$.

Note that in general, the derived type system is recursive. This is why the two-step unification is needed as described above. First we need to incorporate t' into t, only then can we call *unify* for the subtypes; if we called *unify* for the subtypes first, and only then defined t based on $t^\dagger$ and t', then we would destroy unifications of t and t' that resulted from recursive calls.

Consider the ctype system in the example from Section 6.4.2. Initially, we have one type for each of the 9 ctypes: *type*$(c_0) = t_0, \ldots,$ *type*$(c_8) = t_8$. From *implies* we conclude that t_1 and t_6 should be unified, say the resulting type is called $t_{1,6}$, and then $t_{1,6}$ should be unified with t_5, say the resulting type is called $t_{1,5,6}$. Note that unification of t_1 and t_6 leads recursively to a unification of t_3 and t_7 and of t_4 and t_8. We obtain:

$$
\begin{array}{rcl}
t_0(\texttt{a}) & = & (t_{1,5,6}) \\
t_0(\texttt{b}) & = & (t_2) \\
t_{1,5,6}(\texttt{f}) & = & (t_{3,7}, t_{4,8}) \\
t_{1,5,6}(\texttt{g}) & = & (t_{1,5,6}) \\
t_2(\texttt{h}) & = & (t_{1,5,6}) \\
t_{3,7}(\texttt{x}) & = & () \\
t_{4,8}(\texttt{y}) & = & ()
\end{array}
$$

Many useful properties can be derived from the type system. We first define the notion of *proper-subtype* inductively as:

- For all types t and t' and functors f such that $t(f) = (t_1, \ldots, t_k)$ and $t' = t_j$, $1 \le j \le k$, t' is a proper-subtype of t and each proper-subtype of t' is a proper-subtype of t.

We say t' is a *subtype* of t if t' is a proper-subtype of t, or if $t' = t$.

$$
\begin{array}{rcl}
\textit{recursivetype}(t) & \equiv & t \text{ is proper-subtype of } t \\
\textit{infinitetype}(t) & \equiv & \exists t'[t' \text{ is subtype of } t \wedge \textit{recursivetype}(t')] \\
\textit{typesize}(t) & = & \text{IF } \textit{infinitetype}(t) \\
& & \text{THEN } \infty \\
& & \text{ELSE } \displaystyle\sum_{f \in F} \textit{functorsize}(f) \\
& & \text{where } \ \textit{functorsize}(f) = \\
& & \qquad\qquad \text{IF } t(f) \text{ is undefined} \\
& & \qquad\qquad \text{THEN } 0 \\
& & \qquad\qquad \text{ELSEIF } t(f) = (t_1, \ldots, t_k) \\
& & \qquad\qquad \text{THEN } \prod_{j \in \{1,\ldots,k\}} \textit{typesize}(t_j) \\
\textit{singletontype}(t) & \equiv & |\{f \in F \mid t(f) \text{ is defined}\}| = 1
\end{array}
$$

The function *typesize* indicates the number of possible (ground) features in a certain type. For the representation of features by means of bit strings, this is the required length of those strings (Nederhof, 1996).

The determination of singleton types may be useful for simplifying the grammar. For example, if we have a category $a(X)$, and X is of a type t which is a singleton type, and f of arity 3 is the only functor for which $t(f)$ is defined, then we may safely replace $a(X)$ in grammar rules by $a(f(X, Y, Z))$ where X, Y, Z are free variables. The advantage of such a simplification is that the analysis of flow (Section 6.4.2) can be more accurate. This transformation is enabled by option `+t`. Any transformations of this kind are reported in the log file.

The type system is only computed provided the option `+h` is in effect. Note that because the types are computed from the ctypes, and *singletontype* from the types, the transformation enabled by option `+t` may require the ctypes to be computed two times.

The type system is printed in the log file. Usually subtypes are printed with proper indentation. However, if a type is a subtype of other types in more than one way, then a new paragraph in the output is reserved for this type, and it is given a tag of the form <*number*> for the benefit of identification. Types which correspond only with uninstantiated variables are denoted by <>.

6.5 The generation of parsers

At the moment, the parser generator creates a set of tables in the form of Prolog predicates. A driver, which is itself a Prolog program, combined with these tables constitutes the parser. In the future, the module that generates the tables may be replaced by a variant that generates tables interpretable by a C program, and even the tables themselves may be turned into C code, for very fast parsing (cf. recursive-descent and recursive-ascent parsing (Leermakers, 1992), and compilation of Prolog predicates (Aït-Kaci, 1991)).

The tables consist of the predicates that are described in Section 6.3.

6.6 Auxiliary analyses

Apart from the main function of generating a parser, `ovispg` may also perform pretty printing of the grammar (option `-p`, and also `-v`), and two kinds of analysis. The analyses are described below.

Concerning the pretty printer, two warnings should be given:

- The grammar is read, deleted and then rewritten by pretty printing. If the system breaks down during this time or if the user accidentally aborts the process, the grammar, in part or in whole, may be lost.
- Pretty printing involves inserting many blanks in the text. This slows down considerably the reading of the text.

6.6.1 Occur check

The first kind of analysis is activated by the option `-o` *depth*. It investigates whether it is possible that the parser creates a feature that is its own subfeature, a *cyclic feature*. Such features are generally considered incorrect, but avoiding such features by means of a special unification algorithm (using the *occur check*) is expensive. Therefore, it is desirable to be able to verify before parsing that such features may not occur. We have tried to do this in `ovispg` by means of abstract evaluation.

Regrettably, this problem is undecidable, and a useful "sufficient condition" is lacking at the moment. Instead, we have made a function that investigates all possible subparses up to a certain depth (the argument of the option). In these subparses, the root is an arbitrary category, and the leaves are lexical entries. As soon as an instance is found where normal unification of two features in a subparse succeeds but the safe unification extended with the occur check fails, this fact is reported to standard output, and a description of how this may result is written to the log file. This fact then indicates that cyclic features may occur in the parsing process, and the grammar writer would be wise to alter his grammar in order to avoid this problem.

The notion of *reentrancy* is defined to be a multiple occurrence of a single variable in a term. In other words, reentrancy is the phenomenon that two subfeatures of a feature are both uninstantiated, but bound to one another.

Cyclicity in features always originates in *reentrancy*. Because of this fact our analysis has been optimized to remove all subfeatures in the intermediate results that do not include any reentrancy.

6.6.2 Generation of example sentences

Before one can experiment with parsers, test their correctness, measure their run-time properties, etc., one needs a set of sentences to run that parser on. Compiling a list of such sentences manually is an arduous task, and may not reveal all defects of the parser.

Following (Nederhof, 1994) we have incorporated an automatic generator of sentences in `ovispg`, which is activated by the option `-g` *number*. It tries to generate *number* sentences, and writes them to file *filename*`.ex`. Words are separated by blanks and sentences are ended by a period. If the generator fails a certain number of times (say 3 times as many as *number*), then it stops.

In `ovispg`, the generation, which is performed top-down, is preceded by a bottom-up analysis, which computes for each feature that may occur at the root of a subparse, what should be the minimal depth of such subparses, as explained in (Nederhof, 1994). In order to ensure termination of this analysis, these features are truncated at a certain depth. Because of this truncation, which is essentially a simplification of the results yielded by the analysis, the top-down generation is not guaranteed to succeed (other than in the context-free case in (Nederhof, 1994)).

The top-down process is performed using a random generator. This means that "ad random" rules and lexical entries are selected for the construction of parses, guided by the preceding analysis. For technical reasons, we have implemented a particular method to select the lexical entries: after each, say, 15 attempts to generate sentences we arrange the lexicon in a random order. Each time we need a lexical entry, we either take the next entry from the list that fits in the parse under construction, or take one that we had seen before, depending on a random generator.

For rules we always first try to apply one of those that have been applied the least number of times before.

Appendix A. The relevant files

The following files are read or written by a call of `ovispg` *options filename*.

filename`.g` contains the source grammar. See Section 2 for its format. This file is overwritten by a pretty printed form of the grammar in the case of option `-p`.

filename`.pl` contains the parse tables after parser generation, provided none of the options `-g`, `-o`, `-p` or `-v` is in effect.

filename`.trans` contains the grammar as it results from internal transformations. This file is only created provided the option `-v` is in effect.

filename`.log` contains some data collected during analyses of the grammar. This file is only created provided the option `+l` is in effect.

filename`.xlog` contains some intermediate data from the analysis of the lexicon. It is created when `ovispg` is called with option `+h` but *without* option `+x` and it is consulted when `ovispg` is called with option `+h` and with option `+x`.

filename`.ex` contains the random sentences generated after `ovispg` has been called with option `-g`.

Appendix B. The modules in the implementation

The ovispg system is a program in (Sicstus) Prolog consisting of the following modules.

ovispg.pl is the main module. It treats processing of the command line with the options, and calls routines in other modules depending on the options. Also the opening and closing of files is done in this module.

utilities.pl contains library routines.

wellformed.pl checks some well-formedness conditions, and assigns unique identifiers to grammar rules, lexical entries and top-category declarations. It also contains access routines to the grammar.

pretty.pl contains the code to pretty-print grammar rules, Prolog terms, etc.

abstract.pl performs abstract evaluation to find possible cyclic features that may occur during parsing. See further Section 6.6.1.

trans.pl eliminates epsilon rules from the grammar, as well as unreachable rules and lexical entries, as described in Section 6.4.1.

flows.pl computes directions of flow of feature values. It also determines the "semantic filters", based on either the directions of flow, or on the sem_filter declarations from the grammar. It also contains the routines to truncate categories according to the semantic filters. See further Section 6.4.2.

types.pl determines the type system of all possible feature values. Also some properties of types are determined, and also some operations on features which rely on such properties can be found here. See further Section 6.4.3.

generate.pl produces a number of example sentences in the language described by the grammar. Before actual generation, an analysis is performed, the results of which help to avoid some failure of the generation process. See Section 6.6.2 for more details.

cattrans.pl instantiates variables in categories in case the corresponding types are singleton types, as explained in Section 6.4.3, and it replaces categories that (in conflict with the official syntax) are variables by proper categories in all possible ways.

index.pl computes the information necessary to construct the parse tables corresponding to the first-member index, the nonfirst-member index, and the top-cat index (see Section 6.3).

writeparser.pl constructs the parse tables and writes them to a file.

Chapter 7

A bottom-up Earley Parser

In this section we describe one of the parsers that we are experimenting with. Many variants are possible using the parse tables currently produced by the parser generator.

The parsing algorithm is basically a tabular bottom-up algorithm without top-down filtering, using sharing but no packing (Nederhof, 1994). More precisely, we have a table U which is a bag (not set) of items of the form $(i, j, \mathit{id}, A', m, \beta', P, t)$. The elements in such an item are discussed below.

First we describe the structure of the input to the parser: A word graph is a labelled, directed (and usually acyclic) graph which represents all hypotheses of a speech recognizer for a spoken sentence (Oerder and Ney, 1993). Usually there is a unique node representing the beginning of the sentence (the *initial* node) and a unique node representing the end of the sentence (the *final* node). The arrows are labelled with words and with a *score* which indicates the likelihood that the word perceived is the word actually spoken. Each path from the initial to the final node is one interpretation of the spoken input. An example of a word graph is given in Figure 7.1.

The elements i and j of an item in the table U denote nodes in the word graph. They indicate that the item represents some parse of some path in the word graph from node i to node j. P is an indication of the likelihood of the correctness of the parse; at the moment it may be computed as the sum of the scores of the arrows in the corresponding path.

At the top of the parse represented by the item we have some rule which is used in some way. This is indicated by *id*, which is the identifier of some grammar rule, say `grammar_rule(`$\mathit{id}, A, \alpha\beta$`)`, where the initial (non-empty) list of members in the right-hand side α generates the paths from i to j, β is a (non-empty) list of remaining members in the right-hand side. (We denote the concatenation of two lists α and β simply by $\alpha\beta$.) The left-hand side A may have been instantiated to A' because of the subparses found for the members in α, and similarly β' as it occurs in the item is just β but possibly further instantiated. The number m indicates that the next member to be found in the right-hand side (the first member of β) is the m-th one, in other words $m = |\alpha| + 1$.

The list t in the items is a representation for the subparses found for the members in α. One could define the structure of such subparses inductively as:

- A subparse may be of the form `leaves(`*LexId*`, `*Ids*`)`, where *LexId* is the identifier of a lexical entry, and *Ids* is a list of consecutive arrows in the word graph of which the words in the labels correspond with the words in the lexical entry.
- A subparse may be of the form `tree(`*RuleId*`, `*Subparses*`)`, where *RuleId* is the identifier of a rule, and *Subparses* is a list of subparses (actually a list of *pointers* to subparses), one for each member in the right-hand side of the rule.

In order to simplify the discussion, we omit below the computation of parses and of scores. We also simplify the unification of features by assuming the grammar is context-free, i.e. the categories are atoms. Further, we omit in the algorithm below the components m from items,

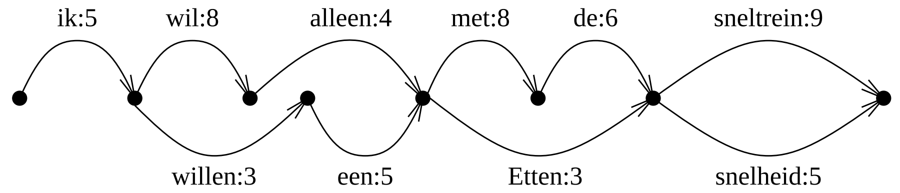


Figure 7.1: An example of a word graph.

which in the actual implementation are used to allow fast access to items, together with the first-member, nonfirst-member, and top-cat tables (see Section 6.3). Also *id* is no longer needed. The items in U are therefore simplified to (i, j, A, β)

The main version of the parsing algorithm processes the arrows in the word graph one by one, according to some predefined order of the arrows. The algorithm assumes that the order is such that an arrow from node i to node j occurs before all arrows from node j to some node h. The procedure applied to each arrow is:

procedure process_arrow (arrow: a):
say a is from node i to node j;
for all `lex(`*id*`,` A_1`,` A_2`,` w`,` v`)`
such that w = the word in arrow a
do for all paths p in the word graph from node j to some node h
    such that the words in p are v
    do process_completed(i, h, A_2)
    end
end.

The procedure "process_completed" is not unlike the completer step from Earley's algorithm (Earley, 1970), but it includes more cases, which are needed due to the absence of the predict step.

procedure process_completed(nodes: i, j, feature: A):
initiate(i, j, A);
extend(i, j, A);
find_top_cat(i, j, A).

procedure initiate(nodes: i, j, feature: A):
for all `grammar_rule(`*id*`,` A_1`,` A_2`,` $[A|\alpha]$`)`
do if $\alpha = []$
    then process_completed(i, j, A_2)
    else add (i, j, A_2, α) to U
    end
end.

procedure extend(nodes: j, h, feature: A):
for all $(i, j, B, [A|\alpha]) \in U$
do if $\alpha = []$
    then process_completed(i, h, B)
    else add (i, h, B, α) to U
    end
end.

procedure find_top_cat(nodes: i, j, feature: A):
if i and j are the initial and final nodes and there is some `top_category(`*id*`,` A`)`
then report "parse found"
end.

In the actual implementation, not the A_2 from the lexical entries and grammar rules are used in "initiate" and "extend", but A_1, the truncated left-hand sides. This requires a second pass to go through each parse that is found in procedure "find_top_cat" in order to find the full features at the roots of parses, making use of the original left-hand sides A_2. This is made possible by recording the parses as components t of items, as we mentioned at the beginning of this section.

Further, the first-member index is used to speed up the location of grammar rules in procedure "initiate", the nonfirst-member index is used to speed up the location of items in U in procedure "extend". The top-cat index is used to speed up the location of the proper top categories in procedure "find_top_cat". All these indexes require the identifiers of lexical entries or rules to be passed on through procedure "process_completed". The non-firstmember index also requires the member number m to be recorded.

A Prolog version of the parser is given in chapter 8.

Chapter 8

Some Variations of Bottom-up Parsing

As indicated in chapter 7 a number of variations are possible on the bottom-up Earley parser discussed there. In this chapter we briefly indicate a few other parsers that we have implemented.

Most importantly, all parsers that we have constructed can work in two phases. In the first phase we ignore semantic constraints, and we combine derivations that give rise to the same result (*packing*). In the second phase we then construct analyses including the semantics on the basis of the items derived in the first phase.

Given the fact that we need a second phase to reconstruct parses it turned out to be worthwile to consider a technique we call *item weakening*. This technique can decrease the number and size of items drastically. This is very important since the memory requirements of the Bottom-up Earley Parser discussed before are quite demanding.

The following variants of the bottom-up parser have been implemented; each of these variants include packing and item weakening. Also note that each of the parsers is able to create all possible top categories anywhere in the input (word graph), as a first step towards *robustness*. This approach to robustness is not yet discussed in this deliverable, but will be subject to further study in the coming months.

- bottom-up Earley with indexing on category
- inactive chart parser: no active items are constructed (shift-reduce parser).
- active chart parser: version of the Earley parser in which both active and inactive items are maintained.

A number of head-driven parsers are discussed in chapter 9. These head-driven parsers are sufficiently different from the chart parsers discussed here to warrant a special treatment.

8.1 Packing

If semantic constraints are ignored then there usually can be different ways to derive the fact that there is, say, a VP from a position p_0 to a position p. If this VP includes prepositional phrases then each derivation might be related to a different internal structure of that VP, one for each possible PP-attachment.

The idea of *packing* is that even if there are different ways to construct such an item, we only use that item once to compute its consequences. For this reason we maintain a set (rather than a bag) of items. Such items are triples `item( P0, P, Cat)` indicating that there is a derivation of `Cat` from `P0` to `P`. Information concerning the way in which such an item was constructed (to be used in the second phase) is asserted in a different table. Every time we create an item we check whether a more general item already existed; only if the item is new we add it to the table.

8.2 Item Weakening

Item weakening is a technique that can be useful for complex-feature-based grammatical analysis. The idea is that we *weaken* the information in each item, by replacing the item with a more general (subsuming) item. Thus, if we find an item of the form `item( P0, P, Cat)` we add an item `item( P0, P, General)` to the table, where `General` subsumes `Cat`. The detailed definition of this weakening operation might differ for different grammars.

Clearly, this technique implies that the table will be unsound in general; on the other hand items will be (much) smaller. Moreover there will be much less possible items, and hence much less work to be done. We achieve a kind of packing even for items that were not (completely) the same, but were similar to a certain degree. During the second phase we have to take into account all information from the grammar rules, and we filter out the incorrect analyses introduced by item weakening.

The definition of an 'appropriate' item weakening operation is crucial for efficient processing. The trick is to come up with an item weakening operation that reduces the size of items as much as possible but gives rise only to a small number of incorrect derivations.

Item weakening is related to a similar notion (*goal-weakening*) that comes up in the discussion of the head-driven parsers in the next chapter.

8.3 Appendix: Prolog implementations of four bottom-up parsers

8.3.1 Bottom-up Earley Parser

This is the parser as described in chapter 7.

```
:- module(bu_earley, [ parse/2 ]).

% lexical analysis has already been performed: lex:lex_analysis/6
parse(Cat,Score) :-
    process_word_graph,
    wordgraph:start(Begin),
    wordgraph:final(End,Conf2),
    second_phase(Begin,End,Cat,Conf1),
    multiply_scores(Conf1,Conf2,Score).

process_word_graph :-
    ( lex:clause(lex_analysis(Begin,Lex_End,LHS,_,LHS_Conf,Id,_,_),_,Ref),
      chart:assert(leaves(Ref), HRef),
      process_completed(Begin, Lex_End, Id, LHS, LHS_Conf, HRef,1),
      fail
    ; true
    ).

process_completed(Begin, End, Id, LHS, Conf, Ref,I0) :-
    initiate_entries(Begin, End, Id, LHS, Conf, Ref,I0),
    extend_entries(Begin, End, Id, LHS, Conf, Ref,I0),
    find_top_cat(Begin, End, Id, LHS, Conf, Ref).

initiate_entries(Begin, End, Id, Mem, Conf, Ref,I0) :-
    ( I is I0+1,
      grammar:first_member_index(Id, Members),
      lists:member(RuleId, Members),
      grammar:grammar_rule(RuleId, LHS, _, [Mem | RHS_suf]),
```

```
      continue(RHS_suf,RuleId,[Ref],Begin,End,LHS,Conf,I,1),
      fail
    ; true
    ).

extend_entries(Begin, End, Id, Mem, New_Conf, New_Ref,I0) :-
    ( I is I0+1,
      grammar:nonfirst_member_index(Id, Members),
      lists:member(RuleId - MemNr, Members),
      completer_index(Begin, RuleId, MemNr, ItemRef),
      chart:clause(item(ItBegin, LHS, [Mem | RHS_suf], Old_Conf, Old_Refs),
         true, ItemRef),
      lists:append(Old_Refs, [New_Ref], Refs),
      probabilities:multiply_scores(New_Conf, Old_Conf, Conf),
      continue(RHS_suf,RuleId,Refs,ItBegin,End,LHS,Conf,I,MemNr),
      fail
    ; true
    ).

continue([],RuleId,Refs,ItBegin,End,LHS,Conf,I,_) :-
    chart:assert(tree(RuleId,Refs),Ref),
    process_completed(ItBegin,End,RuleId,LHS,Conf,Ref,I).
continue([H|T],RuleId,Refs,ItBegin,End,LHS,Conf,_,MemNr) :-
    NewMemNr is MemNr+1,
    store_item(ItBegin,End,RuleId,LHS,NewMemNr,[H|T],Conf,Refs).

find_top_cat(Begin,End,Id,Cat,Score,Ref) :-
    (  grammar:top_cat_index(Id,TopCatIds),
       lists:member(TopCatId,TopCatIds),
       grammar:top_category(TopCatId,Cat),
       chart:assertz(second(Begin,End,Cat,Ref,Score)),
       fail
    ;  true
    ).

store_item(Begin, End, RuleId, LHS, MemNr, RHS_suffix, Conf, Subtrees) :-
    chart:assert(item(Begin, LHS, RHS_suffix, Conf, Subtrees), Ref),
    chart:assert(completer_index(End, RuleId, MemNr, Ref)).

second_phase(P0,P,Cat,Conf) :-
    chart:second(P0,P,Cat,Ref,Conf),
    second_pass(Ref,Cat).

second_pass(Ref,LHS) :-
    chart:clause(leaves(LRef), true, Ref),
    lex:clause(lex_analysis(_,_,LHS,LHS,_,_),_,LRef).

second_pass(Ref,LHS) :-
    chart:clause(tree(RuleId, Refs), true, Ref),
    grammar:grammar_rule(RuleId, _, LHS, RHS),
    second_pass_list(Refs,RHS).

second_pass_list([],[]).
second_pass_list([Ref|Refs],[El|Els]) :-
    second_pass(Ref,El),
    second_pass_list(Refs,Els).
```

8.3.2 Bottom-up Earley Parser with Indexing

This is another version of the bottom-up Earley parser in which we perform indexing using categories. Note that we assume a slightly different representation of grammar rules in order to obtain indexing of grammar-rules on the functor of the left-most right-hand-side member.

```
:- module(bu_earley_index, [ parse/2 ]).

% this uses grammar_rule1(Leftmost_daughter, RuleId, SynMother, OtherDs)
parse(Cat,Score) :-
    process_word_graph,
    wordgraph:start(Begin),
    wordgraph:final(End,Conf2),
    second_phase(Begin,End,Cat,Conf1),
    multiply_scores(Conf1,Conf2,Score).

process_word_graph :-
    ( lex:clause(lex_analysis(Begin,Lex_End,LHS,_,LHS_Conf,Id),_,Ref),
      chart:assert(leaves(Ref), HRef),
      process_completed(Begin, Lex_End,  LHS, LHS_Conf, HRef),
      fail
    ; true
    ).

process_completed(Begin, End,  LHS, Conf, Ref) :-
    initiate_entries(Begin, End, LHS, Conf, Ref),
    extend_entries(Begin, End, LHS, Conf, Ref),
    find_top_cat(Begin, End, LHS, Conf, Ref).

initiate_entries(Begin, End, Mem, Conf, Ref) :-
    ( grammar_rule1(Mem, RuleId, LHS, RHS_suf),
      continue(RHS_suf,RuleId,[Ref],Begin,End,LHS,Conf),
      fail
    ; true
    ).

extend_entries(Begin, End, Mem, New_Conf, New_Ref) :-
    ( Mem =.. [F|Args],
      ITEM =.. [F,Begin,ItBegin,LHS,RHS_Suf,Old_Conf,Old_Refs,RuleId|Args],
      chart:ITEM, append(Old_Refs, [New_Ref], Refs),
      multiply_scores(New_Conf, Old_Conf, Conf),
      continue(RHS_Suf,RuleId,Refs,ItBegin,End,LHS,Conf),
      fail
    ; true
    ).

continue([],RuleId,Refs,ItBegin,End,LHS,Conf) :-
    chart:assert(tree(RuleId,Refs),Ref),
    process_completed(ItBegin,End,LHS,Conf,Ref).
continue([H|T],RuleId,Refs,ItBegin,End,LHS,Conf) :-
    store_item(ItBegin,End,RuleId,LHS,[H|T],Conf,Refs).

find_top_cat(Begin,End,Cat,Score,Ref) :-
    ( grammar:top_category(Cat),
      chart:assertz(second(Begin,End,Cat,Ref,Score)),
      fail
    ; true
    ).
```

```
% this uses Prolog's first argument indexing.
store_item(Begin, End, Id, LHS, [Hd|Tail], Conf, Ts) :-
    Hd =.. [F|Args],
    ITEM =.. [F,End,Begin,LHS,Tail,Conf,Ts,Id|Args],
    chart:assert(ITEM).

second_phase(P0,P,Cat,Conf) :-
    chart:second(P0,P,Cat,Ref,Conf),
    second_pass(Ref,Cat).

second_pass(Ref, LHS) :- chart:clause(leaves(LRef), true, Ref),
    lex:clause(lex_analysis(_,_,LHS,LHS,_,_,_,_),_,LRef).

second_pass(Ref, LHS) :- clause(tree(Id, Refs), true, Ref),
    grammar_rule(Id, _, LHS, RHS),
    second_pass_list(Refs,RHS).

second_pass_list([],[]).
second_pass_list([Ref|Refs],[El|Els]) :-
    second_pass(Ref,El),
    second_pass_list(Refs,Els).
```

8.3.3 Bottom-up Earley Parser with Packing and Item Weakening

This is yet another version of bottom-up Earley. This version implements both packing and item weakening. In order to obtain packing we keep the inactive edges in the chart too.

```
:- module(active,[ parse/2 ]).

% bottom-up chart parser with both active and inactive items. This
% one uses packing, and a restriction in the first phase, to have
% fewer and smaller items. The second phase adds full constraints.
parse(Cat,Score) :-
    process_word_graph,
    wordgraph:start(Begin),
    wordgraph:final(End,Conf2),
    second_phase(Begin,End,Cat,Conf1),
    multiply_scores(Conf1,Conf2,Score).

process_word_graph :-
    ( lex:clause(lex_analysis(P0,P,LHS,_,_,Id),_,Ref),
      debug_message(1,"~w ~w ~w~n",[P0,P,Id]),
      store([],LHS,P0,P,leaves(Ref),[]),
      fail
    ; true
    ).

store([],LHS0,P0,P,RuleId,His) :-
    weaken(LHS0,LHS),
    LHS =.. [F|Args],
    INACTIVE =.. [F,P0,P|Args],
    assertz_if_new(chart:INACTIVE,Ref,New),
    chart:assertz(his(Ref,RuleId,His)),
    continue_inactive(New,P0,P,LHS,Ref).

store([H0|T0],LHS0,P0,P,RuleId,His) :-
    weaken(LHS0,LHS),
    weaken_list([H0|T0],[H|T]),
    H =.. [F|Args],
```

```
    ACTIVE =.. [F,P,P0,LHS,T,RuleId,His|Args],
    chart:assertz(ACTIVE).

continue_inactive(not_new,_,_,_,_).
continue_inactive(    new,P0,P,LHS,Ref) :-
    initiate(P0,P,LHS,Ref),
    extend(P0,P,LHS,Ref),
    add_for_second_phase(P0,P,LHS,Ref).

initiate(P0,P,First,Ref) :-
    (  grammar_rule1(First,Id,LHS,Rest),
       store(Rest,LHS,P0,P,rule(Id),[Ref]),
       fail
    ;  true
    ).

extend(P1,P,First,Ref) :-
    (  First =.. [F|Args],
       ACTIVE =.. [F,P1,P0,LHS,Rest,Id,Refs0|Args],
       chart:ACTIVE, append(Refs0,[Ref],Refs),
       store(Rest,LHS,P0,P,Id,Refs),
       fail
    ;  true
    ).

add_for_second_phase(P0,P,LHS,Ref) :-
    (  grammar:top_category(LHS),
       chart:assertz(second(P0,P,LHS,Ref)),
       fail
    ;  true
    ).

second_phase(P0,P,Cat,Conf) :-
    chart:second(P0,P,Cat,Ref),
    second_phase(Ref,Cat,Conf).

second_phase(Ref,Cat,Score) :- % 2nd phase is memo-ized.
    (  chart:reconstructed(Ref)
    -> true
    ;  chart:assertz(reconstructed(Ref)),
       (  second_phase0(Ref,Cat0,Score0),
          chart:assertz(reconstructed(Ref,Cat0,Score0)),
          fail
       ;  true
       )
    ),
    chart:reconstructed(Ref,Cat,Score).

second_phase0(Ref,Cat0,Sc) :-
    chart:his(Ref,Rule,Refs),
    second_phase2(Rule,Cat0,Refs,Sc).

second_phase2(rule(Rule),Cat,Refs,Sc) :-
    grammar_rule(Rule,Cat,Rhs),
    second_phase_list(Refs,Rhs,0,Sc).
second_phase2(leaves(Ref),Cat,[],Sc1) :-
    lex:clause(lex_analysis(_,_,Cat,Cat,Sc1,_),_,Ref).
```

```
second_phase_list([],[],S,S).
second_phase_list([H|T],[H2|T2],S0,S) :-
    second_phase(H,H2,S1),
    multiply_scores(S0,S1,S2),
    second_phase_list(T,T2,S2,S).
```

8.3.4 Shift-reduce Parser with Packing and Item Weakening

This is a shift-reduce parser with packing and item weakening. No active edges are ever constructed. Note that we assume a special variant of grammar-rules in which the order of right-hand-side members is inverted.

```
:- module(inactive,[ parse/2 ]).

%% inv_rule(Cat,X0,Mother,Body,RuleId) version of grammar rule where the
%% list of daughters is inverted; X0 is variant of Mother without semantics
parse(Cat,Score) :-
    process_word_graph,
    wordgraph:start(Start),
    wordgraph:final(Final,Score1),
    second_phase(Start,Final,Cat,Score2),
    multiply_scores(Score1,Score2,Score).

process_word_graph :-
    (  lex:clause(lex_analysis(P0,P,M,_,_Sc,Id),_,Ref),
       store(M,P0,P,leaves(Ref),[]),
       fail
    ;  true
    ).

store(M0,P0,P,RuleId,Refs):-
    weaken(M0,M),
    assertz_if_new(chart:inactive(P,P0,M),Ref,New),
    chart:assertz(his(Ref,RuleId,Refs)),
    continue_inactive(New,M,P0,P,Ref).

continue_inactive(not_new,_,_,_,_).
continue_inactive(new,M,P0,P,Ref) :-
    extend(M,P0,P,Ref),
    add_for_second_phase(P0,P,M,Ref).

extend(Cat,P1,P,Ref) :-
    (  inv_rule(Cat,X0,_,Body,RuleId),
       process_body(Body,P0,P1,Refs),
       store(X0,P0,P,rule(RuleId),[Ref|Refs]),
       fail
    ;  true
    ).

process_body([],P,P,[]).
process_body([H|T],P0,P,[Ref|Refs]) :-
    chart:clause(inactive(P,P1,H),_,Ref),
    process_body(T,P0,P1,Refs).

add_for_second_phase(P0,P,M,Ref) :-
    (  grammar:top_category(M),
       chart:assertz(second(P0,P,M,Ref)),
       fail
```

```
    ;  true
    ).

second_phase(P0,P,Cat,Conf) :-
    chart:second(P0,P,Cat,Ref),
    second_phase(Ref,Cat,Conf).

second_phase(Ref,Cat,Score) :-
    (  chart:reconstructed(Ref)
    -> true
    ;  chart:assertz(reconstructed(Ref)),
       (  second_phase0(Ref,Cat0,Score0),
          chart:assertz(reconstructed(Ref,Cat0,Score0)),
          fail
       ;  true
       )
    ),
    chart:reconstructed(Ref,Cat,Score).

second_phase0(Ref,Cat0,Conf) :-
    chart:his(Ref,Rule,Refs),
    second_phase2(Rule,Cat0,Refs,Conf).

second_phase2(rule(Rule),Cat,Refs,Conf) :-
    inv_rule(D1,Cat,Cat,Ds,Rule),
    second_phase_list(Refs,[D1|Ds],0,Conf).

second_phase2(leaves(Ref),Cat,[],Conf) :-
    lex:clause(lex_analysis(_,_,Cat,Cat,Conf,_),_,Ref).

second_phase_list([],[],S,S).
second_phase_list([Ref0|Refs],[D0|Ds],S0,S) :-
    second_phase(Ref0,D0,S1),
    multiply_scores(S0,S1,S2),
    second_phase_list(Refs,Ds,S2,S).
```

Chapter 9

An Efficient Prolog Implementation of the Head-corner Parser

This chapter describes an efficient Prolog implementation of a bi-directional, head-driven parser for constraint-based grammars. Search is implemented using a memoization technique. Parse-trees can be reconstructed on the basis of sets of packed items. A new techniques called goal-weakening is described.

9.1 Motivation

In this chapter we discuss in full detail the implementation of the head-corner parser developed as one of the parsers for the OVIS2 prototype. This project is part of the Priority Programme on Language and Speech Technology of the Dutch Research Organization. The aim of this program is to construct a Spoken Dialog System for Public Transportation Information. For an overview of the Programme we refer to Boves et al. (1995).

Lexicalist grammar formalisms, such as HPSG and Categorial Unification Grammar (CUG) have two characteristic properties. Lexical elements and phrases are associated with categories that have considerable internal structure. Second, instead of construction-specific rules, a small set of generic rule schemata is used. Consequently, the set of constituent structures defined by a grammar cannot be 'read off' the rule set directly, but is defined by the interaction of the rule schemata and the lexical categories. Applying standard parsing algorithms to such grammars is unsatisfactory for a number of reasons. Earley parsing is intractable in general, as the rule set is simply too general. For some grammars , naive top-down prediction may even fail to terminate. Shieber (1985) therefore proposes a modified version of the Earley-parser, using *restricted* top-down prediction. While this modification leads to better termination properties of the parsing method, in practice it easily leads to a complete trivialization of the top-down prediction step, thus leading to inferior performance.

Bottom-up parsing is far more attractive for lexicalist formalisms, as it is driven by the syntactic information associated with lexical elements. Certain inadequacies remain, however. Most importantly, the selection of rules to be considered for application may not be very efficient. Consider, for instance, the following DCG rule:

```
s([],Sem) --> Arg, vp([Arg],Sem).
```

A parser in which application of a rule is driven by the left-most daughter, as it is for instance in a standard bottom-up active chart parser, will consider the application of this rule each time an arbitrary constituent `Arg` is derived. For a bottom-up active chart parser, for instance, this may lead to the introduction of large numbers of active items. Most of these items will be useless.

For instance, if a determiner is derived, there is no need to invoke the rule, as there are simply no VP's selecting a determiner as subject. Parsers in which the application of a rule is driven by the rightmost daughter, such as shift-reduce and inactive bottom-up chart parsers, encounter a similar problem for rules such as:

```
vp(As,Sem) --> vp([Arg|As],Sem), Arg.
```

Each time an arbitrary constituent `Arg` is derived, the parser will consider applying this rule, and a search for a matching VP-constituent will be carried out. Again, in many cases (if `Arg` is instantiated as a determiner or preposition, for instance) this search is doomed to fail, as a VP subcategorizing for a category `Arg` may simply not be derivable by the grammar. The problem may seem less acute than that posed by uninstantiated left-most daughters for an active chart parser, as only a search of the chart is carried out and no additional items are added to it. Note, however, that the amount of search required may grow exponentially, if more than one uninstantiated daughter is present:

```
vp(As) --> vp([A1,A2|As]), A1, A2.
```

or if the number of daughters is not specified by the rule:

```
vp([A0]) --> vp([A0,...,An]), A1,..., An.
```

as appears to be the case for some of the rule-schemata used in Head-driven Phrase Structure Grammar.

Several authors have suggested parsing algorithms which appear to be more suitable for lexicalist grammars. Kay (1989) discusses the concept of *head-driven* parsing. The key idea underlying this concept is that the linguistic concept *head* can be used to obtain parsing algorithms which are better suited for typical natural language grammars. Most linguistic formalisms assume that among the daughters introduced by a rule or rule-schema there is one daughter which can be identified as the *head* of that rule. There are several criteria for deciding which daughter is the head. Two of these criteria seem relevant for parsing. First of all, the head of a rule determines to a large extent what other daughters may or must be present, as the head *subcategorizes* for the other daughters. Second, the syntactic category and morphological properties of the mother node are, in the default case, identical to the category and morphological properties of the head daughter. These two properties suggest that it may be possible to design a parsing strategy in which one first identifies a potential head of a rule, before starting to parse the non-head daughters. By starting with the head, important information about the remaining daughters is obtained. Furthermore, since the head is to a large extent identical to the mother category, effective top-down identification of a potential head should be possible. A head-driven parsing strategy is particularly interesting for lexicalist grammars, as these grammars normally suffer most from the problem that rules or rule-schemata hardly constrain the search-space of the parser.

In Kay (1989) two different head-driven parsers are presented. First, a 'head-driven' shift-reduce parser is presented which differs from a standard shift-reduce parser in that it considers the application of a rule (i.e. a reduce step) only if a category matching the head of the rule has been found. Furthermore, it may shift elements onto the parse-stack which are in a sense similar to the active items (or 'dotted rules') of active chart parsers. By using the head of rule to determine whether a rule is applicable, the head-driven shift-reduce parser avoids the disadvantages of parsers in which either the leftmost or rightmost daughter is used to drive the selection of rules. Kay also presents a 'head-corner' parser. The striking property of this parser is that it does not parse a phrase from left to right, but instead operates 'bidirectionally'. It starts by locating a potential head of the phrase and then proceeds by parsing the daughters to the left and the right of the head. Again, this strategy avoids the disadvantages of parsers in which rule selection is uniformly driven by either the leftmost or rightmost daughter. Furthermore, by selecting potential heads on the basis of a 'head-corner table' (comparable to the left-corner table of a left-corner parser) it may use top-down filtering to minimize the search-space. Head-corner parsing has also been considered elsewhere. In Satta and Stock (1989),Sikkel and op den

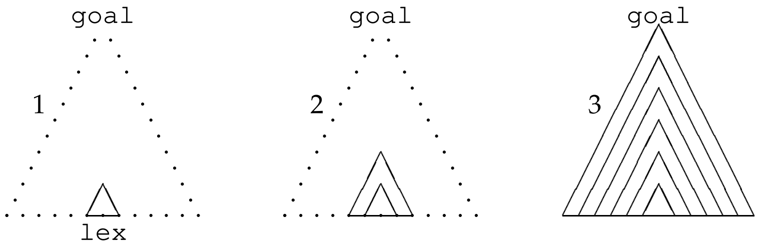


Figure 9.1: The head-corner parser. In order to prove that a string is of category `goal`, the parser selects the head of the string (1), and proves that this element is the head-corner of the goal. To this end, a rule is selected of which this lexical entry is the head daughter. Then the other daughters of the rule are parsed recursively in a bidirectional fashion: the daughters to the left of the head are parsed from right to left (starting from the head), and the daughters right of the head are parsed from left to right (starting from the head). The result is a slightly larger head-corner (2). This process repeats itself until a head-corner is constructed which dominates the whole string (3).

Akker (1992),Sikkel and op den Akker (1993), Sikkel (1993) chart-based head-corner parsing for context-free grammar is considered. It is shown that, in spite of the fact that bidirectional parsing seemingly leads to more overhead than left-to-right parsing, the worst-case complexity of a head-corner parser does not exceed that of an Earley parser. Some further variations are discussed in Nederhof and Satta (1994).

van Noord (1991),van Noord (1993) argues that head-corner parsing is particularly useful for parsing with non-concatenative grammar formalisms. In Lavelli and Satta (1991),van Noord (1994) head-driven parsing strategies for Lexicalized Tree Adjoining Grammars are presented.

Finally, Bouma and van Noord (1993) present some results indicating that head-driven parsing strategies indeed can improve the average-case time efficiency. These results are supported by preliminary experiments with the OVIS grammar. In these experiments we also find that the head-corner parser uses *much* less space than other parsers.

9.2 Non-deterministic Versions of the Head-corner Parser

9.2.1 A Simple Head-corner Parser

Head-corner parsing is a radical approach to head-driven parsing in that it gives up the idea that parsing should proceed form left to right. Rather, the order of processing in a head-corner parser is bidirectional, starting from a head outward ('island'-driven). A head-corner parser can be thought of as a generalization of the left-corner parser (Rosenkrantz and Lewis-II, 1970). As in the left-corner parser, the flow of information in a head-corner parser is both bottom-up *and* top-down.

The basic idea of the head-corner parser is illustrated in figure 9.1. The parser selects the head of the string (1), and proves that this element is the head-corner of the goal. To this end, a rule is selected of which this lexical entry is the head daughter. Then the other daughters of the rule are parsed recursively in a bidirectional fashion: the daughters to the left of the head are parsed from right to left (starting from the head), and the daughters right of the head are parsed from left to right (starting from the head). The result is a slightly larger head-corner (2). This process repeats itself until a head-corner is constructed which dominates the whole string (3).

Note that a rule is triggered only with a fully instantiated head-daughter. The 'generate-and-test' behavior observed above is avoided in a head-corner parser, because the rule is applied only if the `VP` is found, and hence `Arg` is instantiated. For example if `Arg = np(sg3,[],Subj)`, the parser continues to search for a singular `NP`, and need not consider other categories.

In this figure we have made certain assumptions concerning the representation of the grammar rules and the input sentence that turn out to be useful. These assumptions are explicated now as follows.

The Representation of a Grammar. In this section we will assume that grammar rules are represented by the predicate `headed_rule/9` in which the first argument is the head of the rule, the second argument is the mother node of the rule, the third argument is a list consisting of the reversed list of daughters to the left of the head, and the fourth argument is a list of the daughters right of the head. The other arguments are ignored for the moment. Such a representation will be constructed by a parser-generator on the basis of a grammar defined in the appropriate format (cf. section 2).

As an example, the rule

```
grammar_rule(id23, x(A,E), [a(A), b(B,A), x(C,B), d(C,D), e(D,E) ]).
```

of which the third daughter constitutes the head, is represented now as:

```
headed_rule( x(C,B), x(A,E), [b(B,A), a(A)],
                              [d(C,D), e(D,E)],_,_,_,_,_).
```

This notation is the result of a compilation step.

The Representation of the Input String. In this section we will be assuming that lexical lookup has been performed already by another module. This module has asserted clauses for the predicate `lexical_analysis/6` where the first two arguments are the string positions and the third argument is the (lexical) category. For the moment we will ignore the last three arguments. For an input sentence 'Time flies like an arrow' this module may produce the following set of clauses:

```
lexical_analysis(0,1,verb,_,_,_).
lexical_analysis(0,1,noun,_,_,_).
lexical_analysis(0,2,noun,_,_,_).
lexical_analysis(1,2,noun,_,_,_).
lexical_analysis(1,2,verb,_,_,_).
lexical_analysis(2,3,prep,_,_,_).
lexical_analysis(2,3,verb,_,_,_).
lexical_analysis(3,4, det,_,_,_).
lexical_analysis(4,5,noun,_,_,_).
```

A simple non-deterministic version of the head-corner parser is defined as follows:

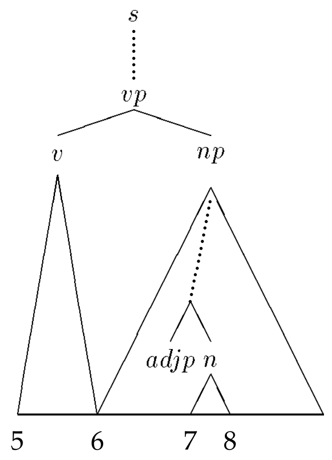
```
% parse(?Cat,+P0,+P)
parse(Cat,P0,P) :-
    parse(Cat,P0,P,P0,P).

% parse(?Cat,?P0,?P,+E0,+E)
% there is a category Cat from P0-P within the interval E0-E
parse(Cat,P0,P,E0,E) :-
    predict(Cat,P0,P,SmallCat,Q0,Q),
    head_corner(SmallCat,Q0,Q,Cat,P0,P,E0,E).

% head_corner(?Small,+P0,+P,?Cat,?Q0,?Q,+E0,+E)
% Small from P0-P is a head-corner of Cat from Q0-Q where Q0-Q occurs
% within E0-E
head_corner(Cat,P0,P,Cat,P0,P,_,_).
head_corner(Small,Q0,Q,Cat,P0,P,E0,E) :-
    headed_rule(Small,Mother,RevLeftDs,RightDs,_,_,_,_,_),
    parse_left_ds(RevLeftDs,QL,Q0,E0),
    parse_right_ds(RightDs,Q,QR,E),
```

```
    head_corner(Mother,QL,QR,Cat,P0,P,E0,E).

% parse_left_ds(+RevLeftDs,-Qleft,+Qright,+ExtremeLeft)
% there is a sequence of categories LeftDs from Qleft to Qright
% s.t. RevLeftDs is the reverse of LeftDs, and ExtremeLeft < Qleft.
parse_left_ds([],Q,Q,_).
parse_left_ds([H|T],Q0,Q,E) :-
    parse(H,Q1,Q,E,Q),
    parse_left_ds(T,Q0,Q1,E).

% parse_right_ds(+RightDs,+Qleft,-Qright,+ExtremeRight)
% there is a sequence of categories RightDs from Qleft to Qright
% s.t. Qright < ExtremeRight.
parse_right_ds([],Q,Q,_).
parse_right_ds([H|T],Q0,Q,E) :-
    parse(H,Q0,Q1,Q0,E),
    parse_right_ds(T,Q1,Q,E).

% predict(?Cat,-P0,-P,+E0,+E,-SmallCat,-Q0,-Q)
% SmallCat from Q0-Q is a lexical category and possible head-corner
% of Cat from P0-P, where P0-P is within E0-E.
predict(_Cat,_P0,_P,E0,E,Small,Q0,Q) :-
    lex:lexical_analysis(Q0,Q,Small,_,_,_),
    between(Q0,Q,E0,E).

% between(+P0,+P,+E0,+E)
% P0-P occurs within E0-E
between(P0,P,E0,E) :-
    E0 =< P0, P =< E.
```

The Quicksort Effect. Note that unlike the left-corner parser, the head-corner parser may need to consider alternative words as a possible head-corner of a phrase, e.g. when parsing a sentence which contains several verbs. This problem is somewhat reduced because of the following observation.

The main difference of the head-corner parser on page 90 with a simple version of the left-corner parser is — apart from the head-driven selection of rules — the use of two pairs of indices, to implement the bidirectional way in which the parser proceeds through the string.

Observe that each parse-goal in the left-corner parser is provided with a category and a left-most position. In the head-corner parser a parse-goal is provided either with a begin or end position (depending on whether we parse from the head to the left or to the right) but also with the extreme positions between which the category should be found. In general the parse predicate is thus provided with a category and two pairs of indices. The first pair indicates the begin and end position of the category, the second pair indicates the extreme positions between which the first pair should lay. The following figure illustrates this point with an example:



Suppose we found for a goal category s a possible head-corner v from position 5 to 6. In order to construct a complete tree s for this head-corner, a rule is selected which dictates that a category np should be parsed to the right, starting from position 6. To parse np, we predict the head-corner n between 7 and 8. Suppose furthermore that in order to connect n to np a rule is selected which requires a category $adjp$ to the left of n. It will be clear that this category should end in position 7, but can never start before position 6. Hence the only candidate head-corner of this phrase is to be found between 6 and 7. This example illustrates how the use of two pairs of string positions reduces the number of possible head-corners for a given goal.

9.2.2 Adding Top-down Filtering

Category Information

The head-relation of two categories h, m holds with respect to a grammar iff the grammar contains a rule with left hand side m and head daughter h. The relation 'head-corner' is the reflexive and transitive closure of the head relation. As in the left-corner parser, a 'linking' table is maintained which represents important aspects of this head-corner relation. For some grammars this table simply represents the fact that the HEAD features of a category and its head-corner are shared.

For the moment we will assume that this table consists of clauses for the predicate `head_link/2` where the second argument is a category for which the first argument is a possible head-corner. A sample linking table may be:

```
head_link(verb,    s).
head_link(verb,   vp).
head_link(  vp,    s).
head_link(noun,   np).
head_link(prep,   pp).
head_link(comp,sbar).
head_link(   X,    X).
```

String Position Information

The head-corner table also includes information about begin and end positions, following an idea in Sikkel (1993). For example, if the goal is to parse a phrase with category *sbar* from position 7, and within positions 7 and 12, then for some grammars it can be concluded that the only possible head-corner for this goal should be a complementizer starting at position 7. Such information is compiled into the table as well. Hence the number of possible head-corners is reduced. Thus, rather than the predicate `head_link/2` we have a predicate `head_link/6` in which the fourth, fifth and sixth arguments are a category and its start and end position, where the first, second and third argument represents a possible head-corner of that triple. The previous example may then become something like:

```
head_link(verb,_,_,   s,_,_).
head_link(verb,P,_,  vp,P,_).
head_link(  vp,_,P,   s,_,P).
head_link(noun,_,_,  np,_,_).
head_link(prep,P,_,  pp,P,_).
head_link(comp,P,_,sbar,P,_).
head_link(   X,P,Q,   X,P,Q).
```

Integrating the top-down information

The linking information can now be integrated into the `predict/8` predicate as follows:

```
% predict(?Cat,-P0,-P,+E0,+E,-SmallCat,-Q0,-Q)
% SmallCat from Q0-Q is a lexical category and possible head-corner
% of Cat from P0-P, where P0-P is within E0-E.
predict(Cat,P0,P,Small,Q0,Q,E0,E) :-
    head_link(Small,Q0,Q,Cat,P0,P),  %%% new
    lexical_analysis(Q0,Q,Small,_,_,_),
    between(Q0,Q,E0,E).
```

A few potential problems may arise in connection with the use of linking tables.

Firstly, for constraint-based grammars of the type assumed here the number of possible non-terminals is infinite. Therefore, we generally cannot use all information available in the grammar but rather we should compute a 'weakened' version of the linking table. This can be accomplished for example by replacing all terms beyond a certain depth by anonymous variables (or by other *restrictors* (Shieber, 1985)).

Secondly, the use of a linking table may give rise to spurious ambiguities if we are not careful. Consider the case in which the category we are trying to parse can be matched against two different items in the linking table, but in which case the predicted head-category may turn out to be the same.

The first thing we should note here is that the memoization technique discussed in section 9.3 takes care of this problem.

Another possibility is to use the linking table only as a check, but not as a source of information, by encapsulating the call within a double negation. The prediction predicate then may be defined as:

```
% predict(?Cat,-P0,-P,+E0,+E,-SmallCat,-Q0,-Q)
% SmallCat from Q0-Q is a lexical category and possible head-corner
% of Cat from P0-P, where P0-P is within E0-E.
predict(Cat,P0,P,Small,Q0,Q,E0,E) :-
    lexical_analysis(Q0,Q,Small,_,_,_),
    \+ \+ head_link(Small,Q0,Q,Cat,P0,P),  %%% new
    between(Q0,Q,E0,E).
```

This approach also solves another potential problem: the linking table may give rise to (undesired) cyclic terms due to the absence of the occur-check. The double negation takes care of this possible problem too.

Note that we have changed the order of the goals of the body of the `predict/8` predicate, for obvious reasons. In fact, it may be worthwile in this approach to add another linking check after each rule application:

```
% head_corner(?Small,+P0,+P,?Cat,?Q0,?Q,+E0,+E)
% Small from P0-P is a head-corner of Cat from Q0-Q where Q0-Q occurs
```

```
% within E0-E
head_corner(Cat,P0,P,Cat,P0,P,_,_).
head_corner(Small,Q0,Q,Cat,P0,P,E0,E) :-
    headed_rule(Small,Mother,RevLeftDs,RightDs,_,_,_,_,_),
    parse_left_ds(RevLeftDs,QL,Q0,E0),
    parse_right_ds(RightDs,Q,QR,E),
    \+ \+ head_link(Mother,QL,QR,Cat,P0,P),
    head_corner(Mother,QL,QR,Cat,P0,P,E0,E).
```

9.2.3 Dealing with Empty Productions

In the preceding subsection we have said nothing about empty productions. It turns out that empty productions can be dealt with in the version given on page 90, provided empty productions are also asserted as possible 'lexical analyses'. In such an approach the result of lexical analysis may contain clauses like:

```
lexical_analysis(0,0,np/np,_,_,_).
lexical_analysis(1,1,np/np,_,_,_).
lexical_analysis(2,2,np/np,_,_,_).
lexical_analysis(3,3,np/np,_,_,_).
lexical_analysis(4,4,np/np,_,_,_).
```

There are two objections to this approach. The first objection is that this is a task that can hardly be expected from a lexicon lookup procedure. The second, more important, objection is that empty categories are hypothesized essentially everywhere. A straightforward generalization would of course be to represent the previous set of clauses compactly by:

```
lexical_analysis(X,X,np/np,_,_,_).
```

But in order for this approach to work we must slightly change some of the predicates on page 90 because the instantiation status of some of the arguments now changes. For example we cannot use Prolog's builtin comparison operator anymore, since that operator requires that its arguments are ground.

The version below takes the two objections into account; empty productions are inserted by a special clause for the `predict/8` predicate, and variables are used to indicate their corresponding string positions. Here we assume that empty rules are available as clauses for the predicate `gap/3` where the category is the first argument. The other arguments are ignored for now.

```
% parse(?Cat,+P0,+P)
parse(Cat,P0,P) :-
    parse(Cat,P0,P,P0,P).

% parse(?Cat,?P0,?P,+E0,+E)
% there is a category Cat from P0-P within the interval E0-E
parse(Cat,P0,P,E0,E) :-
    predict(Cat,P0,P,E0,E,SmallCat,Q0,Q),
    head_corner(SmallCat,Q0,Q,Cat,P0,P,E0,E).

% head_corner(?Small,?P0,?P,?Cat,?Q0,?Q,+E0,+E)
% Small from P0-P is a head-corner of Cat from Q0-Q where Q0-Q within E0-E
head_corner(Cat,P0,P,Cat,P0,P,_,_).
head_corner(Small,Q0,Q,Cat,P0,P,E0,E) :-
    headed_rule(Small,Mother,RevLeftDs,RightDs,_,_,_,_,_),
    parse_left_ds(RevLeftDs,QL,Q0,E0,E), parse_right_ds(RightDs,Q,QR,E0,E),
    head_corner(Mother,QL,QR,Cat,P0,P,E0,E).

% parse_left_ds(+RevLeftDs,-Qleft,?Qright,+ExtremeLeft,+ExtremeRight)
```

```
% there is a sequence of categories LeftDs from Qleft to Qright
% s.t. RevLeftDs is the reverse of LeftDs, and ExtremeLeft < Qleft.
parse_left_ds([],Q,Q,_,_).
parse_left_ds([H|T],Q0,Q,E0,E) :-
    ( var(Q) -> RightEdge=E ; RightEdge=Q ),
    parse(H,Q1,Q,E0,RightEdge), parse_left_ds(T,Q0,Q1,E0,RightEdge).

% parse_right_ds(+RightDs,?Qleft,-Qright,+ExtremeLeft,+ExtremeRight)
% there is a sequence of categories RightDs from Qleft to Qright
% s.t. Qright < ExtremeRight.
parse_right_ds([],Q,Q,_,_).
parse_right_ds([H|T],Q0,Q,E0,E) :-
    ( var(Q0) -> LeftEdge=E0 ; LeftEdge=Q0 ),
    parse(H,Q0,Q1,LeftEdge,E), parse_right_ds(T,Q1,Q,LeftEdge,E).

% predict(?Cat,-P0,-P,+E0,+E,-SmallCat,-Q0,-Q)
% SmallCat from Q0-Q is a lexical category and possible head-corner
% of Cat from P0-P, where P0-P is within E0-E.
predict(Cat,P0,P,E0,E,Small,Q0,Q) :-
    head_link(Small,Q0,Q,Cat,P0,P),
    lexical_analysis_or_gap(Q0,Q,Small), between(Q0,Q,E0,E).

% lexical_analysis_or_gap(Q0,Q,Small)
% Small is a lexical category or gap from Q0 to Q
lexical_analysis_or_gap(Q0,Q,Small) :-
    ( lexical_analysis(Q0,Q,Small,_,_,_) ; gap(Small,_,_), Q0=Q ).

% between(?P0,?P,?E0,?E) P0-P occurs within E0-E
between(P0,P,E0,E) :-
    smaller_or_equal(E0,P0), smaller_or_equal( P, E).

% smaller_or_equal(A,B) if A and B are both nonvar, then A =< B.
smaller_or_equal(A,B) :-
    ( ( var(A) ; var(B) ) -> true ; A =< B ).
```

It should be noted that both versions of the head-corner parser are not guaranteed to terminate, even if the grammar defines only a finite number of derivations for all input sentences. Thus, even if the head-corner parser proceeds in a bottom-up direction, it can run into left-recursion problems (just like the left-corner parser can). This is because it may be possible that an empty category is predicted as the head, after which trying to construct a larger projection of this head gives rise to a parse goal for which a similar empty category is a possible candidate head, etcetera.

This problem can be solved in some cases by a good (but relatively expensive) implementation of the memoization technique, e.g. along the lines of Warren (1992). The simplified (but more efficient) memoization technique that we use (cf. section 9.3) however does not. Experience suggest that grammars exhibiting the undesired behavior can be avoided.

9.2.4 Underspecification of the extreme positions

Certain approaches towards robust parsing use the partial results of the parser. In order for such approaches to work it is often assumed that a bottom-up parser is essential: parsers that use top-down information (such as the head-corner parser) may fail to recognize relevant sub-parsers in the context of an ungrammaticality. In this subsection we show that we may still use the head-corner parser in such circumstances.

Normally the head-corner parser such as on page 90 will be called as e.g. :

```
?- parse(s(Sem),0,12).
```

indicating that we want to parse a sentence from position 0 to 12 with category `s(Sem)` (a sentence with a semantic representation that is yet to be discovered). Suppose however that a specific robustness module is interested in all 'maximal projections' anywhere in the sentence. Such a maximal projection may be represented by a term `xp(Sem)`. Furthermore there may be unary grammar rules rewriting such an `xp` into appropriate categories, e.g.:

```
xp(Sem) --> np(Sem,_).
xp(Sem) --> s(Sem).
xp(Sem) --> pp(Sem,_).
xp(Sem) --> advp(Sem).
```

If we want to recognize all maximal projections at all positions in the input, then we can simply give the following parse predicate:

```
?- parse(xp(Sem),_,_).
```

Now it may seem that such an underspecified goal will dramatically slow down the head-corner parser, but this turns out to be a false expectation. In actual fact we have experienced an increase of efficiency using underspecification. This can only be understood in the light of the use of memoization (as discussed in the following section). Even though we now have a much more general goal, the number of different goals that we need to solve is much smaller.

Also note that even though the first call to the parse predicate has variable extreme positions, this does not imply that all power of top-down prediction is lost by this move; recursive calls to the parse predicate may still have instantiated left and/or right extreme positions. The same applies with even more force for top-down information on categories.

9.2.5 Removing the extreme positions

Even though we argued in the previous subsection that underspecification of the extreme positions in the parse predicate does not eliminate top-down prediction completely, it is nevertheless true that its effect is reduced. It may therefore be worthwile to consider a version of the head-corner parser that does not take into account these extreme positions at all.

Clearly this approach leads to a situation in which derivations are attempted that cannot possibly lead to a useful result. However, it turns out that the cost of computing these spurious derivations is sometimes worth it. In the next subsection we define a deterministic implementation of the head-corner parser. The number of 'items' that need to be asserted in the database is much lower if no extreme positions are taken into account. Also, there are much less different goals to be considered (similar to the reduction of the number of goals that is obtained with goal-weakening).

For the current version of the OVIS-grammar removing the extreme positions from the head-corner parser completely is on average as efficient as the version which includes these positions. For particular sentences the differences can be significant. This suggests that for other grammars it may make a difference whether or not extreme positions are taken into account.

9.3 Deterministic Versions of the Head-corner Parser

9.3.1 Memoization

The basic idea behind memoization is simple: do not compute things twice. In Prolog we can keep track of each goal that has already been solved and keep a list of the corresponding solution(s). If the same goal needs to be solved later, then we can skip the computation but simply do a table lookup.

It turns out that the cost of maintaining a table and doing the table lookup is rather expensive itself. Therefore, we slightly change the basic intuition behind memoization into: do not compute *expensive* things twice. It generally takes some experimenting to find out what predicates are good candidates for memoization. In the current setting it turns out that the `parse/5` predicate is a very good candidate for memoization. The other predicates are not.

The current version of our implementation of memoization uses Prolog's internal database to store the tables. Thus, it may be that a meta-interpreter along the lines of the XOLDT system (Warren, 1992) in which the tables are maintained dynamically may improve the current implementation.

Memoization is implemented by two different tables. The first table encodes which goals have already been solved. Items in the first table are called *goal items*. The second table represents all solved (i.e. instantiated) goals. Items in this second table are called *cky items*. One may be tempted to use only the second table. But in that case we would not be able to tell the difference between a goal which has already been searched, but did not result in a solution ('fail-goal') and a goal which has not been searched at all. If we have two tables then we can also immediately stop working on branches in the search space for which it has already been shown that there is no solution.

The first table is represented by the predicate `'GOAL_ITEM'/1`. This predicate simply consists of a number of unit-clauses indicating all goals that have been searched completely. Thus, before we try to attempt to solve `Goal` we first check whether `'GOAL_ITEM'(Goal)` already exists. Given the fact that `Goal` may contain variables we should be a bit careful here. Unification is clearly not appropriate here since that may result in a situation in which a more general goal is not searched because a more specific variant of that goal was solved. We want exactly the opposite: if a more general version of `Goal` is included in the `'GOAL_ITEM'/1` table, then we can continue to look for a solution in the second table.

The second table is represented by the predicate `'CKY_ITEM'/1`. It is defined by unit-clauses which each represent an instantiated goal (i.e. a solution). The memoization of a predicate can now be defined in Prolog essentially as follows. If a given goal is not present in the first table, then all solutions for this goal are collected and asserted in the second table. After that, the first table is updated to represent the fact that this goal now has been searched completely. Finally a solution is picked from the second table.

```
memo(Pred) :-
    (   in_table1(Pred)
    ->  true
    ;   (   call(Pred),
            assert_table2(Pred),
            fail
        ;   assert_table1(Pred)
        )
    ),
    in_table2(Pred).
```

The auxiliary predicates can now be defined as follows. Note that the following definitions are supposed to clarify the way memoization is defined. Note that the predicate `assert_table2` removes solutions that are found to be more specific than other solutions. This is not strictly necessary but is often useful because it decreases the size of the tables; in this approach tables are redundancy-free and hence minimal.

```
in_table1(Specific) :-
```

```
    'GOAL_ITEM'(General), subsumes_chk(General,Specific).

assert_table1(Goal) :-
    assertz('GOAL_ITEM'(Goal)).

in_table2(Specific) :-
    'CKY_ITEM'(General), subsumes_chk(General,Specific).

assert_table2(Goal) :-
    (  'CKY_ITEM'(General), subsumes_chk(General,Goal)
    -> true
    ;  retractall_more_specific(Goal),
       assertz('CKY_ITEM'(Goal))
    ).

retractall_more_specific(Goal) :-
     ( clause(Specific,_,Ref),
       subsumes_chk(Goal,Specific),
       erase(Ref),
       fail
     ; true
     ).
```

In the actual implementation a somewhat more efficient but more complex definition is used (cf. the appendix). Furthermore, the CKY_ITEMs do not keep track of the extreme positions. This implies that in order to see whether a CKY_ITEM is applicable we check whether the interval covered by the CKY_ITEM lies within the extreme positions of the current goal.

The head-corner parser can now be memoized by replacing the `parse/5` goals by `memo(parse/5)` goals.

Note that nothing would prevent us to memo other predicates as well. Experience suggests that the cost of maintaining tables for e.g. the `head_corner` relation is (much) higher than the associated profit.

The use of memoization for only the `parse/5` goals implies that the memory requirements of the head-corner parser in terms of the number of items that is being recorded is much smaller than in ordinary chart parsers. Not only do we refrain from asserting so-called *active* items, but we also refrain from asserting *inactive* items for partial analyses of projections of heads. In practice the difference in space requirements is enormous (2 or 3 orders of magnitude). This difference may in turn be a significant reason for the practical efficiency of the head-corner parser.

In section 9.4 we show how the information contained in the second table can be used to recover parse-trees.

9.3.2 The Occurs Check

It turns out that the use of tables defined in the previous subsection can lead to a problem with cyclic unifications. If we assume that Prolog's unification includes the occurs-check then no problem would arise. But since most versions of Prolog do not implement the occurs-check it is worthwile to investigate this potential problem.

The problem arises because cyclic solutions can be constructed that would not have been constructed by ordinary SLD-resolution. Furthermore, these cyclic structures lead to practical problems because items containing such a cyclic structure may have to be put in the table. In SICStus Prolog this results in a crash.

An example may clarify the problem. Suppose we have a very simple program containing the following unit clause:

```
x(A,B).
```

Furthermore suppose that in the course of the computation a goal of the form

```
?- x(f(X),X)
```

is attempted. This clearly succeeds. Furthermore an item of that form is added to the table. Later on it may be the case that a goal of the form

```
?- x(Y,Y)
```

is attempted. Clearly this is not a more specific goal as we solved before, so we need to solve this goal afresh. This succeeds too. Now we can continue by picking up a solution from the second table. However, if we pick the first solution then a cyclic term results.

A possible approach to deal with this situation is to index the items of the second table with the item of the first table from which the solution was obtained. In other words: you can select a solution from the second table not only if the solution matches your goal, but also if the corresponding goal of the solution is more general than your current goal. This strategy works, but turns out to be considerably slower than the original version given above. The reason seems to be that the size of the second table is increased quite drastically, because solutions may now be added to the table more than once (for all goals that could give rise to that solution).

It turns out that an improvement of the head-corner parser using a *goal weakening* technique often eliminates this occurs-check problem. Goal weakening is discussion in the following subsection.

9.3.3 Goal Weakening

The insight behind 'goal weakening' in the context of memoization is that we may combine a number of slightly different goals into a single more general goal. Very often it is much cheaper to solve this single (but more general) goal, than to solve each of the specific goals in turn. Clearly, one must be careful not to remove essential information from the goal (in the worst case this may even lead to non-termination of otherwise well-behaved programs).

Depending on the datastructures used in a particular grammar, it may for example be worthwile to *restrict* a given category to its syntactic features before we attempt to solve the parse goal of that category. Shieber's restriction operator (Shieber, 1985) can be used here. Thus we essentially throw some information away before an attempt is made to solve a (memoized) goal. For example, the category

```
x(A,B,f(A,B),g(A,h(B,i(C))))
```

may be weakened into:

```
x(A,B,f(_,_),g(_,_))
```

Note that goal weakening is sound. An answer a to a weakened goal g is only considered as an answer for g if a and g unify. Also note that goal-weakening is complete in the sense that for an answer a to a goal g there will always be an answer a' to the weakening of g such that a' subsumes a.

We want to stress that for practical implementations the use of goal weakening is very important. It is our experience that a well-chosen goal weakening operator may reduce parsing times with one or two orders of magnitude.

If we assume that the predicate `weaken/2` relates a predicate with a weakened version, then the following is the improved version of the memo predicate:

```
memo(Pred0) :-
    weaken(Pred0,Pred),
    (   in_table1(Pred)
    ->  true
    ;   (   call(Pred),
            assert_table2(Pred),
            fail
        ;   assert_table1(Pred)
    )   ),
    in_table2(Pred0).
```

The goal weakening technique can also be used to eliminate typical instances of the problems concerning the occurs-check (discussed in the previous subsection). Coming back to the example in the previous subsection, if our first goal

```
x(f(X),X)
```

were weakened into

```
x(f(_),_)
```

then the problem would not occur. If we want to guarantee that no cyclic structures can be formed then we would need to define goal-weakening in such a way that no variable sharing would occur in the weakened goal. For the current version of the grammar of OVIS2, it turned out that weakening the goal category in such a way that all information below a depth of 6 is replaced by fresh variables eliminated the problem.

9.4 Compact Representation of Parse Trees

9.4.1 Introduction

Traditionally a distinction is made between *recognition* and *parsing*. Recognition checks whether a given sentence can be generated by a grammar. Usually recognizers can be slightly adapted to be able to recover the possible parse-trees of that sentence (if any).

In the context of Definite Clause Grammar this distinction is often blurred because it is possible to build up the parse-tree as part of the complex non-terminal symbols. Thus the parse-tree of a sentence may be constructed as a side-effect of the recognition phase. If we are interested in logical forms rather than in parse trees a similar trick may be used. The result of this however is that already during recognition ambiguities will result in an explosion of processing time.

For this reason we will assume that parse trees are *not* built by the grammar, but rather are the responsibility of the parser. This allows the use of efficient *packing* techniques. The result of the parser will be a *parse forest*: a compact representation of all possible parse trees rather than an enumeration of all parse trees.

Semantic information is only added once parse trees are extracted from this parse forest. The motivation for this is that the use of semantic information during the recognition phase would minimize the number of situations in which packing applies.

9.4.2 History Items

The structure of the 'parse-forest' in the head-corner parser is rather unusual, and therefore we will take some time to explain it. Rather than a table of local trees, the head-corner parser maintains a table of (partial-) parse-trees which represent a succesful path from a lexical head up to a goal category. The table consisting of such partial parse-trees is called the history table; its items are history-items.

More specifically, each history-item is a triple consisting of a cky-item reference, a rule name and a list of triples. The rule name is always the name of a rule without daughters (i.e. a lexical entry or a gap): the (lexical) head. Each triple in the list of triples represents a local tree. It consists of the rule name, and two lists of cky-item references (representing the list of daughters to the left of the head in reverse, and the list of daughters right of the head). An example will clarify this. Suppose we have a history-item:

```
'HISTORY_ITEM'(112,give22,[rule(vp_v,[],[]),
                           rule(vp_np_pp,[],[121,125]),
                           rule(s_np_vp,[87],[]),
                           rule(s_adv_s,[46],[])
                          ]).
```

This item indicates that there is a possible derivation of the category defined in cky-item 112 of the form illustrated in figure 9.2. In this figure the labels of the nodes are either rule-names or references to cky-items.

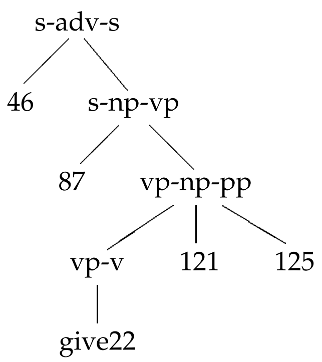


Figure 9.2: Example of a partial parse-tree projected by a history-item.

The following subsection explains how history-items can be constructed during recognition. We then show how history-items can be used to recover full parse-trees, if so desired.

9.4.3 Constructing history-items

Let us now look how the parser of the previous section can be adapted to be able to assert history-items. The basic idea is that for each item in the second table we keep track of the rules and items that were used to construct that item. Furthermore, each such item represents a path starting from a 'predict' clause up through the head-corner relation.

Firstly we add an (output-) argument to the `parse/5` predicate. This sixth argument is the reference to the cky-item that was actually used. The predicates to parse a list of daughters are augmented with a list of such references. This enables the construction of a term for each local tree in the head_corner predicate consisting of the name of the rule that was applied and the list of references of the cky-items used for the left and right daughters of that rule. Such a local tree representation is an element of a list that is maintained for each lexical head upward to its goal. Such a list thus represents in a bottom-up fashion all rules and cky-items that were used to show that that lexical entry was indeed a head-corner of the goal. If a parse goal has been solved then this list containing the history information is asserted in a new kind of table: the

'HISTORY_ITEM'/3 table.

A complication is caused by those cases where items are removed later because a more general item has been found. It may be the case that such items were referred to by history items. Therefore we should keep track of the fact that an item has been replaced by a more general one using a fourth (and final) kind of item: 'REPLACED_ITEM'/2.

The resulting program is given in the appendix.

9.4.4 The Recovery of Parse Trees and Logical Forms

The head-corner parser we have discussed above produces a compact representation of parse-trees. In this subsection we show how individual parse trees may be constructed on the basis of this representation.

Parse-trees can be recovered by the following non-deterministic procedure. We start by taking a reference of a cky-item that matches the top-category. For this reference we look-up its history-item. This history-item is a recipe indicating which rules need to be applied. For every cky-item reference in this history-item we find another possible history-item (taking the replaced-items into account). In Prolog we have:

```
parse(Cat,tree(Cat,Ds),P0,P) :-
    parse(Cat,P0,P,P0,P,Ref),
    recover_tree(Ref,Cat,tree(Cat,Ds)).

recover_tree(Ref0,Cat,tree(Cat,Ds)) :-
    up_to_date(Ref0,Ref),
    'HISTORY_ITEM'(Ref,HeadRef,His),
    recover_seed(HeadRef,Small),
    recover_tree_hc(His,Small,tree(Cat,Ds)).

recover_seed(lex(LexRef),tree(LexCat,[lex(Word)])) :-
    clause(lexical_analysis(_,_,LexCat,_,_,Word),_,LexRef).

recover_seed(gap(GapRef),tree(GapCat,[])) :-
    gap(Small,_,GapRef).

recover_tree_hc([],C,C).
recover_tree_hc([rule(Name,Lhis,Rhis)|His],tree(Small,SmallDs),Goal) :-
    headed_rule(Small,Mid,Lds,Rds,_,_,_,_,Name),
    recover_tree_lds(Lhis,Lds,[tree(Small,SmallDs)|Rtrees],Trees),
    recover_tree_rds(Rhis,Rds,Rtrees),
    recover_tree_hc(His,tree(Mid,Trees),Goal).

recover_tree_lds([],[],H,H).
recover_tree_lds([H|T],[Hd|Td],His0,His) :-
    recover_tree(H,Hd,Htree), recover_tree_lds(T,Td,[Htree|His0],His).

recover_tree_rds([],[],[]).
recover_tree_rds([H|T],[Hd|Td],[Htree|His]) :-
    recover_tree(H,Hd,Htree), recover_tree_rds(T,Td,His).

up_to_date(Ref0,Ref) :-
    ( 'REPLACED_ITEM'(Ref0,Ref1) -> up_to_date(Ref1,Ref) ; Ref0=Ref ).
```

The head-corner parser defined sofar exhibits 'packing' in the sense that if there are multiple ways to derive a category between two positions, then this category is used only once to derive larger constituents. The practical usefulness of this notion of packing is of course improved if two categories are often indistinguishable. For context-free grammars this is often the case. But for feature-based grammars of the kind assumed here, it may be the case that categories contain explicit or implicit information about their derivational history. In such cases packing will not be very useful, since two categories will almost never be identical.

We already argued above that parse-trees should not be explicitly defined inside the category. Logical forms can be seen as an implicit definition of the internal structure of a category. Therefore, the common use of logical forms as part of the categories will imply that you will hardly ever find two different analyses for a single category, because two different analyses will also have two different logical forms. Therefore, no packing is possible and the recognizer will behave as if it is enumerating all parse-trees.

The solution to this problem is to delay the evaluation of semantic constraints. During the first phase all constraints referring to logical forms are ignored. Only if a parse-tree is recovered from the parse-forest we add the logical form constraints. This is similar to the approach worked out in CLE (Alshawi, 1992).

This approach may lead to a situation in which the second phase actually filters out some otherwise possible derivations, in case the construction of logical forms is not *compositional* in the appropriate sense. In such cases the first phase may be said to be unsound in that it derives ungrammatical derivations. The first phase combined with the second phase is of course still sound. Furthermore, if this situation would arise very often, then the first phase would tend to be useless, and all work would have to be done during the recovery phase. The present architecture of the head-corner parser embodies the assumption that such cases are rare, and that the construction of logical forms is (grosso modo) compositional.

The distinction between semantic and syntactic information is compiled into the grammar rules on the basis of a user declaration. We simply assume that in the first phase the parser only refers to syntactic information, whereas in the second phase both syntactic and semantic information is taken into account. In the final version of the head-corner parser presented in the appendix, this distinction is implemented.

If we assume that the grammar constructs logical forms, then it is not clear that we are interested in parse-trees at all. A simplified version of the recover predicate may be defined in which we only recover the semantic information of the root category, but in which we don't build parse trees. The simplified version may be regarded as the run-time version, whereas parse-trees may still be very useful for grammar development.

9.5 Parsing Word-graphs with Probabilities

The head-corner parser is one of the parsers developed within the NWO Priority Programme on Language and Speech Technology. In this programme a spoken dialog system is developed for public transport information.

In this system the input for the parser is not a simple list of words, as we have assumed up to now, but rather a word-graph: a directed, acyclic graph where the states are points in time, and the edges are labelled with word hypotheses and their corresponding probability. This imposes two further complications. Firstly, we cannot use string indices anymore. Secondly we need to keep track of the probabilities of the words used in a certain derivation.

9.5.1 From string positions to state names.

Instead of using string indices we simply use the names of the states of the word-graph. Refer to van Noord (1995) for an example. The only predicate that needs to be changed is `between/4`. Rather than a simple integer comparison, we now need to check that a derivation from `P0` to `P` can be extended to a derivation from `E0` to `E` iff there are edges from `E0` to `P0` and from `P` to `E`.

A somewhat different approach is to use the ordinary comparison operator that we used in the original definition of the head-corner parser. We then assume that positions in the word-graph are integers such that transitions always connect to a larger integer (hence this technique does not generalize to finite-state automata in general). The possible extra cost of allowing impossible partial analyses is worthwile if the more precise check would be more expensive.

9.5.2 Accounting for Word-graph Probabilities.

In order to account for the probabilities of a derivation (simply defined as the product of the probabilites associated with all words from the word-graph involved in the derivation) we assume that the predicate `lexical_analysis/6` represents the probability of the piece of the word-graph that it covers in its fifth argument. During the first phase probabilities are ignored. During the second phase (when a particular derivation is constructed) the probabilities are combined. Again we refer to the final version of the head-corner parser in the appendix for the actual implementation.

Acknowledgements

Some of the introductory material of this report is taken from Bouma and van Noord (1993). Gosse Bouma and Mark-Jan Nederhof provided useful feedback.

9.6 Appendix: Current Version of the Head-corner Parser

The current version of the head-corner parser uses the following predicates. Note that we are still assuming that lexical lookup has already been performed. The variable names should be self-explanatory.

```
lex:lexical_analysis(FromState,ToState,SynCat,SemCat,Score,Words).
grammar:headed_rule(Head,MotherSyn,RevLds,Rds,RuleName).
grammar:headed_rule_i(RuleName,Head,Mother,RevLds,Rds),
grammar:head_corner_link(GoalCat,FromGoal,ToGoal,HeadCat,FromHead,ToHead).
grammar:gap(SynCat,RuleName).
grammar:gap_i(RuleName,Cat).
```

Now follows the definition of the current version of the head-corner parser.

9.6.1 Head-corner recognition

```
:- module(hc,[ parse/2 ]).

parse(Cat,Score) :-
    % first phase parsing:
    % Note: at this point Start and Final are not yet
    % instantiated. We could just as well start parsing
    % with parse(Cat,Start,Final,_,_,_,_,Ref)
    parse(Cat,Start,Final,Start,Final,l,r,Ref),
    wordgraph:start(Start),
    wordgraph:final(Final,Score0),
    % second phase parsing: it uses the history items built
    % by the first phase parser. These history items use
    % _references_ to result items, but these result items
    % themselves are not used anymore at this point.
    parse_2nd(Ref,Cat,Start,Final,Score1,_,_),
    multiply_scores(Score0,Score1,Score).
```

```
% parse(Cat,Begin,End,LeftExtreme,RightExtreme,L,R,Reference)
% is true if there is a derivation from Begin to End of Cat such
% that <Begin,End> is an interval within <LeftExtreme,RightExtreme>
% If L is ground then we also know that Begin==LeftExtreme;
% If R is ground then we also know that End==RightExtreme
%
% Note that we assume that as a side-effect of calling this predicate
% all solutions will be found for weakening(Cat), any P0, P. That is,
% in proving all solutions we ignore P0 and P, and we weaken Cat.
parse(Cat,P0,P,E0,E,LeftEq,RightEq,Ref) :-

    % get a more general copy, this reduces the number of different goals,
    % reduces the size of the tables and increases number of solutions
    % If you weaken too much, non-termination might occur. If you
    % weaken too little, the parser will be slower and memory consumption
    % will be higher. Also note that we first ignore P0 and P.
    weaken(Cat,Cat0),

    % Take into account information for each of the edges whether the
    % edge is identical to the extreme position. Since we don't use
    % P0,P directly, but a variant (PP0,PP) we unify this information in
    % explicitly. The reason for not using P0,P directly is that we
    % don't keep P0,P in the goal table (kind of goal weakening too).
    (  nonvar(LeftEq) -> PP0=E0 ; true ),
    (  nonvar(RightEq) -> PP=E ; true ),

    % constructs the form of the items that will be asserted.
    % Sole purpose is to obtain indexing. This really makes a difference.
    Cat0=..[F|Args],
    MemoCall =.. [F,E0,E,LeftEq,RightEq|Args],
    MemoItem =.. [F,PP0,PP|Args],

    % first check whether this goal has been attempted before:
    (   a_more_general(table:MemoCall)
    ->  % we already solved this
    true
    ;   % if not: find all solutions for MemoCall
        (
      % predict a lexical entry or gap that might be the
         % head-corner of this goal.
      predict(Cat0,PP0,PP,E0,E,Small,QL,QR,His0),

      % prove that the prediction was correct
      head_corner(Small,QL,QR,Cat0,PP0,PP,E0,E,His),

      % add the result to the result table, if it is new.
      % Currently we don't check for more specific items
      % (these could be thrown away if we update the history
      % table) since no more specific items will be
      % constructed anyway. If packing occurs, then the
      % categories are always (?) equivalent, hence a check
      % in one direction is enough. If this assumption is wrong
      % no harm is done, we only do some work extra.
      add_item(MemoItem,His0,His),

      fail       % fail to go back and find all solutions
    ;
```

```
      % at this point the result table is complete for the
      % current weakened goal. Therefore we assert the current goal in
      % the goal table.
      assertz(table:MemoCall)
)     % assert that this goal is solved now
    ),
    % all solutions have now been found for the weakened goal. At
    % this point we re-introduce the information in the original goal
    % and select an item from the result table
    Cat=Cat0,
    P=PP,
    P0=PP0,
    table:clause(MemoItem,_,Ref),

    % make sure the result is within bounds
    between(P0,P,E0,E,table).


add_item(Item,His0,His) :-
    add_item(table:Item,IRef),
    % we keep a separate table for the history information
    % so, for each item in the result table we know how we
    % built that item. For a single item there might be
    % various history items (packing).
    % Also note that a single history item represents
    % the path from a lexical head-corner up to the goal node,
    % with rule names on the one hand, and a reference of
    % a memo-item for each of the (non-head) daughters of
    % each of the rules.

    % we decompose the reference functor for indexing.
    % again, this makes a real difference
    IRef='$ref'(IRefA,IRefB),

    % we check whether MEMO_HIS is already there, because it might
    % be the case that the current item has already been found before
    % using the same rules, but on the basis of a more specific goal.
    (  table:'MEMO_HIS'(IRefA,IRefB,His0,His)
    -> true
    ;  table:assertz('MEMO_HIS'(IRefA,IRefB,His0,His))
    ).

% case 2. epsilon-rule head-corner. This case first since it gives
% rise to more general goals because of underspecification of position.
% Later more specific goals thus are subsumed by these..
predict(Cat,P0,P,_E0,_E,Small,Q,Q,gap(Name)) :-
    % again, use the table to limit the attention to possible
    % head-corner categories.
    grammar:head_corner_table(Cat,P0,P,Small,Q,Q),
    grammar:gap(Small,Name).

% predict(+Cat,?P0,?P,?E0,?E,-Small,-QL,-QR,-Ref)
% find a possible head-corner for Cat
% case 1. lexical head-corner
predict(Cat,P0,P,E0,E,Small,QL,QR,lex(Ref)) :-
    \+ E0==E,    % quick check to see whether there can be a lexical head-corner

    % use head-corner table to find a possible category, string
```

```
    % position pair.
    hc_in:head_corner_table(Cat,P0,P,Small,QL,QR),

    % lexical lookup is already performed beforehand; its results
    % are available thru the following unit clauses:
    lex:clause(lex_analysis(QL,QR,Small,_,_Score,_Name,_,_x),_,Ref),

    % check that head-corner is within bounds:
    between(QL,QR,E0,E,table).


% head_corner(Small,SmallP0,SmallP,Goal,GoalP0,GoalP,Extr0,Extr,His)
% case 1: we are done
head_corner(X,Y,Z,X,Y,Z,_,_,[]).
% case 2: find a rule whose head unifies with Small, parse the other
% daughters of the rule, prove that the mother node is a head-corner
% of Goal.
head_corner(Small,Q0,Q,Goal,P0,P,E0,E,[rule(Name,Lhis,Rhis)|His]) :-
    % select a rule:
    grammar:headed_rule(Small,Mid,RevLefties,Righties,Name),
    % make sure there is a link possible from Mother to Goal
    grammar:head_corner_table(Goal,P0,P,Mid,QL,QR),
    parse_left_ds(RevLefties,QL,Q0,E0,E,Lhis),
    parse_right_ds(Righties,Q,QR,E0,E,Rhis),
    head_corner(Mid,QL,QR,Goal,P0,P,E0,E,His).

% parse_left_ds(+Ds,?P0,?P,+E0,+E)
% parse a reversed list of daughters from position P0 to P
% where E0 is smaller than P0 and E larger than P (E0-E are
% the extremes within which the interval P0-P should be
% found).

% If P0 is instantiated (because we parse from a given head
% to the left) we should use P0 as
% the extreme right position. If P0 is not instantiated (in case
% of a gap) than we should use E0. Similar for P,E.
% Furthermore we also instantiate LeftEq (RightEq) if it
% is known that L0=E0 (L=E).
parse_left_ds([],L,L,_,_,[]).
parse_left_ds([H|T],L0,L,E0,E,[Ref|His]):-
    ledge(L0,E0,Left,LeftEq),
    redge(L,E,Right,RightEq),
    parse(H,L1,L,Left,Right,LeftEq,RightEq,Ref),
    parse_left_ds(T,L0,L1,E0,E,His).

% find most constraining information for edges
% ledge(LeftEdge,LeftExtremeIn,LeftExtremeOut,Eq)
% LeftExtremeOut will be the most informative of LeftEdge
% and LeftExtremeIn.
% Eq will be instantiated in case LeftEdge==LeftExtremeOut
ledge(L0,E0,Left,Eq) :-
    (  var(L0)
    -> Left=E0
    ;  var(E0)
    -> Left=L0, Eq=l
    ;  L0 < E0
    -> Left=E0
    ;  Left=L0, Eq=l
```

```
    ).

% similar to ledge, but vv
redge(L,E,Right,Eq) :-
    (  var(L)
    -> Right=E
    ;  var(E)
    -> Right=L, Eq=r
    ;  L < E
    -> Right=L, Eq=r
    ;  Right=E
    ).

% similar to parse_left_ds, but now we go from left to right,
% (and the list of daughters is not reversed, of couse).
parse_right_ds([],L,L,_,_,[]).
parse_right_ds([H|T],L0,L,E0,E,[Ref|His]):-
    ledge(L0,E0,Left,LeftEq),
    redge(L,E,Right,RightEq),
    parse(H,L0,L1,Left,Right,LeftEq,RightEq,Ref),
    parse_right_ds(T,L1,L,E0,E,His).
```

9.6.2 Head-corner reconstruction

```
:- module(hc_reconstruct, [ parse_2nd/7 ]).

% parse_2nd(+Ref,?Cat,+P0,+P,-Score,Words0,Words)
% second_phase variant of the parse predicate.
% check for existence is useful if the grammar has a lot of
% so called spurious ambiguities...  It is also quite expensive
% for sentences with many readings
% note that we could also delete categories that use
% a different path thru the graph, but have worse score...
% that's what we do now!
parse_2nd('$ref'(Refa0,Refb0),Cat,P0,P,Sc,RWds0,RWds) :-
    up_to_date(Refa0,Refb0,Refa,Refb),
    (  table:reconstructed(Refa,Refb)
    -> true
    ;  table:assertz(reconstructed(Refa,Refb)),
       (  table:'MEMO_HIS'(Refa,Refb,Seed,His),
          predict_2nd(Seed,Small,QL,QR,Sc00,SeedWds0,SeedWds),
          head_corner_2nd(His,Small,Cat0,QL,QR,Q0,Q,Sc00,Sc0,
                          SeedWds0,SeedWds,Wds0,Wds,Module),
          assertz_if_new(Refa,Refb,Cat0,Q0,Q,Sc0,Wds0,Wds,Module),
          fail
       ;  true
       )
    ),
    table:reconstructed(Refa,Refb,P0,P,Cat,Sc,RWds0,RWds).

% if a more general / better one exists, then use that one..
% if a more specific/worse one exists, delete these
assertz_if_new(Refa,Refb,Cat0,Q0,Q,Sc0,Wds0,Wds,Module) :-
    (  table:reconstructed(Refa,Refb,Q0,Q,Cat,Sc,_,_),
       subsumes_chk(Cat,Cat0),
       Sc =< Sc0
    -> true    % ignore this result. We already have better results
    ;  % this is a new result. Throw away all worse results;
```

```
        % and assert this one
        (  table:clause(reconstructed(Refa,Refb,Q0,Q,Cat,Sc,_,_),_,Cref),
           subsumes_chk(Cat0,Cat),
           Sc0 =< Sc,
           erase(Cref),
           fail
        ;  table:assertz(reconstructed(Refa,Refb,Q0,Q,Cat0,Sc0,Wds0,Wds)))
    ).

% second_phase variant of prediction
predict_2nd(lex(LexRef),Small,P0,P,Sc,Wds0,Wds) :-
    lex:clause(lex_analysis(P0,P,Small,Small,Sc,_,Wds1,_Trans),_,LexRef),
    (  atomic(Wds1)
    -> Wds0 = [Wds1|Wds]
    ;  lists:append(Wds1,Wds,Wds0)
    ).

predict_2nd(gap(Name),Small,Q,Q,0,Ws,Ws) :-
    hc_in:gap_i(Name,Small).

head_corner_2nd([],C,C,P0,P,P0,P,S,S,W0,W,W0,W,_).
head_corner_2nd([rule(Name,Lhis,Rhis)|His],Small,Goal,P1,P2,Q0,Q,
                                         S0,S,W1,W2,X0,X,Module) :-
    hc_in:headed_rule_i(Name,Small,Mid,Lds,Rds),
    parse_lds_2nd(Lhis,Lds,P0,P1,S0,S1,W0,W1,Module),
    parse_rds_2nd(Rhis,Rds,P2,P,S1,S2,W2,W,Module),
    head_corner_2nd(His,Mid,Goal,P0,P,Q0,Q,S2,S,W0,W,X0,X,Module).

parse_lds_2nd([],[],P,P,S,S,W,W,_).
parse_lds_2nd([H|T],[Hd|Td],P0,P,S0,S,W0,W,Module) :-
    parse_2nd(H,Hd,P1,P,S1,W1,W,Module),
    probabilities:multiply_scores(S0,S1,S2),
    parse_lds_2nd(T,Td,P0,P1,S2,S,W0,W1,Module).

parse_rds_2nd([],[],P,P,S,S,W,W,_).
parse_rds_2nd([H|T],[Hd|Td],P0,P,S0,S,W0,W,Module) :-
    parse_2nd(H,Hd,P0,P1,S1,W0,W1,Module),
    multiply_scores(S0,S1,S2),
    parse_rds_2nd(T,Td,P1,P,S2,S,W1,W,Module).

up_to_date(Module,Aref0,Bref0,Aref,Bref) :-
    (  table:'MEMO_REPLACED'(Aref0,Bref0,'$ref'(Aref1,Bref1))
    -> up_to_date(Module,Aref1,Bref1,Aref,Bref)
    ;  Aref0=Aref, Bref0=Bref
    ).
```

9.6.3 Item administration

```
:- module(item, [ a_more_specific/1,
                  a_more_specific/2,
                  a_more_general/1,
                  a_more_general/2,
                  add_item/2,
                  add_item_no_delete/2,
                  remove_more_specific/2
                ]).

a_more_specific(Module:Item) :-
```

```
    a_more_specific(Module:Item,_).

a_more_specific(Module:Item,Ref) :-
    copy_term(Item,Copy),
    Module:clause(Copy,_,Ref),      % quick check for unification
    Module:clause(Specific,_,Ref), % new copy for subsumption check
    subsumes_chk(Item,Specific).

a_more_general(Module:Item) :-
    a_more_general(Module:Item,_).

a_more_general(Module:Item,Ref):-
    copy_term(Item,Copy),
    Module:clause(Copy,_,Ref),
    Module:clause(General,_,Ref),
    subsumes_chk(General,Item).

add_item(Module:Item,Ref) :-
    (  a_more_general(Module:Item,Ref)
    -> true
    ;  findall(SpecRef,( a_more_specific(Module:Item,SpecRef),
         erase(SpecRef)
                      ), SpecRefs),
       Module:assertz(Item,Ref),
       assert_replaced(SpecRefs,Ref,Module)
    ).

assert_replaced([],_,_).
assert_replaced([Spec|T],Ref,Module):-
    Module:assertz('MEMO_REPLACED'(Spec,Ref)),
    assert_replaced(T,Ref,Module).

add_item_no_delete(Module:Item,Ref) :-
    (  a_more_general(Module:Item,Ref)
    -> true
    ;  Module:assertz(Item,Ref)
    ).

remove_more_specific(Module:Item,Ref) :-
    ( a_more_specific(Module:Item,Ref0),
      \+ Ref0=Ref,
      erase(Ref0),
      Ref0='$ref'(Ref0A,Ref0B),
      Module:assertz('MEMO_REPLACED'(Ref0A,Ref0B,Ref)),
      fail
    ; true
    ).
```

9.6.4 Interval administration

```
:- module(connection, [ between/5,
                        connection/3
                      ]).

% between(+L0,+R0,+L,+R)
between(L0,R0,L,R,Module) :-
    connection(L,L0,Module),
    connection(R0,R,Module).
```

```
connection(A,B,Module):-
    (  var(A)
    -> true
    ;  var(B)
    -> true
    ;  A=:=B
    -> true
    ;  B < A
    -> fail
    ;  Module:ok_connection(A,B)
    -> true
    ;  Module:fail_connection(A,B)
    -> fail
    ;  wordgraph:trans(A,_,X,_),
       connection(X,B,Module)
    -> Module:assertz(ok_connection(A,B))
    ;  Module:assertz(fail_connection(A,B)),
       fail
    ).
```

9.6.5 Goal weakening

```
:- module(weaken, [ weaken/2,
                    weaken_list/2
                  ]).

% the value 6 turns out to be good for the OVIS grammar
weaken(Term0,Term) :-
    restrict(Term0,6,Term).

weaken_list([],[]).
weaken_list([H|T],[H2|T2]) :-
    weaken(H,H2),
    weaken_list(T,T2).

restrict(Term,Depth,Restr):-
    integer(Depth),
    restrict(Term,0,Depth,Restr).

restrict(Term,D0,D,Restr):-
    (  D0=D
    -> true
    ;  (  var(Term)
       -> Term = Restr
       ;  functor(Term,F,A),
          functor(Restr,F,A),
          D1 is D0 + 1,
          restrict_args(0,A,Term,D1,D,Restr)
       )
    ).

restrict_args(P0,A,Term,D1,D,Restr):-
    (  P0=A
    -> true
    ;  P is P0 + 1,
       arg(P,Term,TP),
       arg(P,Restr,RP),
```

```
   restrict(TP,D1,D,RP),
   restrict_args(P,A,Term,D1,D,Restr)
).
```

Chapter 10

Preliminary Experiments

We have performed a number of experiments with the parsers presented in the previous chapters, on the basis of versions of the OVIS grammar. In this chapter we present some preliminary results of these experiments. These results should not be taken as conclusive but are only meant to give an idea of the current status of the project.

Although we have experimented with different test corpora we will give parse times here for the corpus of 500 wordgraphs that is available through the documentation system: `http://grid.let.rug.nl:4321/tst/graphs/`.

We have timed the performance of the parsers presented in the previous chapters. In the tables below these parsers are referred to by their system-internal names. These names correspond to:

hc9_mixtus Best version of a collection of different versions of the head-corner parser. This version is partially evaluated using the Mixtus partial-evalutation sytem (Sahlin, 1991).

act Simple version of the bottom-up Earley parser as described in chapter 7.

act2 The same as *act* but this version uses a different means of indexing. This parser is described in section 8.3.2.

bu Bottom-up Earley parser with packing and item-weakening; described in section 8.3.3.

inact_p Shift-reduce parser with Packing and Item-weakening. Described in section 8.3.4.

Note that the parse-times below include lexical lookup, and a preliminary version of the robustness and disambiguation component that is not described here. It should be kept in mind that these parse times were obtained with the current version of the Ovis grammar. This grammar is not finished yet; parse times for the full grammar might be slower than the times given here.

10.1 Parse times

The parse times are in milliseconds (cputime) on a fast HP 7000/935 machine running HP-UX 9.5. Total parse times are given in figure 10.2. Note that the hc9_mixtus parser incorporates a few optimizations that could be applied to the other parsers as well. It is expected that implementing the same optimizations for the other parsers reduces their parse times with 10 or 20%. Note that the cputime statistics are obtained with the SICStus built-in predicate `statistics/2`.

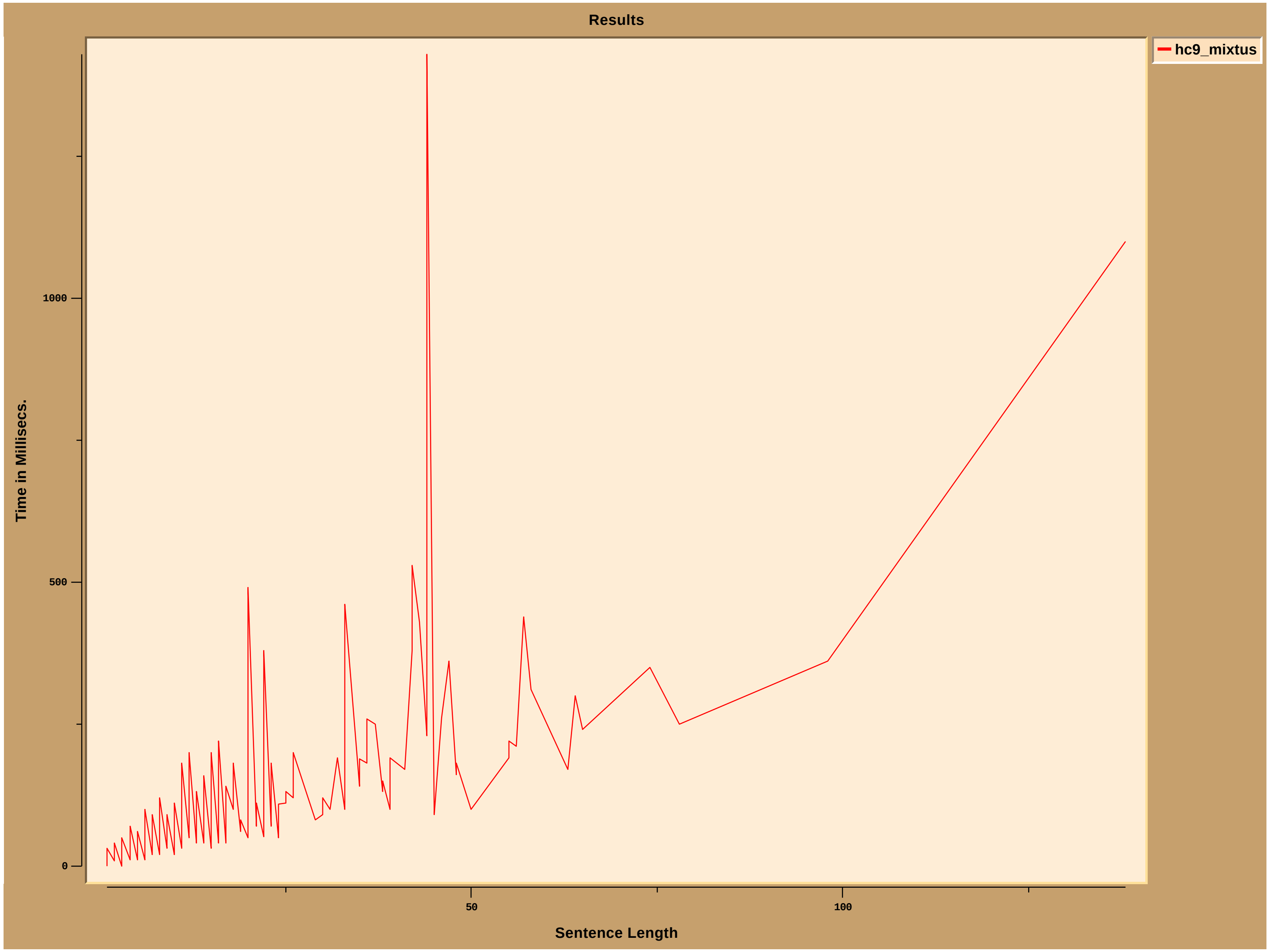


Figure 10.1: Parse-times in milliseconds versus the number of transitions in the wordgraph.

parser	total cputime (msec)	average per wordgraph	maximum
hc9_mixtus	27257	54	1320
inact_p	182661	365	26770
act2	235470	470	36660
bu	237116	474	66120
act	248411	496	47939

Figure 10.2: Cputime required for 500 wordgraphs (`http://grid.let.rug.nl:4321/tst/graphs/`).

We have made a graph (figure 10.1) of the parse-times of the hc9_mixtus parser with respect to the number of transitions (somewhat confusingly called sentence-length in the graph).

10.2 Memory Requirements

In figure 10.3 we give the total memory usage of each of the parsers. In this table we have summed the total size (in kilobytes) of the tables (charts) each of the parser constructs during a parse. These statistics are obtained using the SICStus built-in predicate `statistics` with first argument `program_space`. Therefore, these statistics indicate only the increase of the space used for the tables that are asserted in the database, but do not take into account the sizes of stacks during processing.

parser	total memory (Kbytes)	average per wordgraph	maximum
hc9_mixtus	5214	10	226
inact_p	23239	46	1577
act2	221013	442	37197
bu	52685	105	4765
act	174071	348	26358

Figure 10.3: Sum of the size of the tables (chart) for each parse. Sizes in Kilobytes.

References

Agnäs, M-S., H. Alshawi, I. Bretan, D. Carter, K. Ceder, M. Collins, R. Crouch, V. Digalakis, B. Ekholm, B. Gambäck, J. Kaja, J. Karlgren, B. Lyberg, P. Price, S. Pulman, M. Rayner, C. Samuelsson, and T. Svensson. 1994. Spoken Language Translator: First-year report. Technical report, SICS and SRI Cambridge, Cambridge Computer Research Centre. Available as crc043/paper.ps.Z from http://www.cam.sri.com/tr/.

Aït-Kaci, H. 1991. *Warren's abstract machine: a tutotial reconstruction*. MIT Press.

Alshawi, Hiyan, editor. 1992. *The Core Language Engine*. ACL-MIT press.

Alshawi, Hiyan, David Carter, Richard Crouch, Steve Pulman, Manny Rayner, and Arnold Smith. 1992. CLARE: A contextual reasoning and cooperative response framework for the Core Language Engine. Technical report, SRI International, Cambridge Computer Research Centre. Available as cmp-lg/9411002 from http://xxx.lanl.gov/cmp-lg.

Block, Hans Ulrich. 1994. Compiling trace & unification grammar. In Tomek Strzalkowski, editor, *Reversible Grammar in Natural Language Processing*. Kluwer Academic Publishers.

Bouma, Gosse and Gertjan van Noord. 1993. Head-driven parsing for lexicalist grammars: Experimental results. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht.

Boves, Lou, Jan Landsbergen, Remko Scha, and Gertjan van Noord. 1995. *Language and Speech Technology*. NWO Den Haag. Project plan for the NWO Priority Programme 'Language and Speech Technology'.

Cooper, Robin, Richard Crouch, Jan van Eijck, Chris Fox, Josef van Genabith, Jan Jaspars, Hans Kamp, Manfred Pinkal, Massimo Poesio, and Stephen Pulman. 1994. Fracas deliverable d8: Describing the approaches. Technical report, Centre For Congitive Science, Edinburgh. Available by ftp from ftp.cogsci.ed.ac.uk, pub/FRACAS/.

Davidson, Donald. 1967. The logical form of action sentences. In Nicholas Rescher, editor, *The Logic of Decision and Action*. University of Pittsburgh Press.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February.

Flickinger, Dan. 1995. Pieces of time: Toward the syntax of temporal expressions in English. Talk presented in Groningen, October, 31, 1995.

Frank, Annette. 1994. Verb second by lexical rule or by underspecification. Technical report, Institute for Computational Linguistics, Stuttgart.

Gazdar, Gerald, Ewan Klein, Geoffrey Pullum, and Ivan Sag. 1985. *Generalized Phrase Structure Grammar*. Blackwell.

Hukari, Thomas E. and Robert D. Levine. 1994. Adjunct extraction. ms. Univerity of Victoria and Ohio State Univerity.

Kay, Martin. 1989. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburg.

Lavelli, Alberto and Giorgio Satta. 1991. Bidirectional parsing of lexicalized tree adjoining grammar. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, Berlin.

Leermakers, R. 1992. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312.

Mellish, C.S. 1988. Implementing systemic classification by unification. *Computational Linguistics*, 14(1).

Nederhof, M.-J. 1994. Efficient generation of random sentences. Technical report csi-r9408, University of Nijmegen, Department of Computer Science, August.

Nederhof, Mark-Jan and Giorgio Satta. 1994. An extended theory of head-driven parsing. In *32th Annual Meeting of the Association for Computational Linguistics*, New Mexico State University.

Nederhof, M.J. 1994. *Linguistic Parsing and Program Transformations*. Ph.D. thesis, University of Nijmegen.

Nederhof, M.J. 1996. On the borderline between finite and infinite argument domains. In C.H.A. Koster and E. Oltmans, editors, *Proceedings of the first AGFL Workshop*, pages 17–43, University of Nijmegen, January.

Nerbonne, John. 1992. Constraint-based semantics. Technical Report RR-92-18, DFKI Saarbrücken.

Netter, Klaus. 1992. On non-head non-movement. In G. Görz, editor, *KONVENS 92*. Springer-Verlag.

van Noord, Gertjan. 1991. Head corner parsing for discontinuous constituency. In *29th Annual Meeting of the Association for Computational Linguistics*, Berkeley.

van Noord, Gertjan. 1993. *Reversibility in Natural Language Processing*. Ph.D. thesis, University of Utrecht.

van Noord, Gertjan. 1994. Head corner parsing for TAG. *Computational Intelligence*, 10(4).

van Noord, Gertjan. 1995. The intersection of finite state automata and definite clause grammars. In *33th Annual Meeting of the Association for Computational Linguistics*, MIT Boston.

van Noord, Gertjan and Gosse Bouma. 1995. Dutch verb clustering without verb clusters. submitted.

Oerder, Martin and Hermann Ney. 1993. Word graphs: An efficient interface between continuous-speech recognition and language understanding. In *ICASSP Volume 2*, pages 119–122.

Pereira, F.C.N. and D.H.D. Warren. 1980. Definite clause grammars for language analysis—a survey of the formalism and a comparison with the augmented transition networks. *Artificial Intelligence*, 13:231–278.

Pinkal, Manfred. 1995. *Logic and Lexicon*. Kluwer.

Pollard, Carl and Ivan Sag. 1994. *Head-driven Phrase Structure Grammar*. University of Chicago / CSLI.

Rosenkrantz, D.J. and P.M. Lewis-II. 1970. Deterministic left corner parsing. In *IEEE Conference of the 11th Annual Symposium on Switching and Automata Theory*, pages 139–152.

Sag, Ivan. 1995. English relative clause constructions. Draft, Stanford University, October, 1995.

Sahlin, Dan. 1991. *An Automatic Partial Evaluator for Full Prolog*. Ph.D. thesis, The Royal Institute of Technology (KTH) Stockholm, Sweden. SICS Dissertation Series 04.

Satta, Giorgio and Oliviero Stock. 1989. Head-driven bidirectional parsing. a tabular method. In *Proceedings of the Workshop on Parsing Technologies*, pages 43–51, Pittsburg.

Shieber, Stuart M. 1985. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23th Annual Meeting of the Association for Computational Linguistics*, Chicago.

Sikkel, Klaas. 1993. *Parsing Schemata*. Ph.D. thesis, Twente University, Enschede.

Sikkel, Klaas and Rieks op den Akker. 1992. Head-corner chart parsing. In *Proceedings Computer Science in the Netherlands (CSN '92)*, Utrecht.

Sikkel, Klaas and Rieks op den Akker. 1993. Predictive head-corner chart parsing. In *IWPT 3, Third International Workshop on Parsing Technologies*, pages 267–276, Tilburg/Durbuy.

Veldhuijzen van Zanten, Gert. 1995. The interface between the natural-language interpreter and the dialogue manager in OVIS2. Priority Programme Language and Speech Technology, document number 15. Not publicly available.

Warren, David S. 1992. Memoing for logic programs. *Communications of the ACM*, 35(3):94–111.