Grammar-based Natural Language Understanding

Gertjan van Noord

May 8, 2003

1 Natural Language Understanding

This chapter describes the *grammar-based* natural language understanding (NLU) component of the OVIS system. The NLU component receives its input from the speech recognizer and passes the result of linguistic analysis on to the pragmatic interpretation component. The output of the speech recognizer is a *word graph*, which represents all different hypotheses for the spoken input from the user. Linguistic analysis of such a structure is more complicated than analysis for a single input sentence (cf. linguistic analysis for *written* input), as we will show later. The interface between the NLU component and the Dialogue Manager is defined by a special interface language, called the *Update Language*, described elsewhere.

In the design of the NLU component, the following problems must be addressed.

Firstly, a NLU component will be faced with *ambiguity*. The combination of lexical and structural ambiguities often leads to an enormous amount of possible readings for an input utterance. In OVIS this problem is even more acute because of the use of word graphs. Techniques must be available to be able to deal with large numbers of analyses, and to be able to choose the most appropriate reading from such a set of candidate analyses.

While the system must somehow deal with large numbers of analyses, many of which may eventually turn out to be useless, another requirement is that the system should be *robust*, i.e. it should do something useful even if something happens that the system didn't expect, or if something goes wrong. For example, the input utterance may contain disfluencies or ungrammaticalities. Or the speech recognition component may have failed to recognize part of the user utterance. In such cases, a linguistic analysis component might proceed by computing a partial analysis of the input.

NLU needs to be robust for three reasons. Firstly, it is quite difficult to anticipate in the grammar all linguistic constructions that might occur. This is one of the traditional problems for grammar-based NLU. Secondly, spoken language is full of hesitations, corrections, false starts etc. which are not always easy to detect. The third reason is that the utterance that was actually spoken is not guaranteed to be a path in the word graph, due to limitations of state-of-the-art speech recognition, as well as external factors such as background noise, etc.

These observations indicate that *robustness* and *disambiguation* are two very important problems to be solved in the NLU component. Given the nature of the proposed application, it will be clear that the system is supposed to run in *real-time*, i.e. it is not supposed to leave the user waiting for the requested information. Therefore a requirement of *efficiency* provides a further challenge for the NLU component.

2 Grammar-based NLU

In order to meet the requirements the grammar-based NLU component consists of a number of modules. Firstly, a computational grammar of Dutch is defined. This grammar (described in section 3) defines the relation between sequences of words in the input and meaning representations in the output. The grammar is fully declarative, and therefore it has been possible to implement a number of different parsing algorithms (section 4). Such parsing algorithms compute for a given sequence of words the corresponding meaning representation according to the rules of the grammar. The parsers have been extended to accept word graphs as their input.

The meaning representations defined by the grammar are translated into *update expressions* by means of a translation component which is tailored to the application (section 5).

These components provide input for the *robustness and disambiguation* component. This component is responsible for determining the most likely analysis of the input, or parts of the input; the component is described in section 6.

3 A computational grammar for Dutch

In developing the grammar we combined the short-term goal of developing a grammar which meets the requirements imposed by the application (i.e. robust analysis of the output of the speech recogniser, extensive coverage of locative phrases and temporal expressions, and the construction of semantic representations) with the long-term goal of developing a general, computational, grammar which covers the major constructions of Dutch.

The design and organisation of the grammar, as well as many aspects of the particular grammatical analyses we propose, are based on Head-driven Phrase Structure Grammar [21]. We depart from this formalism mostly for computational reasons. As is explained below, the grammar is compiled into a restricted kind of definite clause grammar for which efficient analysis is feasible. The semantic component follows the approach to monotonic semantic interpretation using simplified *quasi-logical forms* presented originally in Alshawi [1].

The grammar covers the majority of verbal subcategorisation types (intransitives, transitives, verbs selecting a PP, and modal and auxiliary verbs), NP-syntax (including pre- and post-nominal 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. Due to restrictions imposed by the speech recogniser, the lexicon is relatively small (4000 word forms, many of which are names of stations and cities).

3.1 Formalism

The grammar for OVIS 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.

The OVIS-grammar formalism is essentially equivalent to Definite Clause Grammar (DCG) [19]. The choice for DCG is motivated by the fact that this formalism provides a balance between computational efficiency and linguistic expressiveness, and the fact that it is closely related to constraint-based grammar formalisms, such as HPSG, Categorial Unification Grammar, and Lexical Functional Grammar. Another important reason to choose DCG instead of a more restricted formalism such as context-free grammar, is the fact that DCG allows the integration of syntax and semantics that is standard in constraint-based formalisms such as HPSG.

Grammar rules consist of a context-free skeleton to which feature-constraints are added. The context-free skeleton is important, as it ensures a reasonable level of processing efficiency and facilitates experimentation with different parsing techniques.

The central formal operation in constraint-based grammar formalisms is unification of (typed or untyped) feature-structures [25]. The OVIS-formalism employs typed feature-structures in the definition of rules as well as lexical entries. During the construction of the parser, feature-structures are translated into Prolog terms. Because of this translation step, parsing can make use of Prolog's built-in term-unification, instead of the more expensive feature-unification. Similar formalisms have been successfully used a number of times before [7, 1, 4, 22]

The grammar consists of a set of grammar rules, and a set of lexical entries. In both cases, we make use of inheritance of constraints to express generalizations.

Grammar rules. A grammar rule is defined by a ternary predicate, `rule/3`. The first argument of this predicate is a ground Prolog term indicating the rule identifier. The second argument of the rule is the mother *category*. Categories are non-variable Prolog terms. The third argument of the rule is a (possibly empty) list of categories. Note that we require that the length of the list is given, and that none of the categories appearing in the list is a variable. An example of a grammar rule is provided:

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

Terminal symbols cannot be introduced in rules directly, but are introduced by means of lexical entries.

Lexical entries. The lexicon is defined by the predicate `lex/2`. As an example, the lexical entry 'sleeps' could be encoded as:

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

The first argument is the terminal symbol (the word) introduced by this lexical category. The second argument is the category associated with this word. In cases where a lexical entry introduces a sequence of terminal symbols (multi-word unit) the first argument is also allowed to be a (non-empty) list of atoms.

Top category. The top category for the grammar (or *start symbol*) is defined by the unary predicate `top_category`. Its argument is a category.

Feature constraints. Almost all work in computational grammar writing uses ‘feature-structures’ of some sort. It is fairly standard to compile (descriptions of) such features-structures into first-order terms (see [22] for a recent overview). We use the HDRUG development platform [34], which contains a library for compiling feature constraints into Prolog terms, and various predicates to visualise such Prolog terms as feature structures in matrix notation.

The most important operators provided by the HDRUG library are the type assignment operator ('`=>`'), the path equality operator ('`<=>`'), and the path operator ('`:`'). The categories that are used in the grammar are all defined through these operators. A small grammar fragment employing those operators is:

```
rule(1,S,[Np,Vp]) :-
    S => s, np(Np), vp(Vp),
    Vp:vform => finite,
    subj_agreement(Vp,Np).

np(Np) :- Np => np, Np:lex => -.
vp(Vp) :- Vp => v,  Vp:lex => -.
subj_agreement(Vp,Np) :- Vp:agr <=> Np:agr.
```

In this rule, the constraint `Np:lex => -` indicates that the value of the `lex` attribute of `Np` is of type `-`. The constraint `Vp:agr <=> Np:agr` indicates that the value of the `agr` attribute of `Vp` is identical to the value of the `agr` attribute of `Np`. Internally, such a rule could be represented as follows (the actual result of the compilation depends on what attributes are allowed for what types; declarations of this sort are part of the grammar):

```
rule(1,s,[np(Agr,-),v(Agr,-,finite,_,_)]).
```

To the grammar writer, such rules are displayed in matrix notation, as follows:

$$\texttt{rule}(1, s, \langle \begin{bmatrix} np & \\ \mathsf{agr} & \boxed{1} \\ \mathsf{lex} & \text{-} \end{bmatrix}, \begin{bmatrix} v & \\ \mathsf{agr} & \boxed{1} \\ \mathsf{lex} & \text{-} \\ \mathsf{vform} & \mathsf{finite} \end{bmatrix} \rangle).$$

The feature library also supports boolean combinations of atomic values; these are compiled into Prolog terms using a technique described in [15] (who attributes it to Colmerauer) and [22]. Thus, we may specify `agr` values such as `sg` $\wedge$ `(sec` $\vee$ `thi)`, denoting an agreement value which is singular and either second or third person.

We have also found it useful to provide the predicates `unify_ifdef/3` and `ifdef/4`. The predicate `unify_ifdef(C1,C2,Att)` can be used to require that if both `C1` and `C2` can have the attribute `Att` (i.e. `C1, C2` are of a type for which `Att` is a possible feature), then the values `C1:Att` and `C2:Att` must be identical. The predicate `ifdef(Att,Cat,Val,Otherwise)` is used to require that `Cat:Att` is identical to `Val` if `Att` is an appropriate feature for `Cat`. Otherwise `Val` is identical to `Otherwise`.

These predicates allow very compact and simple data-structures. For instance, whereas in HPSG the head-feature principle is defined by refering to a special feature HEAD, we define a variant of the head-feature principle which lists the features that are required to percolate between heads directly:

```
head_feature_principle(Head,Mother) :-
    unify_ifdef(Head,Mother,vform),
    unify_ifdef(Head,Mother,agr),
    unify_ifdef(Head,Mother,case),
    ...
```

This has the advantage that categories which do not have a particular feature can simply do without that feature. Moreover, no distinct 'head' attribute needs to be defined in feature-structures. The `ifdef/4` constraint can be used in a similar way. The valence principle (defined below) needs to ensure that, in a rule, the subcat list of the mother is the concatenation of the subcat list of the head and the daughters. Rather than providing all categories with such a subcat feature, instead the principle assumes that if the feature is not defined for a given category, its value is assumed to be the empty list. As a consequence, only a few categories specify a subcat feature, whereas the valence principle still applies globally.

3.2 Signs

Each word or phrase in the grammar is associated with a feature-structure, in which syntactic and semantic information is bundled. Within Head-driven Phrase Structure Grammar (HPSG), such feature-structures are called *signs*, a terminology which we will follow here. The grammar makes use of some 15 different types of sign, 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, case and semantics, respectively.

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 (SUBCATEGORISATION) (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 subcategorisation properties of a given sign.

The feature SLASH is present on *v, ques* and *sbar*. Its value is a list of signs. It is used to implement a (restricted) version of the account of non-local dependencies proposed in [21] and [23]. The value of SLASH is the list of signs which are 'missing' from a given constituent. Such a 'missing' element is typically connected to a preposed element in a topicalisation sentence or WH-question. The same mechanism can also be used 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 analyse 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 orthographic representation of the sentence). This analysis has the advantage that all other rules for verb phrases need not distinguish between main clauses and subordinate clauses.

The feature SEM is present on all signs. It is used to encode the semantics of a word or phrase. Such a semantic representation is a simplified *quasi logical form* [1], called *predicate form*. A predicate form consists of a predicate name (an arbitrary first-order term), a list of predicate forms representing the arguments of the predicate, and a list of semantic representations representing the modifiers of the predicate. For instance, the sentence

(1) ik wil om tien uur naar assen
I want at ten o'clock to assen
I want to go to assen at ten

is assigned the predicate form:

$$\left[\begin{array}{ll}
\text{pred} & \text{willen} \\
\text{args} & \left\langle \left[\begin{array}{ll} \text{pred} & \text{ik} \\ \text{args} & \langle\rangle \\ \text{adjs} & \langle\rangle \end{array}\right] \right\rangle \\
\text{adjs} & \left\langle \begin{array}{l}
\left[\begin{array}{ll} \text{pred} & \text{om} \\ \text{args} & \left\langle \left[\begin{array}{ll} \text{pred} & \text{hour(10)} \\ \text{args} & \langle\rangle \\ \text{adjs} & \langle\rangle \end{array}\right] \right\rangle \\ \text{adjs} & \langle\rangle \end{array}\right], \\
\left[\begin{array}{ll} \text{pred} & \text{naar} \\ \text{args} & \left\langle \left[\begin{array}{ll} \text{pred} & \text{place(town(assen))} \\ \text{args} & \langle\rangle \\ \text{adjs} & \langle\rangle \end{array}\right] \right\rangle \\ \text{adjs} & \langle\rangle \end{array}\right]
\end{array} \right\rangle
\end{array}\right]$$

An important restriction imposed by the formalism is that each rule must specify the category of its mother and daughters. A consequence of this requirement is that general rule-schemata as in HPSG cannot be used. 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 cannot be implemented directly, as it leaves the categories of the daughters and mother unspecified. Nevertheless, generalisations 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*.

Rules normally introduce a structure in which one of the daughters can be identified as the *head*. The head daughter either subcategorises 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. In figure 1 we list the definition for the head-complement structure

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

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

valence_principle(Head,Complements,Mother) :-
    ifdef(sc,Head,HeadSc,[]), ifdef(sc,Mother,MotherSc,[]),
    append(Complements,MotherSc,HeadSc)
```

Figure 1: Structures and Principles

and the principles it refers to, except for the *filler* principle, which is presented later.

The *head-complement* structure is an instance of a *headed* structure. The definition of *headed* structure refers to the HEAD-FEATURE, VALENCE, and FILLER principles, 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 and possibly a modifier.

The structures defined in figure 1 are used in the definition of grammar rules. The `np-det-n` rule introduces a *head-complement* structure in which (following traditional semantic analysis) the determiner is the head, and the noun the complement:

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

The `n-adj-n` rules introduces a *head-modifier* structure where the adjective is the modifier:

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

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 straightforwardly.

3.3 The lexicon

The lexicon is a list of clauses, associating a word (or sequence of words) with a specific sign. Constraint-based grammars tend to store lots of grammatical information in the lexicon. 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 for which quite detailed constraints may

```
intransitive(Pred,Sign) :- iv(Sign), iv_sem(Sign,Pred).

transitive(Pred,Sign) :-   tv(Sign), tv_sem(Sign,Pred).

v(V) :- V => v, V:lex => basic,
    V:vslash => [], V:subj <=> [Subj],
    Subj => np, Subj:nform => norm.

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

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

Figure 2: Fragment of the lexical hierarchy

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 2, 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 normal (non-locative and non-temporal) NP as subject. This is expressed by the predicate `v(V)`. Intransitive verbs (`iv(V)`) can now be characterised syntactically as verbs which do not subcategorise for any (non-subject) complements. Transitive verbs (`tv(V)`) subcategorise 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. A limited form of non-monotonic inheritance is supported.

An extended description of the coverage of the grammar is given in [36].

4 Parsing

We first describe the complications which arise in order to generalise parsing strategies to allow word graphs as input. A more extended version of this material is presented elsewhere [29]. We then continue to describe in somewhat more detail the actual parsing strategy used in the system; again a more elaborated presentation is available elsewhere [31].

4.1 Parsing Word Graphs

Pause-transition Elimination The input to the parser consists of word graphs produced by the speech recogniser ([17]). A word graph is a compact representation for all sequences of words that the speech recogniser hypothesises for a spoken utterance. The states of the graph represent points in time, and a transition between two states represents a word that may have been uttered between the corresponding points in time. Each transition is associated with an *acoustic score* representing a measure of confidence that the word perceived there was actually uttered. These scores are typically derived from negative logarithms of probabilities and therefore require addition as opposed to multiplication in order to combine two scores.

A word graph is a weighted connected directed acyclic graph $G = \langle \Sigma, V, v_s, T, F \rangle$ where Σ is a set of words, V is a set of states (vertices), $v_s \in V$ is the start state, T is a set of transitions and F is a set of final states. Transitions are tuples $\mathit{trans}(v_i, v_j, w, a)$ for a transition from v_i to v_j ($v_i, v_j \in V$) with label $w \in \Sigma$ and acoustic score a. Finals are tuples $\mathit{final}(v_i, a)$ where $v_i \in V$ indicating that v_i is a final state with associated acoustic score a.

At an early stage, so-called *pause transitions* are eliminited from the word-graph. Such transitions represent periods of time for which the speech recogniser hypothesises that no words are uttered. Depending on the details of the grammar, filled pause transitions and noise transitions can be treated as pauses as well. The motivation for pause transition elimination is that the grammar will not anticipate such transitions. The elimination might also result in graphs with a smaller number of states (but typically with a larger number of transitions). Finally, pause transition eliminiation turns out to be more efficient than extending lexical lookup in a way which would extend lexical entries to include any neighbouring pause transitions. Pause-transition elimination is similar to ϵ-removal for weighted finite automata.

For a given input word graph $G = \langle \Sigma, V, v_s, T, F \rangle$ and a set of pause labels Σ_{pause}, pause transition elimination is the word graph $G' = \langle \Sigma - \Sigma_{pause}, V, v_s, T', F' \rangle$ such that:

1. $T' = \{\mathit{trans}(v_i, v_k, w, a + \mathrm{pause}(v_i, v_j)) | \mathit{trans}(v_j, v_k, w, a) \in T\}$
2. $F' = \{\mathit{final}(v_i, a + \mathrm{pause}(v_i, v_f)) | final(v_f, a) \in F\}$

For each two states $v_i, v_j \in V$ the function $\mathrm{pause}(v_i, v_j)$ is the score of the best path from v_i to v_j using pause transitions only. The computation of this function reduces to a well-known problem for graphs, namely the all-pairs shortest path problem for directed acyclic graphs (the graph made up of all pause transitions).

Note that the resulting word graph could contain non-start states without any incoming transitions. These states can be removed (as well as any transitions leaving these states). From now on, we will assume word graphs are pause eliminated. After pause transition elimination, the word graph contains exactly one start state and one or more final states, associated with a score, representing a measure of confidence that the utterance ends at that point.

Parsing as Intersection. Ordinarily, the input for parsing is a sequence of symbols (words). In the case of word graphs the input is a compact representation of a finite number of such sequences of words.[1] Thus a naive solution would be to parse each of these sequences in turn. Such an approach, however, is impracticable because the number of paths in a given word graph can be enormous. It turns out that it is possible to generalize parsing algorithms in such a way that the compact word graph representation can be used directly as input. The generalization is very similar to algorithms which compute the *intersection* of context-free grammars (CFG) and *finite state automata* (FSA).

[1]For the moment we ignore the complication introduced by the acoustic scores.

It can be shown that the computation of the intersection of a FSA and a CFG requires only a minimal generalization of existing parsing algorithms. We simply replace the usual string positions with the names of the states in the finite state automaton (this is explained in more detail below). It is also straightforward to show that the complexity of this process is cubic in the number of states of the finite state automaton (in the case of ordinary parsing the number of states equals $n+1$ where n is the number of words of the input sentence) ([13, 3]).

Let a finite-state machine M be specified by a tuple $(Q, \Sigma, \delta, Q^S, Q^F)$ where Q is a finite set of states, Σ is a finite alphabet, δ is a function $Q \times \Sigma \rightarrow 2^Q$. Furthermore, $Q^S \subseteq Q$ is the set of start states, and $Q^F \subseteq Q$ is the set of final states. [2] The language defined by such an automaton is defined in the normal way (e.g. as in [10]).

Following [2], a CFG defining the intersection of a given CFG and a FSA is constructed by keeping track of the state names in the non-terminal category symbols in the new CFG. For each rule $X_0 \rightarrow X_1 \ldots X_n$ there are rules $\langle X_0 q_0 q\rangle \rightarrow \langle X_1 q_0 q_1\rangle\langle X_2 q_1 q_2\rangle \ldots \langle X_n q_{n-1} q\rangle$, for all $q \in Q$. Furthermore for each $q_k \in \delta(q_i, \sigma)$ we have a rule $\langle \sigma q_i q_k\rangle \rightarrow \sigma$. Finally, there are rules rewriting the start symbol S into tuples $\langle S\ q_s\ q_f\rangle$ for each $q_s \in Q^S, q_f \in Q^F$. The intersection grammar not only defines exactly the strings in the intersection, but it also defines the same parse-trees for each of these sentences (ignoring the position markers in the category names).

Although this construction shows that the intersection of a FSA and a CFG is itself a CFG, it is not of practical interest. The reason is that this construction typically yields an enormous amount of rules with *useless* symbols ([10]), i.e. symbols which are not derivable from the start state, and/or from which no sequence of terminal symbols is derivable. In fact the (possibly enormously large) parse forest grammar might define an empty language (if the intersection was empty). Luckily 'ordinary' parsers for CFG can be generalized to construct this intersection yielding (in typical cases) a much smaller grammar. Pure bottom-up parsers ensure that the symbols in the parse forest grammar do indeed derive a number of terminal symbols; pure top-down parsers ensure that the symbols in the parse forest grammar are derivable from the start symbol. For some parsers (e.g. a straightforward generalization of the parser described in [9]) it can be shown that the parse forest grammar will never contain useless symbols. Checking whether the intersection is empty or not then reduces to the question whether the parse forest grammar contains rules or not.

As an example, consider in figure 3 the simple bottom-up inactive chart parser presented abstractly in the style of [26] and [28]. The parser maintains items of the form $[p_i, X, p_j]$ indicating that a category X has been shown to exist between position p_i and p_j. The inference rules indicate how new items can be constructed from existing items. Side-effects are attached to these inference rules which assert the existence of rules in the parse forest grammar. The intersection is non-empty if and only if the parse forest grammar contains a rule rewriting the start symbol.

In figure 4 this parser is generalized to take a FSA as its input. Instead of string positions, the items now keep track of the state names of the FSA. Other parsing algorithms can be generalized in the same manner.

[2] For simplicitly we do not allow ϵ-moves in the finite state automaton, although the construction could easily be extended to treat those.

Scan $\dfrac{}{[p_i, w, p_{i+1}]}$

condition: w is the $i+1$th word of the string
side-effect: $\langle w\ p_i\ p_{i+1}\rangle \rightarrow w$

Complete $\dfrac{[p_k, X_1, p_{k'}][p_{k'}, X_2, p_{k''}]\ldots[p_{m'}, X_l, p_m]}{[p_k, X_0, p_m]}$

condition: $X_0 \rightarrow X_1, X_2 \ldots X_l$
side-effect: $\langle X_0\ p_k\ p_m\rangle \rightarrow \langle X_1\ p_k\ p_{k'}\rangle\langle X_2\ p_{k'}\ p_{k''}\rangle\ldots\langle X_l\ p_{m'}\ p_m\rangle$

Finish $\dfrac{[p_0, S, p_n]}{}$

condition: n is the length of the string
side-effect: $S \rightarrow \langle S\ 0\ n\rangle$

Figure 3: Specification of a simple bottom-up inactive chart parser. The parse forest grammar is constructed as a side-effect. It contains derivations if and only if a rule rewriting the start symbol has been constructed.

Admittedly, some differences between parsing of strings and parsing of FSA are ignored because of the abstract presentation chosen here. For instance, the parser in figure 3 can be implemented under a left-to-right processing regime in such a way that for each item i that is constructed it is only neccessary to consider the completion inference rule using i as the right-most antecedent item (such a simplication is possible if the grammar does not contain ϵ productions). In constrast, for the parser in figure 4 an item i could trigger the completion inference rule as any one of the antencedent items.

The same generalization applies to existing parsers for (off-line parsable) Definite Clause Grammars. In [30, 29] we showed that the intersection of off-line parsable DCG and FSA is undecidable. However, in the case of word-graphs, it is easy to verify that the question whether the intersection of an acyclic FSA and an off-line parsable DCG is empty or not is decidable since it reduces to checking whether the DCG derives any one of a finite number of strings. A practical implementation will not use this method, but instead will use a generalized DCG parser along the lines of the generalizations discussed here.

4.2 Efficient Parsing

Although we have chosen to illustrate the generalization required to parse word graphs as opposed to strings using an item-based presentation in the tradition of [28, 26], the actual parser that is employed in the current version of the OVIS system is a left/head-corner parser in the logic programming tradition, heavily influenced by [14, 18, 12, 27]. Such a parser can be seen as a bottom-up parser with top-down filtering. This parser is described in full detail elsewhere [31].

Careful evaluations on actual word graphs have shown that this left/head-corner parser performs much better than any of the chart-based implementa-

Scan	$\overline{[q_i, w, q_j]}$
condition:	$q_j \in \delta(q_i, w)$
side-effect:	$\langle w\ q_i\ q_j \rangle \rightarrow w$
Complete	$\dfrac{[q_k, X_1, q_{k'}][q_{k'}, X_2, q_{k''}] \ldots [q_{m'}, X_l, q_m]}{[q_k, X_0, q_m]}$
condition:	$X_0 \rightarrow X_1, X_2 \ldots X_l$
side-effect:	$\langle X_0\ q_k\ q_m \rangle \rightarrow \langle X_1\ q_k\ q_{k'} \rangle \langle X_2\ q_{k'}\ q_{k''} \rangle \ldots \langle X_l\ q_{m'}\ q_m \rangle$
Finish	$\underline{[q_s, S, q_f]}$
condition:	$q_f \in Q^F, q_s \in Q^S$
side-effect:	$S \rightarrow \langle S\ q_s\ q_f \rangle$

Figure 4: Specification of a simple bottom-up inactive chart parser, generalized to compute the intersection of a FSA and a CFG. As before, the parse forest grammar is constructed as a side-effect and it contains derivations if and only if a rule rewriting the start symbol has been constructed.

tions.[3] These competing implementations included active and inactive chart parsers, and LR parsers. The implementations are described in full detail in [37], [35], [16].

The characteristics of the left/head-corner parsers are described as follows.

Selective Memoization. In a chart parser for context free grammars, each computation step is performed only once. Moreover the administration involved in adding categories to the chart, and looking up categories in the chart is guaranteed to be efficient, because categories are atomic (and hence simple hashing techniques are applicable to ensure that chart items can be consulted in constant time). In feature-based formalisms, categories can be very complex. Searching whether a particular category already exists in the chart therefore is computationally expensive. For this reason, we have found it more practical to allow during parsing that certain computations are performed repeatedly. Thus, a selective use is made of memoization in such a way that only relatively large chunks of computation are memoized (i.e. performed only once). The computation of each such chunk does involve a depth-first backtracking search (because is readily available in a Prolog environment, and does not require much memory).

In our implementation of the head-corner parser, each non-head daughter in a rule relates to a parse goal. Each such non-head parse goal is memoized. Assuming traditional syntactic structures, this has the effect that only *maximal projections* such as *np, pp, s* are added to the chart; whereas non-maximal projections such as unsaturated verb phrases are not. As a result, the costs associated with the adminstration of the chart are reduced considerably.

[3] For the current version of the grammar, it hardly makes a differences whether we use the left-corner variant or the head-corner variant.

Goal Weakening. Another important technique which is applied for efficiency reasons is *goal weakening* (or *abstraction* [11]; or *restriction* [24]). Although top-down filtering in a bottom-up parser is crucial for efficiency, it can also have an undesirable effect: the same parse goal might have to be re-computed simply because the top-down expectations are slightly different. For instance, a rule might have been applied which expects an accusative noun-phrase to occur at some position p. Later, another rule might be applied which expects a nominative noun-phrase to occur at p. Yet, most noun-phrases can be analysed both as nominative and accusative. Instead of searching for a noun-phrase at position p twice, it might be better to ignore the case requirements temporarily and only search for a generic noun-phrase at p once.

The idea of goal-weakening is that before a parse goal is attempted, a filter is applied. This filter generalizes the parse goal somewhat. As a consequence, all solutions for this more general parse goal are collected and end up in the chart. Only those solutions that match the original goal are then passed on as solutions of the original goal. A later goal which is only slightly different will typically generalize to the same weakened goal. As a consequence it can simply use those results in the chart which match this later goal.

Compact Representation of Parse Trees Often a distinction is made between *recognition* and *parsing*. Recognition checks whether a given sentence can be generated by a grammar. Usually recognizers can be 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 a (possibly exponential) increase 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.

The structure of the 'parse-forest' in the head-corner parser is rather unusual, and therefore we will take some time to explain it. Because the head-corner parser uses selective memoization, conventional approaches to construct parse forests [3] are not applicable. The head-corner parser maintains a table of partial derivation-trees which each represent a successful path from a lexical head (or gap) 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 result-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 result-item references (representing the list of daughters left of the head in reverse, and the list of daughters right of the head).

The history table is a lexicalized tree substitution grammar, in which all nodes (except substitution nodes) are associated with a rule identifier (of the original grammar). This grammar derives exactly all derivation trees of the input. The tree substitution grammar is lexicalized in the sense that each of the trees has an associated anchor, which is a pointer to either a lexical entry or a gap.

Because we use chunks of parse trees less packing is possible than in their approach. Correspondingly, the theoretical worst-case space requirements are worse too.

We already argued above that parse trees should not be explicitly defined in the grammar. Logical forms often implicitly represent the derivational history 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 [1].

5 Translation into Updates

The grammar assigns a predicate form, a domain-independent semantic representation to a given utterance. In OVIS this predicate form is translated into a domain-specific *update* expression, which is passed on to the dialogue manager for further processing.

The dialogue manager keeps track of the information provided by the user by maintaining an *information state* or *form* [39]. This form is a hierarchical structure, with slots and values for the origin and destination of a connection, for the time at which the user wants to arrive or depart, etc. Each user utterance leads to an *update* of the information state. An update is an instruction for updating the information in an information state. Updating can mean that new information is added or that given information is confirmed, retracted or corrected. For example, given the information state:

$$\begin{bmatrix} \mathsf{travel} & \begin{bmatrix} \mathsf{origin} & \begin{bmatrix} \mathsf{place} & \begin{bmatrix} \mathsf{town} & \mathsf{groningen} \end{bmatrix} \\ \mathsf{moment} & \begin{bmatrix} \mathsf{at} & \begin{bmatrix} \mathsf{time} & \begin{bmatrix} \mathsf{clock\_hour} & 3 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathsf{destination} & \begin{bmatrix} \mathsf{place} & \begin{bmatrix} \mathsf{town} & \mathsf{leiden} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

the update

```
travel.destination.([# place.town.leiden]; [! place.town.abcoude])
```

the translation of *No, I do not want to travel to Leiden but to Abcoude*) leads to the information state:

$$\begin{bmatrix} \mathsf{travel} & \begin{bmatrix} \mathsf{origin} & \begin{bmatrix} \mathsf{place} & \begin{bmatrix} \mathsf{town} & \mathsf{groningen} \end{bmatrix} \\ \mathsf{moment} & \begin{bmatrix} \mathsf{at} & \begin{bmatrix} \mathsf{time} & \begin{bmatrix} \mathsf{clock\_hour} & 3 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathsf{destination} & \begin{bmatrix} \mathsf{place} & \begin{bmatrix} \mathsf{town} & \mathsf{abcoude} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The # -operator indicates that the information within its scope (indicated by square brackets) is to be retracted, and the '!'-operator indicates a correction. These operators are called 'communicative function' below.

The translation of a list of predicate forms into an *update* is done by applying translation-rules. Such a list of predicate forms correspond to the list of categories found by the robustness component, discussed below. These translation rules, expressed by a set of definite clauses, recursively define a corresponding update for a given predicate form.

A first set of such rules applies certain application-specific simplication rules. For example, the sentence

(2) nee naar assen
no to assen
no to assen

is assigned two separate predicate forms (one for 'nee' and one for 'naar assen'). A simplication rule is defined which combines the two predicate forms into a new single predicate form headed by the predicate 'correction'. Another application-specific simplification rule is a rule which simplifies a predicate form with predicate 'graag' and empty lists of arguments and adjuncts into a predicate form with predicate 'confirmation'.

A second set of rules translates each of the predicate forms in the list in turn, and combines the result by conjunction. Such rules generally take the form

```
pred_to_u(Pred,Args,Adjs,Update,CF,FrameStrNode)
```

where `Pred` is the name of the predicate, `Args` is the list of argument predicate forms, `Adjs` is the list of modifier predicate forms, `Update` will be bound to the resulting update, `CF` is the communicative function (such as 'assignment', 'correction', 'confirmation' etc). `FrameStrNode` is the name of a node in the frame structure; this is used to ensure that the resulting update is constructed in accordance with the frame structure.

As an example consider the following (simplified) rule that might be used to translate example (1).

```
pred_to_u(place(town(A)),[],[],U,CF,N0):-
    ftrs_assignment(U,CF,[town,S],N0).

pred_to_u(willen,[User],Adjs,U,CF,N0) :-
    user(User),
    ftrs_to_update([user,wants],_,UpdateHole,Hole,N0,N1),
    adjs_to_updates(Adjs,CF,U0,N1),
    if_nil(U0,U,UpdateHole,Hole).

if_nil(nil,nil,_,_).
if_nil(nonnil(Hole),U,U,Hole).
```

The rule for `willen` can be understood as follows. The first goal in the `pred_to_u` clause checks that the first argument is the predicate form associated with expressions such as 'ik' (I) and 'we' (we). The second goal creates a partial update starting with 'user.wants' and a hole. In this hole we will fill in the updates associated with the adjuncts. However, in case the adjuncts are translated to a special 'nil' update (indicating the empty update), then the resulting update for this rule will be the empty update too.

The first rule is used to translate city names. It uses a generic predicate `ftrs_assignment` which creates an update with an assignment operator (determined by `CF`).

Note that the updates in these rules are Prolog terms. A simple definite clause grammar is used both for parsing and formatting updates as sequences of terminal symbols. These terminal symbols themselves are defined by a small Elex script. Elex is a scanner generator for multiple languages including Prolog output [20, 33].

6 Robustness

In this section we describe the robustness component of the grammar-based linguistic analysis component. The robustness component is described in more detail in [36, 29].

6.1 Introduction

In the ideal case, the parser will find a path in the word graph that can be assigned an analysis according to the grammar, such that the path covers the complete time span of the utterance, i.e. the path leads from the start state to a final state. The analysis gives rise to a semantic representation which is then passed on to the next processing component.

However, often no such paths can be found in the word graph, due to:

- errors made by the speech recognizer,
- linguistic constructions not covered in the grammar, and
- irregularities in the spoken utterance.

Even if no full analysis of the word graph is possible, it is often the case that useful information can be extracted from the word graph. Consider for example the utterance:

(3)	Ik	reizen	van	Assen	naar	Amsterdam
	I	travel	from	Assen	to	Amsterdam
	I want to travel from Assen to Amsterdam					

The grammar might not assign an analysis to this utterance due to the missing finite verb. However, it would be useful if the parser discovered the prepositional phrases *van Assen* and *naar Amsterdam* since in that case the important information contained in the utterance can still be recovered. Thus, in cases where no full analysis is possible we would like to fall back on an approach reminiscent of concept spotting. The following proposal implements this idea.

Firstly, the grammar is defined in such a way that each *maximal projection* such as S, NP, PP, etc., can be analysed as a top category. This is well-motivated because utterances very often consist of a single NP or PP anyway.

Secondly, the parser is required to discover all instances of the top category *anywhere in the word graph*, i.e. for all partial paths in the word graph. This has the desired effect for the example (3): both PPs will be found by the parser (as well as the noun phrases "Ik", "Assen" and "Amsterdam".

Thirdly, an appropriate search algorithm will find a non-overlapping sequence of such top categories connected by uncovered words in the word-graph. For the example on page 16 this results for instance in the sequences:

⟨NP reizen van NP naar NP⟩
⟨NP reizen van NP PP⟩
⟨NP reizen PP naar NP⟩
⟨NP reizen PP PP⟩

Using appropriate heuristics, including a heuristic to minimize the number of uncovered words, we are then able to pick out the last path as the best path in this particular case.

Thus, we are interested in paths from the start state to the final state consisting of a number of categories and transitions in the word graph (the latter are called *skips*). The problem is to find the optimal path, according to a number of criteria. This problem is formalized by defining the annotated word graph as a directed acyclic graph. The vertices of this graph are the states of the word graph; the edges are the transitions of the word graph and the categories found by the parser.

The criteria which are used to favor some paths over other paths are expressed as a weight function on the edges of the graph. The criteria we might take into account are discussed in below. For instance, a typical criterion will favor paths consisting of a small number of categories, and a small number of skips. The case in which the parser found a full analysis from the start state of the word graph to a final state then reduces to a special case: the analysis solely consisting of that category will be favored over sequences of partial analyses.

Obviously, it is not a good idea to generate all possible sequences of categories and skips, and then to select the best path from this set: in typical word graphs there are simply too many different paths. If a certain uniformity requirement on weights is met, however, then efficient graph search algorithms are applicable. The particular algorithm implemented is a variant of the DAG-SHORTEST-PATH algorithm [8]. The implementation is discussed in detail in [36], [29].

6.2 Criteria

The robustness search algorithm combines the following criteria.

- Acoustic score. Obviously, the acoustic score present in the word graph is an important factor. The acoustic scores are derived from probabilities by taking the negative logarithm. For this reason we aim to minimize this score. If edges are combined, then we have to sum the corresponding acoustic scores.
- Number of 'skips'. We want to minimize the number of skips, in order to obtain a preference for the maximal projections found by the parser. Each time we select a skip edge, the number of skips is increased by 1.
- Number of maximal projections. We want to minimize the number of such maximal projections, in order to obtain a preference for more extended

linguistic analyses over a series of smaller ones. Each time we select a category edge, this number is increased by 1.

- Ngram statistics. We have experimented with bigrams and trigrams. Ngram scores are expressed as negative logarithms of probabilities. This implies that combining Ngram scores requires addition, and that lower scores are to be preferred.

The only requirement we make to ensure that efficient graph searching algorithms are applicable is that weights are *uniform*. This means that a weight for an edge leaving a vertex v_i is independent of how state v_i was reached. We have experimented with a variety of different *methods*. For example, the *nlp_speech_trigram* method mentioned takes into account trigram statistics as well as the acoustic scores and the number of skips and maximal projects.

6.3 Filtering Word Graphs

The method which involves a full parse of the word graph, is impracticable for large word graphs: both CPU-time and memory requirements become too demanding. For the test set described later, we were only able to apply this method for word graphs of up to about 150 transitions.

For this reason we have experimented with methods in which the word graph is filtered before it is passed on to the parser. Thus we make two passes in the word graph. In the first pass the best P paths are computed using Ngram statistics and acoustic scores. Transitions not taking part in any of these paths are removed from the word graph. In the second pass the *nlp_speech_trigram* method is applied to the filtered word graph. We refer to these methods as B-P and T-P, for various values of P:

B-P In the first pass this method selects the P best paths using bigram statistics. The second pass applies the *nlp_speech_trigram* method.

T-P In the first pass this method selects the P best paths using trigram statistics. The second pass applies the *nlp_speech_trigram* method.

6.4 A More Efficient Approximation

It turns out that the filtering methods discussed in the previous section still require unattractive processing times for larger word graphs. Therefore, this section introduces an efficient approximation technique. This variant is not sound, since it does not always yield the best path possible. In practice, however, it turns out (as we will show in a later section) that the resulting implementation almost always produces the same results, using only a fraction of the processing time required by the sound method.

The approximation is very similar to beam search techniques in that a constant b (the "beam") is assumed which indicates the maximum number of paths associated with a given vertex. We maintain for each vertex the best b paths to that vertex. But in case Ngrams are added to the weight function, we cannot guarantee that we find the best path, because weights are not uniform anymore. It may be possible that a path ending in some vertex v_i is discarded by the algorithm, whereas the last sequence of words of that path would ensure that extending that path yields a very good score and in fact even better than any of the scores that result from extending any of the paths ending in v_i which were maintained by the algorithm.

Larger values for b entail more chance of obtaining the best path (therefore it makes sense to talk about approximation here); and for every word graph there will be some b such that the best path is produced.

Note that the complexity of the search algorithm is linearly dependent on b. Finally, b is always at least as great as P, i.e. the number of solutions we want to obtain.

We write M, b for a method M with beam b. For example, the method B-4, 10 is the method which filters the word-graph in the first phase using the approximation search with a beam $b = 10$ to select the best 4 paths using bigram statistics (and acoustic scores). The second phase, as always, consists of the (sound) application of the *nlp_speech_trigram* method.

7 Evaluation

In order to test the adequacy of the NLP component we have performed a formal evaluation three years after the start of the Programme. The evaluation measures string accuracy, semantic accuracy and computational resources. For comparison we also list the best result of the data-oriented approach. More details of this evaluation can be found in [38, 6, 32].

A training set of 10K richly annotated word graphs was available. The 10K training corpus is annotated with the user utterance, a syntactic tree and an update. This training set was used to train the DOP system. It was also used by the grammar-based component for reasons of grammar maintenance and grammar testing.

A further training set of about 90K user utterances was available as well. It was used for constructing the Ngram models incorporated in the robustness search algorithm.

The NLP components were evaluated on 1000 unseen user utterances. The latest version of the speech recogniser produced 1000 word graphs on the basis of these 1000 user utterances. For these word graphs, annotations consisting of the actual sentence ('test sentence'), and an update ('test update') were assigned semi-automatically, without taking into account the dialogue context in which the sentences were uttered. These annotations were unknown to both NLP groups. The annotation tools are described in Bonnema (1996).

Some indication of the difficulty of the set of 1000 word graphs is presented in table 1. A further indication of the difficulty of this set of word graphs is obtained if we look at the word and sentence accuracy obtained by a number of simple methods. The method *speech* only takes into account the acoustic scores found in the word graph. No language model is taken into account. The method *possible* assumes that there is an oracle which chooses a path such that it turns out to be the best possible path. This method can be seen as a natural upper bound of what can be achieved.

The methods *speech_bigram* and *speech_trigram* use a combination of bigram (resp. trigram) statistics and the speech score. In the latter four cases, a language model was computed from about 50K utterances (not containing the utterances from the test set). The results are summarised in table 2.

	graphs	trans	states	words	t/w	max(t)	max(s)
input	1000	48215	16181	3229	14.9	793	151
normalised	1000	73502	11056	3229	22.8	2943	128

Table 1: **Characterisation of test set (1).** This table lists the number of transitions, the number of states, the number of words of the actual utterances, the average number of transitions per word, the maximum number of transitions, and the maximum number of states. The first row provides those statistics for the input word graph; the second row for the so-called normalised word graph in which all ϵ-transitions (to model the absence of sound) are removed. The number of transitions per word is an indication of the extra ambiguity for the parser introduced by the word graphs in comparison with parsing of an ordinary string.

method	WA	SA
speech	69.8	56.0
possible	90.5	83.7
speech_bigram	81.1	73.6
speech_trigram	83.9	76.2

Table 2: **Characterisation of test set (2).** Word accuracy and sentence accuracy based on acoustic score only (speech); using the best possible path through the word graph, i.e. based on acoustic scores only (possible); and using a combination of bigram (resp. trigram) scores and acoustic scores.

This section lists the results for word graphs. In table 3 we list the results in terms of string accuracy, semantic accuracy and the computational resources required to complete the test.

7.1 Results for Approximation Methods

The second set of experiments compares the sound implementation with the approximation defined in subsection 6.4. This second set of experiments was performed later, using better hardware, and using a better N-gram language model (more dialogues had been collected in the mean time).

In table 4 we compare the results for the approximating filtering methods for different values of b. The tables also list the results for the sound implementation. As can be observed, a value of $b = 4$ yielded for T-1 a concept accuracy

Method	String Acc		Semantic Acc	CPU		Mem
	WA	SA		total	max	max
DOP	76.8	69.3	75.5	7011	648	619
B-16	83.8	76.4	82.6	1659	757	60
T-4	84.3	76.4	83.0	5524	2791	177

Table 3: **Accuracy and Computational Resources for 1000 word graphs.** String Accuracy and Semantic Accuracy is given as percentages; total and maximum CPU-time in seconds, maximum memory requirements in Megabytes.

Method	CPU		WA	CA
	mean	max		
	msec	sec	%	%
T-1,1	26	0.4	79.6	78.1
T-1,2	30	0.6	82.2	80.6
T-1,3	33	0.8	83.7	82.1
T-1,4	37	1.0	84.0	82.3
T-1,5	40	1.2	84.0	82.3
T-1,10	58	2.7	84.3	82.4
T-1,20	102	6.5	84.4	82.4
T-1	168	22.1	84.5	82.4
T-3,3	45	1.1	84.4	83.1
T-3,6	56	1.6	84.9	83.7
T-3,10	71	2.8	84.9	83.7
T-3,20	118	7.1	85.0	83.6
T-3	449	67.3	85.0	83.4

Table 4: Results for the full test set. The top rows of the table compare for T-1 the approximating filtering method (using various beams) with the sound implementation. Using a beam of $b = 4$ already obtains results similar to the sound implementation; the CPU-time requirements are far more modest though. The lower rows of the table provide a similar comparison for T-3.

which was already close to the sound implementation (82.3% versus 82.4%). The difference in speed was tremendous though. The difference in maximum amount of CPU-time is even more dramatic. The second part of the table shows that for methods in which more than a single result is obtained it hardly makes a difference to increase b.

Final Remarks

In this chapter we presented the grammar-based linguistic analysis component of the OVIS system. We shortly described a computational grammar for Dutch, which defines the relation between spoken utterances and semantic representations. It was shown how standard parsing algorithms can be generalized for word graph input, and we described the actual parser employed in the system.

In many cases, the parser will not assign a single analysis to a word graph. In some cases, the parser will find too many analyses (in case of ambiguity), whereas in other cases too few analyses result (in case of speech recognition errors, disfluencies in the input, gaps in the grammar). Therefore, the parser was extended to find all meaningful phrases anywhere in the word graph. A shortest path algorithm is used which finds the best sequence of partial parses, incorporating disambiguation techniques in a multidimensional weight function. The resulting algorithm is accurate but slow if this weight function takes Ngram statistics into account. A very efficient approximation is possible by filtering the word graph using an approximation search; the resulting smaller word graph can then be analysed accurately and efficiently. The accuracy of the approximation turns out to be very good in practice.

The resulting system performs well for typical word-graphs: accuracy is competitive and efficiency is acceptable.

References

[1] Hiyan Alshawi, editor. *The Core Language Engine*. ACL-MIT press, Cambridge Mass., 1992.

[2] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, SprachWissenschaft und Kommunicationsforschung*, 14:143–172, 1961. Reprinted in Bar-Hillel's Language and Information – Selected Essays on their Theory and Application, Addison Wesley series in Logic, 1964, pp. 116-150.

[3] S. Billot and B. Lang. The structure of shared parse forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver, 1989.

[4] Hans Ulrich Block. Compiling trace & unification grammar. In Tomek Strzalkowski, editor, *Reversible Grammar in Natural Language Processing*, pages 155–174. Kluwer Academic Publishers, Dordrecht, 1994.

[5] R. Bonnema. Data oriented semantics. Master's thesis, Department of Computational Linguistics, University of Amsterdam, 1996.

[6] Remko Bonnema, Gertjan van Noord, and Gert Veldhuizen van Zanten. Evaluation results NLP components OVIS2. Technical Report 57, NWO Priority Programme Language and Speech Technology, 1998.

[7] Ted Briscoe, Claire Grover, Bran Boguraev, and John Carroll. A formalism and environment for the development of a large grammar of english. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 703–708, Milan, 1987.

[8] Cormen, Leiserson, and Rivest. *Introduction to Algorithms*. MIT Press, Cambridge Mass., 1990.

[9] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 14, 1970.

[10] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[11] Mark Johnson and Jochen Dörre. Memoization of coroutined constraints. In *33th Annual Meeting of the Association for Computational Linguistics*, pages 100–107, Boston, 1995.

[12] Martin Kay. Head driven parsing. In *1st International Workshop on Parsing Technologies (IWPT '89)*, Carnegie Mellon University, Pittsburg, 1989.

[13] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, 1974. Also: Rapport de Recherche 72, IRIA-Laboria, Rocquencourt (France).

[14] Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom up parser embedded in Prolog. *New Generation Computing*, 1(2), 1983.

[15] C. S. Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, 1988.

[16] M. J. Nederhof and G. Satta. Efficient tabular LR parsing. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 239–246, Santa Cruz, 1996.

[17] Martin Oerder and Hermann Ney. Word graphs: An efficient interface between continuous-speech recognition and language understanding. In *ICASSP Volume 2*, pages 119–122, 1993.

[18] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford, 1987.

[19] Fernando C. N. Pereira and David Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980.

[20] Matthew Phillips. Elex user's guide, 1997.

[21] Carl Pollard and Ivan Sag. *Head-driven Phrase Structure Grammar*. University of Chicago / CSLI, 1994.

[22] Steve Pulman. Unification encodings of grammatical notations. *Computational Linguistics*, 22(3):295–328, 1996.

[23] Ivan Sag. English relative clause constructions. *Journal of Linguistics*, 1997. to appear.

[24] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23th Annual Meeting of the Association for Computational Linguistics*, Chicago, 1985.

[25] Stuart M. Shieber. *Introduction to Unification-Based Approaches to Grammar*. Center for the Study of Language and Information Stanford, 1986.

[26] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. Technical Report TR-11-94, Center for Research in Computing Technology, Division of Applied Sciences, Harvard University, 1994. CMP-LG 9404008.

[27] Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C. N. Pereira. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42, 1990.

[28] Klaas Sikkel. *Parsing Schemata*. PhD thesis, Twente University, Enschede, 1993. Published in 1997 by Springer Verlag, Texts in Theoretical Computer Science, An EATCS Series.

[29] Gertjan van Noord. Robust parsing of word graphs. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, Dordrecht. In Preparation.

[30] Gertjan van Noord. The intersection of finite state automata and definite clause grammars. In *33th Annual Meeting of the Association for Computational Linguistics*, pages 159–165, MIT Cambridge Mass., 1995. cmp-lg/9504026.

[31] Gertjan van Noord. An efficient implementation of the head corner parser. *Computational Linguistics*, 23(3):425–456, 1997. cmp-lg/9701004.

[32] Gertjan van Noord. Evaluation of OVIS2 NLP components. Technical Report 46, NWO Priority Programme Language and Speech Technology, 1997.

[33] Gertjan van Noord. Prolog(elex): A new tool to generate prolog tokenizers, 1998.

[34] Gertjan van Noord and Gosse Bouma. Hdrug, A flexible and extendible development environment for natural language processing. In *Proceedings of the EACL/ACL workshop on Environments for Grammar Development, Madrid*, 1997.

[35] Gertjan van Noord, Gosse Bouma, Rob Koeling, and Mark-Jan Nederhof. Conventional natural language processing in the NWO priority programme on language and speech technology. October 1996 Deliverables. Technical Report 28, NWO Priority Programme Language and Speech Technology, 1996.

[36] Gertjan van Noord, Gosse Bouma, Rob Koeling, and Mark-Jan Nederhof. Robust grammatical analysis for spoken dialogue systems. *Journal of Natural Language Engineering*, 5(1):45–93, 1999.

[37] Gertjan van Noord, Mark-Jan Nederhof, Rob Koeling, and Gosse Bouma. Conventional natural language processing in the NWO priority programme on language and speech technology. January 1996 Deliverables. Technical Report 22, NWO Priority Programme Language and Speech Technology, 1996.

[38] Gert Veldhuijzen van Zanten, Gosse Bouma, Khalil Sima'an, Gertjan van Noord, and Remko Bonnema. Evaluation of the nlp components of the OVIS2 spoken dialogue system. In Frank van Eynde, Ineke Schuurman, and Ness Schelkens, editors, *Computational Linguistics in the Netherlands 1998*, pages 213–229. Rodopi Amsterdam, 1999.

[39] Gert Veldhuijzen van Zanten. Semantics of update expressions. Technical Report 24, NWO Priority Programme Language and Speech Technology, 1996.