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

## October 1996 Deliverables

Gertjan van Noord
Gosse Bouma
Rob Koeling
Mark-Jan Nederhof

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

December 13, 1996

# Contents

# Chapter 1

# Introduction

This document contains the deliverables after one year and nine months 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).

This document should be read in conjunction with the first package of deliverables (van Noord et al., 1996). That package explains the basic building blocks of the system. It first described the grammar formalism: definite-clause grammars. Furthermore, the grammar was documented extensively, as well as the component which compiles quasi-logical forms into update expressions. These update expressions are constructed according to the directions given in Veldhuijzen van Zanten (1996). Furthermore, the parser generator and a number of different parsers were implemented and described: two bottom-up Earley parsers; a bottom-up shift-reduce parser; and a head-corner parser. The parsers are consistent with the requirements documented in Boves (1996).

This deliverable consists of the following parts. In the first part we document the changes and extensions in the grammar, and the grammar-to-update module. The second part documents a new and very fast left-corner parser, and a new LR parser. Furthermore, comparisons between all of the available parsers are described. The third part consists of a description of the robustness and disambiguation component. The final part describes the evaluation of the full NLP component in terms of word accuracy and sentence accuracy, for a test-set extracted from the corpus of more than 20000 word-graphs.

Everything that is described in this deliverable is implemented and integrated. If alternative methods are described then each of these methods can be obtained by the setting of a small number of parameters. The program is available in two forms. On the one hand as a SICStus Prolog runtime system (essentially a program written in C, linked with a large `.o` file containing the Prolog interpreter, and a number of compiled Prolog files containing the parsers and grammars). This version is ready to be plugged into the full system as a module: it follows the instructions given in Kersten (1995). For development and demonstration purposes we have also constructed a version of the system which is based on the Hdrug system. This provides an easily extendible graphical user interface (van Noord, 1994), and various other useful utilities for grammar and parser development.

A number of utilities for development were implemented. Since these are not described elsewhere, we mention these here. Utilities were implemented in order to:

- automatically parse a given test-suite (set of word-graphs) and compute the word-accuracy and sentence-accuracy, for a number of different methods. Such methods are described in section 4.
- compute the statistical significance of the difference in word-accuracy for each given pair of methods.
- compare the efficiency of different parsers. Output can be generated in a number of forms:

charts of cputime against word-graph size, memory consumption against word-graph size, charts of the percentage of word-graphs that can be treated within certain time thresholds, etc.

- visualisation of word-graphs (by providing an interface to the FSA Utilities toolbox (van Noord, 1996)).

The previous versions of the system were implemented in SICStus Prolog 2.1 #8. The latest version is now ported to SICStus Prolog 3. This port resulted in an efficiency increase. Moreover the interface between C and Prolog should now be more robust.

# Chapter 2

# Grammar Development

In this chapter, we describe the grammar development efforts carried out after completion of the first deliverable (van Noord et al., 1996). The first section reports changes in the grammar which resulted from evaluating the performance of the grammar on a corpus that was created using the experimental OVIS1 system. The second section reports work on the semantic component of the grammar. The format of the updates produced by the system was made compatible with the latest version of the Update language (Veldhuijzen van Zanten, 1996), and a first attempt was made to incorporate information from the system question in determining the correct meaning of the answer.

## 2.1 Using the Corpus

To improve the performance of the grammar in the domain of human-machine dialogs on railway timetable information, we have used the corpus of $21K^+$ word-graphs which was constructed using the experimental OVIS1 system. Our goal was to tune the grammar in such a way that:

1. Semantic accuracy would increase,

2. Efficiency would increase.

The performance with respect to the first goal cannot be measured automatically at the moment. Therefore we used word and sentence accuracy as an approximation (cf. chapter 5). Determining whether the system provides an accurate translation of an utterance currently requires a human expert. Efficiency increase was obtained by removing a number of spurious derivations.

We used the following procedure to find shortcomings in the original grammar:

1. The four project members each carefully evaluated the perfomance of the grammar on a test set of 500 utterances (i.e. a total of 2000 utterances was inspected).

2. On the basis of this inspection, a list of problems for the original grammar was compiled. This list was our guide in trying out various modifications and extensions of the grammar.

3. In case the grammar was missing certain words, a search of the full set of actually uttered sentences was performed, in order to determine in which contexts, and how frequently, the word occurred.

Below, we describe the most important changes to the grammar (as compared to the grammar described in the first deliverable). We conclude with a summary of results.

### 2.1.1 Spoken Language Constructs

The fact that the utterances which must be analyzed are part of a dialogue, and often answers to questions, implies that certain constructions which are typical for spoken language and question answering occur frequently. We give some examples below.

Often, the speaker will respond to a yes/no question. This leads to the following types of response:

```
dat klopt
inderdaad
ja
ja dat klopt
ja dat klopt zei ik al
ja dat is juist
ja dat is mogelijk
ja dat is prima
ja dat lijkt me fijn
ja dat lijkt me heel zinvol
ja dat lijkt me uitstekend
ja doe maar
ja graag
jawel
zo ongeveer ja
dank u wel nee dank u
dat is niet nodig
dat hoeft niet
liever niet
nee
nee bedankt
nee dank je wel
nee dank u
nee dank u wel
nee dank u wel zo is het voldoende
nee dat is niet nodig
nee dat hoeft niet
nee dat wil ik niet
nee hoor
nee hoor dank u wel
nee hoor dat hoeft niet
nee hoor dat hoeft niet dank u wel
nee laat maar
niet nodig
ook niet
```

Another typical situation is one where the system asks (implicitly or explicitly) for confirmation. This leads to responses such as:[1]

```
nee ik wil op acht februari reizen
nee ik wil vertrekken om twee uur s-middags
nee ik wil van den haag naar utrecht
nee ik wil geen andere verbinding weten
ja ik wil een doorgaande trein van amsterdam amstel naar alexanderpolder
```

[1] Note that we did not have access to the full transcription of the dialogues. Therefore, the distinction between an answer to a yes/no question (concerning information not previously mentioned in the conversation), and confirmations (of information previously mentioned), is not always clear.

```
rotterdam
ja graag van den haag naar groningen
ja zeven uur klopt
ja franeker harlingen
ik wil om acht uur vertrekken ja
```

Users sometimes start or end their utterances with a politeness expression, although this is not as common in man-machine dialogue as it is in ordinary conversation:

```
graag van zandvoort naar utrecht terug
goeiendag ik wil graag van voorburg naar amsterdam
vanmiddag als het kan
doe maar van utrecht naar groningen
nou doe dan maar amsterdam
ja hallo ik wil van schiphol naar beilen
nee hallo ik wil van voorburg naar amsterdam lelylaan
amsterdam centraal graag
hallo ik wil graag van rotterdam naar maastricht
```

To account for these phenomena, the grammar allows a phrase (i.e. a sentence, or meaningful maximal projection, such as a temporal or locative phrase) to start with a spoken language expression. Spoken language expressions also may occur as single utterances. The semantics of utterances such as *ja* and *nee*, and their equivalent is straightforward. The semantics of other expressions, such as *graag*, is simply ignored, if they are followed or preceded by a meaningful expression. In utterances such as *nee, naar zwolle*, finally, the spoken language expression *nee* indicates that the following information is a correction.

The grammar for spoken language expressions is a separate module in the grammar, where no attempt is made to follow the principles of linguistic organization adopted in the 'core' grammar (van Noord et al., 1996, chapter 3). Currently, it consists mainly of a long list of (multiword) phrases. Before inspecting the corpus, we had already compiled a substantial list. Inspection of the corpus led to the addition of such phrases as:

```
dit is fout
dit is foutief
dit is weer fout
dit is wederom fout
fout
dat is niet correct
dit is niet correct
correct
dat heb ik gezegd
dat heb ik niet gezegd
dat zei ik niet

doet u maar
doe maar
doet u dat
doet u dat maar

ik weet voldoende
dit is voldoende


u hebt me niet begrepen
begrepen
```

```
het is begrepen

dat wil ik niet
dat wil ik helemaal niet

juist
niet juist
dat is niet juist

dat is goed
nee dat is goed
dat klopt
klopt jazeker
```

Note that some of these utterances are typical for situations in which misunderstandings are likely to occur.

Our impression is that, while one may obtain quite reasonable coverage by simply collecting phrases from a corpus, it is probably worthwhile to investigate to which extent these expressions could be defined by means of a set of (special-purpose) grammar rules. There is good deal of variation in the expressions one encounters, but the lexical variation (i.e. the number of different words encountered) is not very high. The challenge here is to design a grammar which covers the various utterances that may be encountered in all their variations, and at the same time does not overgenerate (as this would certainly have a negative effect on the accuracy of the system in determining what is being said). We are considering a reimplementation of this part of the grammar in the latter sense.

### 2.1.2 Temporal Expressions

Temporal expressions are handled by means of a separate module in the grammar (van Noord et al., 1996, section 3.4.). It currently consists of about 25 rules, and over 300 lexical items. The grammar was completed to a large extent, before extensive testing with the corpus. Inspection of the corpus did lead to a few additions.

The words *aanstaande* and *omstreeks* were added, to allow phrases such as:

```
zeven januari aanstaande
aanstaande zondag
zondag aanstaande
omstreeks negen uur
zondag morgen omstreeks tien uur
ik wou omstreeks veertien uur reizen
```

Furthermore, years were added:

```
negentienzesennegentig
negentienhonderdzesennegentig
zesennegentig
negentien zesennegentig
```

Also, the grammar of *nul* (*zero*) was added:

```
ja om vijftien nul nul uur
negen punt nul nul uur
om nul zes punt nul nul uur
morgen om nul acht uur
om nul zeven honderd uur
```

The following were added as alternatives for *'s ochtends* (*morning*) and *'s middags* (*afternoon*):

```
in de ochtend
in de morgen
in de middag
```

We added the possibility of using *vanaf* in a temporal sense:

```
vanaf half tien
vanaf zeven uur s-morgens
vanaf woensdag
vanaf morgen drie uur
```

Finally, we altered the rule for 'tussen' (between) constructions. From the corpus data we learned that far more occurrences (more than 70%) of these constructions were covered by the pattern: *tussen X (uur) en Y (uur)* (the word 'uur' is optional). The other utterances of this type differ so much that we need to add several grammar rules to catch all the differences. Each of them occurred so seldom (only once or twice in the $21K^{+}$ corpus) that we decided not to add these rules at this moment.

### 2.1.3 Sortal restrictions

A number of erroneous constructions are ruled out by adding sortal restrictions for locative and temporal prepositional phrases. In the current grammar only some obvious constraints are added. The prepositions *om, na, voor, rond, tegen* subcategorise for a temporal NP of time, the preposition *tussen* for a conjunction of temporal NP's (of time), the prepositions *voor, na, op* for a temporal NP of date and finally the prepositions *van,naar,uit,in,via,richting,vanuit,vanaf* subcategorise for a locative NP.

These sortal restrictions are implemented as follows. A preposition is associated with a feature PFORM which typically simply takes the lexical form of that preposition as its value. Moreover, such a preposition requires that it can only combine with a noun-phrase *with the same specification for its* PFORM *attribute*. This technique is possible, because the value of PFORM for noun-phrases typically is a disjunction. Such a disjunction is coded in a single term using the implementation of a special technique (Mellish, 1988) provided by the feature library discussed in van Noord et al. (1996). For example, a temporal expression such as 'tien voor tien' is a noun-phrase which as the value `voor;tegen;rond;om;na` for its PFORM feature. This implies that it can only combine with one these five prepositions to form a prepositional phrase.

### 2.1.4 Other

**Modification and verb-second.** To avoid spurious ambiguities, we added the restriction that a modifier can only occur at the righthand side of a verb if the verb is not a gap (i.e. the feature NULL is *nonnull*). Thus, the grammar now only derives the first of the following two analyses. Here, `vgap` refers to an empty verbal element that is used for the analysis of verb-second (van Noord et al., 1996):

```
[s ik wil [vp [pp naar assen] vgap]]
[s ik wil [vp vgap [pp naar assen]]]
```

**Denial of modifiers.** A special rule was added to cover the denial of a modifier:

```
niet naar Assen
nee niet om tien uur
niet morgen
```

In our previous grammar, these utterances were (wrongly) given the same updates as examples such as:

```
nee naar Assen
nee om tien uur
nee morgen
```

In the former sentences it is clear that some information is *denied*, and in the latter some information is *corrected*.

***Niet ...maar ...*** Another special rule was added to deal with constructions of the form *nee niet A maar B* (*no not A but B*) and *nee ik wil niet A maar B* (*no I don't want A but B*). A special rule was necessary in this case as the 'core' grammar does not cover conjunctions at the moment. Although the robustness component of the grammar did recognize both conjuncts, it didn't recognize that the actual speech-act is a *correction* instead of a *denial* followed by an *assertion*.

***Eerder* and *later*** Users often ask for an earlier or later connection, or for the latest or earliest connection:

```
nee ik wil een trein eerder
is er ook een eerdere trein
een eerdere verbinding
een latere
ja ik wil een latere verbinding
nee een latere trein
een trein later graag
ja ik wil hem graag van een trein later weten
heeft u ook eentje later
de eerste trein
met de eerste trein die gaat
van hoorn naar enkhuizen hoe vroeg gaat de eerste trein
op dinsdag de eerste trein
de eerste gelegenheid graag
met de laatste trein
de laatste
ik wil graag de laatste
ja de laatste trein morgenavond van amsterdam naar nijmegen
nee ik wil graag de laatste verbinding horen
```

To account for this, the relevant expressions have been added to the grammar. They are translated by creating an update which sets either the value of the slot `users.wants.browse` to `earlier` or `later` (in case the system asks if the user wants the previous or the next connection) or the slot `user.wants.info.quantor` to `earliest`, or `latest` (in case the user adds information about the preferred time of travelling).

Finally, various lexical items were added: past tenses of *willen*, i.e. *wilde, wou*; adverb phrases such as *we, zo spoedig mogelijk, zo snel mogelijk*, and *zo vroeg mogelijk* (where the latter is translated into the value `first` of the slot `user.wants.info.quantor`) and the station name *Almere Buiten*.

### 2.1.5 Results

The results with respect to the first goal (improving word and sentence accuracy) can now be measured quantitatively, by comparing the results of parsing the word graph with the utterance that was actually uttered (see the section on robustness below). [2]

On a test set of 500 utterances, the results were as follows (for a number of other test sets, similar results were obtained):

| | $Aug96$ | $Oct96$ | $Increase$ | $Possible$ |
|---|---|---|---|---|
| Word Accuracy | 71.4 | 73.8 | 2.4 | 88.2 |
| Sentence Accuracy | 58.2 | 61.8 | 3.6 | 72.8 |

The first column gives the score for the original grammar, the second the score for the final grammar, the third column the increase in performance, the fourth column the best possible score (i.e. using an oracle that would make the optimal decision all the time), and the final column the score of the experimental OVIS1 system. Comparing our initial and final result with the results of OVIS1, we observe that the 27% of the difference in word accuracy and 43% of the difference in sentence accuracy has been eliminated. This suggests that the grammar development efforts have led to a significant increase in performance. Note also that we only consider improvements due to changes in the grammar at this point. In the section on robustness, the results of combining various techniques to improve performance are given.

## 2.2 Semantics

The main improvements that were made with respect to the semantics were concerned with construction of the updates consistent with the latest version of the definition of the update-language. Furthermore, a first attempt to disambiguate on the basis of the system question was implemented.

### 2.2.1 The Update Module

#### 2.2.1.1 Format of updates

In van Noord et al. (1996) we reported that the updates produced by the system were not fully compatable with Veldhuijzen van Zanten (1995). In the mean time the definition of the update language has changed. The current update module produces updates that follow the specifications given in Veldhuijzen van Zanten (1996).

We give an example of an update produced by the current update module in figure 2.1:

(2.1) *a.* Ik wil om half vijf vertrekken
*(I want to leave at half past four)*
*b.* `(user.wants.travel.origin.moment.at.[: (time.clock-hour.4; time.minute.30)])`

The communicative function in this update is *assert* (represented by the operator :). In this update the value expressed by the conjunction of the two update expressions `time.clock-hour.4` en `time.minute.30` is inserted in the slot denoted by the square brackets.

#### 2.2.1.2 Construction of updates

The translation of QLF's (produced by the grammar) into updates now handles communicative functions and conjunctions correctly. The scope of communicative functions as well as conjunctions is always made as small as possible. This is important especially for the communicative functions (Veldhuijzen van Zanten, 1996).

[2]Since the utilities for the computation of word and sentence accuracy, and the full corpus of 21K word-graphs are only available since the summer of this year, we can only compare with the grammar of August 1996. In this version some of the modifications discussed here were already present.

#### 2.2.1.3 Interpretation of negation

Utterances like 2.2 are ambiguous with respect to the scope of the negation operator. This utterance might mean that either only the first or only the second or both of the PP's are negated.

(2.2) `Ik wil niet om drie uur naar Assen`
*I do not want to go to Assen at three o'clock*

What exactly is negated depends very much on the current content of the information state. We decided to pass in these cases the disjunction of all possible readings to the pragmatic interpretation module/dialogue manager. Utterance 2.2 is therefore translated into the following update (in which the operator # represents the communicative function *deny*):

```
({ (moment.at.[# time.clock_hour.3];destination.[# place.town.assen])
  |(moment.at.[# time.clock_hour.3];destination.[+ place.town.assen])
  |(moment.at.[+ time.clock_hour.3];destination.[# place.town.assen])})
```

#### 2.2.1.4 Proper names and pronouns

In van Noord et al. (1996) we reported that the first step in the translation process from QLF to update consisted of replacing the terms describing the *proper names* and *pronouns* by constants (i.e. values corresponding to the lexical values). This is possible because the proper names (i.e. mostly station names) have their equivalent in the update expression language, and the pronouns are resolved trivially. In the current grammar we anticipate this step and translate these constituents in the QLF with *constants*. In other domains it might well be the case that this is not desirable, so in fact we favor here a domain specific translation of these expressions.

### 2.2.2 Disambiguation on the basis of the system question

We observed that it occurs quite frequently that although the right interpretation of the user utterance is available in the wordgraph, this interpretation is not chosen by our NLP-module. This is partly due to the algorithm that selects the best path in the word-graph (as descibed in section 4) which favours paths that skip as little information in the word-graph as possible. Although this is a reasonable constraint, it seems to be a good idea to employ other sources of information as well. One of those sources is the information provided by the system question that preceeds the utterance. When the type of the user utterance matches the type of the system question, it is an indication that this particular path in the wordgraph is to be favoured above others. In the current update module we implemented a first version of this idea.

The corpus of over 20.000 wordgraphs which we could use to test our grammar are all isolated utterances, without the preceding system questions. At an earlier stage we received a much smaller corpus of complete dialogues. We could do some tests with this corpus, but it only gives an indication of the results we can expect from this technique. The first tests suggested a small improvement of the performance.

This technique is demonstrated with an example from the corpus. In one of the dialogues the system asked the following question:

(2.3) *a.* `System:` Hoe laat wilt u op donderdag eenentwintig maart reizen?
*: At what time do you want to travel on thursday March twentyone?*
*b.* `User:` Tien uur
*Ten o'clock*

The output of the speech recognizer for the answer of the user resulted in the wordgraph of figure 2.1.

The update of the question is represented by:

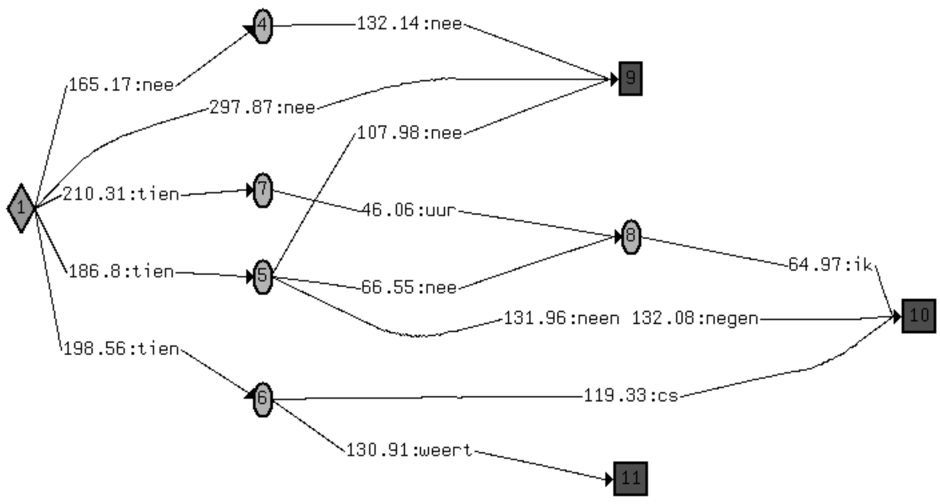


Figure 2.1: The wordgraph for the utterance *Tien uur*

```
open_q(temporal): moment.at.([? time._]
                             ;date.day_of_week.thursday
                             ;date.month.March;date.day.21)
```

The actual utterance (i.e. *tien uur*) is present in the wordgraph. The path however does not end in a final state and therefore a transition must be skipped (which results in a penalty; see section 4). The path that is chosen when only probabilities assigned by the speech-recognizer are considered, is *tien cs*. Without considering the system question, the path denoting *nee* is favoured. When the question is taken into account, the path *tien uur* is chosen as most appropriate path.

At this moment only three types of questions/answers are considered: closed (yes/no) questions and open (temporal and locative) questions. In this domain these seem to be the major discriminating types. Further refinement of this idea will be possible if we take the results on a larger corpus into account.

# Chapter 3

# Parsing

In van Noord et al. (1996) we described a number of different parsing strategies for the Ovis Grammar. These included:

- two bottom-up Earley parsers (`act`, `act2`).
- a bottom-up Earley parser with packing and item weakening (`bu`). This parser is an active chart parser.
- a bottom-up shift-reduce parser (`inact_p`) with packing and item weakening. This is an inactive chart parser.
- a head-corner parser (`hc9_mixtus`) with packing, goal weakening and partial evaluation.

Preliminary experiments were carried out which showed that the head-corner parser is more efficient both in terms of cpu-time and memory requirements.

In this deliverable we present two new parsers that were implemented during the last few months. Inspired by the success of the head-corner parser, a left-corner parser was implemented using many of the techniques that had been proved useful in the implementation of the head-corner parser. This parser is described in section 3.1. Furthermore, we implemented two versions of a tabular LR parser, based on the techniques introduced for context-free parsing in Nederhof and Satta (1996). These parsers are described in section 3.2.

Each of the parsers are capable of producing all parses for a given top-category *occurring anywhere in the input* word-graph, as required by our robustness and disambiguation model (described in chapter 4).

Further experiments were carried out for much larger sets of word-graphs. The results of these experiments are described in section 3.3.

## 3.1 Left-corner parser

The left-corner parser is implemented in a very similar fashion as the head-corner parser described in full detail in van Noord et al. (1996). We will therefore leave out many of the details here, and refer to the previous deliverable for motivations and technicalities of the methods used.

The left-corner parser implemented for Ovis can be compared with the BUP parser (Matsumoto et al., 1983; Pereira and Shieber, 1987). The following characteristics can be given:

- Top-down filtering. Top-down filtering is obtained using a precompiled table of (a generalization of) the left-corner relation. The left-corner relation is the reflexive and transitive closure of the relation 'left-most member', which holds for a pair of categories $c_m, c_i$ iff there is a rule with left-hand-side $c_m$ and left-most member of the right-hand-side $c_i$.

As in the head-corner parser, this table is constructed in such a way (using TERM GENERALIZATION) that for each pair of *functors* of categories, there is only at most one matching clause. This solves potential spurious ambiguity problems. Moreover, this made it possible to implement efficient indexing techniques for the storage of the table.

- Memoization. Memoization is implemented using the primitives previously implemented for the head-corner parser. Memoization is only used for parse-goals. This implies that, in comparison with ordinary chart parsers, more work is done using backtrack search, whereas only the larger chunks of work take part in memoization. This has a very good effect on both processing time and memory usage, because for complex-feature-based grammar formalisms, the costs associated with the maintenance of tables is considerable.

- 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 grammars).

  The implementation re-uses the basic operations implemented previously for the head-corner parser.

- Compact representation of parse-trees (packing). In the left-corner parser, the same techniques for packing are used as in the head-corner parser.

- Underspecification of string positions. It turns out that the possibility to use variables instead of string positions (or rather word-graph state names) is useful for the left-corner parser too. The parser is started without any information concerning the start and input state name: this implies that *all* top-categories occurring anywhere in the input graph will be constructed. This is required by the robustness component (described in chapter 4). Also note that empty productions are treated directly, by treating them as lexical entries with (shared) variable begin and end position.

#### 3.1.0.1 Implementation of the Left-Corner Parser

**Grammar compilation.** The following predicates are the result of grammar compilation:

```
grammar:top_category(Cat).
lc_in:left_corner_table(Lc,Goal).
lc_in:lc_rule(SynLc,SynMother,SynRestRhs,RuleName).
lc_in:lc_rule_i(Lc,Mother,RestRhs,RuleName).
lc_in:gap(SynCat,RuleName).
lc_in:gap_i(RuleName,Cat).
```

Lexical analysis has asserted clauses for the predicate:

```
lex:lex_analysis(FromState,ToState,SynCat,SemCat,Score,Words)
```

**Recognition.** The parser can now be given as follows. Note that the parser refers to the `robust/1` predicate. This predicate implements the robustness model. Note that for the other parsers the interface to the robustness component is defined in a similar fashion.

```
:- module(lc,[]).

parse_upto_robust :-    % recognition of all goals, anywhere.
        (  grammar:top_category(Cat),
           parse(Cat,_,_,_Ref),
```

```
            fail
        ;   true
        ).

parse(o(Cat,_,_)) :-
        parse_upto_robust,
        robust:robust(Cat).

parse(Cat,P0,P,Ref) :-
        weaken:weaken(Cat,Cat0),
        Cat0=..[F|Args],
        MemoCall =.. [F,P0|Args],
        MemoItem =.. [F,P0,PP|Args],
        (    item:a_more_general(table_goal:MemoCall)
        ->   true
        ;    ( lc_in:left_corner_table(Cat0,Small),
               predict(P0,Small,P1,His0),
               left_corner(Small,Cat0,P1,PP,His),
               add_item(MemoItem,His0,His,table_item),
               fail
            ; assertz(table_goal:MemoCall)
            )
        ),
        Cat=Cat0,
        P=PP,
        table_item:clause(MemoItem,_,Ref).

add_item(Item,His0,His,Module) :-
        item:add_item(Module:Item,IRef),
        IRef='$ref'(IRefA,IRefB),
        (  Module:'MEMO_HIS'(IRefA,IRefB,His0,His)
        -> true
        ;  Module:assertz('MEMO_HIS'(IRefA,IRefB,His0,His))
        ).

predict(P,Small,P,gap(Name)) :-
        lc_in:gap(Small,Name).

predict(P0,Small,P,lex(Ref)) :-
        lex:clause(lex_analysis(P0,P,Small,_,_Score,_),_,Ref).

left_corner(X,X,Y,Y,[]).
left_corner(Small,Goal,Q0,Q,[rule(Name,Rhis)|His]) :-
        lc_in:lc_rule(Small,Mid,Righties,Name),
        lc_in:left_corner_table(Goal,Mid),
        parse_right_ds(Righties,Q0,Q1,Rhis),
        left_corner(Mid,Goal,Q1,Q,His).

parse_right_ds([],L,L,[]).
parse_right_ds([H|T],L0,L,[Ref|His]):-
        parse(H,L0,L1,Ref),
        parse_right_ds(T,L1,L,His).
```

**Second phase.** During the construction of the best path through the word-graph this robustness procedure uses the `max_category/5` predicate. This recovers the derivations found in the recognition phase, and adds the semantic constraints.

```
max_category(P0,P,Cat,Conf,Words) :-
        grammar:top_category(Cat),
        second_phase(P0,P,Cat,Conf,Words,table_item).

second_phase(P0,P,Cat,Conf,Words,Module) :-
        Cat =.. [F|Args],
        Left =.. [F,P0,P|Args],
        Module:clause(Left,_Body,Ref),
        parse_2nd(Ref,Cat,P0,P,Conf,Words,[],Module).

parse_2nd('$ref'(Refa,Refb),Cat,P0,P,Sc,RWds0,RWds,Module) :-
        (  Module:reconstructed(Refa,Refb)
        -> true
        ;  Module:assertz(reconstructed(Refa,Refb)),
           (  Module:'MEMO_HIS'(Refa,Refb,Seed,His),
              predict_2nd(Seed,Small,P0,P1,Sc00,Wds0,Wds1),
              left_corner_2nd(His,Small,Cat0,P1,P,Sc00,Sc0,
                              Wds1,Wds,Module),
              assertz(Module:reconstructed(Refa,Refb,P0,P,Cat0,Sc0,Wds0,Wds)),
              fail
           ;  true
           )
        ),
        Module:reconstructed(Refa,Refb,P0,P,Cat,Sc,RWds0,RWds).

% 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),_,LexRef),
        lists:append(Wds1,Wds,Wds0).

predict_2nd(gap(Name),Small,Q,Q,0,Ws,Ws) :-
        lc_in:gap_i(Name,Small).

left_corner_2nd([],C,C,P,P,S,S,W,W,_).
left_corner_2nd([rule(Name,Rhis)|His],Small,Goal,Q0,Q,
                                            S0,S,W0,W,Module) :-
        lc_in:lc_rule_i(Name,Small,Mid,Rds),
        parse_rds_2nd(Rhis,Rds,Q0,Q1,S0,S1,W0,W1,Module),
        left_corner_2nd(His,Mid,Goal,Q1,Q,S1,S,W1,W,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),
        S2 is S0+S1,
        parse_rds_2nd(T,Td,P1,P,S2,S,W1,W,Module).
```

**Other libraries.** The modules `terms` and `lists` are standard SICStus 3 libraries. The `item` module is defined as:

```
:- module(item, [ a_more_specific/1,
```

```
                 a_more_specific/2,
                 a_more_general/1,
                 a_more_general/2,
                 add_item/2,
                 up_to_date/5
               ]).

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
        terms: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),
        terms:subsumes_chk(General,Item).

add_item(Module:Item,Ref) :-
        (   a_more_general(Module:Item,Ref)
        ->  true
        ;   Module:assertz(Item,Ref)
        ).
```

## 3.2 Efficient Tabular LR Parsing

We give a new treatment of tabular LR parsing, which is an alternative to Tomita's generalized LR algorithm. The advantage is twofold. Firstly, our treatment is conceptually more attractive because it uses simpler concepts, such as grammar transformations and standard tabulation techniques also know as *chart parsing*. Secondly, the static and dynamic complexity of parsing, both in space and time, is significantly reduced. We discuss the implications for context-free parsing as in (Nederhof and Satta, 1996), and further we indicate how these ideas may be applied to parsing with unification grammars such as the one we use for the OVIS project.

Giorgio Satta, of Dipartimento di Elettronica ed Informatica, Università di Padova, is coauthor for this section.

### 3.2.1 Introduction

The efficiency of LR($k$) parsing techniques (Sippu and Soisalon-Soininen, 1990) is very attractive from the perspective of natural language processing applications. This has stimulated the computational linguistics community to develop extensions of these techniques to general context-free grammar parsing. The best-known example is *generalized* LR parsing, also known as Tomita's algorithm, described by Tomita (1986) and further investigated by, for example, Tomita (1991) and Nederhof (1994a). Despite appearances, the graph-structured stacks used to describe Tomita's algorithm differ very little from *parse tables*, or in other words, generalized LR parsing is one of the so called *tabular* parsing algorithms, among which also the CYK algorithm (Harrison, 1978) and Earley's algorithm (Earley, 1970) can be found. (Tabular parsing is also known as *chart parsing*.)

Here we investigate the extension of LR parsing to general context-free grammars from a more general viewpoint: tabular algorithms can often be described by the composition of two constructions. One example is given by Lang (1974) and Billot and Lang (1989): the construction of pushdown automata from grammars and the simulation of these automata by means of tabulation yield different tabular algorithms for different such constructions. Another example, on which our presentation is based, was first suggested by Leermakers (1989): a grammar is first transformed and then a standard tabular algorithm along with some filtering condition is applied using the transformed grammar. In our case, the transformation and the subsequent application of the tabular algorithm result in a new form of tabular LR parsing.

Our method is more efficient than Tomita's algorithm in two respects. First, reduce operations are implemented in an efficient way, by splitting them into several, more primitive, operations (a similar idea has been proposed by Kipps (1991) for Tomita's algorithm). Second, several paths in the computation that must be simulated separately by Tomita's algorithm are collapsed into a single computation path, using state minimization techniques. Experiments on practical grammars have indicated that there is a significant gain in efficiency, with regard to both space and time requirements.

Our grammar transformation produces a so called *cover* for the input grammar, which together with the filtering condition fully captures the specification of the method, abstracting away from algorithmic details such as data structures and control flow. Since this cover can be easily precomputed, implementing our LR parser simply amounts to running the standard tabular algorithm. This is very attractive from an application-oriented perspective, since many actual systems for natural language processing are based on these kinds of parsing algorithm.

The remainder is organized as follows. In Section 3.2.2 some preliminaries are discussed. We review the notion of LR automaton in Section 3.2.3 and introduce the notion of 2LR automaton in Section 3.2.4. Then we specify our tabular LR method in Section 3.2.5, and provide an analysis of the algorithm in Section 3.2.6. Some empirical results are given in Section 3.2.7, and our practical experiences with the application of the ideas to the OVIS grammar are discussed in Section 3.2.8. Further discussion of our method is provided in Section 3.2.9.

### 3.2.2 Definitions

We use standard formal language notation. We assume that the reader is familiar with context-free grammar parsing theory (Harrison, 1978).

A context-free grammar (CFG) is a 4-tuple $G = (\Sigma, N, P, S)$, where $\Sigma$ and $N$ are two finite disjoint sets of terminal and nonterminal symbols, respectively, $S \in N$ is the start symbol, and $P$ is a finite set of rules. Each rule has the form $A \rightarrow \alpha$ with $A \in N$ and $\alpha \in V^*$, where $V$ denotes $N \cup \Sigma$. The size of $G$, written $|G|$, is defined as $\sum_{(A\rightarrow\alpha)\in P} |A\alpha|$; by $|\alpha|$ we mean the length of a string of symbols $\alpha$.

We generally use symbols $A, B, C, \ldots$ to range over $N$, symbols $a, b, c, \ldots$ to range over $\Sigma$, symbols $X, Y, Z$ to range over $V$, symbols $\alpha, \beta, \gamma, \ldots$ to range over $V^*$, and symbols $v, w, x, \ldots$ to range over $\Sigma^*$. We write $\epsilon$ to denote the empty string.

A CFG is said to be in *binary form* if $\alpha \in \{\epsilon\} \cup V \cup N^2$ for all of its rules $A \rightarrow \alpha$. (The binary form does not limit the (weak) generative capacity of context-free grammars (Harrison, 1978).) For technical reasons, we sometimes use the *augmented* grammar associated with $G$, defined as $G^\dagger = (\Sigma^\dagger, N^\dagger, P^\dagger, S^\dagger)$, where $S^\dagger$, $\triangleright$ and $\triangleleft$ are fresh symbols, $\Sigma^\dagger = \Sigma \cup \{\triangleright, \triangleleft\}$, $N^\dagger = N \cup \{S^\dagger\}$ and $P^\dagger = P \cup \{S^\dagger \rightarrow \triangleright S \triangleleft\}$.

A pushdown automaton (PDA) is a 5-tuple $\mathcal{A} = (\Sigma, Q, T, q_{in}, q_{fin})$, where $\Sigma, Q$ and $T$ are finite sets of input symbols, stack symbols and transitions, respectively; $q_{in} \in Q$ is the initial stack symbol and $q_{fin} \in Q$ is the final stack symbol.[1] Each transition has the form $\delta_1 \overset{z}{\mapsto} \delta_2$, where $\delta_1, \delta_2 \in Q^*$,

[1]We dispense with the notion of *state*, traditionally incorporated in the definition of PDA. This does not affect the power of these devices, since states can be encoded within stack symbols and transitions.

$1 \leq |\delta_1|$, $1 \leq |\delta_2| \leq 2$, and $z = \epsilon$ or $z = a$. We generally use symbols $q, r, s, \ldots$ to range over $Q$, and the symbol $\delta$ to range over $Q^*$.

Consider a fixed input string $v \in \Sigma^*$. A *configuration* of the automaton is a pair $(\delta, w)$ consisting of a stack $\delta \in Q^*$ and the remaining input $w$, which is a suffix of the input string $v$. The rightmost symbol of $\delta$ represents the top of the stack. The *initial* configuration has the form $(q_{in}, v)$, where the stack is formed by the initial stack symbol. The *final* configuration has the form $(q_{in}\ q_{fin}, \epsilon)$, where the stack is formed by the final stack symbol stacked upon the initial stack symbol.

The application of a transition $\delta_1 \overset{z}{\mapsto} \delta_2$ is described as follows. If the top-most symbols of the stack are $\delta_1$, then these symbols may be replaced by $\delta_2$, provided that either $z = \epsilon$, or $z = a$ and $a$ is the first symbol of the remaining input. Furthermore, if $z = a$ then $a$ is removed from the remaining input. Formally, for a fixed PDA $\mathcal{A}$ we define the binary relation $\vdash$ on configurations as the least relation satisfying $(\delta\delta_1, w) \vdash (\delta\delta_2, w)$ if there is a transition $\delta_1 \overset{\epsilon}{\mapsto} \delta_2$, and $(\delta\delta_1, aw) \vdash (\delta\delta_2, w)$ if there is a transition $\delta_1 \overset{a}{\mapsto} \delta_2$. The recognition of a certain input $v$ is obtained if starting from the initial configuration for that input we can reach the final configuration by repeated application of transitions, or, formally, if $(q_{in}, v) \vdash^* (q_{in}\ q_{fin}, \epsilon)$, where $\vdash^*$ denotes the reflexive and transitive closure of $\vdash$.

By a *computation* of a PDA we mean a sequence $(q_{in}, v) \vdash (\delta_1, w_1) \vdash \ldots \vdash (\delta_n, w_n)$, $n \geq 0$. A PDA is called *deterministic* if for all possible configurations at most one transition is applicable. A PDA is said to be in *binary form* if, for all transitions $\delta_1 \overset{z}{\mapsto} \delta_2$, we have $|\delta_1| \leq 2$.

### 3.2.3 LR automata

Let $G = (\Sigma, N, P, S)$ be a CFG. We recall the notion of LR automaton, which is a particular kind of PDA. We make use of the augmented grammar $G^\dagger = (\Sigma^\dagger, N^\dagger, P^\dagger, S^\dagger)$ introduced in Section 3.2.2.

Let $I_{\mathrm{LR}} = \{A \to \alpha \bullet \beta \mid (A \to \alpha\beta) \in P^\dagger\}$. We introduce the function *closure* from $2^{I_{\mathrm{LR}}}$ to $2^{I_{\mathrm{LR}}}$ and the function *goto* from $2^{I_{\mathrm{LR}}} \times V$ to $2^{I_{\mathrm{LR}}}$. For any $q \subseteq I_{\mathrm{LR}}$, *closure*$(q)$ is the smallest set such that

(i) $q \subseteq$ *closure*$(q)$; and

(ii) $(B \to \alpha \bullet A\beta) \in$ *closure*$(q)$ and $(A \to \gamma) \in P^\dagger$ together imply $(A \to \bullet\ \gamma) \in$ *closure*$(q)$.

We then define

$$\begin{aligned} &\mathit{goto}(q, X) = \\ &\qquad \{A \to \alpha X \bullet \beta \mid (A \to \alpha \bullet X\beta) \in \mathit{closure}(q)\}. \end{aligned}$$

We construct a finite set $\mathcal{R}_{\mathrm{LR}}$ as the smallest collection of sets satisfying the conditions:

(i) $\{S^\dagger \to \rhd \bullet S\lhd\} \in \mathcal{R}_{\mathrm{LR}}$; and

(ii) for every $q \in \mathcal{R}_{\mathrm{LR}}$ and $X \in V$, we have *goto*$(q, X) \in \mathcal{R}_{\mathrm{LR}}$, provided *goto*$(q, X) \neq \emptyset$.

Two elements from $\mathcal{R}_{\mathrm{LR}}$ deserve special attention: $q_{in} = \{S^\dagger \to \rhd \bullet S\lhd\}$, and $q_{fin}$, which is defined to be the unique set in $\mathcal{R}_{\mathrm{LR}}$ containing $(S^\dagger \to \rhd S \bullet \lhd)$; in other words, $q_{fin} =$ *goto*$(q_{in}, S)$.

For $A \in N$, an $A$-*redex* is a string $q_0 q_1 q_2 \cdots q_m$, $m \geq 0$, of elements from $\mathcal{R}_{\mathrm{LR}}$, satisfying the following conditions:

(i) $(A \to \alpha \bullet) \in$ *closure*$(q_m)$, for some $\alpha = X_1 X_2 \cdots X_m$; and

(ii) *goto*$(q_{k-1}, X_k) = q_k$, for $1 \leq k \leq m$.

Note that in such an $A$-redex, $(A \to \bullet\ X_1 X_2 \cdots X_m) \in$ *closure*$(q_0)$, and $(A \to X_1 \cdots X_k \bullet X_{k+1} \cdots X_m) \in q_k$, for $0 < k \leq m$.

The LR automaton associated with $G$ is now introduced.

**Definition 1** *$\mathcal{A}_{\mathrm{LR}} = (\Sigma, Q_{\mathrm{LR}}, T_{\mathrm{LR}}, q_{in}, q_{fin})$, where $Q_{\mathrm{LR}} = \mathcal{R}_{\mathrm{LR}}$, $q_{in} = \{S^\dagger \to \rhd \bullet S\lhd\}$, $q_{fin} =$ goto$(q_{in}, S)$, and $T_{\mathrm{LR}}$ contains:*

(i) $q \overset{a}{\longmapsto} q\ q'$, *for every* $a \in \Sigma$ *and* $q, q' \in \mathcal{R}_{\mathrm{LR}}$ *such that* $q' = goto(q, a)$;

(ii) $q\delta \overset{\epsilon}{\longmapsto} q\ q'$, *for every* $A \in N$, $A$*-redex* $q\delta$, *and* $q' \in \mathcal{R}_{\mathrm{LR}}$ *such that* $q' = goto(q, A)$.

Transitions in (i) above are called *shift*, transitions in (ii) are called *reduce*.

### 3.2.4 2LR Automata

The automata $\mathcal{A}_{\mathrm{LR}}$ defined in the previous section are deterministic only for a subset of the $\mathrm{CFG}$s, called the LR$(0)$ grammars (Sippu and Soisalon-Soininen, 1990), and behave nondeterministically in the general case. When designing tabular methods that simulate nondeterministic computations of $\mathcal{A}_{\mathrm{LR}}$, two main difficulties are encountered:

- A reduce transition in $\mathcal{A}_{\mathrm{LR}}$ is an elementary operation that removes from the stack a number of elements bounded by the size of the underlying grammar. Consequently, the time requirement of tabular simulation of $\mathcal{A}_{\mathrm{LR}}$ computations can be onerous, for reasons pointed out by Sheil (1976) and Kipps (1991).
- The set $\mathcal{R}_{\mathrm{LR}}$ can be exponential in the size of the grammar (Johnson, 1991). If in such a case the computations of $\mathcal{A}_{\mathrm{LR}}$ touch upon each state, then time and space requirements of tabular simulation are obviously onerous.

The first issue above is solved here by recasting $\mathcal{A}_{\mathrm{LR}}$ in binary form. This is done by considering each reduce transition as a sequence of "pop" operations which affect at most two stack symbols at a time. (See also Lang (1974), Villemonte de la Clergerie (1993) and Nederhof (1994a), and for LR parsing specifically Kipps (1991) and Leermakers (1992b).) The following definition introduces this new kind of automaton.

**Definition 2** $\mathcal{A}'_{\mathrm{LR}} = (\Sigma, Q'_{\mathrm{LR}}, T'_{\mathrm{LR}}, q_{in}, q_{fin})$, *where* $Q'_{\mathrm{LR}} = \mathcal{R}_{\mathrm{LR}} \cup I_{\mathrm{LR}}$, $q_{in} = \{S^\dagger \rightarrow \triangleright \bullet S \triangleleft\}$, $q_{fin} = goto(q_{in}, S)$ *and* $T'_{\mathrm{LR}}$ *contains:*

(i) $q \overset{a}{\longmapsto} q\ q'$, *for every* $a \in \Sigma$ *and* $q, q' \in \mathcal{R}_{\mathrm{LR}}$ *such that* $q' = goto(q, a)$;

(ii) $q \overset{\epsilon}{\longmapsto} q\ (A \rightarrow \alpha\ \bullet)$, *for every* $q \in \mathcal{R}_{\mathrm{LR}}$ *and* $(A \rightarrow \alpha\ \bullet) \in closure(q)$;

(iii) $q\ (A \rightarrow \alpha X \bullet \beta) \overset{\epsilon}{\longmapsto} (A \rightarrow \alpha \bullet X\beta)$, *for every* $q \in \mathcal{R}_{\mathrm{LR}}$ *and* $(A \rightarrow \alpha X \bullet \beta) \in q$;

(iv) $q\ (A \rightarrow \bullet\ \alpha) \overset{\epsilon}{\longmapsto} q\ q'$, *for every* $q, q' \in \mathcal{R}_{\mathrm{LR}}$ *and* $(A \rightarrow \alpha) \in P^\dagger$ *such that* $q' = goto(q, A)$.

Transitions in (i) above are again called *shift*, transitions in (ii) are called *initiate*, those in (iii) are called *gathering*, and transitions in (iv) are called *goto*. The role of a reduce step in $\mathcal{A}_{\mathrm{LR}}$ is taken over in $\mathcal{A}'_{\mathrm{LR}}$ by an initiate step, a number of gathering steps, and a goto step. Observe that these steps involve the new stack symbols $(A \rightarrow \alpha \bullet \beta) \in I_{\mathrm{LR}}$ that are distinguishable from possible stack symbols $\{A \rightarrow \alpha \bullet \beta\} \in \mathcal{R}_{\mathrm{LR}}$.

We now turn to the second above-mentioned problem, regarding the size of set $\mathcal{R}_{\mathrm{LR}}$. The problem is in part solved here as follows. The number of states in $\mathcal{R}_{\mathrm{LR}}$ is considerably reduced by identifying two states if they become identical after items $A \rightarrow \alpha \bullet \beta$ from $I_{\mathrm{LR}}$ have been simplified to only the suffix of the right-hand side $\beta$. This is reminiscent of techniques of state minimization for finite automata (Booth, 1967), as they have been applied before to LR parsing, e.g., by Pager (1970) and Nederhof and Sarbo (1993).

Let $G^\dagger$ be the augmented grammar associated with a $\mathrm{CFG}$ $G$, and let $I_{2\mathrm{LR}} = \{\beta \mid (A \rightarrow \alpha\beta) \in P^\dagger\}$. We define variants of the *closure* and *goto* functions from the previous section as follows. For any set $q \subseteq I_{2\mathrm{LR}}$, *closure*$'(q)$ is the smallest collection of sets such that

(i) $q \subseteq$ *closure*$'(q)$; and

(ii) $(A\beta) \in$ *closure*$'(q)$ and $(A \rightarrow \gamma) \in P^\dagger$ together imply $(\gamma) \in$ *closure*$'(q)$.

Also, we define

$$goto'(q, X) \quad = \quad \{\beta \mid (X\beta) \in closure'(q)\}.$$

We now construct a finite set $\mathcal{R}_{2\mathrm{LR}}$ as the smallest set satisfying the conditions:

(i) $\{S\triangleleft\} \in \mathcal{R}_{2\mathrm{LR}}$; and

(ii) for every $q \in \mathcal{R}_{2\mathrm{LR}}$ and $X \in V$, we have $goto'(q, X) \in \mathcal{R}_{2\mathrm{LR}}$, provided $goto'(q, X) \neq \emptyset$.

As stack symbols, we take the elements from $I_{2\mathrm{LR}}$ and a subset of elements from $(V \times \mathcal{R}_{2\mathrm{LR}})$:

$$Q_{2\mathrm{LR}} \quad = \quad \{(X, q) \mid \exists q'[goto'(q', X) = q]\} \cup I_{2\mathrm{LR}}$$

In a stack symbol of the form $(X, q)$, the $X$ serves to record the grammar symbol that has been recognized last, cf. the symbols that formerly were found immediately before the dots.

The 2LR automaton associated with $G$ can now be introduced.

**Definition 3** *$\mathcal{A}_{2\mathrm{LR}} = (\Sigma, Q_{2\mathrm{LR}}, T_{2\mathrm{LR}}, q'_{in}, q'_{fin})$, where $Q_{2\mathrm{LR}}$ is as defined above, $q'_{in} = (\triangleright, \{S\triangleleft\})$, $q'_{fin} = (S, goto'(\{S\triangleleft\}, S))$, and $T_{2\mathrm{LR}}$ contains:*

(i) $(X, q) \overset{a}{\longmapsto} (X, q)\ (a, q')$, *for every $a \in \Sigma$ and $(X, q), (a, q') \in Q_{2\mathrm{LR}}$ such that $q' = goto'(q, a)$;*

(ii) $(X, q) \overset{\epsilon}{\longmapsto} (X, q)\ (\epsilon)$, *for every $(X, q) \in Q_{2\mathrm{LR}}$ such that $\epsilon \in closure'(q)$;*

(iii) $(X, q)\ (\beta) \overset{\epsilon}{\longmapsto} (X\beta)$, *for every $(X, q) \in Q_{2\mathrm{LR}}$ and $\beta \in q$;*

(iv) $(X, q)\ (\alpha) \overset{\epsilon}{\longmapsto} (X, q)\ (A, q')$, *for every $(X, q)$, $(A, q') \in Q_{2\mathrm{LR}}$ and $(A \rightarrow \alpha) \in P^{\dagger}$ such that $q' = goto'(q, A)$.*

Note that in the case of a reduce/reduce conflict with two grammar rules sharing some suffix in the right-hand side, the gathering steps of $\mathcal{A}_{2\mathrm{LR}}$ will treat both rules simultaneously, until the parts of the right-hand sides are reached where the two rules differ. (See Leermakers (1992a) for a similar sharing of computation for common suffixes.)

An interesting fact is that the automaton $\mathcal{A}_{2\mathrm{LR}}$ is very similar to the automaton $\mathcal{A}_{\mathrm{LR}}$ constructed for a grammar transformed by the transformation $\tau_{two}$ given by Nederhof and Satta (1994).[2]

### 3.2.5 The algorithm

This section presents a tabular LR parser, which is our main result. The parser is derived from the 2LR automata introduced in the previous section. Following the general approach presented by Leermakers (1989), we simulate computations of these devices using a tabular method, a grammar transformation and a filtering function.

We make use of a tabular parsing algorithm which is basically an asynchronous version of the CYK algorithm, as presented by Harrison (1978), extended to productions of the forms $A \rightarrow B$ and $A \rightarrow \epsilon$ and with a left-to-right filtering condition. The algorithm uses a parse table consisting in a 0-indexed square array $U$. The indices represent positions in the input string. We define $U_i$ to be $\bigcup_{k \leq i} U_{k,i}$.

Computation of the entries of $U$ is moderated by a filtering process. This process makes use of a function *pred* from $2^N$ to $2^N$, specific to a certain context-free grammar. We have a certain nonterminal $A_{init}$ which is initially inserted in $U_{0,0}$ in order to start the recognition process.

We are now ready to give a formal specification of the tabular algorithm.

**Algorithm 1** Let $G = (\Sigma, N, P, S)$ be a CFG in binary form, let *pred* be a function from $2^N$ to $2^N$, let $A_{init}$ be the distinguished element from $N$, and let $v = a_1 a_2 \cdots a_n \in \Sigma^*$ be an input string. We compute the least $(n+1) \times (n+1)$ table $U$ such that $A_{init} \in U_{0,0}$ and

[2] For the earliest mention of this transformation, we have encountered pointers to Schauerte (1973). Regrettably, we have as yet not been able to get hold of a copy of this paper.

(i) $A \in U_{j-1,j}$
if $(A \rightarrow a_j) \in P$, $A \in pred(U_{j-1})$;

(ii) $A \in U_{j,j}$
if $(A \rightarrow \epsilon) \in P$, $A \in pred(U_j)$;

(iii) $A \in U_{i,j}$
if $B \in U_{i,k}$, $C \in U_{k,j}$, $(A \rightarrow BC) \in P$, $A \in pred(U_i)$;

(iv) $A \in U_{i,j}$
if $B \in U_{i,j}$, $(A \rightarrow B) \in P$, $A \in pred(U_i)$.

The string has been accepted when $S \in U_{0,n}$.

We now specify a grammar transformation, based on the definition of $\mathcal{A}_{2\mathrm{LR}}$.

**Definition 4** *Let* $\mathcal{A}_{2\mathrm{LR}} = (\Sigma, Q_{2\mathrm{LR}}, T_{2\mathrm{LR}}, q'_{in}, q'_{fin})$ *be the* 2LR *automaton associated with a* CFG $G$. *The* **2LR cover** *associated with* $G$ *is the* CFG $\mathcal{C}_{2\mathrm{LR}}(G) = (\Sigma, Q_{2\mathrm{LR}}, P_{2\mathrm{LR}}, q'_{fin})$*, where the rules in* $P_{2\mathrm{LR}}$ *are given by:*

(i) $(a, q') \rightarrow a$,
*for every* $(X, q) \stackrel{a}{\mapsto} (X, q)\ (a, q') \in T_{2\mathrm{LR}}$*;*

(ii) $(\epsilon) \rightarrow \epsilon$,
*for every* $(X, q) \stackrel{\epsilon}{\mapsto} (X, q)\ (\epsilon) \in T_{2\mathrm{LR}}$*;*

(iii) $(X\beta) \rightarrow (X, q)\ (\beta)$,
*for every* $(X, q)\ (\beta) \stackrel{\epsilon}{\mapsto} (X\beta) \in T_{2\mathrm{LR}}$*;*

(iv) $(A, q') \rightarrow (\alpha)$,
*for every* $(X, q)\ (\alpha) \stackrel{\epsilon}{\mapsto} (X, q)\ (A, q') \in T_{2\mathrm{LR}}$.

Observe that there is a direct, one-to-one correspondence between transitions of $\mathcal{A}_{2\mathrm{LR}}$ and productions of $\mathcal{C}_{2\mathrm{LR}}(G)$.

The accompanying function *pred* is defined as follows ($q$, $q'$, $q''$ range over the stack elements):

$$\begin{aligned} pred(\tau) \quad = \quad & \{q \mid q'q'' \stackrel{\epsilon}{\mapsto} q \in T_{2\mathrm{LR}}\} \ \cup \\ & \{q \mid q' \in \tau,\ q' \stackrel{z}{\mapsto} q'\ q \in T_{2\mathrm{LR}}\} \ \cup \\ & \{q \mid q' \in \tau,\ q'\ q'' \stackrel{\epsilon}{\mapsto} q'\ q \in T_{2\mathrm{LR}}\}. \end{aligned}$$

The above definition implies that only the tabular equivalents of the shift, initiate and goto transitions are subject to actual filtering; the simulation of the gathering transitions does not depend on elements in $\tau$.

Finally, the distinguished nonterminal from the cover used to initialize the table is $q'_{in}$. Thus we start with $(\triangleright, \{S\triangleleft\}) \in U_{0,0}$.

The 2LR cover introduces spurious ambiguity: where some grammar $G$ would allow a certain number of parses to be found for a certain input, the grammar $\mathcal{C}_{2\mathrm{LR}}(G)$ in general allows *more* parses. This problem is in part solved by the filtering function *pred*. The remaining spurious ambiguity is avoided by a particular way of constructing the parse trees, described in what follows.

After Algorithm 1 has recognized a given input, the set of all parse trees can be computed as $tree(q'_{fin}, 0, n)$ where the function $tree$, which determines sets of either parse trees or lists of parse trees for entries in $U$, is recursively defined by:

(i) $tree((a, q'), i, j)$ is the set $\{a\}$. This set contains a single parse tree consisting of a single node labelled $a$.

(ii) $tree(\epsilon, i, i)$ is the set $\{\epsilon\}$. This set consists of an empty list of trees.

(iii) $tree(X\beta, i, j)$ is the union of the sets $\mathcal{T}^k_{(X\beta),i,j}$, where $i \leq k \leq j$, $(\beta) \in U_{k,j}$, and there is at least one $(X, q) \in U_{i,k}$ and $(X\beta) \to (X, q)\ (\beta)$ in $\mathcal{C}_{2\mathrm{LR}}(G)$, for some $q$. For each such $k$, select one such $q$. We define $\mathcal{T}^k_{(X\beta),i,j} = \{t \cdot ts \mid t \in tree((X, q), i, k) \wedge ts \in tree(\beta, k, j)\}$. Each $t \cdot ts$ is a list of trees, with head $t$ and tail $ts$.

(iv) $tree((A, q'), i, j)$ is the union of the sets $\mathcal{T}^\alpha_{(A,q'),i,j}$, where $(\alpha) \in U_{i,j}$ is such that $(A, q') \to (\alpha)$ in $\mathcal{C}_{2\mathrm{LR}}(G)$. We define $\mathcal{T}^\alpha_{(A,q'),i,j} = \{glue(A, ts) \mid ts \in tree(\alpha, i, j)\}$. The function $glue$ constructs a tree from a fresh root node labelled $A$ and the trees in list $ts$ as immediate subtrees.

We emphasize that in the third clause above, one should not consider more than one $q$ for given $k$ in order to prevent spurious ambiguity. (In fact, for fixed $X, i, k$ and for different $q$ such that $(X, q) \in U_{i,k}$, $tree((X, q), i, k)$ yields the exact same set of trees.) With this proviso, the degree of ambiguity, i.e. the number of parses found by the algorithm for any input, is reduced to exactly that of the source grammar.

A practical implementation would construct the parse trees on-the-fly, attaching them to the table entries, allowing packing and sharing of subtrees (cf. the literature on *parse forests* (Tomita, 1986; Billot and Lang, 1989)). Our algorithm actually only needs one (packed) subtree for several $(X, q) \in U_{i,k}$ with fixed $X, i, k$ but different $q$. The resulting parse forests would then be optimally compact, contrary to some other LR-based tabular algorithms, as pointed out by Rekers (1992), Nederhof (1993) and Nederhof (1994b).

### 3.2.6 Analysis of the algorithm

In this section, we investigate how the steps performed by Algorithm 1 (applied to the 2LR cover) relate to those performed by $\mathcal{A}_{2\mathrm{LR}}$, for the same input.

We define a subrelation $\models^+$ of $\vdash^+$ as: $(\delta, uw) \models^+ (\delta\delta', w)$ if and only if $(\delta, uw) = (\delta, z_1 z_2 \cdots z_m w) \vdash (\delta\delta_1, z_2 \cdots z_m w) \vdash \ldots \vdash (\delta\delta_m, w) = (\delta\delta', w)$, for some $m \geq 1$, where $|\delta_k| > 0$ for all $k$, $1 \leq k \leq m$. Informally, we have $(\delta, uw) \models^+ (\delta\delta', w)$ if configuration $(\delta\delta', w)$ can be reached from $(\delta, uw)$ without the bottom-most part $\delta$ of the intermediate stacks being affected by any of the transitions; furthermore, at least one element is pushed on top of $\delta$.

The following characterization relates the automaton $\mathcal{A}_{2\mathrm{LR}}$ and Algorithm 1 applied to the 2LR cover. Symbol $q \in Q_{2\mathrm{LR}}$ is eventually added to $U_{i,j}$ if and only if for some $\delta$:

$$(q'_{in}, a_1 \ldots a_n) \vdash^* (\delta, a_{i+1} \ldots a_n) \models^+ (\delta q, a_{j+1} \ldots a_n).$$

In words, $q$ is found in entry $U_{i,j}$ if and only if, at input position $j$, the automaton would push some element $q$ on top of some lower-part of the stack $\delta$ that remains unaffected while the input from $i$ to $j$ is being read.

The above characterization, whose proof is not reported here, is the justification for calling the resulting algorithm tabular LR parsing. In particular, for a grammar for which $\mathcal{A}_{2\mathrm{LR}}$ is deterministic, i.e. for an LR(0) grammar, the number of steps performed by $\mathcal{A}_{2\mathrm{LR}}$ and the number of steps performed by the above algorithm are exactly the same. In the case of grammars which are not LR(0), the tabular LR algorithm is more efficient than for example a backtrack realisation of $\mathcal{A}_{2\mathrm{LR}}$.

For determining the order of the time complexity of our algorithm, we look at the most expensive step, which is the computation of an element $(X\beta) \in U_{i,j}$ from two elements $(X, q) \in U_{i,k}$ and $(\beta) \in U_{k,j}$, through $(X, q)\ (\beta) \overset{\epsilon}{\mapsto} (X\beta) \in T_{2\mathrm{LR}}$. In a straightforward realisation of the algorithm, this step can be applied $\mathcal{O}(|T_{2\mathrm{LR}}| \cdot |v|^3)$ times (once for each $i, k, j$ and each transition), each step taking a constant amount of time. We conclude that the time complexity of our algorithm is $\mathcal{O}(|T_{2\mathrm{LR}}| \cdot |v|^3)$.

As far as space requirements are concerned, each set $U_{i,j}$ or $U_i$ contains at most $|Q_{2\mathrm{LR}}|$ elements. (One may assume an auxiliary table storing each $U_i$.) This results in a space complexity $\mathcal{O}(|Q_{2\mathrm{LR}}| \cdot |v|^2)$.

The entries in the table represent single stack elements, as opposed to pairs of stack elements following Lang (1974) and Leermakers (1989). This has been investigated before by Nederhof (1994a, p. 25) and Villemonte de la Clergerie (1993, p. 155).

| $G = (\Sigma, N, P, S)$ | $\lvert G\rvert$ | $\lvert N\rvert$ | $\lvert P\rvert$ | $\lvert w\rvert$ |
|---|---|---|---|---|
| ALGOL 68 | 783 | 167 | 330 | 13.7 |
| CORRie | 1141 | 203 | 424 | 12.3 |
| Deltra | 1929 | 281 | 703 | 10.8 |
| Alvey | 5072 | 265 | 1484 | 10.7 |

Table 3.1: The test material: the four grammars and some of their dimensions, and the average length of the test sentences (20 sentences of various length for each grammar).

### 3.2.7 Empirical results

We have performed some experiments with Algorithm 1 applied to $\mathcal{A}_{2\mathrm{LR}}$ and $\mathcal{A}'_{\mathrm{LR}}$, for 4 practical context-free grammars. For $\mathcal{A}'_{\mathrm{LR}}$ a cover was used analogous to the one in Definition 4; the filtering function remains the same.

The first grammar generates a subset of the programming language ALGOL 68 (van Wijngaarden and others, 1975). The second and third grammars generate a fragment of Dutch, and are referred to as the CORRie grammar (Vosse, 1994) and the Deltra grammar (Schoorl and Belder, 1990), respectively. These grammars were stripped of their arguments in order to convert them into context-free grammars. The fourth grammar, referred to as the Alvey grammar (Carroll, 1993), generates a fragment of English and was automatically generated from a unification-based grammar.

The test sentences have been obtained by automatic generation from the grammars, using the Grammar Workbench (Nederhof and Koster, 1992), which uses a random generator to select rules; therefore these sentences do not necessarily represent input typical of the applications for which the grammars were written. Table 3.1 summarizes the test material.

Our implementation is merely a prototype, which means that absolute duration of the parsing process is little indicative of the actual efficiency of more sophisticated implementations. Therefore, our measurements have been restricted to implementation-independent quantities, viz. the number of elements stored in the parse table and the number of elementary steps performed by the algorithm. In a practical implementation, such quantities will strongly influence the space and time complexity, although they do not represent the only determining factors. Furthermore, all optimizations of the time and space efficiency have been left out of consideration.

Table 3.2 presents the costs of parsing the test sentences. The first and third columns give the number of entries stored in table $U$, the second and fourth columns give the number of elementary steps that were performed.

An elementary step consists of the derivation of one element in $Q'_{\mathrm{LR}}$ or $Q_{2\mathrm{LR}}$ from one or two other elements. The elements that are used in the filtering process are counted individually. We give an example for the case of $\mathcal{A}'_{\mathrm{LR}}$. Suppose we derive an element $q' \in U_{i,j}$ from an element $(A \rightarrow \bullet\ \alpha) \in U_{i,j}$, warranted by two elements $q_1, q_2 \in U_i$, $q_1 \neq q_2$, through *pred*, in the presence of $q_1\ (A \rightarrow \bullet\ \alpha) \overset{\epsilon}{\mapsto} q_1\ q' \in T'_{\mathrm{LR}}$ and $q_2\ (A \rightarrow \bullet\ \alpha) \overset{\epsilon}{\mapsto} q_2\ q' \in T'_{\mathrm{LR}}$. We then count *two* parsing steps, one for $q_1$ and one for $q_2$.

Table 3.2 shows that there is a significant gain in space and time efficiency when moving from $\mathcal{A}'_{\mathrm{LR}}$ to $\mathcal{A}_{2\mathrm{LR}}$.

Apart from the dynamic costs of parsing, we have also measured some quantities relevant to the construction and storage of the two types of tabular LR parser. These data are given in Table 3.3.

We see that the number of states is strongly reduced with regard to traditional LR parsing. In the case of the Alvey grammar, moving from $\lvert\mathcal{R}_{\mathrm{LR}}\rvert$ to $\lvert\mathcal{R}_{2\mathrm{LR}}\rvert$ amounts to a reduction to 20.3 %. Whereas time- and space-efficient computation of $\mathcal{R}_{\mathrm{LR}}$ for this grammar is a serious problem, computation of $\mathcal{R}_{2\mathrm{LR}}$ will not be difficult on any modern computer. Also significant is the reduction from $\lvert T'_{\mathrm{LR}}\rvert$ to $\lvert T_{2\mathrm{LR}}\rvert$, especially for the larger grammars. These quantities correlate with the

| $G$ | $\mathcal{A}'_{\mathrm{LR}}$ space | time | $\mathcal{A}_{2\mathrm{LR}}$ space | time |
|---|---|---|---|---|
| ALGOL 68 | 327 | 375 | 234 | 343 |
| CORRie | 7548 | 28028 | 5131 | 22414 |
| Deltra | 11772 | 94824 | 6526 | 70333 |
| Alvey | 599 | 1147 | 354 | 747 |

Table 3.2: Dynamic requirements: average space and time per sentence.

| $G$ | $\mathcal{A}'_{\mathrm{LR}}$ $\lvert\mathcal{R}_{\mathrm{LR}}\rvert$ | $\lvert Q'_{\mathrm{LR}}\rvert$ | $\lvert T'_{\mathrm{LR}}\rvert$ | $\mathcal{A}_{2\mathrm{LR}}$ $\lvert\mathcal{R}_{2\mathrm{LR}}\rvert$ | $\lvert Q_{2\mathrm{LR}}\rvert$ | $\lvert T_{2\mathrm{LR}}\rvert$ |
|---|---|---|---|---|---|---|
| ALGOL 68 | 434 | 1,217 | 13,844 | 109 | 724 | 12,387 |
| CORRie | 600 | 1,741 | 22,129 | 185 | 821 | 15,569 |
| Deltra | 856 | 2,785 | 54,932 | 260 | 1,089 | 37,510 |
| Alvey | 3,712 | 8,784 | 1,862,492 | 753 | 3,065 | 537,852 |

Table 3.3: Static requirements.

amount of storage needed for naive representation of the respective automata.

### 3.2.8 Application to the OVIS grammar

In this section, we discuss a number of properties of our implementation of the LR algorithms that are specific for the OVIS grammar.

#### 3.2.8.1 The grammar

As we will explain later, there are certain advantages to having a grammar without *epsilon rules*, i.e. rules of the form $A \rightarrow \epsilon$. Therefore, the parser generator first eliminates all epsilon rules from the grammar. Our procedure is to repeat the following until there are no more epsilon rules:

- Take an epsilon rule $A \rightarrow \epsilon$.
- Repeat for each non-epsilon rule $B \rightarrow \beta$ presently in the grammar:
  - Repeat for each non-empty sublist $\beta'$ of $\beta$:
    - If different occurrences of $A \rightarrow \epsilon$ can be substituted for all categories of $\beta'$, then add the resulting rule (a modified *copy* of the original) to the grammar.
- Remove epsilon rule $A \rightarrow \epsilon$.

The definite-clause grammar for OVIS has 66 rules, 7 of which are epsilon rules. After elimination of the epsilon rules, there are 92 rules.

Note that one iteration of the above algorithm may introduce several new epsilon rules into the grammar (take $\beta' = \beta$), while eliminating just one. Consequently, this way of epsilon-rule elimination is not guaranteed to terminate for all grammars.

#### 3.2.8.2 The LR table

In unification grammars, we have no straightforward equivalent of the notion of "nonterminal". Instead we have complex categories that may be further instantiated by the interaction of different rules. One implication for the construction of an LR table is that it is not immediately clear what to choose as the second arguments of the function *goto* (or *goto'* in the case of $\mathcal{A}_{2\mathrm{LR}}$). One option would be to take categories that occur in the left-hand or right-hand sides of grammar rules.

The problem with this approach is that spurious ambiguity may result, in the sense that one parse may be computed in several ways: Say we have two categories $A$ and $A'$ from the grammar, and an LR state $q$, and assume that both *goto*$(q, A)$ and *goto*$(q, A')$ are defined and evaluate to different states. When a parse-time category $B$ is found which unifies with both $A$ and $A'$, then both *goto*$(q, A)$ and *goto*$(q, A')$ can be added to the parse table. This may in the end lead to *two* parses being found instead of one. Note that these two parses may be identical, except that the corresponding categories may be differently instantiated due to unification with either $A'$ or $A$.

To avoid this phenomenon, we make sure that every parse-time category $B$ unifies with exactly one of the categories $A$ for which we define entries *goto*$(q, A)$ in the parse table. We find the set $CATS$ of such $A$ as follows.

Let $CATS$ initially be the empty set. Then repeat the following procedure for each category $A$ occurring in the grammar:

- If $A$ does not unify with any category in $CATS$, then add $A$ to $CATS$.
- Otherwise:
  - (i) Let $A'$ be a category in $CATS$ with which $A$ unifies.
  - (ii) Let $A''$ be the most specific category that generalizes $A$ and $A'$. (In SICStus Prolog this operation is available as `term_subsumer/3`.)
  - (iii) Remove $A'$ from $CATS$.
  - (iv) Call the procedure recursively with $A''$ instead of $A$.

It is clear that this process terminates: At every recursive call, the size of $CATS$ shrinks by one category. It is also clear that after this process each category from the grammar and also each parse-time category unifies with exactly one category from $CATS$. That is because 1) for each category in the grammar we make sure there is a more general category in $CATS$, and 2) an invariant of the algorithm is that no two categories in $CATS$ can unify.

The reasoning is as follows: Suppose $B$ is a parse-time category. Then there must be a more general category $B'$ in the grammar, and due to 1) there must be an even more general $A$ in $CATS$. Suppose now that $B$ also unifies with a second $A'$ in $CATS$. Then $A$, which is more general than $B$, must also unify with $A'$, which is in contradiction with 2).

A different application of the set $CATS$ was described in (Briscoe and Carroll, 1993).

An elegant property of the algorithm is that the result is uniquely determined, i.e. that the order in which the categories from the grammar are input into $CATS$ does not affect the end result.

Another way of expressing this is that two sets $CATS_0$ and $CATS_1$ resulting from processing the categories from the grammar in two different orders must be equivalent. (As usual, two categories which are identical up to renaming of variables are considered to be equivalent.)

Before discussing the proof, first note that for every $A$ in $CATS_i$, $i = 0, 1$, we can make a binary tree corresponding to how that element originated from elements in intermediate stages of $CATS_i$ and from categories in the grammar; the leaves in this tree are the categories in the grammar. It can be said that $A$ *represents* those categories in the grammar. Note that a category which represents a set of categories from the grammar is more general than all of those categories.

It is sufficient to prove that for any category $A$ in $CATS_i$, $i = 0, 1$, there is a more general category in $CATS_{1-i}$. We generalize this to the hypothesis that for any category $A$ that occurs in intermediate stages in the computation of $CATS_i$ there is a more general category that occurs at some intermediate stage in the computation of $CATS_{1-i}$. This we prove by induction on the number of categories in the grammar that such categories $A$ represent.

The base step of this proof is easy: for such an $A$ representing a single category from the grammar there must be a corresponding category in $CATS_{1-i}$ that represents (amongst others) $A$, since all categories from the grammar are represented by some element from $CATS_{1-i}$.

| $\mathcal{A}'_{\mathrm{LR}}$ | | | $\mathcal{A}_{2\mathrm{LR}}$ | | |
|---|---|---|---|---|---|
| $\lvert\mathcal{R}_{\mathrm{LR}}\rvert$ | $\lvert Q'_{\mathrm{LR}}\rvert$ | $\lvert T'_{\mathrm{LR}}\rvert$ | $\lvert\mathcal{R}_{2\mathrm{LR}}\rvert$ | $\lvert Q_{2\mathrm{LR}}\rvert$ | $\lvert T_{2\mathrm{LR}}\rvert$ |
| 158 | 484 | 5121 | 51 | 234 | 3271 |

Table 3.4: Static requirements for the OVIS grammar.

For the induction step, take an $A$ that resulted from generalizing two categories $A_1$ and $A_2$ that unify. Obviously, $A_1$ and $A_2$ represent fewer categories from the grammar than $A$ and therefore we can apply the induction hypothesis: There are $A'_1$ and $A'_2$, more general than $A_1$ and $A_2$ respectively, that occur in intermediate stages in the computation of $CATS_{1-i}$. During this computation, it must happen at some point that a pair of elements $A''_1$ and $A''_2$, derived from $A'_1$ and $A'_2$, will exist at the same time in $CATS_{1-i}$. Naturally, $A''_1$ and $A''_2$ will be more general than $A'_1$ and $A'_2$, respectively. Therefore, $A''_1$ and $A''_2$ must also be more general than $A_1$ and $A_2$. This means that they unify to one another, and the most specific category $A''$ generalizing both will be more general than category $A$. This category $A''$ will replace $A''_1$ and $A''_2$ in $CATS_{1-i}$ during some step of the algorithm. This ends the proof.

At the moment, the grammar for OVIS results in a set $CATS$ of 50 categories.

Regarding the computation of the closure function, derivation of an item $A \rightarrow \bullet\ \gamma$ from an item $B \rightarrow \alpha\ \bullet\ A\beta$ is determined solely by whether $A$ and $B$ unify, where $A$ and $B$ are both categories occurring in the grammar, and the instantiations resulting from this unification do not affect the variables in the items $A \rightarrow \bullet\ \gamma$ as they are added to *closure*$(q)$. In other words, the parser generator does not make predictions about how categories may be further instantiated during actual parsing. Naturally, analogous statements hold for our construction of *closure*$'$.

#### 3.2.8.3 The push-down automata

After determination of $CATS$, and computation of $\mathcal{R}_{\mathrm{LR}}$, *closure* and *goto*, our parser generator produces a set of PDA transitions, which is similar to $\mathcal{A}'_{\mathrm{LR}}$. (If a certain flag is set differently, PDA transitions are produced corresponding rather to $\mathcal{A}_{2\mathrm{LR}}$, based on $\mathcal{R}_{2\mathrm{LR}}$, *closure*$'$ and *goto*$'$. In the sequel we will concentrate on $\mathcal{A}'_{\mathrm{LR}}$. We merely state that the case $\mathcal{A}_{2\mathrm{LR}}$ is treated analogously in the implementation.)

The stack elements manipulated by the PDA are pairs, of which the first element is either taken from $\mathcal{R}_{\mathrm{LR}}$ and then the second element is a category at the root of a piece of parse tree, or it is taken from $I_{\mathrm{LR}}$, i.e. it is of the form $A \rightarrow \alpha\ \bullet\ \beta$, and then the second element is a list of categories at the roots of pieces of parse tree corresponding to the categories in $\beta$.

Conceptually, one could say that parse trees are constructed bottom-up, in such a way that pieces of parse tree are connected to stack symbols, but actually we do not compute the parse trees themselves but merely the categories in the roots.

One important difference between the PDA that we actually use for OVIS and Definition 2 is that clause (ii) and one application of clause (iii) are merged into one new clause, which is given by:

(v) $q \overset{\epsilon}{\mapsto} (A \rightarrow \alpha\ \bullet\ X)$, for every $q \in \mathcal{R}_{\mathrm{LR}}$ and $(A \rightarrow \alpha X\ \bullet) \in q$;

(A similar modification was made to Definition 3.)

The advantage is that fewer steps will be performed by the PDA; a second advantage is discussed in the next section. A disadvantage is that this modification is not correctness-preserving when the grammar contains epsilon rules, since then an application of clause (ii) may need to be followed by an application of clause (iv). The way we deal with this is to eliminate all epsilon rules from the grammar before constructing the LR table (Section 3.2.8.1).

The dimensions of the modified forms of $\mathcal{A}'_{\mathrm{LR}}$ and $\mathcal{A}_{2\mathrm{LR}}$ for the OVIS grammar are given in Table 3.4. We see that as in the case of context-free grammars, discussed earlier, there is a significant improvement of the static requirements with regard to traditional LR parsing.

#### 3.2.8.4 The parser

In the case of unification grammars, the elements stored in the parse table are complex categories that may be fairly large. (In Prolog, such storage would typically be done using the predicate `assert`.) Therefore, the costs of parsing with unification grammars are largely determined by the number of such table elements. We have been able to restrict this number by keeping the elements from $I_{\mathrm{LR}}$ (or $I_{2\mathrm{LR}}$, respectively) in local variables instead of adding them to the table. For the other types of stack elements, we restrict usage of the agenda to a minimum.[3] All this is related to epsilon-rule elimination (Section 3.2.8.1) and the modification of the PDA (Section 3.2.8.3).

To illustrate the advantages of all this, we consider the structure that the parser would have if an ordinary construction such as Definition 2 were used. Six steps are needed then, which can be divided into three groups:

- Remove from the agenda some occurrence of $q \in \mathcal{R}_{\mathrm{LR}}$ in $U_j$ and apply the following:

  **1** For every $q \stackrel{a}{\mapsto} q\ q'$ such that $a$ is in the input at position $j+1$, put $q'$ in the table at $U_{j,j+1}$.

  **2** For every $q \stackrel{\epsilon}{\mapsto} q\ (A \to \alpha\ \bullet)$, put $(A \to \alpha\ \bullet)$ in the table at $U_{j,j}$.

  **3** For every $q\ (A \to \bullet\ \alpha) \stackrel{\epsilon}{\mapsto} q\ q'$ and $(A \to \bullet\ \alpha) \in U_{j,i}$, put $q'$ in the table at $U_{j,i}$.

- Remove from the agenda some occurrence of $(A \to \alpha\ \bullet\ \beta) \in I_{\mathrm{LR}}$ in $U_{i,j}$ and apply the following:

  **4** For every $q\ (A \to \alpha' X\ \bullet\ \beta) \stackrel{\epsilon}{\mapsto} (A \to \alpha'\ \bullet\ X\beta)$ and $q \in U_{k,i}$ such that $\alpha = \alpha' X$, put $(A \to \alpha'\ \bullet\ X\beta)$ in the table at $U_{k,j}$.

  **5** In the case $\alpha = \epsilon$, for every $q\ (A \to \bullet\ \beta) \stackrel{\epsilon}{\mapsto} q\ q'$ such that $q \in U_i$, put $q'$ in the table at $U_{i,j}$.

- Remove from the agenda some occurrence of $q \in \mathcal{R}_{\mathrm{LR}}$ in $U_{i,j}$ and apply the following:

  **6** For every $q\ (A \to \alpha X\ \bullet\ \beta) \stackrel{\epsilon}{\mapsto} (A \to \alpha\ \bullet\ X\beta)$ and $(A \to \alpha X\ \bullet\ \beta) \in U_{j,k}$, put $(A \to \alpha\ \bullet\ X\beta)$ in the table at $U_{i,k}$.

These steps are applied as long as there are elements in the agenda.

The phrases "put $q$ in the table at $U_{i,j}$" in the above can be refined to:

- If $q$ already occurs in $U_{i,j}$ then halt.
- Otherwise, let $U_{i,j}$ be $U_{i,j} \cup \{q\}$, and put this occurrence of $q$ in the agenda.
- If $q$ already occurs in $U_j$ then halt.
- Otherwise, let $U_j$ be $U_j \cup \{q\}$, and put this occurrence of $q$ in the agenda.

. Note that the parsing process is initiated by putting $\{S^\dagger \to \triangleright\ \bullet\ S\triangleleft\}$ in the table at $U_{0,0}$.

The phrases "put $(A \to \alpha\ \bullet\ \beta)$ in the table at $U_{i,j}$" can be refined to:

- If $(A \to \alpha\ \bullet\ \beta)$ already occurs in $U_{i,j}$ then halt.
- Otherwise, let $U_{i,j}$ be $U_{i,j} \cup \{(A \to \alpha\ \bullet\ \beta)\}$, and put this occurrence of $(A \to \alpha\ \bullet\ \beta)$ in the agenda.

Note that steps **3** and **5** above treat the same PDA transitions, and so do steps **4** and **6**. This may lead to the same work being done twice. One solution to solve this problem is the idea of *time stamps*: every element in the table is marked with its time of creation. One then prevents combination of older elements from the agenda with newer elements in the table, which obviously prevents pairs of elements from being combined twice.

[3] An *agenda* is a set (or queue, priority queue, stack, etc.) of elements in the parse table $U$ that are waiting to be combined with other table elements to produce new table elements.

We have implemented this idea, but it proved to be inferior to the following solution: We compute the table elements in a strict order: first those in $U_{0,0}$ and $U_0$, then those in $U_{i,1}$ and $U_1$, etc. Further, we need to make sure that if an element in $U_{i,j}$ or $U_j$ is derived from another element in some $U_{i',j}$ or $U_j$, then no other elements from any $U_{i'',j}$ or $U_j$ may be involved in the proviso of the step.

Investigation of the 6 steps above shows that this last condition is fulfilled when there are no table elements in $U_{j,j}$, for any $j$. This is made true by eliminating the epsilon rules from the grammar (Section 3.2.8.1) and modifying the PDA construction (Section 3.2.8.3). Instead of the 6 steps above we can then suffice with repeating the following:

- Remove from the agenda some occurrence of $q \in \mathcal{R}_{\mathrm{LR}}$ in $U_j$ and apply the following:

  **1** For every $q \overset{a}{\mapsto} q\ q'$ such that $a$ is in the input at position $j+1$, put $q'$ in the table at $U_{j,j+1}$.

- Remove from the agenda some occurrence of $(A \rightarrow \alpha \bullet \beta) \in I_{\mathrm{LR}}$ in $U_{i,j}$ and apply the following:

  **2** For every $q\ (A \rightarrow \alpha' X \bullet \beta) \overset{\epsilon}{\mapsto} (A \rightarrow \alpha' \bullet X\beta)$ and $q \in U_{k,i}$ such that $\alpha = \alpha' X$, put $(A \rightarrow \alpha' \bullet X\beta)$ in the table at $U_{k,j}$.

  **3** In the case $\alpha = \epsilon$, for every $q\ (A \rightarrow \bullet \beta) \overset{\epsilon}{\mapsto} q\ q'$ such that $q \in U_i$, put $q'$ in the table at $U_{i,j}$.

- Remove from the agenda some occurrence of $q \in \mathcal{R}_{\mathrm{LR}}$ in $U_{i,j}$ and apply the following:

  **4** For every $q \overset{\epsilon}{\mapsto} (A \rightarrow \alpha \bullet X)$ , put $(A \rightarrow \alpha \bullet X)$ in the table at $U_{i,j}$.

We have gone one step further and restructured the parser so that only elements resulting from step **1** are put in the agenda, the other elements are used immediately to derive yet other elements. One other modification is that we do not store any elements from $I_{\mathrm{LR}}$ in the table; this is done to restrict the use of the table and thereby to improve the time complexity. (For context-free parsing, this modification would result in an increase in time-complexity from cubic to $\mathcal{O}(n^{p+1})$, where $p$ is the length of the longest right-hand side in the grammar, as in the case of Tomita's algorithm (Tomita, 1986; Nederhof, 1994a).)

More specifically, the phrases "put $q$ in the table at $U_{i,j}$" in the above can be refined to:

- If $q$ already occurs in $U_{i,j}$ then halt.
- Let $U_{i,j}$ be $U_{i,j} \cup \{q\}$.
- If the occurrence of $q$ at $U_{i,j}$ resulted from step **1**, then put this occurrence of $q$ in the agenda and halt.
- Use step **4** above to derive new elements from $q$ at $U_{i,j}$.
- If $q$ already occurs in $U_j$ then halt.
- Let $U_j$ be $U_j \cup \{q\}$.
- Use step **1** above to derive new elements from $q$ at $U_j$.

.

The phrases "put $(A \rightarrow \alpha \bullet \beta)$ in the table at $U_{i,j}$" can be refined to simply:

- Use steps **2** and **3** above to derive new elements from $(A \rightarrow \alpha \bullet \beta)$ at $U_{i,j}$.

In other words, the elements from $I_{\mathrm{LR}}$ are not actually placed in the table; their existence is merely used as an intermediate step in finding elements in $\mathcal{R}_{\mathrm{LR}}$, which *are* placed in the table.

On the highest level, the algorithm we have implemented is given by:

- Place $\{S^\dagger \rightarrow \rhd \bullet S \lhd\}$ in $U_{0,0}$, and put this element in the agenda.

- For every input position, from $j = 0, \ldots, n$, repeat the following until there are no more elements in the agenda for that $j$:
  - Remove some $q \in U_{i,j}$, some $i$, from the agenda.
  - Use step **4** above to derive new elements from $q$ at $U_{i,j}$.
  - If $q$ already occurs in $U_j$ then halt.
  - Let $U_j$ be $U_j \cup \{q\}$.
  - Use step **1** above to derive new elements from $q$ at $U_j$.

For each newly derived table element we first make sure that no equivalent table element already exists before we add it to the table. Therefore, spurious ambiguity in the sense of adding to the table more than one copy of a category representing a single parse is avoided. Yet, it is still possible that the same subparse is found twice, in the sense of doing redundant work: In step **3** there may be several $q \in U_i$ which all warrant deriving $q' \in U_{i,j}$ from $(A \to \alpha \bullet \beta) \in U_{i,j}$. Therefore we have compiled this particular PDA step in such a way that we can apply the following instead:

**3'** In the case $\alpha = \epsilon$, for every $q'$ such that there is at least one $q\ (A \to \bullet\ \beta) \overset{\epsilon}{\mapsto} q\ q'$ such that $q \in U_i$, put $q'$ in the table at $U_{i,j}$.

This means that each such a $q'$ is derived only once from $(A \to \alpha \bullet \beta)$, although there may be several transitions of the form $q\ (A \to \bullet\ \beta) \overset{\epsilon}{\mapsto} q\ q'$ such that $q \in U_i$.

All these ideas have of course been generalized to word graphs as resulting from speech recognition (Oerder and Ney, 1993) instead of to linear input, which requires only little modification of the structure of the parser. One other modification is that we put $\{S^\dagger \to \rhd \bullet S \lhd\}$ in $U_{j,j}$ for any $j = 0, \ldots, n$. This causes the algorithm to start looking for any *substring* in the input that is a sentence according to the grammar.

The increase in efficiency from $\mathcal{A}_{\mathrm{LR}}$ to $\mathcal{A}_{2\mathrm{LR}}$ that was observed for practical context-free grammars in Section 3.2.7 is also present in the case of the OVIS grammar, as shown in Section 3.3.

### 3.2.9 Discussion

Our treatment of tabular LR parsing has two important advantages over the one by Tomita:

- It is conceptually simpler, because we make use of simple concepts such as a grammar transformation and the well-understood CYK algorithm, instead of a complicated mechanism working on graph-structured stacks.
- Our algorithm requires fewer LR states. This leads to faster parser generation, to smaller parsers, and to reduced time and space complexity of parsing itself.

The conceptual simplicity of our formulation of tabular LR parsing allows comparison with other tabular parsing techniques, such as Earley's algorithm (Earley, 1970) and tabular left-corner parsing (Nederhof, 1993), based on implementation-independent criteria. This is in contrast to experiments reported before (e.g. by Shann (1991)), which treated tabular LR parsing differently from the other techniques.

The reduced time and space complexities reported in the previous section pertain to the tabular realisation of two parsing techniques, expressed by the automata $\mathcal{A}'_{\mathrm{LR}}$ and $\mathcal{A}_{2\mathrm{LR}}$. The tabular realisation of the former automata is very close to a variant of Tomita's algorithm by Kipps (1991). The objective of our experiments was to show that the automata $\mathcal{A}_{2\mathrm{LR}}$ provide a better basis than $\mathcal{A}'_{\mathrm{LR}}$ for tabular LR parsing with regard to space and time complexity.

Parsing algorithms that are not based on the LR technique have however been left out of consideration.[4]

[4] As remarked before by Nederhof (1993), the algorithms by Schabes (1991) and Leermakers (1989) are not really related to LR parsing, although some notation used in these papers suggests otherwise.

Theoretical considerations (Leermakers, 1989; Schabes, 1991; Nederhof, 1994b) have suggested that for natural language parsing, LR-based techniques may not necessarily be superior to other parsing techniques, although convincing empirical data to this effect has never been shown. This issue is difficult to resolve because so much of the relative efficiency of the different parsing techniques depends on particular grammars and particular input, as well as on particular implementations of the techniques. We hope the conceptual framework presented here may at least partly alleviate this problem.

## 3.3 Experiments

In this section we compare the different parsers with respect to practical effiency for typical input, both in terms of cpu-time and memory usage. The parsers are compared with respect to a set of word-graphs (the expected input for the system), and also with respect to ordinary strings. [5]

The cpu-time and memory usage statistics that are reported on include lexical lookup, and the resources used by the robustness component. This introduces a small factor which is the same for each of the parsers compared. Lexical lookup is fast enough that we can safely ignore this. The resources taken up by the robustness component cannot easily be separated from the resources taken up by the parser because of the use of lazy evaluation of the second phase of the parser. Therefore, some of the differences in speed are somewhat larger than the timings below suggest.

We first report on a small experiment for each of the parsers described in the previous and current deliverable. Some of these parsers cannot treat all of the word-graphs of a larger set of examples because of memory problems. Therefore, we only compare the more successfull parsers on this larger test set.

The results seem to indicate that the parsers perform slower than in the preliminary experiments reported on in the first deliverable. However, in those experiments no robustness component was present yet. No requirement that all top categories anywhere in the input were to be recognized was yet effective. The grammar has undergone some extensions. Finally, the average number of transitions per word-graph is much higher for the current test-sets (this number was around 2.7 for the previous tests).

### 3.3.1 A small experiment for all parsers

The small experiment consisted of a test-set of hundred word-graphs: the first word-graphs from the set of 21K word-graphs. The difficulty of this test-set in terms of the average number of transitions in the word graph per word in the utterance is given in table 3.5.

| | graphs | transitions | words | w/g | t/w |
|---|---|---|---|---|---|
| test | 100 | 1141 | 358 | 3.6 | 3.2 |
| total | 21288 | 264431 | 72400 | 3.4 | 3.7 |

Table 3.5: This table lists for the test set and the total set the number of transitions, the number of words of the actual utterance, and the average number of words per graph (i.e. the average length of the actual utterance), and the average number of transitions per word. Note that the number of transitions is calculated after the removal of pauses. The most difficult word-graph in this set consists of 157 transitions.

#### 3.3.1.1 Word graphs

The results in terms of cpu-time, for the full wordgraph, are given in table 3.6. Memory usage is documented in table 3.7.

[5] The word-graphs are taken from a corpus which was made available to us in July 1996. These word-graphs are therefore constructed with a rather old version of the speech-recognizer.

| Parser | total cpu-time | average per word graph | Maximum cpu-time |
|---|---|---|---|
| lc_mixtus | 10060 | 101 | 1160 |
| hc9_mixtus | 13810 | 138 | 4690 |
| act | 15960 | 160 | 2210 |
| act2 | 23850 | 238 | 7230 |
| inact_p | 79250 | 792 | 33320 |
| bu | 100190 | 1002 | 35110 |
| lr_cyk2 | 51880 | 519 | 24740 |
| lr_cyk | 117510 | 1175 | 87380 |

Table 3.6: Results of a comparison of all parsers for a small test-set of 100 word-graphs (cpu-time in milliseconds).

| Parser | average memory per word graph | Maximum memory |
|---|---|---|
| lc_mixtus | 20 | 236 |
| hc9_mixtus | 18 | 266 |
| act | 117 | 1645 |
| act2 | 359 | 11941 |
| inact_p | 68 | 1379 |
| bu | 169 | 3822 |
| lr_cyk2 | 39 | 992 |
| lr_cyk | 57 | 1834 |

Table 3.7: Results of a comparison of all parsers for a small test-set of 100 word-graphs (memory in Kbytes).

It can be concluded from these experiments that the left-corner and head-corner parser work considerably more efficient than the other parsers, both in terms of cpu-time requirements and memory requirements. The bottom-up Earley parser without packing (`act`) performs reasonably well too.

### 3.3.2 A large experiment for some parsers

The large experiment consisted of a test-set of 5000 word-graphs: randomly chosen from the set of 21K word-graphs. The difficulty of this test-set in terms of the average number of transitions in the word graph per word in the utterance is given in table 3.8.

| | graphs | transitions | words | w/g | t/w |
|---|---|---|---|---|---|
| test | 5000 | 61351 | 17040 | 3.41 | 3.60 |
| total | 21288 | 264431 | 72400 | 3.40 | 3.65 |

Table 3.8: This table lists for the test set and the total set the number of transitions, the number of words of the actual utterance, and the average number of words per graph, and transitions per word. Note that the number of transitions is calculated after the removal of pauses. The most difficult word-graph consists of 342 edges.

In order to obtain a better idea of the difficulty of the task, consider table 3.9 in which we list the number of word graphs with a certain number of transitions. The graph in figure 3.1 illustrates the distribution of the size of word-graphs in the test set in some more detail.

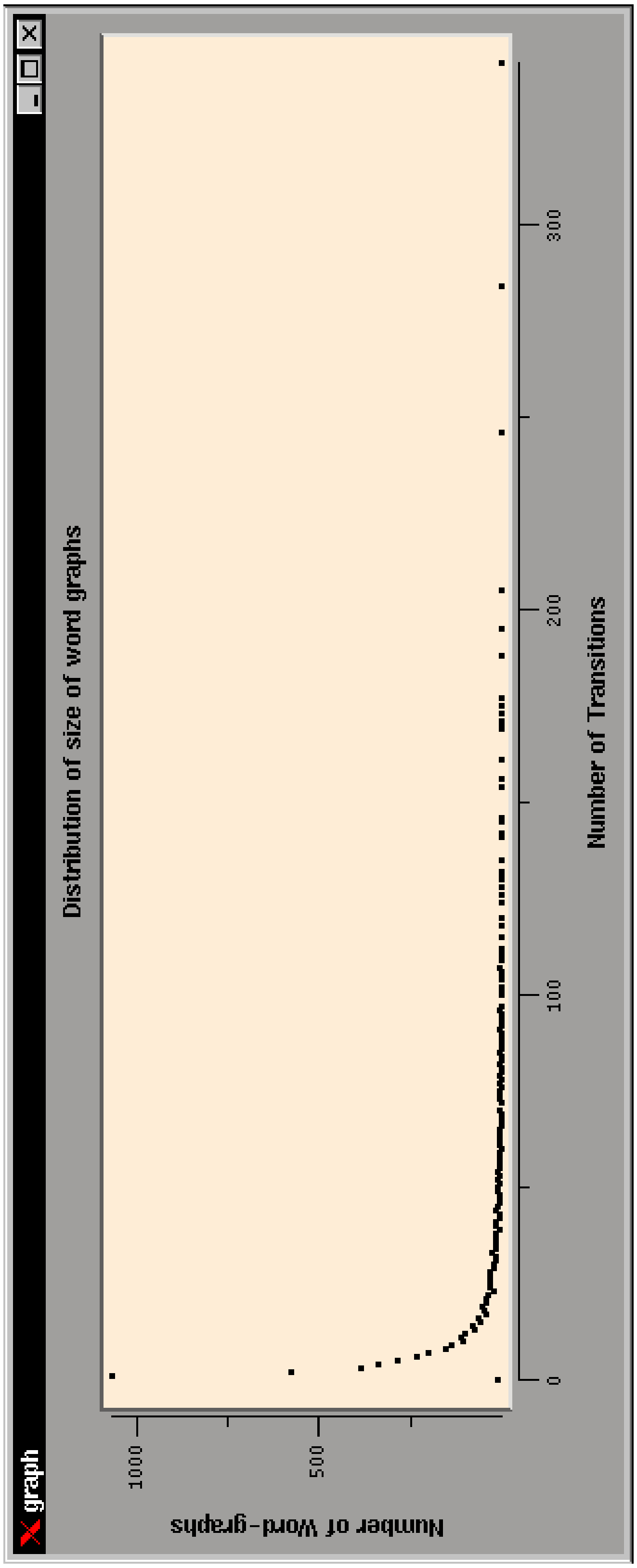


Figure 3.1: The distribution of number of transitions per word-graph.

| Number of transitions | % of word-graphs |
|---|---|
| 0-5 | 53.3 |
| 6-10 | 16.6 |
| 11-25 | 17.2 |
| 26-50 | 11.1 |
| 51-75 | 2.6 |
| 76-100 | 1.2 |
| 101-150 | 0.7 |
| 151-200 | 0.2 |
| 200+ | 0.1 |

Table 3.9: This table lists the proportion of word-graphs with a given number of transitions.

#### 3.3.2.1 Word graphs

This experiment has only been performed for some parsers, because the memory requirements of the `act`, `act2`, `bu`, `inact_p`, `lr_cyk` and `lr_cyk2` parsers made it impossible to complete the experiments. The system typically ran out of space if in the order of 65 Megabytes of memory was required. The results are given in the tables 3.10 and 3.11.

| Parser | total cpu-time | average per word graph | Maximum cpu-time |
|---|---|---|---|
| lc_mixtus | 1861250 | 372 | 820380 |
| hc9_mixtus | 1955120 | 390 | 807840 |

Table 3.10: Results of a comparison of head-corner and left-corner parsers for a test-set of 5000 word-graphs (cpu-time in milliseconds).

| Parser | average memory per word graph | Maximum memory |
|---|---|---|
| lc_mixtus | 30 | 9069 |
| hc9_mixtus | 27 | 8843 |

Table 3.11: Results of a comparison of head-corner and left-corner parsers for a test-set of 5000 word-graphs (memory usage in Kbytes).

As can be seen from these tables, the average cpu-time is much worse in comparison with the small experiment. The reason is the fact that there is a very small number of word-graphs for which the parsers take a very long time. In fact, for both parsers more than 800000 milliseconds (about 40% of the total amount of cpu-time) was spent on a single word-graph!

For this reason we give in table 3.12 the percentage of word-graphs that can be treated within one second of cpu-time. We also give the results for the other parsers because the memory problems only occurred for word-graphs which took (much) longer than 1 second of cpu-time to complete. In figure 3.2 we present a chart of the percentage of word-graphs that can be treated within a certain amount of cpu-time, for each of the parsers.

For the left-corner parser there are 90 word-graphs which take longer than one second of cpu-time. This subset of 1.8% of the graphs takes 77% of the total cpu-time. Moreover, the average time required to parse the remaining 4910 'easy' word-graphs is less than 90 milliseconds.

#### 3.3.2.2 Best path only

Finally, we give the results for the large test set for an approach in which only the best path through the word-graph is considered by the parser. This best path is obtained using acoustic and bigram

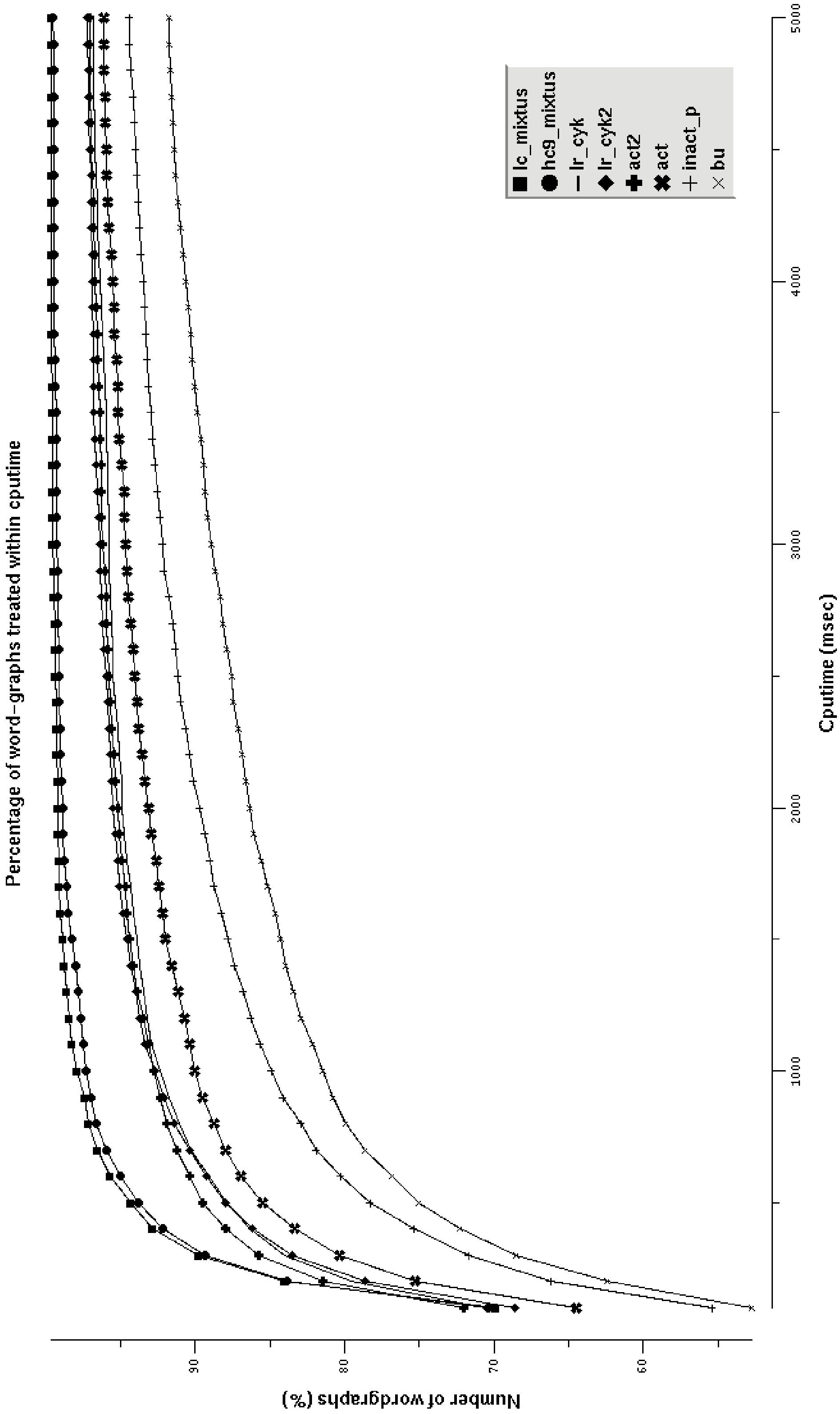


Figure 3.2: This figure shows how many of the word-graphs (percentage of the test-set of 5000) can be treated per amount of cpu-time (in milliseconds).

| lc_mixtus | hc9_mixtus | act2 | lr_cyk2 | lr_cyk | act | inact_p | bu |
|---|---|---|---|---|---|---|---|
| 98.2 | 97.5 | 92.8 | 92.7 | 92.4 | 90.0 | 84.9 | 81.4 |

Table 3.12: Results of a comparison of the parsers for a test-set of 5000 word-graphs (cpu-time in milliseconds). In this table we list the percentage of word-graphs that can be treated within a second of cpu-time.

scores only. The parsers therefore perform as if they were given an ordinary string of words. In chapter 4 this approach is referred to as the `best_1` method. The results are listed in the tables 3.13 and 3.14.

| Parser | total cpu-time | average per utterance | Maximum cpu-time |
|---|---|---|---|
| lc_mixtus | 218550 | 43 | 530 |
| hc9_mixtus | 209840 | 41 | 510 |
| act | 288870 | 57 | 1690 |
| act2 | 336600 | 67 | 30880 |
| inact_p | 567950 | 113 | 12940 |
| bu | 1006550 | 201 | 31790 |
| lr_cyk2 | 722480 | 144 | 194550 |
| lr_cyk | 1699200 | 339 | 594990 |

Table 3.13: Results of a comparison of all parsers for a test-set of 5000 utterances (cpu-time in milliseconds). In this test the parser only considers the best path through the word graph.

| Parser | average per utterance | Maximum memory |
|---|---|---|
| lc_mixtus | 8 | 106 |
| hc9_mixtus | 7 | 98 |
| act | 20 | 366 |
| act2 | 45 | 16255 |
| inact_p | 17 | 761 |
| bu | 40 | 2516 |
| lr_cyk2 | 11 | 3331 |
| lr_cyk | 16 | 6134 |

Table 3.14: Results of a comparison of all parsers for a small test-set of 5000 utterances (memory usage in Kbytes). In this test the parser only considers the best path through the word graph.

Parsing of strings rather than word-graphs is thus much faster. If we were to consider parsing the input utterance *as spoken by the user* rather than parsing the best path through the graph, then the results are similar. Parsing all of the test sentences of the corpus (21288 utterances) takes less than 900 cpu seconds, for an average of 42 milliseconds per utterance (using the head-corner parser), and a maximum of 910 milliseconds. [6]

### 3.3.3 Analysis of difficult word-graphs

In this section we take a closer look at those word-graphs which are hard to process. Somewhat arbitrarily, we have chosen to look at the 90 word-graphs of the test-set for which the left-corner

[6] For the (suprisingly grammatical) sentence 'Ik wil graag op zondag achttien februari 's morgens om negen uur reizen van Amsterdam naar Gouda'.

parser uses more than one cpu second. As could be expected, this set consists of large word-graphs, as can be seen from table 3.15.

| | graphs | transitions | words | w/g | t/w |
|---|---|---|---|---|---|
| test | 90 | 9782 | 852 | 9.5 | 11.5 |

Table 3.15: Characteristics of the set of ninety most difficult word-graphs.

The transitions per word ratio indicates that in these examples the output of the speech recognizer is rather poor, in the sense that there is a lot of uncertainty about the actual utterance. It can be expected, therefore, that the results of the parser will generally not be very good either. Using the evaluation method to be introduced in chapter 5, this expectation turns out to be correct (cf. 3.16).

| Method | WA | SA |
|---|---|---|
| Acoustic | 36.5 | 0.0 |
| Possible | 72.9 | 8.9 |
| VIOS | 53.1 | 4.4 |
| NLP + Speech + Bigram (-keep) | 51.7 | 3.3 |

Table 3.16: Evaluation in terms of word-accuracy (WA) and sentence-accuracy (SA) for the set of 90 difficult word-graphs. The different methods are explained in more detail in chapter 5.

The conclusion which follows from this is twofold:

- the fact that the parser cannot complete the job on certain difficult word-graphs is not that problematic, because those word-graphs are typically useless anyway.
- it would be interesting to have a good heuristic to determine *a priori* which word-graphs are useless in this sense, or possibly which *parts of* word-graphs could be ignored.

### 3.3.4 Conclusion

The experiments warrant the following conclusions:

- parsing of word-graphs is considerably more difficult than parsing of strings: almost ten times slower, even if the transitions per word ratio is only around 3.5.
- the left-corner parser is the fastest parser at the moment for word-graphs. It treats more than 98% of the input word-graphs in less than one cpu second. The head-corner parser is only slightly slower.
- the head-corner parser is the fastest parser for input strings, but the differences are much smaller in the case of strings. Parsing of strings is 'real-time': in the test set all 5000 examples were treated in less than 510 milliseconds.
- memory consumption of both the left-corner parser and the head-corner parser is considerably less than for the other parsers.
- the average time required to parse a word-graph is acceptable. The examples that cannot be treated efficiently do not occur often. However, some of these problematic cases take a very long time to process. It is therefore hard to increase the percentage of graphs processed within e.g. one second significantly.

- the example word-graphs which need long processing time, typically yield very poor results. Therefore, heuristics which could identify such word-graphs *a priori* would be very useful.

# Chapter 4

# Robustness and Disambiguation

In some cases the parser assigns a single update to a given word-graph. In many cases, however, the parser assigns either a number of competing updates, or none at all. In this chapter we present the disambiguation and robustness component. Its purpose is to use the (partial) analyses of the parser in order to construct a single update: hopefully an appropriate one.

There can be a number of reasons why a given word-graph does not give rise to an update:

- error(s) in the speech recognizer
- use of linguistic constructions not covered in the grammar
- irregularities in the spoken utterance

In such cases, the parser typically constructs a number of partial analyses. These partial analyses are sometimes useful in themselves. Consider an example in which the user utters:

(1) Ik wil om tien uur naar Eindhoven

Suppose that for some reason (e.g. background noise) the speech recognizer fails to understand 'Eindhoven'. If the parser is supposed to parse the sentence 'ik wil om tien uur naar' it wil not produce a full parse, but it probably wil have parsed the temporal expression 'om tien uur' correctly. Therefore, the strategy of the robustness component will be to combine the partial results of the parser in a meaningful way.

## 4.1 Parsing with underspecification

In existing approaches towards robust parsing which use the partial results of the parser, it is often assumed that a bottom-up parser is essential: parsers that use top-down information (such as the head-corner parser or the left-corner parser) may fail to recognize relevant sub-parses in the context of an ungrammaticality. Thus, it seems that we are faced with a problem: on the one hand we have fast parsers, which can be fast because of the exploitation of top-down information; on the other hand robustness seems to require the use of a (slow) bottom-up parser.

We show that this problem can be solved by the use of *underspecification*. Rather than assuming that all inactive items of the parser can be useful for robustness, we assume that the grammar specifies which categories are relevant for robustness. The goal of the parser will be to construct these categories (i.e., the start symbol of the grammar will be defined in the appropriate way). The key idea then is to ensure that the parser is supposed to find all such categories *anywhere in the input*.

Normally the parser will be called as e.g. :

(4.1) `?- parse(s(Sem),0,8).`

indicating that we want to parse the sentence from position 0 to 8 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.:

(4.2)
```
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 goal:

(4.3)
```
?- parse(xp(Sem),_,_).
```

Now it may seem that such an underspecified goal will dramatically slow down the 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 memo-ization. Even though we now have a much more general goal, the number of different goals that we need to solve is much smaller.

## 4.2 Best-first search in the word-graph

We have implemented in OVIS the techniques described in the preceding paragraph. The top category (start symbol) of the OVIS grammar is defined as the category `max(Sem)`. Moreover there are unary rules such as `max(Sem)` $\rightarrow$ `np(Sem,..)` for NP, S, PP, ADVP.

In the first phase, the parser finds all occurrences of the top category in the input word-graph. Thus, we obtain items for all possible maximal projections anywhere in the input graph. To obtain the best analysis for the word-graph, the following search procedure is being used.

This robustness procedure consists of a best-first search from the beginning of the graph to the end of the graph. A path in the input graph can be constructed by taking steps of the following two types. To move from position $P$ to $Q$ you can either:

(i) use a maximal projection from $P$ to $Q$ (as constructed by the parser)

(ii) use a transition from $P$ to $Q$. In this case we say that we 'skip' that transition.

In order to compare paths in the best-first search method, we define a score function which includes:

(i) the number of skips. We prefer paths with a smaller number of such skips.

(ii) the number of maximal projections. We prefer paths with a smaller number of such projections.

(iii) the combined acoustic score as defined in the word-graph.

This best-first search method can be implemented very efficiently; it is essentially the Viterbi algorithm for finding the most likely path through an HMM.

The resulting 'best' path in general consists of a number of maximal projections (connected by skips). In the OVIS application these often are simple time or place expressions. The pragmatic module is able to deal with such pieces of information and will perform better if it is given such partial parse results (as compared with an approach in which parser failure does not produce any information). In section 5 we discuss the evaluation procedures we have been using for the robustness component.

Note that disambiguation is a special case of the procedure described here. If the parser does find one or more maximal projections from the beginning of the word-graph to the end of the word-graph, then the search procedure will consider these projections; clearly such large projections will be very good candidates.

In practice, the best-first search will not consider all categories as constructed by the parser. For this reason we have implemented the second phase of the parser (the phase in which derivations are completed with semantic information) in a lazy fashion: the second phase is only applied for a certain category if that category is considered in the best-first search. As before, a table is maintained to make sure that for each category the second phase is only applied once.

## 4.3 Other Score Criteria

We have experimented with other criteria to be included in the scoring function:

- The semantic score. The resulting semantic representation is compared with the question asked previously (in the application all user utterances are reactions to the questions asked by the system). If the answer matches the question then a better score is obtained.
- The bigram score. Because of the success of the bigram language model for the speech recognizer we have experimented with the use of bigrams as a further criterion for the best-first search method.

The results of the use of bigram information are reported in section 5.2. Because of the lack of corpus data, we cannot yet report on results of the use of semantic scoring.

# Chapter 5

# Evaluation

## 5.1 Evaluation Method

In order to evaluate our work, we use a corpus of 21000 word-graphs for which we know the corresponding actual utterances. However, at the moment those word-graphs are not yet annotated with the corresponding update. We therefore cannot yet compare at the level of updates (i.e., we cannot yet determine CONCEPT ACCURACY). We compare the sentence associated with the 'best' path in the word-graph with the sentence that was actually spoken. If the robustness component more often uses the information that was actually uttered then we have more confidence in that component. If, on the other hand, the robustness component more often uses words from the word-graph that were not actually uttered, then of course we have less confidence in the robustness component. Therefore, it seems reasonable to use word accuracy and sentence accuracy as an approximation of the evaluation of the NLP component.

The string comparison on which sentence accuracy and word accuracy are based is defined by the minimal number of substitutions, deletions and insertions that is required to turn the first string into the second (Levenshtein distance), although it may be worthwile to invest other measures. For example, it seems likely that for our application it is less problematic to 'miss' information, whereas "hallucination" is a more severe problem. If information is missed, then the dialogue might become somewhat longer than neccessary. But if information is assumed for which no evidence existed in the actual utterences then misunderstandings might easily lead to confusion. This intuition could be formalized by a scoring function in which insertion is cheaper than deletion. We have chosen not to use these alternative methods because these are non-standard.

Currently the string that is being compared with the actual utterance is defined as the best path through the word-graph, given the best-first search procedure defined in chapter 4. However note that this implies that 'skips' are also part of that string. This seems undesirable because such skips have no effect on the semantic representations that result. Therefore, it may be better (at least for internal evaluation purposes) to count such skips as 'deletions' even if those words were actually spoken. In the reports below the two ways of counting are labelled `+keep` and `-keep` respectively.

In the next section we present our results both in terms of word-accuracy and sentence-accuracy. Word accuracy is defined as

$$1 - \frac{d}{n}$$

where $n$ is the length of the actual utterance and $d$ is the distance as defined above.

Note that we do not present results of the use of the parser without the robustness component. This would be misleading because many decisions in the grammar were motivated by the availability of the robustness component. In order to get some idea: without the robustness component word-accuracy would drop by around 20%.

| | graphs | transitions | words | tpw |
|---|---|---|---|---|
| test | 5000 | 61351 | 17040 | 3.60 |
| total | 21288 | 264431 | 72400 | 3.65 |

Table 5.1: This table lists for the test set and the total set the number of transitions, the number of words of the actual utterance, and the average number of transitions per word. Note that the number of transitions is calculated after the removal of pauses.

| Method | WA | SA |
|---|---|---|
| Acoustic | 67.7 | 53.7 |
| Possible | 86.9 | 72.4 |
| Bigram | 75.1 | 64.0 |
| Acoustic + Bigram | 78.1 | 67.1 |
| VIOS | 77.8 | 66.8 |

Table 5.2: Word accuracy and sentence accuracy for the best path based on acoustic score only (Acoustic); using the best possible path through the word graph (based on acoustic scores only) (Possible); using bigrams only (Bigram); our local implementation of a combination of acoustic score and bigram score (Acoustic + Bigram); and the best sentence as reported by VIOS (combining acoustic score and bigram score) (VIOS). Note that the difference between the latter two methods is on the edge of statistical significance (standard error 0.155).

## 5.2 Results

This section presents a number of results to indicate how well the NLP component currently performs. We used a corpus of 21000 word-graphs, output of a preliminary version of the speech recognizer, and typical of the intended application. From these, we have randomly constructed a set of 5000 word-graphs. The experiments were carried out for this set. Some characteristics of this set are given in table 5.1.

In order to characterize the testsets somewhat further, table 5.2 lists the word and sentence accuracy both of the best path through the wordgraph (acoustic score only), the best possible path through the wordgraph, and the best sentence of the speech recognizer as reported by VIOS (includes the language model, i.e. bigrams), and our implementation of the bigram language model (as used in some of the other experiments).

In the following table (5.3) we list the results of the robustness component under a number of different assumptions.

## 5.3 Results for a Pipeline Architecture

We have experimented with a version of the system in which the best sentence is determined on the basis of the word-graph and bigram scores. This best sentence is then input for the parser. Even if such a version of the system would not perform as well as the version reported in the previous tables, such an approach could still be of practical relevance. If the parser is given only the best sentence and not the full word-graph it will perform much quicker: almost 10 times faster.

As table 5.4 indicates, this approach turns out to be more or less equivalent with the approach in which the full word-graph is used. This indicates that a speed-up of a factor 10 can be obtained without any loss of accuracy. Note that average cputimes have only limited importance; it is more important to know how many of the input word-graphs can be treated in a reasonable amount of time. In the case of full word-graphs the parser succeeds in 95.1% of the cases within 500 milliseconds; 98.2% of the word-graphs can be parsed within one cpu-second. In the pipeline approach,

| Method | WA | SA |
|---|---|---|
| NLP | 71.7 | 58.9 |
| NLP + Speech (-keep) | 73.8 | 62.4 |
| NLP + Speech (+keep) | 75.3 | 63.7 |
| NLP + Speech + Bigram (-keep) | 75.7 | 65.5 |
| NLP + Speech + Bigram (+keep) | 77.4 | 67.1 |

Table 5.3: Evaluation of the robustness component, for a number of different methods. The first method (NLP) only uses the number of skips and the number of categories. The second and third method incorporates the acoustic score from the word-graph (two variants depending on how the words that are skipped are treated). The fourth and fifth method further incorporate bigram information. All differences in WA are statistically significant.

| Method | WA | SA | Milliseconds per word-graph |
|---|---|---|---|
| NLP + Speech + Bigram (-keep) | 75.7 | 65.5 | 385 |
| Pipeline (-keep) | 75.7 | 65.0 | 41 |

Table 5.4: Word accuracy and sentence accuracy for an approach in which bigram and acoustic score is used to select the best sentence, which is then passed on to the parser and robustness component (Pipeline). For comparison we repeat the results for the NLP+Speech+Bigram method. The table also lists the average number of milliseconds cputime that are required. These parse-times include all parts of the NLP component (from word-graph to update).

there is one word-graph which takes 510 milliseconds; all others are treated faster. [1]

[1] This was the sentence *het gaat om dinsdag drieentwintig januari in ik wil van gorinchem naar brummen*.

## References


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

Booth, T.L. 1967. *Sequential Machines and Automata Theory*. New York: Wiley.

Boves, Lou. 1996. Interface tussen de phicos spraakherkenner en nlp. NWO Priority Programme Language and Speech Technology, document number 23. http://grid.let.rug.nl:4321/.

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'. Available from http://grid.let.rug.nl:4321/.

Briscoe, T. and J. Carroll. 1993. Generalized probabilistic LR parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics*, 19(1):25–59.

Carroll, J.A. 1993. Practical unification-based parsing of natural language. Technical Report No. 314, University of Cambridge, Computer Laboratory, England. PhD thesis.

Villemonte de la Clergerie, E. 1993. *Automates à Piles et Programmation Dynamique — DyALog: Une application à la Programmation en Logique*. Ph.D. thesis, Université Paris VII.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February.

Harrison, M.A. 1978. *Introduction to Formal Language Theory*. Addison-Wesley.

Johnson, M. 1991. The computational complexity of GLR parsing. In Tomita (Tomita, 1991), chapter 3, pages 35–42.

Kersten, Danny. 1995. The interface between the components of ovis2 at implementation level. NWO Priority Programme Language and Speech Technology, document number 14. http://grid.let.rug.nl:4321/.

Kipps, J.R. 1991. GLR parsing in time $\mathcal{O}(n^3)$. In Tomita (Tomita, 1991), chapter 4, pages 43–59.

Lang, B. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269, Saarbrücken. Springer-Verlag.

Leermakers, R. 1992a. A recursive ascent Earley parser. *Information Processing Letters*, 41(2):87–91, February.

Leermakers, R. 1992b. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312.

Leermakers, Rene. 1989. How to cover a grammar. In *27th Annual Meeting of the Association for Computational Linguistics*, pages 135–142, Vancouver.

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

Mellish, C.S. 1988. Implementing systemic classification by unification. *Computational Linguistics*, 14(1).

Nederhof, M.J. 1993. Generalized left-corner parsing. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, pages 305–314, Utrecht, The Netherlands, April.

Nederhof, M.J. 1994a. *Linguistic Parsing and Program Transformations*. Ph.D. thesis, University of Nijmegen.

Nederhof, M.J. 1994b. An optimal tabular parsing algorithm. In *32th Annual Meeting of the Association for Computational Linguistics*, pages 117–124, New Mexico.

Nederhof, M.J. and K. Koster. 1992. A customized grammar workbench. In J. Aarts, P. de Haan, and N. Oostdijk, editors, *English Language Corpora: Design, Analysis and Exploitation*, Papers from the thirteenth International Conference on English Language Research on Computerized Corpora, pages 163–179, Nijmegen. Rodopi.

Nederhof, M.J. and J.J. Sarbo. 1993. Increasing the applicability of LR parsing. In *Third International Workshop on Parsing Technologies*, pages 187–201, Tilburg (The Netherlands) and Durbuy (Belgium), August.

Nederhof, M.J. and G. Satta. 1994. An extended theory of head-driven parsing. In *32th Annual Meeting of the Association for Computational Linguistics*, pages 210–217, New Mexico.

Nederhof, M.J. and G. Satta. 1996. Efficient tabular LR parsing. In *34th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 239–246, Santa Cruz, California, USA, June.

van Noord, Gertjan. 1994. Hdrug: a graphical user environment for natural language processing in prolog. http://www.let.rug.nl/~vannoord/hdrug/.

van Noord, Gertjan. 1996. FSA Utilities: Manipulation of finite-state automata implemented in prolog. In *WIA '96 First International Workshop on Implementing Automata*. Technical Report #495, Department of Computer Science, University of Western Ontario, London Ontario.

van Noord, Gertjan, Mark-Jan Nederhof, Rob Koeling, and Gosse Bouma. 1996. Conventional natural language processing in the nwo priority programme on language and speech technology. january 1996 deliverables. NWO Priority Programme Language and Speech Technology, document number 22. http://grid.let.rug.nl:4321/.

Oerder, M. and H. Ney. 1993. Word graphs: An efficient interface between continuous-speech recognition and language understanding. In *ICASSP*, volume 2, pages 119–122.

Pager, D. 1970. A solution to an open problem by Knuth. *Information and Control*, 17:462–473.

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

Rekers, J. 1992. *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam.

Schabes, Y. 1991. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *29th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 106–113, Berkeley, California, USA, June.

Schauerte, R. 1973. Transformationen von LR(k)-grammatiken. Diplomarbeit, Universität Göttingen, Abteilung Informatik.

Schoorl, J.J. and S. Belder. 1990. Computational linguistics at Delft: A status report. Report wtm/tt 90–09, Delft University of Technology, Applied Linguistics Unit.

Shann, P. 1991. Experiments with GLR and chart parsing. In Tomita (Tomita, 1991), chapter 2, pages 17–34.

Sheil, B.A. 1976. Observations on context-free parsing. *Statistical Methods in Linguistics*, pages 71–109.

Sippu, S. and E. Soisalon-Soininen. 1990. *Parsing Theory, Vol. II: LR($k$) and LL($k$) Parsing*. EATCS Monographs on Theoretical Computer Science, volume 20. Springer-Verlag.

Tomita, M. 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers.

Tomita, M., editor. 1991. *Generalized LR Parsing*. Kluwer Academic Publishers.

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.

Veldhuijzen van Zanten, Gert. 1996. Semantics of update expressions. NWO Priority Programme Language and Speech Technology, document number 24. http://grid.let.rug.nl:4321/.

Vosse, T.G. 1994. *The Word Connection*. Ph.D. thesis, University of Leiden.

van Wijngaarden, A. et al. 1975. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5:1–236.