The quest for probabilistic parsing

Institute for Perception Research
PO Box 513, 5600 MB Eindhoven

21.05.1992

Rapport no. 852

The Quest for Probabilistic Parsing

H.R. Ophoff

The Quest for Probabilistic Parsing

Hielko R. Ophoff
May 1992

master thesis

supervisors:
prof. dr. ir. A. Nijholt
dr. R. Leermakers
dr. ir. H.J.A. op den Akker
drs. N. Sikkel

Contents

preface

This report is the result of my work that started at Philips Research Laboratories in Eindhoven and finished at the Institute for Perception Research in Eindhoven. The work started in September 1991 and finished in May 1992. I worked on the Robust Parser Project. Goal of this project was the development of a parser for natural languages with use of statistics. I did my work as part of my study in Computer Science at the University of Twente.

I would like to thank René Leermakers for his ideas and support. My work is based on his ideas. I would also like to thank Jan Odijk for his assistance with typical linguistic issues and Lex Augusteijn for his assistance with the use of the ELEGANT system. Anton Nijholt, Klaas Sikkel and Rieks op den Akker have given valuable advices, comment and sympathy. Thanks. Thanks to the (social) support of Lisette Appelo, Joep Rous and Paul Jansen, I have worked with much pleasure.

Chapter 1

overview

1.1 introduction

Parsing[1] of a natural language[2] for large domains[3] is a big problem. A lot of information that people use to determine sentence structures is not available for parsers (because of representation problems). Because of this lack of information, the parser can often assign a lot of parse trees (sometimes a few hundred) to one sentence. The computation of these parse trees can take a very long time. For a lot of applications, parsing has to be done quickly.

Another problem is the need for a grammar[4]. It is hard to write a grammar that describes all possible sentence structures of a language.

In the Robust Parser Project, it was tried to find a solution to these problems. It was tried to develop a parser that does not compute all possible parse trees of a sentence, but only the most reasonable one. It has to compute such a parse tree very fast. The parser must also compute parse trees for syntactical correct or almost correct sentences that are not described by the grammar.

During the project this goal of developing a robust parser disappeared. The project became a quest for probabilistic parsing. In this chapter, the development of the project is sketched. The organization of this report is given in the last section of this chapter.

1.2 historical development

1.2.1 goal

The first goal of the project was the development of a robust parser, working in linear[5] (or almost linear) time. Robustness means that the parser will assign an acceptable parse tree to grammatical sentences and to somewhat incorrect sentences that the grammar under consideration cannot generate. Whether a parse tree is acceptable, is an application

[1] A parser assigns one or more sentence structures to a sentence. A sentence structure is named a parse tree. Parsing is the assignation process.

[2] Natural languages are languages like English and Dutch.

[3] A small domain is for example the set of sentences used for weather forecasts.

[4] A grammar describes the possible sentence structures of a language. It consists of grammar rules. A parse tree is made by using a number of grammar rules.

[5] In general, the time needed to parse a sentence grows polynomially with respect to sentence length.

dependent question. Some applications only need the general structure of a sentence. Small errors in the structure are of no consequence. For other applications, one structural error would make the parse tree useless. The parser must deliver exactly one parse tree for a sentence.

If the parser has to work in linear time, the parser typically has to make the correct decisions at once. In practice this is impossible. But it may be possible to make the parser work in linear time by forcing a decision at the moment the parser can do more than one action. Statistical information may indicate which decision is most likely. For a parser working in linear time, it is hard to deliver the best parse tree for a given sentence. However, only an acceptable parse tree has to be delivered.

Because collecting statistical information about parser decisions is only useful if there are significant differences between competitive decisions, we wanted to use a parser that uses look-aheads[6]. The longer the look-ahead, the less the number of parsing conflicts (a parsing conflict is a set competitive decisions) and the more distinguishing statistical information may work. Nonterminal as well as terminal symbols[7] will be used as look-aheads. The parser/recognizer[8] model is the Marcus-like parser of Leermakers ([Leermakers, 1993]).

To reach the goal of a robust parser working in (almost) linear time, a parser/recognizer generator had to be made. A grammar was also needed, together with statistical information.

1.2.2 the recognizer

An implementation of the recognizer was made (see appendix B). Starting point was a grammar of a context-free, attribute-free grammar. With use of the Elegant system ([Augusteijn, 1990]), all the parsing information that can be computed before the actual recognition of sentences is generated. C-code that represents the recognizer is delivered. After compiling this code, there is an executable recognizer. In addition, functions for collecting statistical information were implemented.

1.2.3 corpus and grammars

For the grammar and the statistical information, a corpus with analyzed sentences (for every sentence one parse tree) seems pre-eminently suitable.

We used a part of the LDB[9] corpus from the University of Nijmegen ([Halteren & Heuvel, 1990]). The corpus consists of several pieces of English text from the 'real world'. Together with the corpus we got a grammar from the developers of the LDB-corpus that describes almost all sentences from the corpus. Because this grammar

[6]The parser reads a sentence from left to right. While the parser reads a sentence, it has to take a lot of decisions. If the parser may look at the next occurring words of the sentence, it can take better decisions. These next occurring words that the parser may look at, are named look-aheads. The next occurring word can be used as look-ahead, but also a sequence of next occurring words can be used as look-ahead. In the latter case, the look-ahead is longer than in the former case. Other symbols than words can be used as look-ahead.

[7]Terminal symbols are the syntactical classes of the words of the sentence, nonterminal symbols are syntactical classes of groups of words.

[8]A recognizer says whether there exists a parse tree for a sentence.

[9]Linguistic DataBase

has not been documented and because the grammar seemed difficult to work with, first we did not use this grammar. Later examination showed that this grammar is untractable for our purpose.

From the parse trees of the sentences from a part of the corpus, a grammar was extracted (grammar G_C). Grammar G_C describes precisely the parse trees of the corpus sentences. Hence, it should suffice for collecting statistical information.

Grammar G_C appeared to be relatively big. G_C had more grammar rules than there are sentences from which the grammar rules have been derived. A lot of grammar rules are rarely used (often only once a rule is used in a parse tree of a corpus sentence). So two problems appeared:

- the generated parser is too big for experiments to work with
- collecting statistical information is not useful because too many parser decisions rarely occur

All the grammar rules that were seldomly used in the corpus trees, were removed from the grammar G_C. The resulting grammar is manageable. Our assumption was that a set of commonly used grammar rules exists that forms the principal part of the rules which are used in parse trees. The uncommonly used grammar rules describe exceptional cases and are not interesting for a robust parser. It appeared that the reduced grammar only could generate a restricted number of very short sentences, thus our assumption was too optimistic.

A question is what the reduced grammar actually describes. Apparently, rarely occurring grammar rules are essential for parsing sentences, so the commonly used grammar rules do not fully describe general sentence structures (parse trees). Then it also will not be meaningful to collect statistical information about decisions of the parser based on the reduced grammar. It also will be very difficult to collect statistical information with this reduced grammar.

1.2.4 reflection

On the one hand it was interesting to look more precisely at the grammar G_C. Why uses G_C so many rules to describe the sentences from which it was extracted? Is it possible to use fewer rules? How much rules are absolutely necessary? Which factors are important when the number of rules has to be reduced? What are the effects on parsing when the grammar is transformed?

On the other hand it may be possible to use another grammar. In the ROSETTA system a grammar for the English language is being used.

1.2.5 the ROSETTA grammar

First we tried to use the ROSETTA grammar. Using a grammar that does not correspond with the grammar used to make up the parse trees of the analyzed corpus sentences causes a problem. It will be difficult to collect statistical information from the parse trees. The parse trees have to be rewritten according to the new grammar. We could try to do this by parsing the corpus sentences with the new grammar, using information from the parse trees of the sentences build with the old grammar (G_C). But first we had to build a parser based on the ROSETTA grammar.

The attributes used in the grammar were eliminated. The grammar has been written in regular expressions. We rewrote the grammar in BNF notation (a growth from 20 to 300 grammar rules). The resulting recognizer needed very much time to recognize a sentence. After eliminating the attributes the grammar became strongly over-generating. Because of the existence of many empty rewritings (ϵ-rewritings) it could be that the parser has to do a lot of unnecessary work. So a variant on the recognizer was made that uses ϵ-rules more efficiently. But there were other reasons for the complexity in the parsing process (see chapter 8), so it was not possible to use this grammar.

1.2.6 examining grammars

We had three grammars: the grammar delivered with the corpus, the grammar extracted from the corpus (G_C) and the grammar used in the ROSETTA system. None of these could be used for statistical parsing. So it seemed useful to examine these grammars. Questions mentioned before had to be answered. The grammars could be compared with each other. From an examination of the three grammars, conclusions about writing context-free grammars for probabilistic parsing could be extracted.

1.2.7 probabilistic parsing

Other people have tried to make parsers that work with statistical information. Do they use good models? Is our approach with a Marcus-like parser a good approach? We examined other approaches and criticized them. We concluded that our approach is one of the most realistic approaches for developing a probabilistic parser for common English.

The way a grammar is organized, has an effect on the information that can be expressed with the statistics. These effects were examined.

1.2.8 the parser model

In our approach we promoted the use of the Recursive Ascent Marcus Parsers (RAM parsers). We examined the properties of such a parser. The effects on the parser of the way a grammar is organized were examined. Because RAM parsers could be very large for a large grammar of a natural language, we developed a Reduced Recursive Ascent Parser, the size of which does not exponentially grow with respect to the number of look-aheads.

1.2.9 the result

We were not able to experiment with probabilistic parsers. But we could conclude with the properties of the grammar, the parser, the statistical information and the corpus that are needed for a good, significant experiment with probabilistic parsing. It turned out that a realistic experiment that can lead to a conclusion about the usability of statistical information for parsing will take a lot of work. Such an experiment can only be done in cooperation with other research groups.

1.3 contents

There are three parts in this report. In the first part, we describe the Recursive Ascent Marcus Parser and its properties. We also describe an extension of the Recursive Ascent

LR(0) parser with Marcus-like look-aheads. In the second part, context-free grammars are examined. Probabilistic parsing is examined in part three.

Part I

Recursive Ascent Marcus Look-Ahead Parsers

Chapter 2

recursive ascent Marcus parsers

2.1 introduction

In this chapter we construct a model for Marcus parsers. We use the recursive ascent implementation for these parsers. In this way, parsers can be described functionally. This has some benefits (see [Leermakers, 1992]). The parsers use constituent look-aheads. Because the use of constituent look-aheads is essential for Marcus parsers, we name our parsers recursive ascent Marcus (RAM) parsers.

The chapter starts with some notational remarks. An example that introduces some ideas used in the parser model is given. A formal construction of the parsers follows. In the last section of the chapter we prove the correctness of the parser model. In chapter 3 we examine the (time and space) complexity of the parser. In chapter 4 the parser is extended so that attributes can be used. In chapter 5 another recursive ascent Marcus parser is described. In fact we do not describe parsers but recognizers. The recognizers we describe can always be extended with (synthesized) attribute evaluation components. So the recognizers can always be extended to parsers.

2.2 notational remarks

We expect the reader of this report to have some knowledge of LR-like parsing techniques. V_T is the set of terminal symbols (lexical categories). V_N is the set of nonterminal symbols. $V_T \cap V_N = \emptyset$. $V_T \cup V_N = V$.

In general, symbols like A, B, ...are nonterminals ($A, B, \ldots \in V_N$), $x_1, x_2, \ldots \in V_T$ and $X, Y, Z \in V$. The greek symbols $\alpha, \beta, \ldots$ are used for representing symbols strings, so $\alpha, \beta, \ldots \in V^*$. ϵ represents the empty string.

We want to have a parser with constituent look-aheads. When the parser takes a decision, the parser has to see look-ahead symbols. These look-aheads are the next symbols to process. For representing symbols that have been recognized and the symbols the parser has not yet recognized we use dotted items. An item is of the form $A\gamma \rightarrow \alpha.\beta$ with $A \in V_N, \gamma, \alpha, \beta \in V^*$ and there is a grammar rule $A \rightarrow \delta$ with $\delta\gamma = \alpha\beta$. The symbols α have been recognized, the symbols β have to be recognized. The symbols γ are the look-aheads of A. If an item is of the form $A\gamma \rightarrow \alpha.$ then the symbols α have been recognized, hence the symbols $A\gamma$ have been recognized. Such an item is named a final item. An item of the form $A\gamma \rightarrow .\alpha$ is named an initial item.

2.3 example

Consider the grammar with rules:

$$\begin{array}{ll} S \rightarrow A\,B\,C & C \rightarrow D \\ A \rightarrow a & C \rightarrow c\,E \\ B \rightarrow \epsilon & C \rightarrow c\,A \\ B \rightarrow b & D \rightarrow c \\ & E \rightarrow a\,C \end{array}$$

with $S, A, B, C, D, E \in V_N, a, b, c \in V_T, \epsilon$ is the empty string. The number of look-aheads is one. Input sentence is "abc".

With LR(0) parsing, the parser looks at the next symbol to process. With constituent look-ahead, the parser looks not only at the next symbol to process but it also looks at the look-aheads. Thus the parser looks at the next k symbols to process, with k = number of look-aheads + 1.

The nonterminal $ROOT$ is added to V_N to get a unique root item. Root item is $ROOT \rightarrow .S$. Nonterminal S has to be recognized. The left-most-symbol rewritings of S are:

- $S \rightarrow A\,B\,C$
- $S \rightarrow^* a\,B\,C$ ($S \rightarrow A\,B\,C$ and $A \rightarrow a$)

For recognizing S, the parser has to recognize $A\,B\,C$ or $a\,B\,C$. Therefore initial items originating from the root item $ROOT \rightarrow .\,S$ are:

- $S \rightarrow .\,A\,B\,C$
- $A\,B \rightarrow .\,a\,B$

One can see the work of the look-aheads in the second initial item. The parser sees the first k symbols (here A and B) and their left rewriting. An item $A \rightarrow .\,a$ is not created because the parser sees the first k symbols (in this case two symbols) and not only the first symbol. If the item $A \rightarrow .a$ would be created, the parser would not use existing look-ahead information. If there do not exist $k-1$ look-aheads, as in the item $ROOT \rightarrow .S$, as much look-ahead information as available is used. Thus with an item $A \rightarrow B\,.\,C\,D$ and two look-aheads ($k = 3$), an item is created with $C\,D \rightarrow \ldots\,D$. In this case, the parser only uses one look-ahead (D).

With first input symbol a, the parser has recognized symbol a. It shifts this symbol in item $A\,B \rightarrow .\,a\,B$. The resulting item representing the new situation is:

- $A\,B \rightarrow a\,.\,B$ (a has been recognized, B has not been recognized)

The new item is a product of a transition on the recognized symbol a. The transition has been made from the item describing the original situation ($A\,B \rightarrow .\,a\,B$). This is illustrated in a picture (figure 2.1).

There are two possibilities for recognizing B. In the first case B rewrites to the empty string ϵ, in the second case B rewrites to b. If B rewrites to ϵ, symbol B can be reduced immediately as we will see later. In this example B rewrites to ϵ and the symbol ϵ can be recognized. The resulting item is:

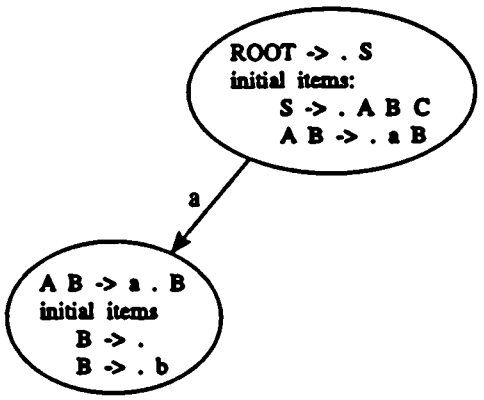


Figure 2.1: states of the recognizer

- $B \rightarrow \epsilon$.

In the other case symbol b has to be recognized. The item representing this situation is $B \rightarrow .b$. With second input symbol b, symbol b will be shifted. This results in item:

- $B \rightarrow b$.

Thus symbol B has been recognized (with $B \rightarrow \epsilon$. and $B \rightarrow b$.). See figure 2.2. Recognizing symbol B started from item $A\ B \rightarrow a\ .\ B$. A reduce symbol B action

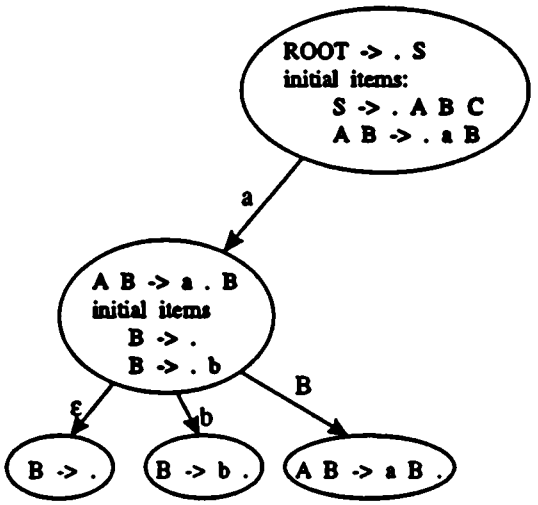


Figure 2.2: states of the recognizer

based on final item $B \rightarrow b$. results in item:

- $A\ B \rightarrow a\ B$.

Symbol string $A\ B$ has been recognized two times: for $B \rightarrow \epsilon$ and for $B \rightarrow b$. Recognizing symbol string $A\ B$ started from item $ROOT \rightarrow .\ S$. Symbol string $A\ B$

cannot be reduced in this item. Symbol string $A\,B$ has been recognized while recognizing the rewritings of symbol S. Symbol S rewrote to $A\,B\,C$ and the prefix of $A\,B\,C$ ($A\,B$) rewrote to $a\,B$. So the parser can reduce symbol string $A\,B$ in the item corresponding with the first rewriting of S. That item is the initial item $S \rightarrow .A\,B\,C$. The reduce action results in item:

- $S \rightarrow A\,B\,.\,C$

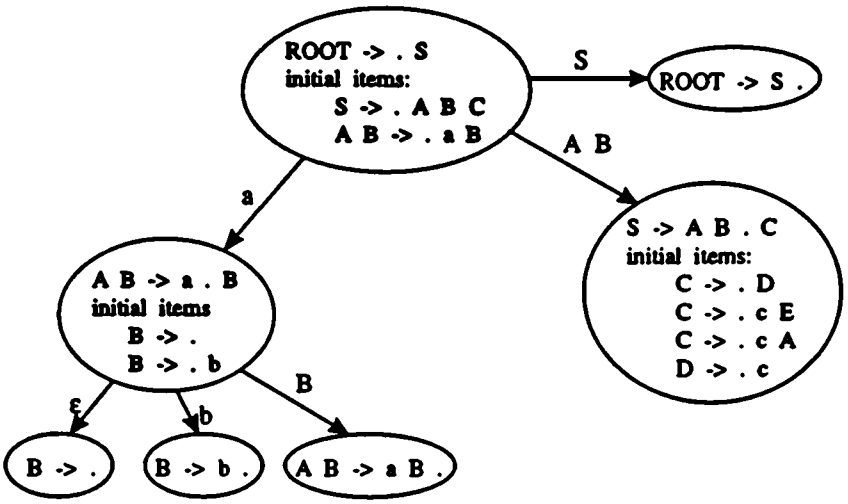


Figure 2.3: states of the recognizer

Now the parser can try to recognize symbol C for two cases. In one case symbol b has been recognized. In the other case symbol b has to be recognized (the case of the ϵ-reduction). Because symbol C does not left-rewrite to terminal b, the second case can be eliminated.

For item $S \rightarrow A\,B\,.\,C$ there are three initial items with shift symbol c (see figure 2.3). After recognizing symbol c there are three new items:

- $D \rightarrow c\,.$
- $C \rightarrow c\,.\,A$
- $C \rightarrow c\,.\,E$

These three items can be taken together, because they are produced on the basis of the same transition symbols. Such a set of items is named a **state**. Thus a state is a set of items derived from a set of items and the related initial items on the basis of the same transition symbol(s). In the example all states except the last exist of only one item. Transitions were made for one item with the corresponding initial items. But for a state with more than one item a transition can be made on the basis of all the items in the state and their corresponding initial items. In the case of the state:

- $D \rightarrow c\,.$
- $C \rightarrow c\,.\,A$

- $C \rightarrow c\,.\,E$

a new state can be made on a transition on symbol a, because there are following initial items:

- $A \rightarrow .\,a$ from $C \rightarrow c\,.\,A$
- $E \rightarrow .\,a\,C$ from $C \rightarrow c\,.\,E$

In figure 2.4 all the states with their transitions are displayed.

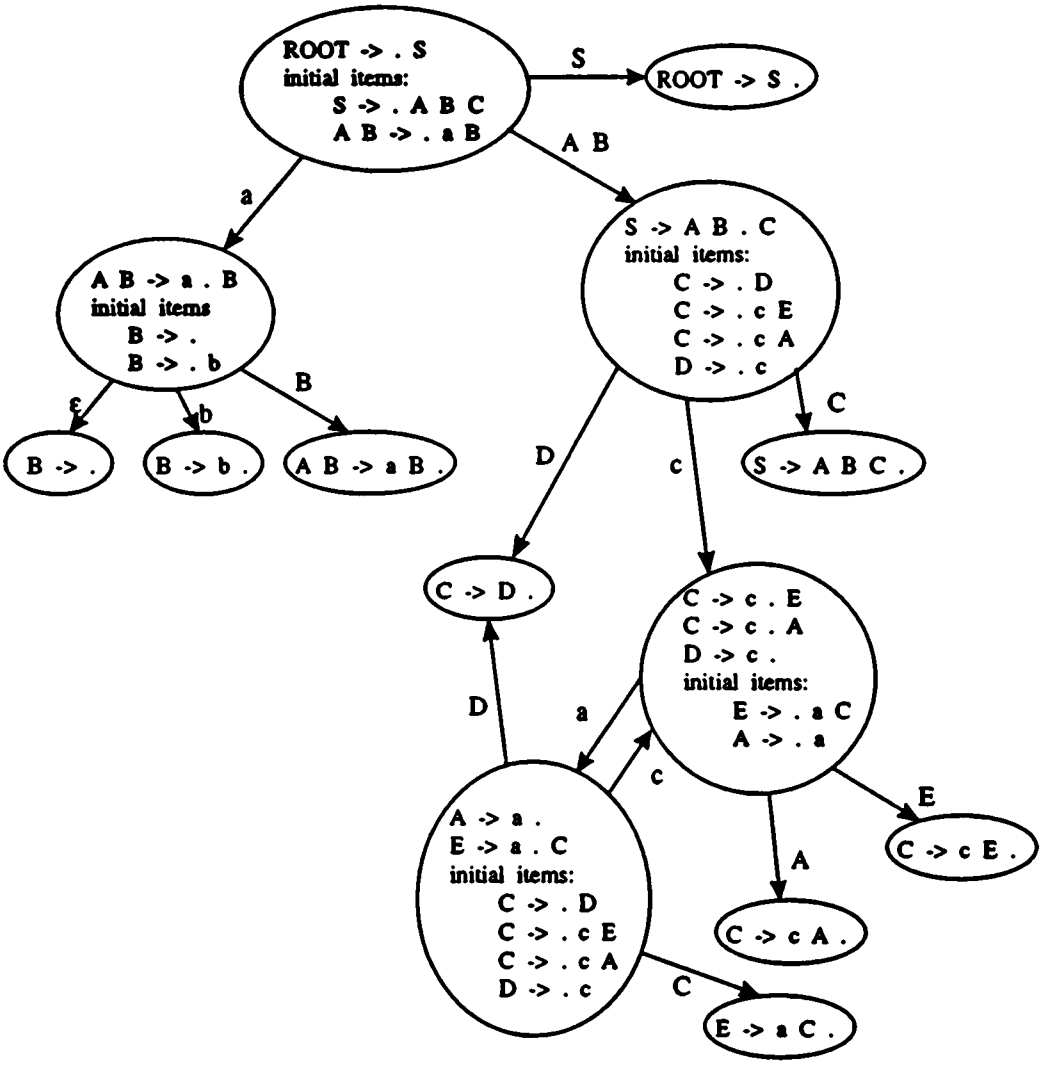


Figure 2.4: states of the recognizer

It is not possible to shift another input symbol because all input symbols have been shifted. The final item $D \rightarrow c\,.$ leads to a reduce action of symbol D (transition to state $\{C \rightarrow D\,.\}$). So symbol C has been recognized and it can be reduced (transition to state $\{S \rightarrow A\,B\,C\,.\}$). Symbol S has been recognized and can be reduced. This leads to the final item $ROOT \rightarrow S\,.\,.$ It is easy to see that $S \rightarrow^* abc$.

2.4 construction of the parser

2.4.1 auxiliary functions

First we define the left-most-symbol rewriting $\Rightarrow$:

$\Rightarrow$: a relation between $V_N V^*$ and V^*

$$\alpha \Rightarrow \beta \equiv \exists_{A\gamma\delta}(\alpha = A\gamma \wedge \beta = \delta\gamma \wedge A \rightarrow \delta)$$

We also need a prefix and a suffix function for splitting up a symbol string in its look-ahead prefix and the remaining string:

$$\begin{array}{ll} k: & : V^* \rightarrow V^\ell \quad , \quad 0 \leq \ell \leq k \\ :k & : V^* \rightarrow V^* \end{array}$$

prefix function

$$\begin{array}{rcl} k:\alpha & = & \epsilon, k = 0 \vee \alpha = \epsilon \\ k:X\alpha & = & X, X \in V_T \\ k:X\alpha & = & X\alpha, X \in V_N \wedge k \geq |X\alpha| \\ k:\alpha & = & \text{prefix of } \alpha \text{ with length } k, \textit{otherwise} \end{array}$$

suffix function

$$\alpha = k:\alpha +\!+^1 \alpha:k$$

The k: and $:k$ functions bind stronger than the $+\!+$ function. The prefix function does not look further than the first symbol if the first symbol is a terminal symbol.

As we have seen before, the parser needs a function that generates the initial items derived from a state. Therefore we define a function *ini* that returns a set of initial items. These initial items are derived from items in a state. The set of all items is I, the set of all states is S.

$$ini : S \rightarrow \mathcal{P}(I)$$

$$\begin{array}{rcl} ini(q) & = & \{B\mu \rightarrow .\nu\mu \mid \exists_{\alpha\beta\gamma\delta}(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* B\mu\delta \\ & & \wedge\, k:B\mu\delta = B\mu \wedge B \rightarrow \nu \wedge \nu \neq \epsilon)\} \end{array}$$

For each item $\gamma \rightarrow \alpha.\beta$ in state q the rewritings of β are computed and the corresponding initial items are produced. The parser sees only $k-1$ look-aheads. For ϵ-rewritings there are no initial items. The parser skips all ϵ-rewritings as we will see later.

The function *goto* returns a new state based on transition symbols δ, derived from the items in state q:

$$goto : S \times (V_T \cup V_N V^\ell) \rightarrow S, 0 \leq \ell < k$$

$$\begin{array}{rcl} goto(q,\delta) & = & \{\gamma \rightarrow \alpha\lambda\delta.\beta \mid \gamma \rightarrow \alpha.\lambda\delta\beta \in (q \cup ini(q)) \\ & & \wedge\, \delta = k:\delta\beta \wedge \lambda \rightarrow^* \epsilon\} \end{array}$$

Symbols rewriting to ϵ are skipped. They can be reduced immediately.

2.4.2 parsing: the action functions

The parser is based on following definition (II is the set of all index-items, see below):

$$[q] \ : \ N \rightarrow \mathcal{P}(II)$$

$$[q](i) \ = \ \{(\gamma \rightarrow \alpha.\beta, j) \mid \gamma \rightarrow \alpha.\beta \in q \wedge \beta \rightarrow^* x_{i+1} \ldots x_j\} \tag{2.1}$$

For each state q there is a function $[q]$. With input $x_1 \ldots x_n$, initial state $\{ROOT \rightarrow .S\}$, $S \rightarrow^* x_1 \ldots x_n$ if and only if $(ROOT \rightarrow .S, n) \in [\{ ROOT \rightarrow .S\}](0)$.

The parsing process works with indexed items (named index-items). The index j of an index-item $(\gamma \rightarrow \alpha.\beta, j)$ indicates up to which input symbol β (thus also γ) rewrites. Other functions as we will see later can use such an index-item for reduce actions. The items in the states determine a situation in the process of parsing a sentence. With the items in a state the parser can see if it can take a shift action or a reduce action, what the next symbols are to determine etc. An **index-item** says what the relation is between an item and the input sentence. A function $[q]$ returns index-items derived from the items of state q.

We will construct a functional implementation for each function $[q](i)$. The construction is based on the equation:

$$\begin{aligned} \beta \rightarrow^* x_{i+1} \ldots x_j \ \equiv \ & \exists_\gamma(\beta \Rightarrow^* x_{i+1}\gamma \wedge \gamma \rightarrow^* x_{i+2} \ldots x_j) \\ & \vee (\beta \rightarrow^* \epsilon \wedge i = j) \end{aligned} \tag{2.2}$$

A string (non)terminals rewrites to ϵ or it has a left most symbol rewriting to a terminal. With equation 2.2, definition 2.1 can be written as:

$$\begin{aligned} [q](i) \ = \ & \{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_\mu(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* x_{i+1}\mu \\ & \qquad \wedge \mu \rightarrow^* x_{i+2} \ldots x_j)\} \\ & \cup \{(\gamma \rightarrow \alpha.\beta, i) \mid \gamma \rightarrow \alpha.\beta \in q \wedge \beta \rightarrow^* \epsilon\} \end{aligned} \tag{2.3}$$

For each item $\gamma \rightarrow \alpha.\beta$ in set q, α has been recognized. For recognizing β the parser looks whether β rewrites to a terminal symbol or $\beta \rightarrow^* \epsilon$ ($\beta \rightarrow^* x_{i+1} \ldots x_i$). In the first case the parser shifts input symbol x_{i+1} and tries to recognize a rewriting of μ. In the second case the function returns an item with index. So we need a function to shift symbols like x_{i+1} in definition 2.3. Such a symbol has been recognized. Therefore for every state q we introduce a function $\overline{[q]}$ with definition:

$$\overline{[q]} \ : \ (V_T \cup V_N V^\ell) \times N \rightarrow \mathcal{P}(II), 0 \leq \ell < k$$

$$\begin{aligned} \overline{[q]}(\delta, i) \ = \ & \{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{\delta\mu}(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* \delta\mu \\ & \qquad \wedge \mu \rightarrow^* x_{i+1} \ldots x_j \wedge k : \delta\mu = \delta\} \end{aligned} \tag{2.4}$$

Each function $\overline{[q]}$ skips symbols δ (and the possible ϵ-rewritings before δ). Because the k-prefix of a symbol list $x\gamma$ with $x \in V_T$ and $\gamma \in V^*$ is x definition 2.3 can be written as:

$$\begin{aligned} [q](i) \ = \ & \{(\gamma \rightarrow \alpha.\beta, j) \mid (\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(x_{i+1}, i+1)\} \\ & \cup \{(\gamma \rightarrow \alpha.\beta, i) \mid \gamma \rightarrow \alpha.\beta \in q \wedge \beta \rightarrow^* \epsilon\} \end{aligned} \tag{2.5}$$

If we have a functional implementation for each function $\overline{[q]}$ we have a functional implementation for each function $[q](i)$.

2.4.3 parsing: climb functions

In the definition of $\overline{[q]}$ (definition 2.4) there are two cases for the rewriting $\beta \Rightarrow^* \delta\mu$. The rewriting consists of zero steps or it consists of at least one step.

An example. In an item $A \rightarrow B\,.\,c$ ($A, B \in V_N, c \in V_T$) the symbol c can be shifted immediately (without rewritings). This equals the case $\beta \equiv c$. In the example in section 2.3 we saw that symbol string AB could not be reduced immediately in item $ROOT \rightarrow .S$. We solved this by reducing string AB in the item corresponding with the first rewriting of S. This equals the situation $S \Rightarrow^* S \wedge S \Rightarrow AB\,C$ with transition symbols $\delta = AB$. If $\beta \Rightarrow \eta$ and $\eta \Rightarrow \eta'$ and so on the parser first tries to recognize the deepest rewriting (recognizing starts at the level of terminal symbols) and then it tries to climb up. In this way it tries to recognize β.

Because the parser skips ϵ-rewritings, an ϵ-rewriting is not considered as a rewriting step. So with a division into zero steps and one or more steps, we get following equation:

$$\begin{aligned}\beta \Rightarrow^* \delta\mu \quad \equiv \quad & \exists_\lambda(\beta = \lambda\delta\mu \wedge \lambda \rightarrow^* \epsilon) \\ & \vee \exists_{\eta\lambda}(\beta \Rightarrow^* \eta \wedge \eta \Rightarrow \lambda\delta\mu \wedge \lambda \rightarrow^* \epsilon)\end{aligned}$$

Because we are not interested in ϵ-rewritings, we also are not interested in left-most-symbol-ϵ rewritings. For a $\lambda\delta\mu$ with $\lambda \rightarrow^+ \epsilon$ it is possible that a lot of left-most-symbol rewritings of the form $\lambda_1\delta\mu \Rightarrow \lambda_2\delta\mu$ with $\lambda_1, \lambda_2 \rightarrow^* \epsilon$ exists. Therefore in the case of $\beta \Rightarrow^* \eta \wedge \eta \Rightarrow \lambda\delta\mu$ the rewriting $\eta \Rightarrow \lambda\delta\mu$ may not be of the form $\lambda_1\delta\mu \Rightarrow \lambda_2\delta\mu$. Thus in the left-most-symbol rewriting $\eta \Rightarrow \lambda\delta\mu$, a prefix of δ has to be written. Thus η must be of the form $A\vartheta$ and $A \rightarrow \lambda$ prefix$(\delta)\nu$ with $\vartheta, \nu \in V^*$ and k : prefix$(\delta)\nu\vartheta = \delta$.

This can be written as ($X \in V$):

$$\begin{aligned}\beta \Rightarrow^* \delta\mu \quad \equiv \quad & \exists_\lambda(\beta = \lambda\delta\mu \wedge \lambda \rightarrow^* \epsilon) \\ & \vee \exists_{\vartheta\lambda AX}(\beta \Rightarrow^* A\vartheta \wedge A \rightarrow \lambda X\nu \wedge k : X\nu\vartheta = \delta \\ & \qquad \wedge X\nu\vartheta : k = \mu \wedge \lambda \rightarrow^* \epsilon)\end{aligned}$$

Now definition 2.4 can be written as:

$$\begin{aligned}\overline{[q]}(\delta, i) \quad = \quad & \{(\gamma \rightarrow \alpha.\lambda\delta\mu, j) \mid \gamma \rightarrow \alpha.\lambda\delta\mu \in q \wedge \mu \rightarrow^* x_{i+1} \ldots x_j \\ & \qquad \wedge k : \delta\mu = \delta \wedge \lambda \rightarrow^* \epsilon\} \\ & \cup \{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{\lambda\vartheta\nu AX}(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* A\vartheta \\ & \qquad \wedge A \rightarrow \lambda X\nu \wedge k : X\nu\vartheta = \delta \\ & \qquad \wedge X\nu\vartheta : k \rightarrow^* x_{i+1} \ldots x_j \\ & \qquad \wedge \lambda \rightarrow^* \epsilon)\}\end{aligned} \tag{2.6}$$

We will rewrite these two sets with use of the functions *ini*, *goto*, [] and $\overline{[]}$. First we will rewrite the first set.

rewriting the first set

In the first set of definition 2.6 one has $\gamma \rightarrow \alpha.\lambda\delta\mu \in q \wedge \mu \rightarrow^* x_{i+1} \ldots x_j \wedge k : \delta\mu = \delta \wedge \lambda \rightarrow^* \epsilon$. Then $\gamma \rightarrow \alpha\lambda\delta.\mu \in goto(q, \delta)$.
Reversily if $\gamma \rightarrow \alpha\lambda\delta.\mu \in goto(q, \delta) \wedge \gamma \rightarrow \alpha.\lambda\delta\mu \in q$ then $k : \delta\mu = \delta$. After recognizing δ the parser goes to a new state (state $goto(q, \delta)$) with new items representing the new

situation.
If

$$\gamma \rightarrow \alpha\lambda\delta.\mu \in goto(q,\delta) \wedge \mu \rightarrow^* x_{i+1} \ldots x_j$$

then

$$(\gamma \rightarrow \alpha\lambda\delta.\mu, j) \in [goto(q,\delta)](i)$$

Reversily if

$$(\gamma \rightarrow \alpha\lambda\delta.\mu, j) \in [goto(q,\delta)](i)$$

then

$$\gamma \rightarrow \alpha\lambda\delta.\mu \in goto(q,\delta) \wedge \mu \rightarrow^* x_{i+1} \ldots x_j$$

So the first set can be written as:

$$\{(\gamma \rightarrow \alpha.\lambda\delta\mu, j) \mid \gamma \rightarrow \alpha.\lambda\delta\mu \in q \wedge \lambda \rightarrow^* \epsilon \\ \wedge (\gamma \rightarrow \alpha\lambda\delta.\mu, j) \in [goto(q,\delta)](i)\}$$

rewriting the second set

We will now rewrite the second set of definition 2.6. With the definition of $ini(q)$ we have:
if

$$\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* A\vartheta \wedge A \rightarrow \lambda X\nu \wedge k : X\nu\vartheta = \delta \wedge \lambda \rightarrow^* \epsilon$$

(see the definition of the second set in 2.6)
then

$$\exists_\xi(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* A\vartheta \wedge k : A\vartheta = A\xi \\ \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge k : X\nu\vartheta = \delta \wedge \lambda \rightarrow^* \epsilon)$$

and reversily.

We have

$$\mid \delta \mid \leq k \wedge \mid X\nu \mid \geq 1 \wedge (\mid \xi \mid = min(k, \mid \vartheta \mid) - 1)$$

(The function min returns the minimum of its arguments.) If $\mid \nu \mid \geq k-1$ then $k : X\nu\vartheta = k : X\nu$. Otherwise $k : X\nu\vartheta = X\nu\varphi$ with $\mid \varphi \mid \leq min(k, \mid \vartheta \mid) - 1$. So $k : X\nu\vartheta = k : X\nu\xi = \delta$.

Hence, the second set in equation 2.6 can be written as:

$$\{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{\vartheta\lambda\nu\xi\ell A X}(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* A\vartheta \\ \wedge k : A\vartheta = A\xi \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \\ \wedge X\nu\xi : k \rightarrow^* x_{i+1} \ldots x_\ell \wedge A\vartheta : k \rightarrow^* x_{\ell+1} \ldots x_j \\ \wedge k : X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon)\}$$

With $\gamma \rightarrow \alpha.\beta \wedge \beta \Rightarrow^* (k : A\vartheta)(A\vartheta : k) \wedge A\vartheta : k \rightarrow^* x_{\ell+1} \ldots x_j$ we can use the definition of $\overline{[q]}$ (definition 2.4) to rewrite this set. The result is:

$$\{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{\vartheta\lambda\nu\xi\ell AX}((\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(A\xi, \ell) \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge X\nu\xi : k \rightarrow^* x_{i+1} \ldots x_\ell \wedge\ k : X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon)\}$$

Symbol string δ has been recognized. The function *goto* can be used to describe this situation. If

$$A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge k : X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon$$

then

$$A\xi \rightarrow \lambda(k : X\nu\xi).(X\nu\xi : k) \in goto(q, \delta)$$

The state $goto(q,\delta)$ contains items of the form $A\xi \rightarrow \lambda(k : X\nu\xi).(X\nu\xi : k)$ with $(X\nu\xi : k) \rightarrow^* x_{i+1} \ldots x_\ell$. This equals the situation described in the definition of the functions $[q]$ (definition 2.1). If

$$A\xi \rightarrow \lambda(k : X\nu\xi).(X\nu\xi : k) \in goto(q, \delta) \wedge (X\nu\xi : k) \rightarrow^* x_{i+1} \ldots x_\ell$$

then

$$(A\xi \rightarrow \lambda(k : X\nu\xi).(X\nu\xi : k), \ell) \in [goto(q, \delta)](i)$$

Reversily if

$$(A\xi \rightarrow \lambda(k : X\nu\xi).(X\nu\xi : k), \ell) \in [goto(q, \delta)](i)$$

then

$$A\xi \rightarrow \lambda(k : X\nu\xi).(X\nu\xi : k) \in goto(q, \delta) \wedge (X\nu\xi : k) \rightarrow^* x_{i+1} \ldots x_\ell$$

So with use of the definitions of the functions *goto*, *ini*, [] and $\overline{[]}$ we can write for the second set of equation 2.6:

$$\{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{A\xi\lambda\nu\ell X}((\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(A\xi, \ell) \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge\ k : X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon \wedge (A\xi \rightarrow \lambda\delta.(X\nu\xi : k), \ell) \in [goto(q, \delta)](i))\}$$

2.5 review

$$\alpha \Rightarrow \beta \quad \equiv \quad \exists_{A\gamma\delta}(\alpha = A\gamma \wedge \beta = \delta\gamma \wedge A \rightarrow \delta)$$

$$k : : V^* \rightarrow V^\ell \quad , \quad 0 \leq \ell \leq k$$
$$: k \ : V^* \rightarrow V^*$$

prefix function

$$\begin{aligned} k:\alpha &= \epsilon, k = 0 \vee \alpha = \epsilon \\ 1:X\alpha &= X, X \in V \\ k:X\alpha &= X, X \in V_T \\ k:X\alpha &= X\alpha, X \in V_N \wedge k \geq |X\alpha| \\ k:\alpha &= \text{prefix of } \alpha \text{ with length } k, \textit{otherwise} \end{aligned}$$

suffix function

$$\alpha = k:\alpha \mathbin{++} \alpha:k$$

$$ini : S \rightarrow \mathcal{P}(I)$$

$$\begin{aligned} ini(q) = \{B\mu \rightarrow .\nu\mu \mid \exists_{\alpha\beta\gamma\delta}(\gamma \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* B\mu\delta \\ \wedge k:B\mu\delta = B\mu \wedge B \rightarrow \nu \wedge \nu \neq \epsilon)\} \end{aligned}$$

$$goto : S \times (V_T \cup V_N V^\ell) \rightarrow S, 0 \leq \ell < k$$

$$\begin{aligned} goto(q,\delta) = \{\gamma \rightarrow \alpha\lambda\delta.\beta \mid \gamma \rightarrow \alpha.\lambda\delta\beta \in (q \cup ini(q)) \\ \wedge \delta = k:\delta\beta \wedge \lambda \rightarrow^* \epsilon\} \end{aligned}$$

$$[q] : N \rightarrow \mathcal{P}(II)$$

$$\begin{aligned} [q](i) = \overline{[q]}(x_{i+1}, i+1) \\ \cup \{(\gamma \rightarrow \alpha.\beta, i) \mid \gamma \rightarrow \alpha.\beta \in q \wedge \beta \rightarrow^* \epsilon\} \end{aligned}$$

$$\overline{[q]} : (V_T \cup V_N V^\ell) \times N \rightarrow \mathcal{P}(II), 0 \leq \ell < k$$

$$\begin{aligned} \overline{[q]}(\delta,i) = \{(\gamma \rightarrow \alpha.\lambda\delta\mu, j) \mid \gamma \rightarrow \alpha.\lambda\delta\mu \in q \wedge \lambda \rightarrow^* \epsilon \\ \wedge (\gamma \rightarrow \alpha\lambda\delta.\mu, j) \in [goto(q,\delta)](i)\} \\ \cup \{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{A\xi\lambda\delta\nu\ell}((\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(A\xi, \ell) \\ \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge k:X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon \\ \wedge (A\xi \rightarrow \lambda\delta.(X\nu\xi : k), \ell) \in [goto(q,\delta)](i))\} \end{aligned}$$

2.6 correctness

The recognizer works for fast all context-free grammars. Only for grammars that cause a cyclicity problem (for the computation of a function call $f(arguments)$ the function f is called with the same arguments), the recognizer does not work. The proof is the construction of the recognizer in section 2.4. If the recognizer has no cyclicity problem then the computation of the recognizer terminates because if $x_{n+1} \notin V$ then $\forall_q goto(q, x_{n+1}) = \emptyset$.

A call $[q](i)$ may lead to a call $\overline{[q]}(x_{i+1}, i+1)$. Because a call of a function $\overline{[q]}(\delta, i)$ cannot lead to a call $[q'](\ell)$ with $\ell < i$, a call $[q](i)$ will never lead to a call $[q](i)$.

For a call $\overline{[q]}(\delta, i)$ the situation is different. A call $\overline{[q]}(\delta, i)$ may lead to a call $[goto(q, \delta)](i)$ but this can never lead to a call $\overline{[q]}(\delta, i)$ (because of the increment of index i). But a call $\overline{[q]}(\delta, i)$ may also lead to a call $\overline{[q]}(A\xi, \ell)$ with $\ell \geq i$ if $A\xi \rightarrow .\lambda X\nu\xi \in ini(q)$, $\lambda \rightarrow^* \epsilon$, $k : X\nu\xi = \delta$ and $(A\xi \rightarrow \lambda\delta.(X\nu\xi : k), \ell) \in [goto(q, \delta)](i)$. If $X\nu\xi : k \not\rightarrow^* \epsilon$ then $\ell > i$. If $X\nu\xi : k \rightarrow^* \epsilon$ it is possible that $\ell = i$. In that case, if $A\xi = \delta$ or $\overline{[q]}(A\xi, i)$ leads to a call $\overline{[q]}(\delta, i)$ then a cyclicity problem exists. Thus if there is a sequence of initial items $\delta \rightarrow .\lambda_n X_n \nu_n \xi_n \in ini(q)$, $k : X_{i+1}\nu_{i+1}\xi_{i+1} \rightarrow .\lambda_i X_i \nu_i \xi_i \in ini(q) (1 \leq i < n)$ and $k : X_1\nu_1\xi_1 \rightarrow .\lambda X\nu\xi \in ini(q)$ with $(X\nu\xi : k), \lambda \rightarrow^* \epsilon, \forall_{i, 1 \leq i \leq n} : (\lambda_i \rightarrow^* \epsilon \wedge (X_i\nu_i\xi_i : k) \rightarrow^* \epsilon) \wedge k : X\nu\xi = \delta$ then the recognizer may have a cyclicity problem.

One can check for a given grammar if this problem may occur. If it occurs, it is possible to overcome the problem by memoization of the functions $\overline{[q]}$. Memoized functions memorize for which arguments they have been called. They do not recompute previous computed results. If a call $\overline{[q]}(\delta, i)$ occurs while computing the function call $\overline{[q]}(\delta, i)$, the parser must stop the computation of the last call $\overline{[q]}(\delta, i)$. The parser will still recognize all sentences the grammar can generate, but an infinity of parse trees that only differ in the derivation $\delta \Rightarrow^* \delta\nu(\nu \rightarrow^* \epsilon)$ will not be computed.

Chapter 3

complexity of RAM parsers

3.1 introduction

In this section we will look at the complexity of the RAM parser. The theoretical upper bound of the space complexity will be compared with some practical results.

3.2 memoization

The functions $[q]$ and $\overline{[q]}$ are implemented as memo-functions. The first time a function is called with a specific argument the function result is memoized. The function does not recompute its result for that argument after this first call. Before recognizing all the possible states are computed (thus all the possible function calls $goto(q, \delta)$). For every state q the corresponding function call $ini(q)$ is computed.

3.3 space complexity

3.3.1 number of items

First we will compute an approximation of the maximum number of different items of a parser for grammar G[1].

An average of the number of symbols in the right-hand-side of a grammar rule is $\frac{|G|}{\#(G)}$ with $\#(G)$ the number of grammar rules and $|G|$ the number of symbols in the right-hand-sides of the grammar rules of G. An item is of the form $A\gamma \rightarrow \alpha.\beta$ ($A \in V_N, \alpha, \beta, \gamma \in V^*$) with $\text{tail}(\alpha\beta) = \gamma$. On average every item in a RAM parser has (at most) $\frac{|G|}{\#(G)} + k - 1$ symbols in its right-hand-side. For an undotted item (an item without a dot) $A \rightarrow B\ C\ D$ with $k = 1$ the dot can be placed at three positions, resulting in the three items:

- $A \rightarrow B\ .\ C\ D$
- $A \rightarrow B\ C\ .\ D$
- $A \rightarrow B\ C\ D\ .$

[1] In this section initial items are not considered as items. For computing the space complexity of the RAM parser, the number of states is important. Only the root state contains an initial item.

So for a $k = 1$ parser of grammar G the maximum number of items is $\mid G \mid$.

For an undotted item $A\ B\ C \rightarrow D\ E\ F\ G\ B\ C$ ($A, B, C, D, E, F, G \in V_N$, $k = 3$) the dot can be placed at two positions (only k-prefixes can be shifted in the item, not parts of a k-prefix), resulting in the two items:

- $A\ B\ C \rightarrow D\ E\ F\ .\ G\ B\ C$
- $A\ B\ C \rightarrow D\ E\ F\ G\ B\ C\ .$

For an undotted item with ℓ symbols in the right-hand-side and ℓ not a multiple of k there are at most $(\ell + k - 1)/k$ positions to place the dot. In the case of terminal symbols in the right-hand-side the dot can be placed at more positions. An approximation of the average number of dot positions in the right-hand-side of an undotted item is $(\frac{\mid G \mid}{\#(G)} + k - 1 + k - 1)/k \approx \frac{\mid G \mid}{\#(G)k}$ ($k \in \{1,2,3,4\}$).

A RAM parser has at most $\sum_{\ell=1}^{k} \#(G) \mid V \mid^{\ell-1}$ undotted items. So an approximation of the number of items of a RAM parser is

$$
\begin{aligned}
& (\sum_{\ell=1}^{k} \#(G) \mid V \mid^{\ell-1}) \frac{\mid G \mid}{\#(G)k} \\
= & \sum_{\ell=1}^{k} (\#(G) \mid V \mid^{\ell-1} \frac{\mid G \mid}{\#(G)k}) \\
= & \sum_{\ell=1}^{k} \frac{\mid V \mid^{\ell-1} \mid G \mid}{k}
\end{aligned}
$$

So the number of items is in an approximation of the worst case of order of magnitude $O(\frac{1}{k} \mid V \mid^{k-1} \mid G \mid)$.

In practice the upper bound $\sum_{\ell=1}^{k} \#(G) \mid V \mid^{\ell-1}$ for the number of undotted items is unrealistic. Linguists write grammars with few terminal symbols in rewriting rules. For example they will not write a rule

$$\ldots \rightarrow \ldots \text{noun} \ldots$$

but

$$\ldots \rightarrow \ldots \text{NP} \ldots$$

together with the rule NP $\rightarrow$ noun. Also more realistic is the upper bound $\sum_{\ell=1}^{k} \#(G) \mid V_N \mid^{\ell-1}$.

This upper bound has been derived from the form $A\gamma \rightarrow \alpha.\beta$ of an item with tail($\alpha\beta$) = γ and $A \rightarrow$ prefix($\alpha\beta$). In practice a lot of possible strings γ ($\gamma \in V_N^*, \mid \gamma \mid < k$) do not occur in combination with a specific grammar rule. For a few very small grammars the value of t in $\frac{1}{k} \mid V \mid^t \mid G \mid$ has been computed (see appendix A for the results). A result is that the value of t will be less than $k - 1$. For the small grammars the value of t strongly depends on the mean length of the right-hand-sides. It seems that a lower

bound of the number of items for our small test grammars is of order of magnitude $O(\frac{1}{k} \mid V_N \mid^{\frac{1}{3}(k-1)} \mid G \mid)$. For a fictitious minimal grammar with 800 grammar rules, 30 nonterminal symbols and a grammar size ($\mid G \mid$) of 1600 the number of different items for different values of k can be seen in figure 3.1.

k	$\frac{1}{k} \mid V_N \mid^{\frac{1}{3}(k-1)} \mid G \mid$
1	1600
2	2500
3	5100

Figure 3.1: number of items with respect to the number of look-aheads

3.3.2 number of states

A state is a set of items with the same last transition, thus the same look-ahead string before the dot. For one collection of g items with the same look-ahead string before the dot there could be in the worst case $2^g - 1$ different sets. The maximum number of different look-ahead strings is $\mid V_N \mid \cdot \mid V \mid^{k-1} + \mid V_T \mid$. The maximum number of items in a state with the same look-ahead string before the dot is smaller than the maximum number of items. So the maximum number of states is of order of magnitude $O((\mid V_N \mid \cdot \mid V \mid^{k-1} + \mid V_T \mid)(2^{\frac{1}{k}\mid V_N \mid^{k-1}\mid G \mid} - 1)) \equiv O(2^{\frac{1}{k}\mid V_N \mid^{k-1}\mid G \mid})$.

For LR parsers the space complexity is of order of magnitude $O(2^{\mid V_T \mid^{k-1}\mid G \mid})$ (the length of the look-ahead is $k-1$). So there is not an important difference in space complexity. For RAM parsers as well for LR parsers the space complexity is very bad. In practice the size of the parsers is much better (smaller).

For the very small test grammars the number of states is displayed against the values of k (figure 3.2).

3.4 time complexity

Let n be the sentence length and s the maximum number of states. We will compute a rough calculation of the (time-)complexity of the recognizer.

There are $O(n \mid V_N \mid^k s)$ different invocations of functions $[q]$ and $\overline{[q]}$. Each invocation of a function $\overline{[q]}$ calls a function $[goto(q,\delta)](i)$. This call returns a set of $O(n)$ elements. An invocation of $\overline{[q]}$ leads to $O(n)$ invocations of functions $\overline{[q]}(A\xi,\ell)$. This results in $O(n)$ sets of $O(n)$ elements. Together with merging these sets and removing duplicates each invocation of a function $\overline{[q]}$ takes $O(n^2)$ time. Hence, the total time complexity is $O(n^3 s \mid V_N \mid^k)$.

Without memo-functions every function result has to be computed by invocation. Because every function $[q]$ and $\overline{[q]}$ invokes other functions, the recognizing takes exponential time without memo-functions. Memo-functions are not necessary if the recognizing process is deterministic. In that case the recognizer knows at every moment what to do and it recognizes a sentence on the basis of only one parse tree. In the case of a fully non-deterministic recognizing process memo-functions are necessary because of the exponential time complexity.

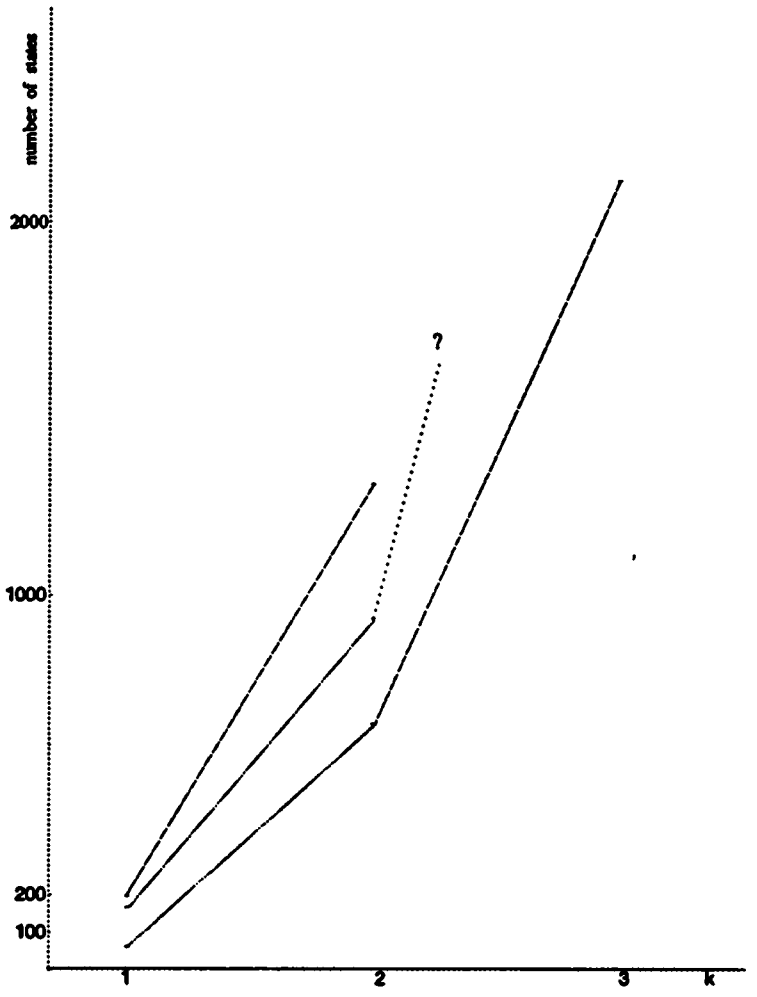


Figure 3.2: number of states with respect to the number of look-aheads

3.5 conclusions

The space complexity of the recursive ascent Marcus parser is of order of magnitude $O(2^{\frac{1}{k}|V_N|^{k-1}|G|})$. The time complexity is of order of magnitude $O(n^3)$ when memoizing the function results of the functions [] and $\overline{[]}$. In practice, the space complexity is less than $O(2^{\frac{1}{k}|V_N|^{k-1}|G|})$ but the number of states of the parser still highly grows with respect to the length of the look-aheads. Time and space complexity of the RAM parser do not really differ from the time and space complexity of LR(k) parsers.

Chapter 4

the use of attributes in RAM parsers

4.1 introduction

Context-free grammars can be extended with attributes. Attributes are useful to make a distinction within syntactical categories (attributes act as features) and they are useful to work with context-sensitive information. Attributes are associated with the symbols of the grammar (different attributes can be associated with different symbols). Attribute-evaluation rules are associated with the grammar rules.

4.2 definition

An attribute grammar G is an extension of a context-free grammar G_{CFG} with $G_{CFG} = (V_N, V_T, \mathrm{P}, \mathrm{S})$. V_N is the set of nonterminal symbols, V_T is the set of terminal symbols, S is the start symbol and P is a set of grammar rules. A grammar rule r in P is written by:

$$\mathrm{r} : A \rightarrow A_1 \ldots A_n$$

where $A \in V_N$, $A_i \in V_N \cup V_T$ $(1 \leq i \leq n)$. For $n = 0$, A rewrites to the empty string ϵ.

Each nonterminal $X \in V_N$ has a set of synthesized attributes $Syn(X)$ and a set of inherited attributes $Inh(X)$. Each terminal $x \in V_T$ has only a set of synthesized attributes $Syn(x)$. $Att(Y) = Inh(Y) \cup Syn(Y)$ $(Y \in V_N \cup V_T)$. Each attribute $a \in Att(Y)$ has a finite domain $\mathrm{W}(a)$ for all $Y \in V_N \cup V_T$.

A rewriting $\mathrm{r} : A \rightarrow A_1 \ldots A_n$ has attribute occurrence (a, k) if $a \in Att(A_k)$. With each rewriting a finite set of semantic rules is associated. With a semantic rule of a rewriting r, the occurrences of synthesized attributes of A and the occurrences of inherited attributes of A_k $(1 \leq k \leq n)$ can be computed from the values of the inherited attributes of A and the synthesized attributes of A_k $(1 \leq k \leq n)$. A semantic rule f related with a rewriting r is of the type :

$$f : \mathrm{W}(a_1) \times \mathrm{W}(a_2) \times \ldots \times \mathrm{W}(a_m) \rightarrow \mathrm{W}(a),\ m > 0$$

with $a \in Syn(A) \cup Inh(A_1) \cup \ldots \cup Inh(A_N)$ and $\forall_i 1 \leq i \leq m : a_i \in Inh(A) \cup Syn(A_1) \cup \ldots \cup Syn(A_n)$.

4.3 effect on parsing

Making a distinction within a syntactical category with attributes (thus defining subcategories with use of attributes), reduces the number of grammar rules, because otherwise the possible values of the attributes have to be described in terms of syntactical categories. Also the use of context-sensitive information by attributes can bring about a reduction of the number of grammar rules. For example the grammar rules

A → B C D

A → B C E

(A ∈ V_N; B, C, D, E ∈ V) can be merged in the rule

A → B C optional_D optional_E

with an attribute optional_D_exists associated with symbol optional_D and an attribute optional_E_exists associated with symbol optional_E and attribute-evaluation rule:

if (optional_D_exists & optional_E_exists) |

NOT(optional_D_exists | optional_E_exists)

then not ok

fi

Although evaluation of attributes takes time, evaluation of attributes while recognizing/parsing (attribute-directed parsing) can speed up the parsing process. In the preceding example in the cases the attribute-evaluation indicates that the rule A → B C optional_D optional_E cannot be applied. Symbol A cannot be reduced. This could have effect on the following parsing actions.

Attribute-directed parsing can also speed up the parsing process if the attributes are used to specify syntactical (sub)categories and if such a specification has a context-sensitive aspect (as with the plurality of a verb and the related subject).

For grammars that describe natural languages, especially synthesized attributes are used. The properties of the symbols on the right-hand-side determine the properties of the symbol(s) on the left-hand-side and not the other way round.[1] Inherited attributes can be used to speed up the parsing process. For example when recognizing the grammar rule

A → B C

(A, C ∈ V_N, B ∈ V) recognizing B may return an attribute value that conflicts with an attribute value that is computed while recognizing C. At the moment the conflict occurs the recognizing of the rule can be stopped. If the conflicting attribute value of symbol B is passed on to C (inherited attribute), the conflict can be recognized as soon as the conflicting value in the recognizing of C has been computed.

[1] Attributes used with a grammar for a formal language are used among other things for semantic representation. In that case it will be necessary to use inherited attributes. For example for passing on information about the declaration of variables.

4.4 evaluation of attributes

The evaluation of inherited attributes with bottom-up parsers is a problem. For example when recognizing a noun as the first symbol of a sentence and grammar rules:

- S → NP VP
- NP → noun
- NP → NP PP

it may be unclear what the values of the inherited attributes of the NP in the rule NP → noun are, because this NP can be the NP from the rule S → NP VP or from the rule NP → NP PP. Only if the attribute evaluation rules meet specific conditions, it is possible to compute the values of the inherited attributes during parsing.

From the definition of the functions $\overline{[q]}$, it turns out that the recognizer/parser works in two phases.

$$\begin{aligned}\overline{[q]}(\delta,i) \;=\; & \{(\gamma \to \alpha.\lambda\delta\mu, j) \mid \gamma \to \alpha.\lambda\delta\mu \in q \wedge \lambda \to^* \epsilon \\ & \qquad \wedge\, (\gamma \to \alpha\lambda\delta.\mu, j) \in [goto(q,\delta)](i)\} \\ & \cup \{(\gamma \to \alpha.\beta, j) \mid \exists_{A\xi\lambda\delta\nu\ell}((\gamma \to \alpha.\beta, j) \in \overline{[q]}(A\xi,\ell) \\ & \qquad \wedge\, A\xi \to .\lambda X\nu\xi \in ini(q) \wedge\; k : X\nu\xi = \delta \wedge \lambda \to^* \epsilon \\ & \qquad \wedge\, (A\xi \to \lambda\delta.(X\nu\xi : k), \ell) \in [goto(q,\delta)](i))\}\end{aligned}$$

The parser first 'walks' from the left to the right through items (by calls of the form $goto(q,\delta)$). As soon as the end of an item is reached (the item is a final item of the form $\gamma \to \alpha.\beta$ with $\beta \to^* \epsilon$) the second phase starts. An index-item related with the final item is returned. For every step from the left to the right in the first phase, the dot in the corresponding index-item is placed one step back from the right to the left.

It would be nice to evaluate the synthesized attributes in the first phase. If a conflict between attribute values appears, the parser would be able to stop the recognizing of the conflicting item immediately. If the conflict appears in the second phase, the parser wastes time. This is an important point if the grammar has a lot of attributes which work like filters (as α_exists attributes). The attribute component of the grammar is a little more complicated because the parser has to know when an evaluation rule can be applied.

The parser has enough information to evaluate synthesized attributes in the first phase. A step from the left to the right in an item is made when the parser shifts or reduces a symbol. If the parser shifts a symbol, the symbol has been recognized and its attribute values are known. If the parser reduces a symbol, the rewriting of that symbol has been recognized. Thus the attribute values of the rewriting are known and the values of the synthesized attributes of the reduction symbol can be computed.

4.4.1 evaluation in the first phase

Computing the synthesized attributes of the right-hand-side of an item in the first phase, requires that the values of the attributes of the symbols left from the dot are dragged along when moving the dot to the right. Thus the values of the attributes have to be dragged along when the parser makes a transition to a new state. There are two possibilities.

The parser works on the same states as for the attribute-free version of the parser, but during parsing an administration with attribute evaluation details is kept up. Another possibility is to compute states with items *and* attribute evaluation details. So states have to be generated during parsing, they cannot be computed before parsing. For an item $\gamma \rightarrow \alpha.\beta$ with different attribute values (here the values of the synthesized attributes of α) in different cases, the item has to be duplicated because the attribute values are connected with the items. So the states are associated with a position in the sentence that is parsed. The nature of LR parsing is affected with the second method.

The first method (keeping up an attribute administration) requires that for all items in a state the attribute administration should keep accounts of the values of the synthesized attributes before the dot. First we will look at a minimal attribute administration.

Suppose the parser concatenates the attribute values of the symbols of the last transition with the list of attribute values of the symbols of transitions that are already made. For an item $\gamma \rightarrow NP\ \lambda\delta.\xi$ in state S with $\delta,\xi \in V^*$, $NP \in V$, $\gamma \in V_N$, $\lambda \in V_N^*$ and $\lambda \rightarrow^* \epsilon$, the parser has made two transitions from state S' with initial item $\gamma \rightarrow .\ NP\ \lambda\delta\xi$ to state S with the first transition on symbol NP and the second transition on symbol δ. So there is an attribute administration list $Attr_\delta : (Attr_{NP} : tail)$ with $Attr_\delta$ the values of the synthesized attributes of δ and $Attr_{NP}$ the values of the synthesized attributes of NP. At this moment it is not always possible to reconstruct the attribute evaluation. It is clear that $Attr_\delta$ must be associated with δ in the item $\gamma \rightarrow NP\ \lambda\delta.\xi$. But for $Attr_{NP}$ it is possible that there is more than one NP with which it can be associated. It is possible that $\lambda \equiv \lambda_1\ NP\ \lambda_2$, with $\lambda_1, \lambda_2 \in V_N^*$, $\lambda_1, \lambda_2 \rightarrow^* \epsilon$ and $NP \rightarrow^* \epsilon$. In that case the attribute administration does not know to which NP $Attr_{NP}$ belongs. This problem occurs because the parser simply skips ϵ-rewritings. (The problem would be solvable if for every two items $\gamma_1 \rightarrow \alpha_1\lambda_1\delta.\xi_1$ and $\gamma_2 \rightarrow \alpha_2\lambda_2\delta.\xi_2$ $(\lambda_1, \lambda_2 \in V_N^*)$ in a state derived from a state with the (initial) items $\gamma_1 \rightarrow \alpha_1.\lambda_1\delta\xi_1$ and $\gamma_2 \rightarrow \alpha_2.\lambda_2\delta\xi_2$ on a transition on δ the equation $|\lambda_1| = |\lambda_2|$ holds.) Thus a list of evaluated attributes of transition symbols does not satisfy.

It is necessary to administrate more than a list of attribute values of transition symbols. Therefore a (dynamic) attribute administration is associated with a state, as soon as the parser makes a transition to that state. In this administration all the items of the state are copied and the values of the synthesized attributes are associated with the correct symbols. The administration of the previous/original state has to be available so that the values of the attributes of previous transitions can be associated with the correct symbols. The values of the synthesized attributes of the left-hand-side of an item can be computed when the right-hand-side has been recognized. They can be associated with the left-hand-side in the corresponding index-item. For different values of attributes of δ, different calls $\overline{[q]}(\delta, i)$ have to be made. So the effect of memoizing function calls reduces.

4.4.2 evaluation in the second phase

It is more natural and less expensive to handle the attribute evaluation in the second phase, thus to link the attribute values with the index-items. Evaluating attributes in the first phase requires that attribute values are associated with all items in a state. Evaluating attributes in the second phase requires only that attributes are associated with one or more index-items. These index-items are just like attributes results of the actual parsing process, items are static objects of the parser.

When replacing the dot from the right of a $\lambda\delta$ to the left of a $\lambda\delta$ in an index-item $(\gamma \rightarrow \alpha\lambda\delta.\mu, j)$, the values of the attributes of the symbols $\lambda\delta$ are connected with the index-item. The symbols δ in a call $\overline{[q]}(\delta, i)$ are attributed symbols. So there is a negative effect on memoizing function calls.

With attribute evaluation in the second phase, the definitions of the functions $[q]$ and $\overline{[q]}$ of the RAM recognizer differ from the functions $[q]$ and $\overline{[q]}$ as defined in the original attribute-free version of the parser. An attributed index-item is an index-item with a corresponding attribute administration. The set of all attributed index-items is AII. The set A is the set of all attribute administrations.

$$[q] \; : \; N \rightarrow \mathcal{P}(AII)$$

$$\begin{aligned}[q](i) \; = \; & \overline{[q]}(x_{i+1}, Eval(x_{i+1}), i+1) \\ & \cup \{(\gamma \rightarrow \alpha.\beta, Attr(\gamma \rightarrow \alpha.\beta), i) \mid \gamma \rightarrow \alpha.\beta \in q \wedge \beta \rightarrow^* \epsilon\}\end{aligned}$$

$$\overline{[q]} \; : \; (V_T \cup V_N V^\ell) \times A \times N \rightarrow \mathcal{P}(AII), 0 \leq \ell < k$$

$$\begin{aligned}\overline{[q]}(\delta, attr\_\delta, i) \; = \; & \{(\gamma \rightarrow \alpha.\lambda\delta\mu, Insert(attr\_\delta, attrib), j) \mid \\ & \quad \gamma \rightarrow \alpha.\lambda\delta\mu \in q \wedge \lambda \rightarrow^* \epsilon \\ & \quad \wedge (\gamma \rightarrow \alpha\lambda\delta.\mu, attrib, j) \in [goto(q, \delta)](i)\} \\ & \cup \{(\gamma \rightarrow \alpha.\beta, attr\_1, j) \mid \exists_{A\xi\lambda\delta\nu\ell\, attr\_2}(\\ & \quad (\gamma \rightarrow \alpha.\beta, attr\_1, j) \in \overline{[q]}(A\xi, \\ & \qquad Eval(Insert(attr\_\delta, attr\_2)), \ell) \\ & \quad \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge \; k : X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon \\ & \quad \wedge (A\xi \rightarrow \lambda\delta.(X\nu\xi : k), attr\_2, \ell) \in [goto(q, \delta)](i))\}\end{aligned}$$

Eval(α): evaluate synthesized attributes of α if $\alpha \in V_T$.
Eval(α): evaluate synthesized attributes of γ if $(\gamma \rightarrow \xi_1.\xi_2, attrib\_\xi_2, i)$ is an attributed index-item and α represents the values of the attributes of $\xi_1\xi_2$.
Insert($attr\_\delta, attributes$): insert $attr\_\delta$ in the attribute administration *attributes*.
Attr($\gamma \rightarrow \alpha.\beta$): returns the attribute administration of $\gamma \rightarrow \alpha.\beta$ with the values of the synthesized attributes of β for $\beta \rightarrow^* \epsilon$.

4.5 conclusions

Evaluation of attributes while parsing can speed up the parsing process. If a bottom-up parser is used, only synthesized attributes can be evaluated while parsing. Evaluation of synthesized attributes can be done as soon as possible, but this will take a lot of administration. More natural is to postpone the attribute evaluation till an index-item is returned.

Chapter 5

Marcus LR parsers

5.1 introduction

The RAM parser uses constituent look-aheads. The look-ahead symbols were represented in the items, so with more look-aheads, the parser size (the number of states in the parser) grows rapidly. In the case of LR(k) parsing the growth of the parser related to the number of look-aheads is also a problem. With LALR(k) parsers the parser size is limited to the size of a LR(0) parser. Terminal look-aheads are added to a LR(0) parser. States of a LR(k) parser which only differ in the look-ahead strings of the items of these states are merged in the LALR(k) variant of this parser.

We want to make a LALR-like variant of the parser with constituent look-aheads. Thus we want to add constituent look-aheads to a LR(0) parser. In the RAM parser there could be two states q_1 and q_2 with q_1:

A B → C D . E B

G J → D . J

and q_2:

A E → C D . E E

G H → D . H

These states only differ in the look-ahead parts of the items (respectively the symbols B,J and E,H). In the LALR variant of the parser (the Marcus LR parser, MaLR parser) the states q_1 and q_2 are merged into the state with items and look-aheads:

A → C D . E {B, E}

G → D . {J, H}

The construction of the MaLR parser with one look-ahead (MaLR(1)) is described in the next sections.

5.2 functional LR(0) parser

Starting-point is the LR(0) parser in functional notation as described in [Leermakers, 1991a]:

$$\alpha \Rightarrow \beta \equiv \exists_{A\delta\gamma}(\alpha = A\gamma \wedge \beta = \delta\gamma \wedge A \rightarrow \delta \wedge \delta \neq \epsilon)$$

$$ini(q) = \{B \rightarrow .\nu \mid \exists_{A\alpha\beta\delta}(A \rightarrow \alpha.\beta \in q \wedge \beta \Rightarrow^* B\delta \wedge B \rightarrow \nu)\}$$

$$goto(q, X) = \{A \rightarrow \alpha X.\beta \mid A \rightarrow \alpha.X\beta \in (q \cup ini(q))\}$$

$$\begin{aligned}
[q](i) &= \{(A \rightarrow \alpha.\beta, j) \mid A \rightarrow \alpha.\beta \in q \wedge \beta \rightarrow^* x_{i+1} \ldots x_j\} \\
&= \overline{[q]}(x_{i+1}, i+1) \\
&\quad \cup \{(A \rightarrow \alpha.\beta, j) \mid \exists_B(B \rightarrow .\epsilon \in ini(q) \\
&\qquad\qquad \wedge (A \rightarrow \alpha.\beta, j) \in \overline{[q]}(B, i))\} \\
&\quad \cup \{(A \rightarrow \alpha., i) \mid A \rightarrow \alpha. \in q\}
\end{aligned}$$

$$\begin{aligned}
\overline{[q]}(X, i) &= \{(A \rightarrow \alpha.\beta, j) \mid \exists_\gamma(A \rightarrow \alpha.\beta \in q \\
&\qquad\qquad \wedge \beta \Rightarrow^* X\gamma \wedge \gamma \rightarrow^* x_{i+1} \ldots x_j)\} \\
&= \{(A \rightarrow \alpha.X\beta, j) \mid A \rightarrow \alpha.X\beta \in q \\
&\qquad\qquad \wedge (A \rightarrow \alpha X.\beta, j) \in [goto(q, X)](i)\} \\
&\quad \cup \{(A \rightarrow \alpha.\beta, j) \mid \exists_{C\delta m}((A \rightarrow \alpha.\beta, j) \in \overline{[q]}(C, m) \\
&\qquad\qquad \wedge C \rightarrow .X\delta \in ini(q) \\
&\qquad\qquad \wedge (C \rightarrow X.\delta, m) \in [goto(q, X)](i))\}
\end{aligned}$$

5.3 auxiliary functions

We need three auxiliary functions. The function *goto* for computing a sequence of transitions, the function *f* for computing the look-aheads (followers) of an item and the function *parse* for computing the possible parses of a set of (non)terminals.

$$goto : S \times V^* \rightarrow S$$

$$\begin{aligned}
goto(q, \epsilon) &= q \\
goto(q, X\delta) &= goto(goto(q, X), \delta) \\
goto(q, X) &= \{A \rightarrow \alpha X.\beta \mid A \rightarrow \alpha.X\beta \in (q \cup ini(q))\}
\end{aligned}$$

$$f \;:\; S \times (I \cup V_T) \to \mathcal{P}(V)$$

$$\begin{aligned} f(q, A \to \alpha.\beta) \;=\; & \{X \mid \exists_{\mu}(\beta = X\mu \wedge X \in V)\} \\ & \cup \{Y \mid \exists_{p\gamma\delta BY}(\beta = \epsilon \wedge q = goto(p,\alpha) \\ & \qquad \wedge B \to \gamma.A\delta \in (p \cup ini(p)) \\ & \qquad \wedge Y \in f(goto(p,A), B \to \gamma A.\delta))\} \\ f(q,x) \;=\; & \{X \mid \exists_{\alpha\beta A}(A \to \alpha.x\beta \in q \wedge X \in f(q, A \to \alpha x.\beta) \wedge x \in V_T)\} \end{aligned}$$

The function f computes the set of symbols that immediately can be reached after the current recognition. For an item $A \to \alpha.\beta$ in state q the current recognition is α in the rewriting $A \to \alpha\beta$ in state q and the look-ahead of $A \to \alpha.\beta$ in state q is defined as the first symbol of β (if $\beta \neq \epsilon$) or (if $\beta = \epsilon$) the first symbol(s) that can be recognized after the rule $A \to \alpha$. (for example the symbol B in item $C \to .A\ B$ in state q' with $goto(q', \alpha) = q$).

The set $\mathcal{F}$ is the set of all possible indexed parses of all (non)terminals ($\mathcal{F} = V \times N$). An indexed parse consists of the rewriting to terminal symbols (the parse) of a (non)terminal, together with the index of the input symbol up to which the (non)terminal rewrites. A symbol f in $(f,j) \in \mathcal{F}$ rewrites up to the jth symbol of the input. The function *parse* is of type $\mathcal{P}(V) \times N \to \mathcal{F}$. A function call $parse(Z,i)$ with $Z \in \mathcal{P}(V)$ returns the set $\{(z,j) \mid z \in Z \wedge z \to^* x_{i+1} \ldots x_j\}$.

5.4 constructing the MaLR(1) parser: step 1

We are now able to write a first version of the MaLR(1) parser. The parser will only shift a terminal symbol if there is a look-ahead (follower) of that symbol that can be recognized after the shift action. The parser will only recognize a rewriting $B \to \epsilon$ if there is a look-ahead of the item $B \to \epsilon.$ that can be recognized. The parser will do the same for a reduction of $A \to \alpha$. The new functions $[q]$ have definition:

$$\begin{aligned} [q](i) \;=\; & \{(A \to \alpha.\beta, j) \mid (A \to \alpha.\beta, j) \in \overline{[q]}(x_{i+1}, i+1) \\ & \qquad \wedge parse(f(q, x_{i+1}), i+1) \neq \emptyset\} \\ & \cup \{(A \to \alpha.\beta, j) \mid \exists_B(B \to .\epsilon \in ini(q) \\ & \qquad \wedge parse(f(q, B \to .\epsilon), i) \neq \emptyset \\ & \qquad \wedge (A \to \alpha.\beta, j) \in \overline{[q]}(B, i))\} \\ & \cup \{(A \to \alpha., i) \mid \exists_{G\alpha'}(A \to \alpha. \in q \\ & \qquad \wedge G = parse(f(q, A \to \alpha.), i) \\ & \qquad \wedge (G \neq \emptyset \vee \alpha = \alpha' \perp))\} \end{aligned}$$

The functions $\overline{[q]}$ will be the same as before.

The set $f(q, x_{i+1})$ is the set of immediate followers of x_{i+1} in the recognition process. So if no element Y from the set $f(q, x_{i+1})$ rewrites to $x_{i+2} \ldots x_\ell$ ($\ell > i+1$) or to ϵ, shifting the symbol x_{i+1} is useless. The set $f(q, B \to .\epsilon)$ is the set of immediate followers of B in the recognition process. So if no element Y from the set $f(q, B \to .\epsilon)$ rewrites to $x_{i+1} \ldots x_\ell$ ($\ell > i$) or to ϵ, recognizing the rule $B \to \epsilon$ leads to a situation in which no further recognition is possible. Recognizing the rule $B \to \epsilon$ is in this case useless.

For the same reason there is no need to reduce a recognized rule $A \rightarrow \alpha$ if the set of recognized followers (followers that rewrite to the next part of the input or to ϵ) is empty (except in the case that the last symbol of the sentence ($\perp$) has been reached).

Thus if the LR(0) parser is correct[1] than this version with constituent look-ahead is also correct.

5.5 constructing the MaLR(1) parser: step 2

The new definition of the functions $[q]$ is not beautiful. Look-aheads are recognized, but the result is thrown away. In two steps we will construct new definitions of $[q]$ and $\overline{[q]}$ which do not waste computed results. First the set of recognized look-aheads of x_{i+1} will be passed on to a function call $\overline{[q]}(x_{i+1}, i+1)$ and the set of recognized look-aheads of an item $B \rightarrow \epsilon.$ will be passed onto a function call $\overline{[q]}(B, i)$ and the set of recognized look-aheads of a final item $A \rightarrow \alpha.$ will be merged with the corresponding index item. Thus an index item is now of type $I \times N \times \mathcal{P}(\mathcal{F})$. The set F in index item $(item, i, F)$ is the set of look-aheads of the (recognized) left-hand-side of *item*. If $(Y, \ell) \in F$ and $(A \rightarrow \alpha.\beta, i, F)$ an index item produced in the recognition of a sentence $x_1 \ldots x_n$ then $Y \rightarrow^* x_{i+1} \ldots x_\ell$. This condition has to hold after every step in the construction of the MaLR(1) parser. The set $F \in \mathcal{P}(\mathcal{F})$ in a function call $\overline{[q]}(X, i, F)$ is the set of recognized look-aheads of X.

The set F of recognized look-aheads of X will be passed onto a function call $[goto(q, X)](i)$. So we get function calls $[q](i, F)$ with F the set of recognized look-aheads of the last transition symbol (the transition symbol that causes the transition to q).

$$
\begin{aligned}
[q](i,F) \;=\; & \{(A \rightarrow \alpha.\beta, j, G) \mid \exists_H((A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(x_{i+1}, i+1, H) \\
& \qquad \wedge\, H = parse(f(q, x_{i+1}), i+1) \\
& \qquad \wedge\, H \neq \emptyset)\} \\
& \cup \{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{HB}(B \rightarrow .\epsilon \in ini(q) \wedge H \neq \emptyset \\
& \qquad \wedge\, H = parse(f(q, B \rightarrow .\epsilon), i) \\
& \qquad \wedge\, (A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(B, i, H))\} \\
& \cup \{(A \rightarrow \alpha., i, G) \mid \exists_{\alpha'}(A \rightarrow \alpha. \in q \\
& \qquad \wedge\, G = parse(f(q, A \rightarrow \alpha.), i) \\
& \qquad \wedge\, (G \neq \emptyset \vee \alpha = \alpha' \perp))\}
\end{aligned}
$$

$$
\begin{aligned}
\overline{[q]}(X,i,F) \;=\; & \{(A \rightarrow \alpha.X\beta, j, G) \mid A \rightarrow \alpha.X\beta \in q \\
& \qquad \wedge\, (A \rightarrow \alpha X.\beta, j, G) \in [goto(q, X)](i, F)\} \\
& \cup \{(A \rightarrow \alpha.\beta, j, H) \mid \\
& \qquad \exists_{GCm\delta}((A \rightarrow \alpha.\beta, j, H) \in \overline{[q]}(C, m, G) \\
& \qquad \wedge\, C \rightarrow .X\delta \in ini(q) \\
& \qquad \wedge\, (C \rightarrow X.\delta, m, G) \in [goto(q, X)](i, F))\}
\end{aligned}
$$

[1] The LR(0) parser is correct as has been proven in [Leermakers, 1991a].

Nothing has really changed, information about parsable look-aheads is only passed onto function calls and index items. The information is never used.

The implication $(Y,\ell) \in F$ and $(A \rightarrow \alpha.\beta, i, F)$ an index item produced in the recognition of a sentence $x_1 \ldots x_n$, implies $Y \rightarrow^* x_{i+1} \ldots x_\ell$ has to hold. Index items are only created when the third set of $[q](i,F)$ is computed. For every $(Y,\ell) \in G$ in an index item $(A \rightarrow \alpha., i, G)$ in the third set of a function call $[q](i,F)$ the condition $Y \rightarrow^* x_{i+1} \ldots x_\ell$ holds.

We will prove that for every $(Y,\ell) \in F$ and $[q](i,F)$ a function call, $Y \rightarrow^* x_{i+1} \ldots x_\ell$. We need this proof for the next step in the construction of the parser.

For every $(Y,\ell) \in F$, Y rewrites up to x_ℓ (by definition). F is the set of parsable followers of X in a function call $\overline{[q]}(X,i,F)$ (otherwise there is no call $[q](i,F)$) and $F \neq \emptyset$. This invocation of $\overline{[q]}$ could have been made by a call $[q](i,J)$ for computing the set:

$$\begin{aligned}\{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{HB}(B \rightarrow .\epsilon \in ini(q) \wedge H \neq \emptyset \\ \wedge H = parse(f(q, B \rightarrow .\epsilon), i) \\ \wedge (A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(B, i, H))\}\end{aligned}$$

It is clear that in this case $Y \rightarrow^* x_{i+1} \ldots x_\ell$ for all $(Y,\ell) \in H$, thus for all $(Y,\ell) \in F$.

The call $\overline{[q]}(X,i,F)$ $(F \neq \emptyset)$ could also have been made by $\overline{[q]}(Z,n,H)$ for computing the set:

$$\begin{aligned}\{(A \rightarrow \alpha.\beta, j, H) \mid \exists_{GC m \delta}((A \rightarrow \alpha.\beta, j, H) \in \overline{[q]}(C, m, G) \\ \wedge C \rightarrow .X\delta \in ini(q) \\ \wedge (C \rightarrow X.\delta, m, G) \in [goto(q,Z)](n, H))\}\end{aligned}$$

In this case $F \equiv G$ and G is the set of parsable followers of C and $(C \rightarrow X.\delta, m, G)$ is an index item. By definition $Y \rightarrow x_{m+1} \ldots x_\ell$ for all $(Y,\ell) \in G$.

□

5.6 constructing the MaLR(1) parser: step 3

5.6.1 introduction

In step two the functions $[q](i)$ and $\overline{[q]}(X,i)$ have been adapted to functions $[q](i,F)$ and $\overline{[q]}(X,i,F)$ to pass on computed look-ahead information. In the third step we will adapt the function $[q](i,F)$ for the case $F \neq \emptyset$, thus for the case there is look-ahead information that can be used.

5.6.2 construction

The function $[goto(q,X)](i,F)$ returns index items of the form $(A \rightarrow \alpha X.\mu, j, G)$ with $A \rightarrow \alpha X.\mu \in goto(q,X)$. If X is the last symbol of the right-hand-side of $A \rightarrow \alpha X.\mu$ $(\mu = \epsilon)$ the set of recognized look-aheads of the final item $A \rightarrow \alpha X.$ is a subset of the set F (the set of recognized look-aheads of X). If the set F is not empty (the recognized look-aheads of X have been computed) then the function $[goto(q,X)]$ can use this information. It is not needed to compute again the set of recognized look-aheads of $A \rightarrow \alpha X.$.

If X is not the last symbol of the right-hand-side of $A \rightarrow \alpha X.\mu$ ($\mu \neq \epsilon$) then the look-ahead of X in the item $A \rightarrow \alpha.X\mu$ in state q is the first symbol of μ. If the set F of recognized look-aheads of X in state q has been computed and the first symbol of μ is element of this set then a transition on the first symbol of μ can be made immediately.

For a more formal proof let us look at the invocation $[goto(q,X)](i,F)$. This function call returns three sets ($q' = goto(q,X)$):

$$\begin{aligned}\text{set 1} \;:\; \{(A \rightarrow \alpha.\beta,j,G) \mid \exists_H((A \rightarrow \alpha.\beta,j,G) \in \overline{[q']}(x_{i+1},i+1,H) \\ \wedge\, H = parse(f(q',x_{i+1}),i+1) \\ \wedge\, H \neq \emptyset)\} \qquad (5.1)\\ \text{set 2} \;:\; \{(A \rightarrow \alpha.\beta,j,G) \mid \exists_{HB}(B \rightarrow .\epsilon \in ini(q') \wedge H \neq \emptyset \\ \wedge\, H = parse(f(q',B \rightarrow .\epsilon),i) \\ \wedge\, (A \rightarrow \alpha.\beta,j,G) \in \overline{[q']}(B,i,H))\} \qquad (5.2)\\ \text{set 3} \;:\; \{(A \rightarrow \alpha X.,i,G) \mid A \rightarrow \alpha X. \in q' \\ \wedge\, G = parse(f(q',A \rightarrow \alpha X.),i) \\ \wedge\, (G \neq \emptyset \vee X = \perp)\} \qquad (5.3)\end{aligned}$$

The third set represents the situation in which X is the last symbol of the right-hand-side of an item $A \rightarrow \alpha X.\mu \in goto(q,X)$. G is the set of parsable followers of $A \rightarrow \alpha X.$. If the parsable followers of X already have been computed, it is evident that it is not needed to compute G. The set F is the set of parsable followers of X in a call $[goto(q,X)](i,F)$, thus the third set can be replaced by:

$$\begin{aligned}\{(A \rightarrow \alpha X.,i,F') \mid A \rightarrow \alpha X. \in q' \wedge F' \neq \emptyset \\ \wedge\, F' = \{(Y,\ell) \mid Y \in f(q',A \rightarrow \alpha X.) \\ \wedge\, (Y,\ell) \in F\}\} \qquad (5.4)\end{aligned}$$

in the case $F \neq \emptyset$. This is only true if for every $(Y,\ell) \in F$, $Y \rightarrow^* x_{i+1} \ldots x_\ell$. We have proved this in section 5.5.

For computing the first and second set of a function call $[goto(q,X)](i,F)$ with use of computed look-ahead information, we will first rewrite the function $\overline{[q']}(Z,i',H)$ ($q' = goto(q,X)$). The function $\overline{[q]}(Z,i,F)$ computes the same set as $\overline{[q]}(Z,i)$ except for the look-ahead sets that are passed on. So we can use the specification

$$\begin{aligned}\overline{[q]}(Z,i) \;=\; \{(A \rightarrow \alpha.\beta,j) \mid \exists_\gamma(A \rightarrow \alpha.\beta \in q \\ \wedge\, \beta \Rightarrow^* Z\gamma \wedge \gamma \rightarrow^* x_{i+1} \ldots x_j)\}\end{aligned}$$

as a specification of the function $\overline{[q']}(Z,i',H)$. In the following equations we will use the fact that there is a follower Y of X that rewrites to Z in zero or more steps. (F is the set of parsable followers of X in $goto(q,X)$.)

$$\begin{aligned}\overline{[q']}(Z,i',H) \;&=\; \{(A \rightarrow \alpha.\beta,j,G) \mid \exists_\gamma(A \rightarrow \alpha.\beta \in q' \wedge \beta \Rightarrow^* Z\gamma \\ &\qquad \wedge\, \gamma \rightarrow^* x_{i'+1} \ldots x_j \\ &\qquad \wedge\, G = parse(f(goto(q',\beta),A \rightarrow \alpha\beta.),j))\} \\ &=\; \{(A \rightarrow \alpha.\beta,j,G) \mid \exists_{Y\mu\nu\ell}(A \rightarrow \alpha.\beta \in q' \wedge \beta = Y\mu\end{aligned}$$

$$\begin{aligned}
&\wedge Y \Rightarrow^* Z\nu \wedge \nu \rightarrow^* x_{i'+1} \ldots x_\ell \\
&\wedge \mu \rightarrow^* x_{\ell+1} \ldots x_j \\
&\wedge G = parse(f(goto(q', \beta), A \rightarrow \alpha\beta.), j))\} \\
= \; &\{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{Y\ell}((A \rightarrow \alpha.\beta, j, G) \in \overline{[q']}(Y, \ell, \emptyset) \\
&\wedge (Y, \ell) \in F \wedge Y \in f(q', A \rightarrow \alpha.\beta) \wedge Y \Rightarrow^* Z\nu \\
&\wedge \nu \rightarrow^* x_{i'+1} \ldots x_\ell)\}
\end{aligned}$$

So we have:

$$\begin{aligned}
&(5.1) \cup (5.2) = \\
&\{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{Y B \ell \nu \nu'}((A \rightarrow \alpha.\beta, j, G) \in \overline{[q']}(Y, \ell, \emptyset) \\
&\qquad \wedge (Y, \ell) \in F \wedge Y \in f(q', A \rightarrow \alpha.\beta) \\
&\qquad \wedge ((Y \Rightarrow^* B\nu \wedge B \rightarrow .\epsilon \in ini(q') \wedge \nu \rightarrow^* x_{i+1} \ldots x_\ell) \\
&\qquad\quad \vee (Y \Rightarrow^* x_{i+1}\nu' \wedge \nu' \rightarrow^* x_{i+2} \ldots x_\ell)))\}
\end{aligned}$$

If Y does not (left) rewrite to $B\nu$ with $B \rightarrow .\epsilon \in ini(q')$ or to $x_{i+1}\nu'$ then $Y \Rightarrow^* C\nu''$ with $C \rightarrow .\epsilon \notin ini(q')$ and $C \in V_N$. In that case $C \Rightarrow^* C'\nu$ with $C' \rightarrow .\epsilon \in ini(q')$ and $\nu \rightarrow^* x_{i+1} \ldots x_\ell$ or $C \Rightarrow^* x_{i+1}\nu'$ with $\nu' \rightarrow^* x_{i+2} \ldots x_\ell$ because $Y \rightarrow^* x_{i+1} \ldots x_\ell$. Thus the condition

$$\begin{aligned}
&((Y \Rightarrow^* B\nu \wedge B \rightarrow .\epsilon \in ini(q') \wedge \nu \rightarrow^* x_{i+1} \ldots x_\ell) \\
&\vee (Y \Rightarrow^* x_{i+1}\nu' \wedge \nu' \rightarrow^* x_{i+2} \ldots x_\ell))
\end{aligned}$$

always holds. So:

$$\begin{aligned}
&(5.1) \cup (5.2) = \\
&\{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{Y\ell}((A \rightarrow \alpha.\beta, j, G) \in \overline{[q']}(Y, \ell, \emptyset) \\
&\qquad \wedge (Y, \ell) \in F \wedge Y \in f(q', A \rightarrow \alpha.\beta))\} \qquad (5.5)
\end{aligned}$$

For a call $[q](i, F)$ there are two cases, the set F is empty or it is not empty. In the case $F = \emptyset$ there is no look-ahead information that can be used. In the case $F \neq \emptyset$ the look-aheads of the transition symbol have been computed and we can replace the old function $[q](i, F)$ by a function that computed the set (5.4) $\cup$ (5.5). So we get:

$$\begin{aligned}
[q](i, \emptyset) \; = \; &\{(A \rightarrow \alpha.\beta, j, G) \mid \exists_H((A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(x_{i+1}, i+1, H) \\
&\qquad \wedge H = parse(f(q, x_{i+1}), i+1) \\
&\qquad \wedge H \neq \emptyset)\} \\
&\cup \{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{HB}(B \rightarrow .\epsilon \in ini(q) \wedge H \neq \emptyset \\
&\qquad\quad \wedge H = parse(f(q, B \rightarrow .\epsilon), i) \\
&\qquad\quad \wedge (A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(B, i, H))\} \\
&\cup \{(A \rightarrow \alpha., i, G) \mid \exists_{\alpha'}(A \rightarrow \alpha. \in q \\
&\qquad \wedge G = parse(f(q, A \rightarrow \alpha.), i) \\
&\qquad \wedge (G \neq \emptyset \vee \alpha = \alpha' \perp))\}
\end{aligned}$$

$$
\begin{aligned}
[q](i,F) \;=\; & \{(A \to \alpha.,i,F') \mid A \to \alpha. \in q \wedge F' \neq \emptyset \\
& \quad \wedge F' = \{(Y,\ell) \mid Y \in f(A \to \alpha.) \wedge (Y,\ell) \in F\}\} \\
& \cup \{(A \to \alpha.\beta,j,G) \mid \exists_{Y\ell}((A \to \alpha.\beta,j,G) \in \overline{[q]}(Y,\ell,\emptyset) \\
& \quad \wedge (Y,\ell) \in F \wedge Y \in f(q,A \to \alpha.\beta))\}
\end{aligned}
$$

5.7 review

$$\alpha \Rightarrow \beta \;\equiv\; \exists_{A\delta\gamma}(\alpha = A\gamma \wedge \beta = \delta\gamma \wedge A \to \delta \wedge \delta \neq \epsilon)$$

$$ini \;:\; S \to \mathcal{P}(I)$$

$$ini(q) \;=\; \{B \to .\nu \mid \exists_{A\alpha\beta\delta}(A \to \alpha.\beta \in q \wedge \beta \Rightarrow^* B\delta \wedge B \to \nu)\}$$

$$goto \;:\; S \times V^* \to S$$

$$
\begin{aligned}
goto(q,\epsilon) &= q \\
goto(q,X\delta) &= goto(goto(q,X),\delta) \\
goto(q,X) &= \{A \to \alpha X.\beta \mid A \to \alpha.X\beta \in (q \cup ini(q))\}
\end{aligned}
$$

$$f \;:\; S \times (I \cup V_T) \to \mathcal{P}(V)$$

$$
\begin{aligned}
f(q,A \to \alpha.\beta) \;=\; & \{X \mid \exists_{\mu}(\beta = X\mu \wedge X \in V)\} \\
& \cup \{Y \mid \exists_{p\gamma\delta BY}(\beta = \epsilon \wedge q = goto(p,\alpha) \\
& \quad \wedge B \to \gamma.A\delta \in (p \cup ini(p)) \\
& \quad \wedge Y \in f(goto(p,A),B \to \gamma A.\delta))\} \\
f(q,x) \;=\; & \{X \mid \exists_{\alpha\beta A}(A \to \alpha.x\beta \in q \wedge X \in f(q,A \to \alpha x.\beta) \wedge x \in V_T)\}
\end{aligned}
$$

$$[q] \;:\; N \times \mathcal{P}(\mathcal{F}) \to \mathcal{P}(II)$$

$$
\begin{aligned}
[q](i,\emptyset) \;=\; & \{(A \to \alpha.\beta,j,G) \mid \exists_H((A \to \alpha.\beta,j,G) \in \overline{[q]}(x_{i+1},i+1,H) \\
& \quad \wedge H = parse(f(q,x_{i+1}),i+1) \\
& \quad \wedge H \neq \emptyset)\} \\
& \cup \{(A \to \alpha.\beta,j,G) \mid \exists_{HB}(B \to .\epsilon \in ini(q) \wedge H \neq \emptyset \\
& \quad \wedge H = parse(f(q,B \to .\epsilon),i)
\end{aligned}
$$

$$\wedge\ (A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(B, i, H))\}$$
$$\cup\ \{(A \rightarrow \alpha., i, G) \mid \exists_{\alpha'}(A \rightarrow \alpha. \in q$$
$$\wedge\ G = parse(f(q, A \rightarrow \alpha.), i)$$
$$\wedge\ (G \neq \emptyset \vee \alpha = \alpha'\ \bot))\}$$

$$[q](i,F) = \{(A \rightarrow \alpha., i, F') \mid A \rightarrow \alpha. \in q \wedge F' \neq \emptyset$$
$$\wedge\ F' = \{(Y,\ell) \mid Y \in f(A \rightarrow \alpha.) \wedge (Y,\ell) \in F\}\}$$
$$\cup\ \{(A \rightarrow \alpha.\beta, j, G) \mid \exists_{Y\ell}((A \rightarrow \alpha.\beta, j, G) \in \overline{[q]}(Y,\ell,\emptyset)$$
$$\wedge\ (Y,\ell) \in F \wedge Y \in f(q, A \rightarrow \alpha.\beta))\}$$

$$\overline{[q]} : V \times N \times \mathcal{P}(\mathcal{F}) \rightarrow \mathcal{P}(II)$$

$$\overline{[q]}(X,i,F) = \{(A \rightarrow \alpha.X\beta, j, G) \mid A \rightarrow \alpha.X\beta \in q$$
$$\wedge\ (A \rightarrow \alpha X.\beta, j, G) \in [goto(q,X)](i,F)\}$$
$$\cup\ \{(A \rightarrow \alpha.\beta, j, H) \mid$$
$$\exists_{CGm\delta}((A \rightarrow \alpha.\beta, j, H) \in \overline{[q]}(C, m, G)$$
$$\wedge\ C \rightarrow .X\delta \in ini(q)$$
$$\wedge\ (C \rightarrow X.\delta, m, G) \in [goto(q,X)](i,F))\}$$

5.8 complexity

Look-aheads must be parsed by a function *parse*. If the function *parse* is not a MaLR(1) parser, the parser will not always use look-ahead information. The results of computing the parses of one look-ahead have to be memorized, otherwise a lot of subparses have to be computed several times. So for the function *parse* the same parser has to be used as the parser that calls the function *parse* and the parse results (in fact the function calls $[q](i,F)$ and $\overline{[q]}(X,i,F)$) have to be memorized. For computing parses of look-aheads, the parser must be extended with states for every nonterminal that can act as look-ahead (in practice all nonterminals may act as look-ahead). Such a state is the root state used for recognizing/parsing the corresponding look-ahead symbol. Thus for every nonterminal $Y \in V_N$ a state $\{LookAhead \rightarrow .Y\ \bot\}$ has to be added to the parser. The symbol $\bot$ is used to indicate that no symbols have to be recognized after the look-ahead symbol. The parser does not stop with recognizing till the $\bot$ symbol has been reached. The symbol *LookAhead* has no meaning. It is used to create an initial item.

The number of states of a MaLR(1) parser equals the number of states of a LR(0) parser ($O(2^{|G|})$). But every item is connected with a set of look-ahead symbols. At most $|\ V\ |$ different look-aheads are connected with an item. If an item has symbols on the right-hand-side of the dot then the item has only one look-ahead.

There are $O(n \mid V \mid)$ different invocations of a function $[q]$ and $O(\mid V \mid n \mid V \mid)$ different invocations of a function $\overline{[q]}$. Each invocation of a function $\overline{[q]}(X,i,F)$ leads to an invocation of $[goto(q,X)](i,F)$. A call $[q](i,F)$ returns a set of $O(n)$ elements. An invocation $\overline{[q]}(X,i,F)$ leads to an invocation of a function $\overline{[q]}(X,i,F)$ for every element of

the set $[goto(q, X)](i, F)$, thus $O(n)$ invocations of functions $\overline{[q]}$. A call $\overline{[q]}(X, i, F)$ returns a set of $O(n)$ elements. So $O(n)$ invocations of functions $\overline{[q]}$ results in $O(n)$ sets of $O(n)$ elements. Together with merging these sets and removing duplicates each invocation of a function $\overline{[q]}$ takes $O(n^2)$ time. Hence the total time complexity of the MaLR(1) parser is $O(2^{|G|} \mid V \mid^2 n^3)$.

5.9 reduction of look-ahead power

The Marcus LR parsers have less look-ahead power than the recursive ascent Marcus parsers. Look-aheads are no longer part of the items. In figure 5.1 and figure 5.2 one can see that this disconnection of look-aheads and items causes a loss of information. As

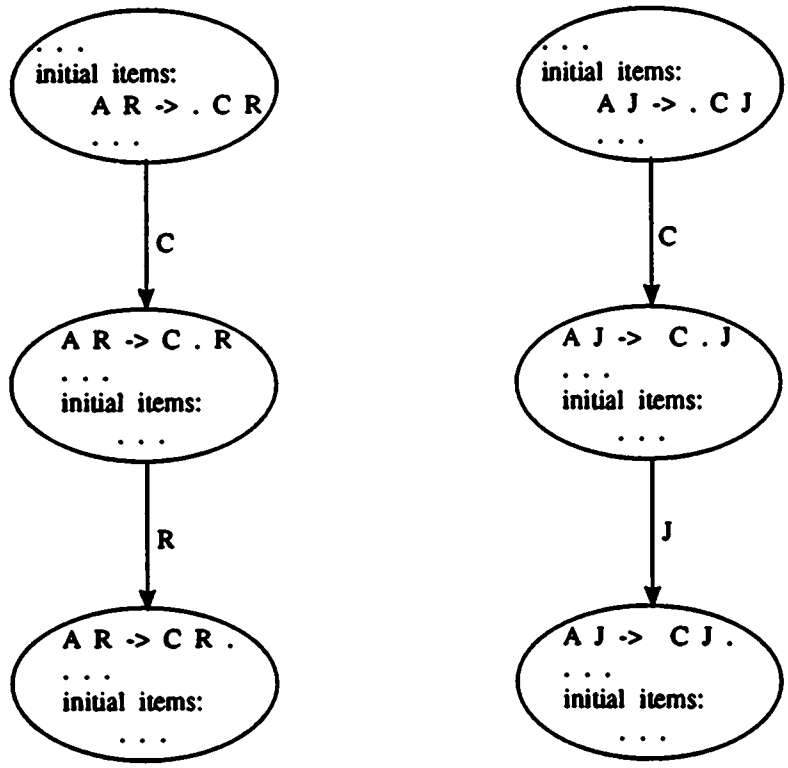


Figure 5.1: use of look-ahead in RAM parsers

we will see in chapter 11, the difference between the RAM parsers and the MaLR parsers has great influence on probabilistic parsing.

5.10 MaLR(k) parser

The number of look-aheads does not play a part in the construction of the MaLR(1) parser. While constructing the parser we assumed one constituent look-ahead. In the following extension of the parser we assume k look-aheads. Only the function f which computes the followers and the functions $[q]$ change. A function is needed to compare a list of followers and a list of parsed followers. This function is named $SymEq$.

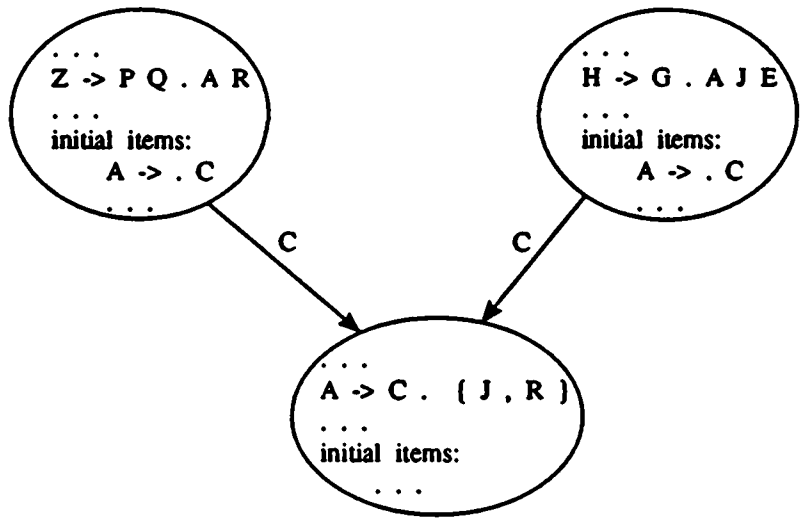


Figure 5.2: use of look-ahead in MaLR parsers

$$f \;:\; S \times (I \cup V_T) \times N \rightarrow \mathcal{P}(V)$$

$$\begin{aligned}
f(q, A \rightarrow \alpha.\beta, k) &= \emptyset, \text{ if } k = 0 \vee \beta = \perp \\
&= \{X : tail \mid \exists_{\mu}(\beta = X\mu \wedge X \in V \\
&\qquad \wedge\, tail \in f(q, A \rightarrow \alpha X.\mu, k-1))\} \\
&\quad \cup \{flist \mid \exists_{p\gamma\delta BY}(\beta = \epsilon \wedge q = goto(p,\alpha) \\
&\qquad \wedge\, B \rightarrow \gamma.A\delta \in (p \cup ini(p)) \\
&\qquad \wedge\, flist \in f(goto(p,A), B \rightarrow \gamma A.\delta, k))\} \\
f(q,x,k) &= \{X \mid \exists_{\alpha\beta A}(A \rightarrow \alpha.x\beta \in q \\
&\qquad \wedge\, X \in f(q, A \rightarrow \alpha x.\beta, k) \wedge x \in V_T)\}
\end{aligned}$$

$$SymEq \;:\; V^* \times \mathcal{F}^* \rightarrow Bool$$

$$\begin{aligned}
SymEq([], plist) &= True, \text{ if plist } = [] \\
&= False, \text{ if plist } \neq [] \\
SymEq(Y : ftail, []) &= False \\
SymEq(Y : ftail, (X, \ell) : ptail) &= SymEq(ftail, ptail), \text{ if } X = Y \\
&= False, \text{ otherwise}
\end{aligned}$$

$$[q] \;:\; N \times \mathcal{P}(\mathcal{F}) \rightarrow \mathcal{P}(II)$$

$$
\begin{aligned}
[q](i,\emptyset) \;=\; & \{(A \to \alpha.\beta, j) \mid \exists_H((A \to \alpha.\beta, j) \in \overline{[q]}(x_{i+1}, i+1, H) \\
& \qquad \wedge H = parse(f(q, x_{i+1}, k), i+1) \\
& \qquad \wedge H \neq \emptyset)\} \\
& \cup \{(A \to \alpha.\beta, j, G) \mid \exists_{BH}(B \to .\epsilon \in ini(q) \wedge H \neq \emptyset \\
& \qquad \wedge H = parse(f(q, B \to .\epsilon, k), i) \\
& \qquad \wedge (A \to \alpha.\beta, j, G) \in \overline{[q]}(B, i, H))\} \\
& \cup \{(A \to \alpha., i, G) \mid \exists_{\alpha'}(A \to \alpha. \in q \\
& \qquad \wedge G = parse(f(q, A \to \alpha., k), i) \\
& \qquad \wedge (G \neq \emptyset \vee \alpha = \alpha' \perp))\}
\end{aligned}
$$

$$
\begin{aligned}
[q](i,F) \;=\; & \{(A \to \alpha., i, F') \mid A \to \alpha. \in q \wedge F' \neq \emptyset \\
& \qquad \wedge F' = \{(Y,\ell) : ptail \mid Y : ftail \in f(q, A \to \alpha., k) \\
& \qquad\qquad \wedge (Y,\ell) : ptail \in F \\
& \qquad\qquad \wedge SymEq(ftail, ptail)\}\} \\
& \cup \{(A \to \alpha.\beta, j, G) \mid \exists_{Y\ell}((A \to \alpha.\beta, j, G) \in \overline{[q]}(Y, \ell, ptail) \\
& \qquad \wedge (Y,\ell) : ptail \in F \\
& \qquad \wedge Y : ftail \in f(q, A \to \alpha.\beta, k) \\
& \qquad \wedge SymEq(ftail, ptail))\}
\end{aligned}
$$

Part II

Grammars

Chapter 6

large grammars of the English language

6.1 introduction

For parsing a large subset of a natural language, we need a grammar which describes such a large subset. Three grammars are described in the following chapters (chapter 7 and 8). These three grammars were available during our research. In this chapter, we describe some other large grammars of the English language. It is almost impossible to compare these grammars. Grammars are written in different forms, with different lexical and syntactical categories, with or without attributes and so on. An approach to compare different grammars/parsers is given in [Grishman et al., 1992].

6.2 augmented phrase structure grammars

DIAGRAM ([Robinson, 1982]) is a system used for interpreting dialogues. The used grammar consists of context-free rules augmented by procedures that constrain the application of a rule, add information to the structure created by the rule and assign one or more interpretations to the resulting enriched structural analyses. Attributes are used in the grammar of DIAGRAM to set context-sensitive constraints on the acceptance of analyses. Nothing is said about the computational aspects of the parser and about the size of the subset of English that is described by the grammar. An important problem of large grammars is mentioned: *"introducing new rules almost inevitably has a perturbing effect as they interact with the old rules in unforeseen ways"*.

Another augmented phrase structure grammar is the grammar used in the CRITIQUE system, developed by IBM ([Richardson et al., 1988]). CRITIQUE is an extension of the EPISTLE project ([Heidorn et al., 1982]). It is a system used to identify grammatical and style errors in English text. The grammar produces parses which are approximate. The system is (beside other tests) tested on 2254 sentences from 411 business letters. The average word number of a sentence is 19. For 64 percent of the sentences, a parse was produced. For 41 percent of the 2254 sentences, one parse tree was produced, for 11 percent two parse trees were produced and for 11 percent three to nine parse trees were produced. For one percent of the 2254 sentences, ten or more parse trees were produced. This is a very small number of ambiguities. A reason may be that the grammar

describes very general structures. The grammar only has to describe structures needed for the critiquing tasks of the system. The CRITIQUE system works relatively fast: for a sentence of 15-20 words, a large IBM-mainframe needs one CPU second to analyze the sentence.

As will be made clear in chapter 9, the augmented phrase structure grammars used in DIAGRAM and CRITIQUE cannot be used for our probabilistic parser.

6.3 other approaches

In [Sager, 1981] a linguistic string grammar is described. The grammar consists of BNF rules together with a kind of attributes. These attributes are used to restrict the number of syntactical analyses. The BNF component consists of three types of rules (string definitions, adjunct set definitions and LXR definitions). A string definition is a rewriting of a syntactical class in a sequence of syntactical classes and adjunct set definitions. Adjunct set definitions are optional additions to sentences. An adjunct set definition may rewrite to LXR definitions. A LXR definition only exists if X is a syntactical category. A LXR definition is of the form

$$\text{LXR} \rightarrow \text{LX X RX}$$

with LX the set of possible left adjuncts of X and RX the set of possible right adjuncts of X. The complexity of the grammar and the size of the subset of English the grammar can generate are not described.

At the university of Nijmegen (the Netherlands), corpus linguists work with a large grammar for English. It is a detailed, attributed grammar. It is not usable for our probabilistic parser. For correctly tagged sentences (words are tagged with their lexical category), still a lot of analyses are produced (in contrast with the CRITIQUE system). More than 100 analyses for one sentences is not an exception. About 80 percent of sentences from written prose can be generated by the grammar. Parsing is really difficult for long sentences. Parsing of a sentence is stopped after one hour (on a SUN3 workstation). Other sentence types (for example from newspapers) are even more difficult to handle.

6.4 conclusions

Only few grammars exists that describe a large subset of the English language. Not all of them have been mentioned in this chapter. Almost all of the existing large grammars use attributes/features to restrict the number of possible analyses. Other grammar formalisms exist like generalized phrase structure grammars ([Gazdar et al., 1985]) and unification grammars ([Shieber, 1986]). In chapter 9 we show why the context-free approach is a more appropriate formalism to use for probabilistic parsing.

Chapter 7

the corpus grammar

7.1 description

Together with the LDB-corpus a grammar exists that describes many of the sentence constructions that occur in the corpus. The grammar is context-free and attribute-free. It describes rewritings of syntactical categories and grammatical relations[1]. So parse trees are represented by rules like:

utterance:sf → subject:pn elliptic:ell

with *sf*[2], *pn*[3], *ell*[4] syntactical categories and *utterance*, *subject*, *elliptic* grammatical relations.

The grammar has rules like[5]:

- f ell elliptic structure → c ell elliptic structure.
- c vitg[6] → nf vi ing code;

  nf vi ing cv;

  nf vi ing coord.
- f ic immediate constituent → nc constituent in ns utterance.
- nc constituent → nc basic;

  nc phrasal.

A string 'f α' represents a grammatical relation (a function). A string 'c α' represents a syntactical category. A string 'nc α' represents an auxiliary syntactical category (aux-category). Such a category only rewrites to other aux-categories and/or to syntactical categories. A string 'nf α' represents an auxiliary grammatical relation (aux-relation), that only rewrites to other aux-relations and/or to grammatical relations.

[1] Grammatical relations are used to distinguish different uses of a syntactical category. For example a Noun Phrase may act as an Indirect Object, as an Object and as a Predicate.

[2] sf = finite sentence

[3] pn = pronoun

[4] ell = elliptic phrase

[5] A semicolon represents OR, a comma represents concatenation.

[6] vitg = verb prespart intransitive

7.2 elimination of auxiliary categories and relations

To get a grammar that corresponds with the trees of the sentences of the corpus we have to eliminate the *nc* and *nf* rewritings. We are not interested in the grammatical relations, so we also want to eliminate the *f* rewritings. Here the first problem occurs: there are cyclic rewritings of *nf* symbols, for example:

- nf maybe inf vp → nf filling, nf inf verb phrase.
- nf inf verb phrase → nf to option maybe ad, nf inf verb phrase tail;
  nf to.
- nf inf verb phrase tail → f vb vap inf, nf maybe ed or ing vp;
  f vb vas inf, nf maybe inf vp;
  nf inf no aux vp.

This makes eliminating *nf* rewritings very difficult. We have erased the cyclic structures (so the grammar describes fewer constructions as before).

The second problem is the explosion of the number of rules if the *nf*, *nc* and *f* rewritings are eliminated (when only the *nf* and *nc* rewritings are eliminated there is also an explosion). There is an explosion because a lot of rules are of the same form as (for example):

- c noca not a sentence → nf left, f ic immediate constituent,
  nf constituent string, nf right.
- nf constituent string → ;
  f ic immediate constituent.
- f ic immediate constituent → nc constituent in ns utterance.
- nc constituent in ns utterance → c coop coordinator phrase;
  c subp subordinator phrase;
  ⋮
  c in interjection phrase;

The explosion is a real explosion. The original grammar (with *nf*, *nc* and *f* rewritings) consists of more than 1400 rules. Eliminating *nf*, *nc* and *f* produces more than 800 rewritings for the category *c noca not a sentence*. For other categories (as *c sf fin sentence*, *c np noun phrase*) there are much, much more rewritings.

In the original grammar the *nf* and *nc* symbols can be seen as extra grammatical relations and syntactical categories. Adding a category to a grammar increases the number of grammar rules if the new category is a special case of an existing category. It could decrease the number of grammar rules if the new category describes a set of existing categories (as with the categories *nc* and relations *nf*).

7.3 over-generation

We have tried to compute the set of states of the parser without look-aheads for the corpus grammar. It turned out that a considerable number of states have more than 1000 initial items. So computing all the states would take a lot of time and space. For our system it was impossible to compute all the states within a week. The grammar highly over-generates.

According to the original users of the corpus grammar (the people who developed the LDB-corpus), the grammar describes in principle common English. For the LDB-developers it was not a problem that the grammar highly over-generates because they always used lexical categories with tree structure as input for the parser of the corpus grammar. They used the grammar to label the tree structures.

7.4 conclusions

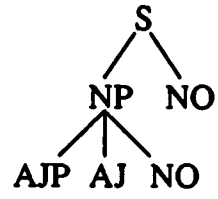
The corpus grammar cannot be used for an experiment with probabilistic parsing. The grammar highly over-generates and it uses a lot of auxiliary categories. Auxiliary categories are of great use for reducing the size of a grammar.

Chapter 8

a grammar extracted from the corpus

8.1 introduction

From two parts of the LDB-corpus we have extracted a grammar. Both the parts are two short stories. There are 3566 analyzed sentences in the two stories. For every sentence there is a unique parse tree. The grammar rules have been extracted from these parse trees. Every node with sons in a tree represents a grammar rule. For example the tree



represents grammar rules:

- S → NP NO
- NP → AJP AJ NO

The resulting grammar consists of more than 4200 rules. The grammar is a flat grammar[1]. The category verb phrase (VP) has not been used in the parse trees. All the possible rewritings of a VP are written out in other rules. An example is:

Finite_Sentence → Interjection Pronoun Verb_finite_modal
Verb_infinite_primary Verb_past_part_monotransitive
Noun_phrase Exclamation_mark

where Verb_infinite_primary Verb_past_part_monotransitive Noun_phrase can be seen as a verb phrase.

In section 8.4 the influence of the absence of the category VP on the number of grammar rules will be examined.

[1] Parse trees written with a flat grammar relatively do not have deeply nested rewritings.

8.2 multiple replication of symbols

8.2.1 existence

The grammar does not contain empty rewritings (ϵ-rules). In the parse trees ϵ-rules are not used, so every category always rewrites to other categories or the category is a terminal symbol. Because of the used method for extracting grammar rules regular expressions with multiple replication of a symbol have been written out, for instance:

- NP → DE *AJ AJ* NOUN PP

  a *pierced decorative* border (in brickwork)

- NP → DE *AJ AJ AJ* NOUN PP

  the *only ordinary normal* person (in the place)

The words between () make up one constituent.
Multiple replication of more than one symbol like

- NP → NP COOR NP COOR NP

  (a great big home) and (a dozen farms in ..) and (a great deal of money ..)

also exists.

8.2.2 representation in grammar rules

Another construction without ϵ-rules for rules as:

- NP → DE *AJ AJ* NOUN PP
- NP → DE *AJ AJ AJ* NOUN PP

is:

- NP → DE NOUN PP
- NP → DE AJ_P NOUN PP
- AJ_P → AJ AJ_P
- AJ_P → *AJ*

With ϵ-rules a shorter notation is possible:

- NP → DE AJ_P NOUN PP
- AJ_P → AJ AJ_P
- AJ_P → ϵ

In general, this construction is better than the written-out notation because in the written-out case the number of AJ replications is bounded. It is also better because in the written-out case the parser has to know the current situation for each applicable rule. So for a noun phrase starting with DE AJ the used rule could be (for example):

- NP → DE AJ NOUN PP
- NP → DE AJ AJ NOUN PP
- NP → DE AJ AJ AJ NOUN PP

But there are also rules like:

- NP → DE AJ NOUN NON_FINITE_SENTENCE
- NP → DE AJ AJ NOUN NON_FINITE_SENTENCE
- NP → DE AJ NOUN PP PP
- NP → DE AJ AJ NOUN PP PP

etcetera

The effect of these written-out rules is a high number of states as the following example shows.

We have the following grammar rules:

- NP → DE NOUN
- NP → DE AJ NOUN
- NP → DE AJ AJ NOUN

A state $\{\gamma \rightarrow \alpha \ .\ \text{NP}\ \beta\}$ has a transition on symbol DE (parsing with k is one). See figure 8.1 for the resulting state with related states.
With grammar rules:

- NP → DE AJ_P NOUN
- AJ_P → AJ AJ_P
- AJ_P → ϵ

a transition on symbol DE leads to a state with item:

- NP → DE . AJ_P NOUN

For this state there are transitions on symbol NOUN (because AJ_P may rewrite to ϵ), on symbol AJ and on symbol AJ_P. The transition on symbol AJ leads to one or more states that describe the recognition of the regular expression AJ_P ::= { AJ }+ [2]. This time fewer states are needed for the recognizing of an AJ-repetition string because the already recognized part of the AJ string is not showed/memorized in the items of the states but in function calls. From the next symbols the parser only knows that AJ_P rewrites to ϵ or to AJ followed by AJ_P.

A disadvantage of the short notation is the change of structure in a sequence of the same symbol (see figure 8.2).

[2] {x}+ means one or more replications of symbol x.

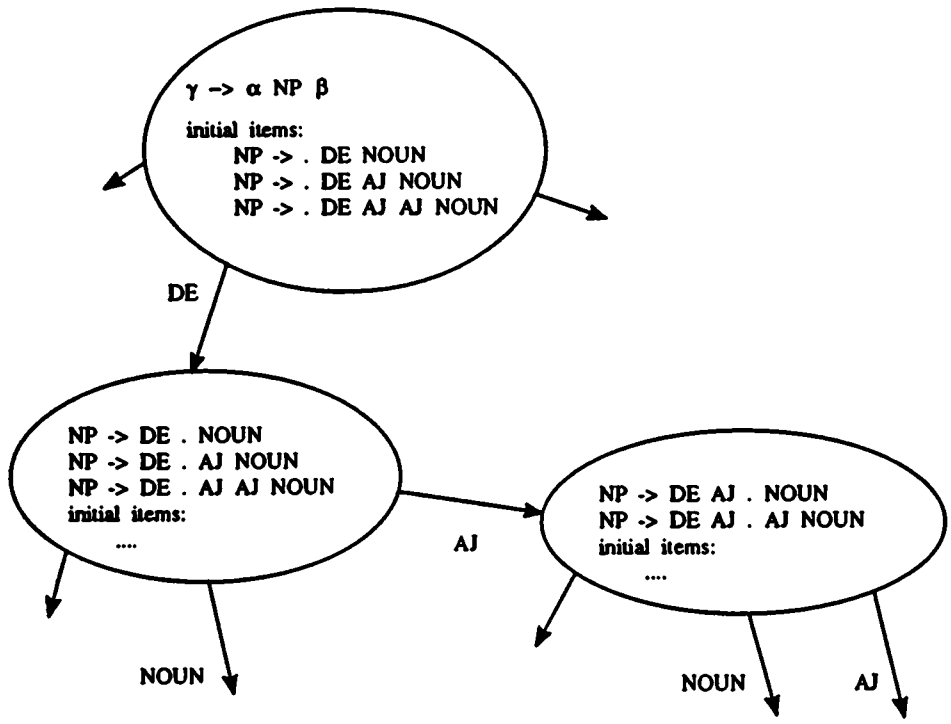


Figure 8.1: states for a grammar without ϵ-rules

8.2.3 effect on the grammar

The number of rules of the grammar could be reduced if multiple replication of a symbol/symbolstring would not be written out. What is the effect on the number of rules of the grammar if we use a shorter notation for representing a multiple replication? How many constructions contain a multiple replication? Or in other words: how bad is the corpus-grammar with respect to multiple replication?

If all multiple replications of one symbol in a right-hand-side of a grammar rule are replaced by a (new) symbol that represents a multiple replication for that symbol and duplicate rules are erased, the number of rules reduces from 4250 to 4025. This means that writing out the multiple replication of one symbol does not create many extra rules.

In the corpus multiple replication of more than one symbol only exists for symbolstrings of length two. Almost all the rules with multiple replication of two symbols are of the form:

Symbol $\rightarrow \alpha$ { SYM COOR } β

or

Symbol $\rightarrow \alpha$ { COOR SYM } β

An example of the first form is:

- NO $\rightarrow$ NO COOR NO COOR SF

  (rock crystal) and shells and (what used to be called fossil bodies)

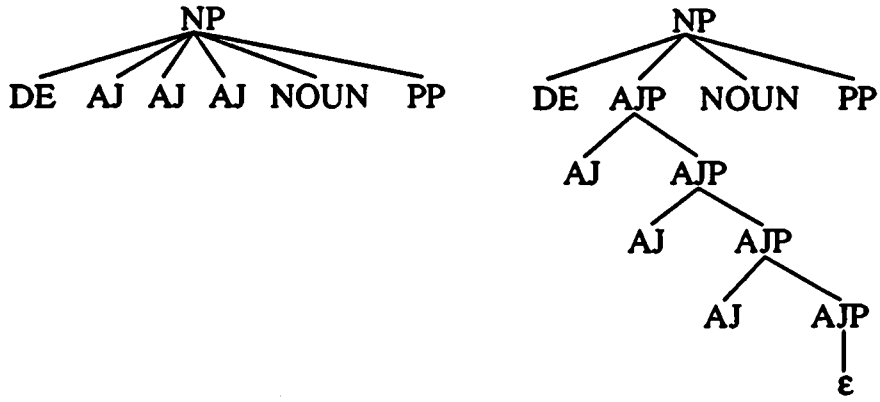


Figure 8.2: change of structure in parse trees

For an example of the second form see section 8.2.1. Thus coordination creates multiple replication of two symbols. Some other rules (not frequently used) are:

- VC[3] → SUB[4] AV PP AV PP

  if not (with authority) (at least) (with an exuberance of ..)

- SF → NO ASP[5] VITF AV PP AV PP

  Straker (as a '.. phenomena') comes home (to us) now (with a ..)

In the corpus constructions with multiple replication of two symbols has been found 89 times partly describing the same rules. Thus multiple replication of two symbols will be of little influence on the number of grammar rules.

8.3 optional symbols

Not only for multiple replication of a symbol the parser could use ϵ-rules. In the grammar extracted from the corpus the following rules occur:

- NP → DE AJ AJ NO

  a slim youngish person

- NP → DE AJ AJ NO PP

  a pierced decorative border (in brickwork)

- NP → DE AJ AJ NO SN

  some shy elderly person (overtaken ..)

- NP → DE AJ AJ NO VC PP

[3]VC = verbless clause
[4]SUB = subordinator
[5]ASP = as phrase

a tall swarthy man (, very much the classic cockney,) (with a ..)

These rules describe the same phenomenon. In a regular expression this can be written as ([x] means zero or one replication of symbol x):

- NP → DE {AJ}+ NO [VC] [PP] [SN]

With use of ϵ-rules this can be written as:

- NP → DE one_or_more_AJ NO opt_VC opt_PP opt_SN
- one_or_more_AJ → AJ one_or_more_AJ
- one_or_more_AJ → AJ
- opt_SYMBOL → SYMBOL
- opt_SYMBOL → ϵ

And without ϵ-rules this can be written as 8 (2^3) rewritings of NP:

- NP → DE one_or_more_AJ NO VC PP SN
- NP → DE one_or_more_AJ NO PP SN
  ⋮
- NP → DE one_or_more_AJ NO

As we can see the number of grammar rules explodes if all the rules would be written out.

For multiple replication of symbol(s) ϵ-rules are not needed. But without ϵ-rules optional symbols have to be written out. So a grammar for natural language without ϵ-rules uses (much?) more grammar rules to describe a language than a grammar with ϵ-rules.

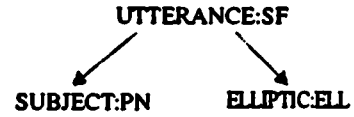
8.4 other reductions

It is not possible to write a grammar that describes precisely all possible sentence structures of a natural language. For context-free grammars it is possible to use attributes to reduce the number of rules. Without attributes the grammar needs more syntactical categories to distinguish different sentence structures. Basically it is possible to write a grammar of one rule with a lot of attributes. By contrast it is possible to write a very, very large grammar without attributes with the same power as the one rule grammar with a lot of attributes.

For a given context-free grammar without attributes, it is difficult to say whether it is a good or a bad grammar. We are searching for a sort of minimal grammar that is over-generating but acceptable with a minimal number of rules. Whether the grammar is acceptable or not is an application dependent question. We are searching for a minimal grammar that describes common English. In practice this grammar will over-generate too much, but when trying to compute the minimal number of rules of a grammar such an

over-generating-grammar is a good goal. A less over-generating-grammar always needs more rules.

The grammar extracted from the corpus is small because it describes only syntactical constructions of 3566 sentences (see section 8.5.3). Nearly 100 syntactical categories are used. In the parse trees of the corpus syntactical categories are used as well as grammatical relations. While extracting the grammar the grammatical relations have been ignored/skipped. Thus for a subtree



the extracted rule is SF → PN ELL. Skipping the grammatical relation creates more over-generating and it reduces the number of grammar rules, because sometimes words belong to the same category but to a different grammatical relation .

Looking for the lower limit for the number of rules of a context-free grammar without attributes and ϵ-rules for common English, other reductions on the number of rules are possible. In the corpus for a number of phrases there is a difference between the phrase consisting of one element and the phrase consisting of more than one element (for example AJ (adjective, one element) and AJP (adjective phrase, more than one element)). Although there is a linguistic motivation for this phenomenon, we replace symbols X (X $\in V_N$) by symbols XP (XP $\in V_N$, XP is a X phrase and X represents a single element XP). Because almost always a single element phrase behaves the same as a multiple element phrase this is a natural reduction that does not create much over-generating. The number of rules will be reduced because of rules like:

- AN → AN COOR AJ
- AN → AN COOR AJP

Because in the corpus spoken text exists, rules for phrases between quotes occur in the grammar. These rules are not general but rather ad hoc as can be seen in following rules (OPEN is an opening quote, CLOS is a closing quote):

- SF → OPEN IN IN PN VCOF AV SF CLOS
- SF → OPEN PN VITF AV CLOS
- SF → OPEN PN VCOF AV AJ
- SF → NP VCOF PP CLOS

With fewer rules the same can be described:

- SF_QUOTE → OPEN_option SF CLOS
- SF_QUOTE → OPEN SF
- NP_QUOTE → OPEN NP CLOS
- ...

It is difficult to see what the effect of the absence of the category VP is on the number of rules of the grammar. For each rule of the form

$$A \rightarrow \alpha \text{ VP } \beta$$

one has to know how many rules of the same form exists in the corpus grammar with a different written out VP.

For a random subset of grammar rules with a verb phrase we have examined this. For each rule a linguist has determined the verb phrase in the right-hand-side. Then the rules with the same left-hand-side and the same right-hand-side except the verb phrase part have been counted. So for rules of the form $A \rightarrow \alpha$ VP β we counted the rules $A \rightarrow \alpha\ \gamma\ \beta$ ($\gamma \in V^*$). It will be clear that γ is not always a verb phrase, so there are fewer rules $A \rightarrow \alpha$ VP β than rules $A \rightarrow \alpha\ \gamma\ \beta$. For some rules there is only one sample of the form $A \rightarrow \alpha\ \gamma\ \beta$. For other rules there are dozens of rules of the form $A \rightarrow \alpha\ \gamma\ \beta$. The experiment indicates that the number of grammar rules reduces with a factor between two and ten if the category VP is added to the grammar.

8.5 number of grammar rules

8.5.1 grammars without ϵ-rules

Grammars without ϵ-rules would not be a problem if they consisted of only a restricted number of rules. Tomita describes a parser for natural language. For testing his parser, the biggest grammar he uses consists of only 400 context-free rules without ϵ-rules and attributes. In his view this grammar is considered as one of the toughest natural language grammars in practice[6]. In this section it will be explained that this is not a realistic view.

As described before, the pure grammar extracted from the corpus consists of more than 4200 rules. Representing multiple replication without ϵ-rules but with a compact notation as described in section 8.2 reduces the number of rules to about 4000. Replacing single element phrases by multiple element phrases as described in section 8.4 reduces the number of grammar rules subsequently to about 3100. Erasing the quote symbols in the rules (see section 8.4) delivers a grammar with about 2700 rules.

Fewer rules are only possible if the grammar is made less flat, if the grammar is made more over-generating or if the grammar describes a smaller subset of English. The grammar already describes a very small subset of English (this aspect will be analyzed in section 8.5.3). As described before, the grammar already highly over-generates. More over-generating with fewer grammar rules involves less sentence structure in parse trees.

With introducing the category VP the grammar will be less flat and the number of rules can be reduced. Parse trees with a normal depth can be generated with the corpus grammar including category VP. A less flat grammar (than a grammar including category VP) is not natural.

In about 500 rules of the 2700 grammar rules there is no nonterminal representing a verb in the right-hand-side. So our expectation is that with category VP the grammar still consists of more than 1000 rules.

[6][Tomita, 1986], page 81

8.5.2 grammars with ϵ-rules

With use of ϵ-rules it is possible to describe a large subset of a natural language with fewer than 1000 rules.

The surface grammar used in the ROSETTA system (a context free grammar for common English) consists of about 20 regular expressions. Replacing the regular expressions for grammar rules in BNF notation with ϵ-rules returns a grammar with about 300 rules. This grammar highly over-generates because the original surface grammar is an attribute grammar and we have erased all the attributes. Some attributes exclude specific rewritings[7]. For example for a regular expression

$$X ::= [\alpha]\ [\beta]\ [\gamma]$$

there could be an attribute α_exists of type Boolean (with α_exists $\equiv \alpha$ rewrites not to ϵ) and evaluation rule

if α_exists

then $X ::= \alpha\ [\beta]$

else $X ::= [\beta]\ [\gamma]$

When replacing this grammar by a grammar without ϵ-rules the number of rules explodes (more than 10000 rules) because of many optional symbols in the right-hand-sides of the grammar rules (see section 8.3). For natural language grammars a lot of grammar rules can be written with optional symbols in the right-hand-side.

8.5.3 the number of sentences with respect to the number of grammar rules

It is important to see if the part of the corpus we used to extract the grammar represents a large part of the possible syntactical constructions. In other words: how does the number of grammar rules increase if the number of sentences the grammar rules are extracted from increases? In figure 8.3 the relation between the number of sentences and the number of grammar rules is shown. We used four parts of the corpus. Starting with one part, we added another part and so on. In the figure one can see the number of rules of the pure grammar (without reductions), the number of rules of the reduced grammar (reduced for multiple replication, single element phrases and quote symbols) and the number of rules without verb of the reduced grammar.
The number of rules grows fast. While adding more analyzed sentences to extract from at a moment the grammar growth will reduce. But with 5000 analyzed sentences a lot of syntactical constructions does not occur.

[7]For this reason we were not able to use the ROSETTA grammar for our research on probabilistic parsing. Besides the problem of the power of some attributes to exclude specific rewritings there is a problem because the grammar uses few syntactical categories because a lot of syntactical information is expressed in attributes. So it was not useful to rewrite the grammar-without-attributes for attributes that exclude specific rewritings (in any case that would be a lot of work) because the syntactical categories are too less expressive. Rewriting the grammar with new syntactical categories would take a lot of time for a linguist.

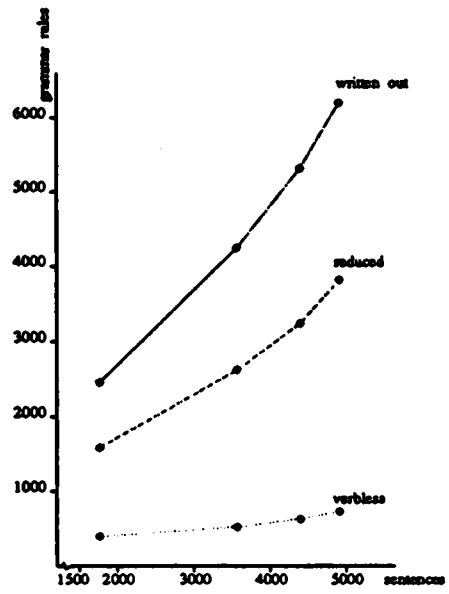


Figure 8.3: number of grammar rules with respect to the number of sentences

8.6 conclusions

A grammar that has been extracted from an analyzed corpus has great disadvantages. Multiple replication of symbols is written out. It is easy to replace multiple replication with a more compact notation. But using optional symbols will be difficult if a consistent, not highly over-generating, manageable grammar is wanted. Using auxiliary categories is possible if the auxiliary categories have been represented in the parse trees. Otherwise a linguist has to rewrite the grammar with use of auxiliary categories if a consistent, not highly over-generating, manageable grammar is wanted.

It is important which syntactic categories are used. The size of a grammar may be large because of the absence of a syntactic category. The choices that have been made when writing a grammar are not always the right choices for creating a minimal grammar.

Chapter 9

context-free grammars

9.1 introduction

In this section we will examine the factors that influence the size of a grammar on the basis of the results described in the preceding chapters (section 9.2). We will look at the effect on the parser if the grammar size is reduced (9.3, 9.4). The effect of the grammar size reductions on probabilistic parsing will be examined in chapter 11. In section 9.5 we will conclude the discussion about grammars with some general conclusions.

9.2 grammar size

9.2.1 introduction

In the original (pure) grammar extracted from the corpus, a lot of rules are needed because all syntactical constructions have to be written out. In this way it is impossible to write a grammar for common English. Such a grammar would need some tens of thousands (or more) rules. A grammar with so many rules is not comprehensible. It is difficult to modify a very large grammar. Adding one rule often has an effect on a lot of rules. Also the parser would be very inefficient. As we will see, with many grammar rules many states are needed for the parser and that has a negative effect on the parsing complexity and on the statistical complexity[1].

The number of rules can be strongly reduced if aux-categories or optional symbols are used. Aux-categories are used in the grammar related with the LDB corpus (section 7). Optional symbols are used in the surface grammar for English in the ROSETTA system. In the following paragraphs these two methods are examined.

9.2.2 aux-categories

If aux-categories are used, the idea of a grammar with written out rules still exists. Different parts of syntactical constructions in right-hand-sides of grammar rules are taken together in aux-categories (or in aux-relations as in the corpus grammar). Aux-categories and aux-relations can be taken together in other (aux-)categories and (aux-)grammatical

[1]The statistical complexity is the complexity with respect to the amount of statistical information that is needed.

relations and so on. An example is the rewriting for category noun phrase in the corpus grammar:

- c np noun phrase:

  nf left, nf np noun phrase inner, nf right.

- nf np noun phrase inner:

  nf np noun phrase simple;

  nf np noun phrase coord;

  nf np noun phrase axb;

  nf np noun phrase apposition.

- nf np noun phrase simple:

  nf normal noun phrase;

  nf nominal adjective phrase;

  nf gnp genitive noun phrase simple.

- nf np noun phrase coord:

  nf coord option, f join np noun phrase, f coor,

  nf coorded anything.

- nf np noun phrase axb:

  f a np noun phrase, f lim pcl, f xb.

- nf np noun phrase apposition:

  f a left appositive, f b right appositive.

- nf normal noun phrase:

  nf np front, f hd in np, nf np tail.

- nf np front:

  nf np det list, nf dumb filler like list, nf np prem list;

  f prem in np, f det in np, nf dumb filler like list,

  nf np prem list.

If a written out grammar is transformed into a grammar with aux-categories/aux-relations, rules of the original grammar and parts of the right-hand-sides of rules are split up in groups. For such a group an aux-category is introduced that rewrites to that group. Because a group may occur at more than one place in the grammar, the number of grammar rules reduces if aux-categories are used.

9.2.3 optional symbols

The method used in the surface grammar for English of the ROSETTA system differs from the method with aux-categories. In fact the grammar rules are not splitted up but they are merged by using optional syntactical categories. For example with one regular expression the rewriting of the prepositional phrase can be written as:

PREPP ::= [ADVP | NP] PREP [NP | PREPP | ADVP | ADJP | SENTENCE]

(Something between [] is optional. The symbol '|' stands for OR. This regular expression can be written in ten rules with BNF notation together with optional rewritings for the categories ADVP, NP, PREPP, ADJP and SENTENCE.)

If attributes may be used the number of rules can be strongly decreased because context-sensitive effects can be controlled by attributes. Then for the rules

A → B C E

A → C E F

we can write

A → optional_B C E optional_F

and a little attribute administration/evaluation.

Without attributes only local optional effects can be used to decrease the number of rules. An example is the rule

A → B optional_C D

derived from the rules

A → B D

A → B C D

Using optional symbols for reducing the grammar size is especially of great use if attributes are used. Otherwise the reduction effect will be less big and over-generating will be easily introduced.

9.3 number of parsing states

For the RAM parser without look-aheads the number of states is at most $O(|V| \cdot 2^{\#(G)})$, with $\#(G)$ the number of grammar rules. This is an extreme upper bound, in practice often the number of states grows linear with the number of grammar rules (see section 3). The number of states depends on the number of (non)terminal symbols and on the number of grammar rules.

With a grammar that uses aux-categories, it is possible to parse. In the produced parse tree(s) the aux-categories can be eliminated (this can be done while parsing). If aux-categories are added, the number of nonterminals increases, so the reduction of the number of states because of fewer grammar rules will be opposed by the increase of the number of nonterminals. Of course the reduction of the grammar size has a much greater

effect than the increase of categories. With aux-categories the use of attributes will be more complex. More evaluation rules have to be written.

Reducing the number of grammar rules by merging grammar rules and using optional symbols does not strongly increase the number of categories, only for optional categories new categories are needed. A disadvantage of this method is that attributes are needed if one wants a considerable reduction of the grammar size (see section 9.2.3). In that case the attribute administration will be complex. Such a grammar cannot be used for probabilistic parsing (see section 11).

9.4 parsing complexity

Suppose we have a grammar G_{red} that describes the same language as grammar G but with fewer rules (reduction with aux-categories and/or with optional symbols). A parser for G needs more states than a parser for G_{red}. So the state complexity (in LR terms the complexity of the parsing table) reduces. But with this transformation, another complexity, the computational complexity of the parsing process increases. The growth of the computational complexity may occur in the use of ϵ-rules and in the parsing search space. In following sections we will examine this complexity.

9.4.1 computational complexity and ϵ-rules

If we use the original Marcus-like parser described in [Leermakers, 1993], the computational complexity of the parsing process for a parser of a grammar with ϵ-rules is greater than the computational complexity of the parsing process for a parser of a grammar without ϵ-rules. This can be seen in the definition/implementation of the functions $[q]$:

$$\begin{aligned}[q](i) \;=\; & \overline{[q]}(x_{i+1}, i+1) \\ & \cup \{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_B(B \rightarrow .\epsilon \in ini(q) \\ & \qquad\qquad \wedge (\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(B, i))\} \\ & \cup \{(\gamma \rightarrow \alpha., i) \mid \gamma \rightarrow \alpha. \in q\}\end{aligned}$$

Without ϵ-rules the set

$$\{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_B(B \rightarrow .\epsilon \in ini(q) \wedge (\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(B, i))\}$$

does not have to be computed. With ϵ-rules the parser has to try all ϵ-rewriting possibilities. Recognizing a ϵ-rewriting causes other reductions. With a lot of ϵ-rules this could be a problem in practice. So if the problem of too much states is partly solved by using optional symbols, a problem with the complexity of the actual parsing process could occur.

In a variant of the original Marcus-like parser (the RAM parser) it was tried to overcome this problem. This parser does not try to compute ϵ-rewritings. The parser simply skips them as can be seen in the definition of *goto*:

$$\begin{aligned}goto(q, \delta) \;=\; & \{\gamma \rightarrow \alpha\lambda\delta.\beta \mid \gamma \rightarrow \alpha.\lambda\delta\beta \in (q \cup ini(q)) \\ & \qquad\qquad \wedge \delta = k{:}\delta\beta \wedge \lambda \rightarrow^* \epsilon\}\end{aligned}$$

This does not affect the parsing process. Before the actual parsing of sentences (or afterwards), it can be computed in which ways a nonterminal rewrites to ϵ.

A call $goto(q,\delta)$ will return a greater set of items with ϵ-skip than without ϵ-skip. Also the number of states will be a little higher. But the profit by using ϵ-rules will be kept. For example for a state q_{PP} = { PP → α . opt_NP PREP β } with

opt_NP → ϵ

opt_NP → NP

there are transitions on NP, opt_NP and PREP. For computing the items of $goto(q_{PP}$, PREP), it is not important to know if opt_NP rewrites to ϵ or to NP. In the case without ϵ-rules we have:

$$q_{PP} = \{\text{PP} \rightarrow \alpha\,.\,\text{NP PREP}\,\beta,\ \text{PP} \rightarrow \alpha\,.\,\text{PREP}\,\beta\}$$

with transitions on PREP and NP. For the state $goto(q_{PP}$, NP) there is also a transition on PREP. It will be clear that the PREP-transition for state q_{PP} corresponds with opt_NP → ϵ and the PREP-transition for state $goto(q_{PP}$, NP) corresponds with opt_NP → NP.

9.4.2 computational complexity and parsing search space

A grammar with aux-categories is less flat than a grammar without aux-categories. That means that for most items $\gamma \rightarrow \alpha\,.\,\beta$ there are more rewritings of the form $\beta \Rightarrow^* v$. With an item $\gamma \rightarrow \alpha\,.\,\beta$, β has longer chains of derivations. This is an effect in the depth of derivations. Shifting a terminal symbol t leads to more reductions in depth. For a grammar with optional symbols the same effect occurs, because of the extra optional categories. Unless there are many extra categories this will not be a real problem. The computational effect will be very small, certainly in contrast with the efficiency of the parser for the original (not reduced) grammar.

The computational complexity may increase if it is possible after introduction of new categories that

$\nu_1 \Rightarrow^* x_i \ldots x_j$

$\nu_2 \Rightarrow^* x_i \ldots x_j$

for an item

$\gamma \rightarrow \alpha.\beta$

with

$\beta \Rightarrow^* \nu_1\eta_1$

$\beta \Rightarrow^* \nu_2\eta_2$

($\nu_1, \nu_2 \in V_N$ (when $k = 1$); $\eta_1, \eta_2 \in V_N^*$; $i,j,l \in$ N; $i \leq j$; $i \leq l \leq j : x_l \in V_T$.) The recognizer has to compute more reductions than before. In practice this problem does not occur if one will try to write consistent grammars.

9.5 conclusions

For a context-free, attribute-free grammar that can be parsed and that describes a large subset of a natural language, the way in which the grammar is written is important. To work with such a grammar the grammar has to be not very big (more than 2000 rules). Grammars with fewer than 1000 rules are more attractive. Without attributes it is difficult to describe a large subset of a natural language in 1000 or fewer context-free rules. Auxiliary categories have to be introduced to make the grammar more compact. Another possibility is to use optional symbols in the grammar rules, but then it will be likely to use attributes for excluding specific rewritings. A combination of both methods is possible. The (time) complexity of parsers for grammars written according to one of these methods does not really differ. A parser for a grammar with a lot of optional symbols has to handle ϵ-rules efficiently.

Part III

Probabilistic Parsing

Chapter 10

probabilistic parsing

10.1 introduction

Parsing with a grammar that describes a large subset of a natural language is a difficult task. Such a grammar is highly ambiguous. Parsing will take cubic time (with respect to sentence length) if only syntactical information is used. It has often been said that the syntactical component has to be followed by a component that handles semantical information. Typically one could read this in the section 'future research'. For the time being it is impossible to model semantical information for large domains.

It is necessary to find another way to reduce the ambiguity problem. Some people think that a statistical approach is an important step to the solution (e.g. [Bod, 1992], [Brill et al., 1990], [Briscoe & Carroll, 1991], [Chitrao & Grishman, 1990], [Fujisaki et al., 1989], [Robinson, 1982], [Sharman et al., 1990]). Statistical information could guide the parsing process in such a way that not all ambiguities have to be computed. A probability is assigned to every parse tree of a sentence. For an ambiguous sentence it is expected that in most cases the desired parse tree will be the parse tree with the highest probability. But parsing with use of statistical information is a negative choice. Because we cannot model the knowledge needed for reducing ambiguity we use a very rough approximation of this knowledge. The use of statistical information has not yet been proven to be part of the solution to the ambiguity problem. In section 10.2 we will examine the main problems of probabilistic parsing. An approach in which a lot of statistical information can be expressed will be discussed. Thereafter two other approaches to probabilistic parsing are presented. In one approach the statistical information is collected about context-free data (section 10.3 and 10.4). In the other approach statistical information is context-sensitive (section 10.5). In the next chapter we will discuss in detail the case of context-sensitive probabilistic parsing with use of LR like parsers.

10.2 statistical significance[1], computability and corpus size

10.2.1 probabilistic parsing with subtrees

For a given sentence, one wants the most likely parse tree. Statistical information can be collected about syntactical structures and semantical information by using parse trees, the words of the sentences and semantical features. With more aspects expressed in the statistical information, more analyzed data will be required to train the 'parser'. Even if only the lexical categories of the words of a sentence are used as input of the parser, it is not possible to compute the most likely parse tree of every sentence (thus of every sequence of lexical categories). Therefore, probabilities are assigned to parts of the parse trees. The probability of a parse tree is computed from the probabilities of parts of the parse tree. In most approaches to probabilistic parsing, grammar rules are used as the basic parts statistical information is collected about. In the approach of Bod and Scha ([Bod, 1992]), these basic parts are parts of parse trees. Such a part can be a complete parse tree, it can also be one rewriting from a parse tree. In this approach something like semantical information has been included, because the words of the sentences are part of the parse trees. With a corpus that consists of the two trees showed in figure 10.1, for the

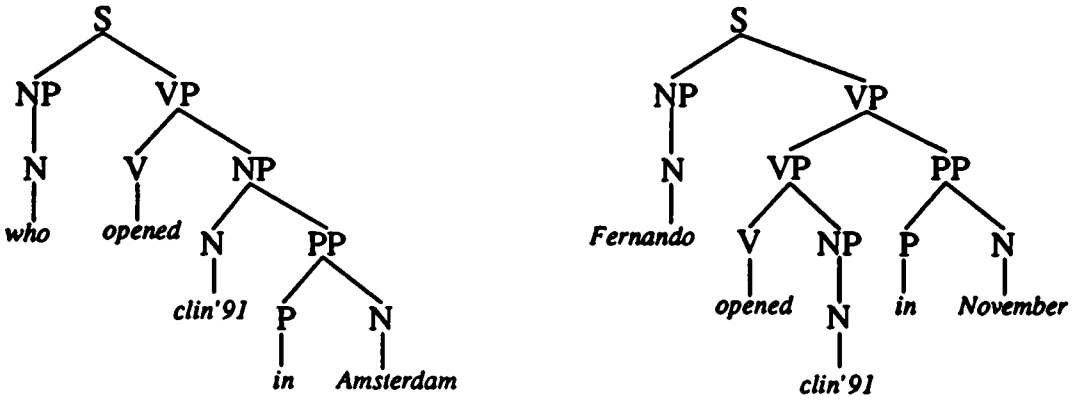


Figure 10.1: corpus consisting of two trees

sentence "Fernando opened clin'91 in Amsterdam in November" a tree can be constructed from three parts of the two corpus sentences (figure 10.2). Because there is a complexity measure for every part of a parse tree, their system handles ambiguous constructions with ease. Only the parse tree with the smallest complexity will be constructed.

10.2.2 criticism

Our criticism against the Data Oriented Parsing (DOP) approach of Bod concerns the use of a formal grammar, the matching process and the size of the analyzed corpus.

[1]We use the word significance to express the fact that statistical information can be collected about data that expresses little information (for example (non)terminals) as well as data that expresses a lot of information (for example parse trees). In the first case the statistical information is less significant than in the second case.

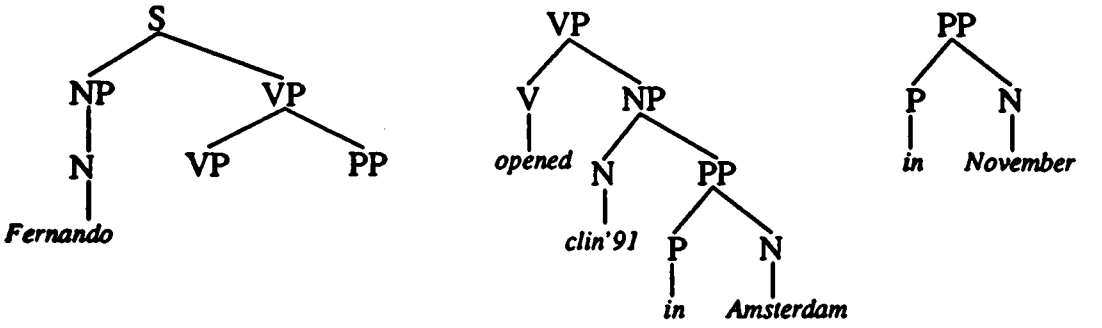


Figure 10.2: parts of the tree of the sentence "Fernando opened clin'91 in Amsterdam in November"

In the view of Bod, language data in the form of an analyzed corpus constitutes the basis for language processing in DOP. The analysis of a sentence is constructed with existing (sub)constructions in the corpus. Thus for analyzing sentences with DOP, an analyzed corpus is needed. This analyzed corpus has to contain all sentence (sub)constructions that cannot be constructed with (sub)constructions that occur in this analyzed corpus. The basis of such an analyzed corpus is a grammar with grammar rules. Thus in fact a formal grammar with grammar rules constitutes the basis for language processing in DOP. The problems with developing a formal grammar for natural language are also problems for the DOP model. For example, the grammar represented by parse trees should not over-generate much. As we have shown in chapter 8, using a grammar that is represented by parse trees (as is done in the DOP model) will give difficulties with for example optional symbols and multiple replication of a symbol. So the use of an analyzed corpus may have undesired effects.

For a very large corpus of analyzed sentences, the matching process used to construct parse trees from parts of parse trees will take a lot of time. With rule based parsing, the PP in figure 10.3 can be a daughter of grammar rules with the nonterminal PP in

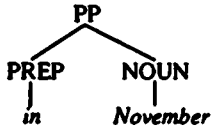


Figure 10.3: a prepositional phrase

the right-hand-side. With Data Oriented Parsing, the PP can be a 'daughter' of a lot of different parts of parse trees that have a nonterminal PP as leaf. The parts of parse trees that can be used to construct new parse trees can be seen as grammar rules. Such a grammar will contain much more rules than the grammar that has been used to make an analyzed corpus. If a lot of parse tree constructions do not have to be computed because of the complexity measure, the matching process will not take as much time as in the worst case. Our expectation is that for a very large corpus a lot of parts of parse trees do not have such a bad complexity that they do not have to be computed.

The number of parse computations will be reduced because a lot of constructions are not possible due to the fact that the words in a lot of parts of parse trees do not correspond with the words of the input sentence. But because a very large corpus is

needed (see hereafter), the matching (parse) process will still need a lot of computations.

Useful statistical information has been collected if almost all possible parse constructions occur in the analyzed corpus with a realistic frequency. Only then a part of a parse tree will not be used because there are no other possibilities, but because it is the most likely one of a set of possible parts of parse trees. As we will show in section 11.2.3, for grammar rules with a little context a very large corpus is needed to get useful statistical information. So, with use of grammar rules with a lot of context (parts of parse trees) a very, very large corpus will be needed to get useful statistical information.

10.2.3 conclusions

An approach like DOP uses large structures as basic units. Statistical information must be collected about these structures. Because large structures are more specific than small structures, more analyzed data is needed to get useful statistical information. Thus there is a trade-off between the size of the train corpus and the size of basic units statistical information is collected about (the statistical significance). With full parse trees as basic units, we know that it is impossible in practice to get useful statistical information. With an approach like DOP it is unrealistic that it is possible to get useful statistical information. Besides this there may be a computational problem with the matching process. In the approaches examined in the following sections, it may be possible to collect a lot of statistical information, but the significance could reduce because the basic units are too small.

10.3 probabilistic context-free grammars

10.3.1 description

To assign a probability to a parse tree, probabilistic context-free grammars (PCFGs) are used (for example in [Jelinek et al., 1991], [Wright et al., 1991], [Ng & Tomita, 1991], [Fujisaki, 1984]). A PCFG is a context-free grammar with a probability of use assigned to every grammar rule. A grammar rule is of the form $A \rightarrow \alpha$ with $A \in V_N$ and $\alpha \in V^* \cup \{\epsilon\}$. So with A a nonterminal with $\#(A)$ different rewritings and $P(A_i)$ the probability of the ith rewriting of A the following holds:

$$\sum_{i=1}^{\#(A)} P(A_i) = 1$$

The probability of a parse tree is computed from the probabilities of the grammar rules used in that tree. The probability of a grammar rule represents its use score. An assumption when using PCFGs is that a rule has a certain probability of use irrespective of the context of the rule. So with a PCFG the probability that the rule $NP \rightarrow DET\ NOUN$ is used as daughter of a rule $NP \rightarrow \ldots NP \ldots$ equals the probability that the rule $NP \rightarrow DET\ NOUN$ is used as daughter of a rule $PP \rightarrow \ldots NP \ldots$ (see figure 10.4 for an example). We have checked this for a part of the corpus. In almost 25% of the rewritings of the NP in the right-hand-side of a rule $NP \rightarrow \ldots NP \ldots$, the rule $NP \rightarrow DET\ NOUN$ is used. The rule $NP \rightarrow DET\ NOUN$ is used in almost 44% of the rewritings of the NP in the right-hand-side of a rule $PP \rightarrow \ldots NP \ldots$. The use of

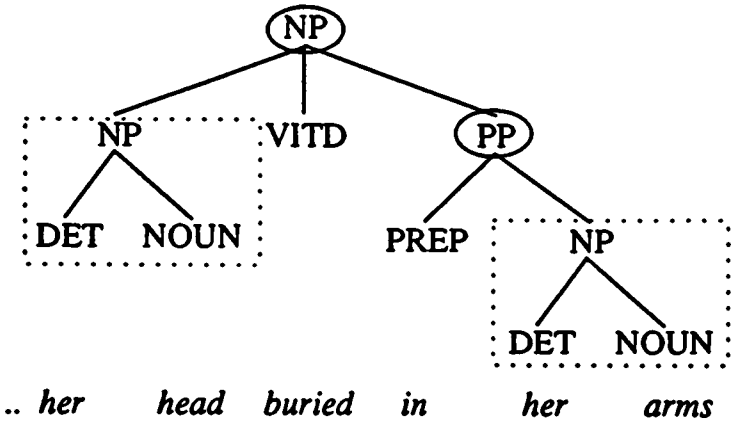


Figure 10.4: context-sensitive use of grammar rules

a rule in a derivation is not always independent of the use of other rules as is suggested by PCFGs.

The statistical information can be collected from an analyzed corpus. In such a corpus one parse tree has been assigned to every sentence. For every grammar rule it can be counted how many times the rule has been applied in the parse trees of the corpus sentences. The grammar that has been used to make the parse trees has to be (almost) the same as the grammar that has to be made probabilistic.

Another possibility is to use the Inside Outside algorithm (derived from a Markov model) as described in [Fujisaki, 1984]. This is an unsupervised method. Sentences are parsed and the statistical information is updated for all parse trees. So all parse trees of an ambiguous sentence contribute to the statistical information. Repetition of the process over the used set of sentences will lead towards convergence. The result will approach the result of the supervised method with an analyzed corpus. It is not clear whether the result of unsupervised training is a good approach of the result of supervised training. With a large, highly ambiguous grammar there is a lot of noise in the statistical information. If one wants to see what is at most feasible with statistical information it is better to use an analyzed corpus.

10.3.2 experiments with PCFGs

Fujisaki describes an experiment with a PCFG in [Fujisaki, 1984].[2] He transforms a grammar in Greibach Normal Form into a grammar with 7550 rules in Chomsky Normal Form. He uses two unanalyzed corpora of 3582 respectively 624 sentences to train and test the probabilistic parser. He does not describe the corpora (style of the sentences etc.). The average wordcount of a sentence is 10.85 in the corpus with 3582 sentences. The average wordcount of a sentence is 12.65 in the other corpus. The parser returns the parse trees of a sentence in order of probability. After collecting statistical information with use of the Inside Outside algorithm, the parser is tested with respectively 63 (in the case of the first corpus) and 21 (in the case of the second corpus) ambiguous sentences from the train set. The delivered parse tree with the highest probability is the good parse

[2] In [Fujisaki et al., 1989] the same experiment is described. It seems that Fujisaki has not made real progress in the research at the possibility of probabilistic parsing.

tree in 85% of the test sentences. Because the experiment is not described in detail, it is not possible to conclude that it is useful to parse statistically with PCFGs.

10.4 ID/LP grammars used for probabilistic parsing

10.4.1 description

In the view of Sharman, Jelinek and Mercer [Sharman et al., 1990] context-free grammars of general natural language are large, very ambiguous and parsing takes a lot of computations. So for unrestricted texts (a large subset of a natural language) compact notations for grammars have to be found. Unification grammars use a compact notation. Sharman et al. use the ID/LP principle of generalized phrase structure grammars for writing a compact grammar. According to this principle a grammar has to be written in two sorts of rules: immediate dominance (ID) and linear precedence (LP) rules. The ID rules express to which symbols a nonterminal could rewrite. LP rules express in which order symbols may appear. So the rules

ID : S → A B C

LP : B < A (B before A)

are equivalent with the context-free grammar rules

S → B A C

S → B C A

To get a statistical parser, probabilities are assigned to ID rules as well as to LP rules. In this case LP rules say with which probability symbols may appear in a specific order. Probabilities are also assigned to the tags in the lexicon. An analyzed corpus and a grammar with 16 nonterminal and 100 terminal symbols are used. The test set exists of 42 sentences. If the test set differs from the learn set the most likely parses are the correct parses in 20 percent of the cases. Nearly correct or correct is 60 percent of the most likely parses. If the sentences of the test set are used to extract the probabilities (test set = learn set) the most likely parses equal the correct parses in 43 percent of the cases. Nearly correct or correct is 88 percent of the most likely parses.

10.4.2 criticism

Sharman, Jelinek and Mercer do not answer the question whether the probabilities of ID and LP rules are independent. But the answer on this question is of great importance. It says whether the use of a probabilistic ID/LP grammar is justified or not. It is very likely that the probability of the linear precedence of two symbols depends on the immediate dominance rule in which the two symbols occur. For the rules

ID : S → A B C

LP : B < A

a probability is assigned to the LP rule on the basis of the use of this rule in the parses of the learn sentences. The probability of the LP rule does not express the linear precedence probability of A and B in the rule S → A B C if there are other rules with the symbols A and B in the right-hand-side. So with probabilities assigned to written-out context-free rules like

S → B A C

more information can be expressed.

The probabilistic ID/LP model is presented for ID rules with two symbols in the right-hand-side. For rewritings with an unlimited number of symbols on the right-hand-side, the rewriting is modeled as successive applications of shorter rules. Here the same disadvantage occurs: there is a difference between the probability of a rule

S → A B C

and the probabilities of the rules

S → A B'

B' → B C

if the rule B' → B C is used in other combinations than with the rule S → A B'.

The conclusion is that with a context-free grammar more information can be expressed than with a ID/LP grammar. So the statistical information that can be collected with the ID/LP grammar is less significant than the statistical information that can be collected with context-free grammars.

Sharman, Jelinek and Mercer do not give enough information about the grammar they used, to criticize their experiment in detail. It seems that it is not easy to attain good results with probabilistic parsing for a subset of a natural language.

10.5 context-sensitive probabilistic parsing

10.5.1 Pearl

Magerman & Marcus [Magerman et al., 1991] criticize PCFGs because of the assumption that the applications of two grammar rules in a parse tree are independent. They make the probability of use of a grammar rule dependent of a small context (a part-of-speech[3] trigram) and a parent theory's rule (a rule with in the right-hand-side the left-hand-side of the child rule). For their parser Pearl they need a big analyzed corpus. May be an unanalyzed corpus is also possible (unsupervised training), but till now this has not been tested. The description of their experiment does not give details of the used grammar, the used corpus and the size of the tests.

10.5.2 probabilistic parsing of messages

Chitrao and Grishman [Chitrao & Grishman, 1990] use probabilistic parsing to parse sentences from a small domain of Navy OPREP messages. A context-sensitive version

[3]A part-of-speech of a word is the lexical category assigned to that word.

of a probabilistic context-free grammar has been used. The probability of a grammar rule depends on the place of occurrence of the left-hand-side of the rule. With a PCFG small structures are preferred over 'large' structures, because when a rule is added to a structure the probability of that structure is multiplied with a value smaller than one. Chitrao and Grishman try to remedy this problem by using penalties for small structures. This does not improve the result.

The grammar was trained with use of the Inside Outside algorithm. The train corpus contained 300 sentences. From the train corpus 140 sentences were used to test the parser. With the context-sensitive probabilistic parser, the correct parse was the most likely parse of 38 percent of the 140 sentences. This is a bad result because only sentences from the train corpus were used to test the parser.

10.5.3 probabilistic LR parsers

An approach in which statistical information is used in a context-sensitive way and that joins with the architecture of LR parsers is proposed by Briscoe & Carroll [Briscoe & Carroll, 1991] and by Leermakers [Leermakers, 1991]. Parsing is a problem because a grammar of a natural language is highly ambiguous. In terms of LR parsers: parsing is a problem because there are shift-reduce and reduce-reduce conflicts in a lot of states. If these conflicts could be eliminated, parsing of a large subset of a natural language would be easy. So the most needed information for a LR parser is not the likelihood of a specific grammar rule being used, but how likely a specific (shift or reduce) action is in contrast with conflicting actions. With this information, the parser could restrict the ambiguity in a state to the most likely actions. In the next chapter this approach is described in detail.

10.6 conclusions

Statistical information can be collected about parse trees, about parts of parse trees, about grammar rules and so on. With more complex structures statistical information is collected about, the size of the analyzed corpus that is needed to get the statistical information increases. A manageable approach seems to be probabilistic parsing with use of probabilistic context-free grammars. With pure PCFGs a lot of context that can increase the statistical power is thrown away. Therefore, people search for manageable, context-sensitive approaches. The results of some (very) small experiments gives someone no cause to expect miracles of probabilistic parsing methods.

Chapter 11

probabilistic LR-like parsing methods

11.1 introduction

The general idea of probabilistic LR-like parsing is the use of statistical information about the shift and reduce actions of the parser. In this way it is possible to compute the most likely parse tree of a sentence from the set of all parse trees of that sentence. If the parser only has to take the most likely action(s) in every situation in the parsing process, the parser can work in (almost) linear time. With PCFGs the parser does not dispose of the right statistical information. With a PCFG the probabilities of the occurrences of grammar rules are known. The probabilities of shift and reduce actions are probabilities of occurrences of grammar rules in specific contexts.

In section 11.2 we will describe and criticize the approach and experiment of Briscoe and Carroll. The experiment is criticized. In section 11.3 we will present a better approach. In section 11.4 we will take some little conclusions about probabilistic LR parsing. In the next chapter we will take general conclusions on the basis of our research results.

11.2 probabilistic LR parsing by Briscoe and Carroll

11.2.1 the idea

Briscoe and Carroll try to develop a model for parsing with large natural language grammars. Their parser is based on a unification grammar. They generated a context-free backbone grammar with attributes from this grammar. The parser they work with is a LALR(1) parser. They choose a LALR parser, because the parsing table for a LALR(1) parser is much smaller than the parsing table of a LR(1) parser. With large grammars this is important. They only use one look-ahead because of the size of the parsing table.

For collecting statistical information sentences are analyzed to get one parse tree per sentence. Briscoe and Carroll do not like the unsupervised learning method because it is not certain how good this method is, compared to the supervised method. They developed a tool to analyze sentences efficiently.

Statistical information is collected about the shift and reduce actions of the parser. Reduce actions in a state depend on the look-ahead symbols. Because reduce actions are

not always distinguishable on the basis of the look-ahead symbols they are also distinguished by the state reached after the reduce action has been applied.

11.2.2 experiment

The grammar consists of 1758 rules. There are 575 syntactic categories. The parser has 3106 states and 1020860 different actions. The parser has been trained and tested with noun definitions[1]. 246 noun definitions were parsed. The parser was able to parse 151 of these definitions correctly. A few rules (14) were added to the grammar. It is not clear why the other 95 definitions were eliminated.

The 151 definitions were analyzed and the results were used to get the statistical information. The parser had to compute all possible parses. Because this took a lot of time only 63 of the 151 definitions were tested. The noun definitions are short. Their mean (word) length was 5.3. None of these 63 definitions had a length of 10 or more words. 13 of the 63 definitions were not ambiguous. For 47 of the 63 definitions the most highly ranked parse tree is also the correct one. For 4 of the remaining cases the most highly ranked was correct, but the analyses of the definitions (used to get the statistical information) were false. This gives a (most highly ranked parse = correct parse)/sentence score of 81%. For a test with 54 noun definitions not in the train set this result was 57%. These 54 definitions consist of at most ten words. 10 of the 54 definitions were not ambiguous. In the view of Briscoe and Carroll, the bad results of this test were mainly caused by the incompleteness of the grammar.

11.2.3 the experiment criticized[2]

The used grammar was derived from a unification based grammar. Syntactical categories were created from sets of features. Briscoe and Carroll tried to define syntactical categories as good as possible. In their view a context-free backbone has to express as much information as available in the unification grammar. In this way a large context-free grammar results that probably expresses more than the needed information. For a context-free grammar for probabilistic LR parsing one needs a grammar that describes legal syntactical constructions and nothing more. Extra information can be expressed in features (attributes).

The parser has 1020860 different shift and reduce actions. Suppose for sentences with a length of ten words the mean number of applied grammar rules is 15. Then with 151 sentences with an average length of ten words or less at most some information about 2300 actions could have been collected. So only a very small part of the parser is trained. Because the noun definitions are very short, the trained part is not the most ambiguous part of the parser.

Our expectation is that for a number of sentences the correct parse tree is the most likely one because the actions applied in the other (ambiguous) parse trees are (very) unlikely due to the small train set. For a number of ambiguities the probabilistic parser

[1] A noun definition is a description of the meaning of a noun. Examples are "the act of washing oneself", "place where one lives" and "strong expressions of approval and praise".

[2] Detailed criticism was only possible because of the detailed description of the experiment. If all people would write so exhaustive as Briscoe and Carroll people could learn much more from each others research than at the moment.

does not know that they could occur, because they occur nowhere in the parse trees of the train set.

So our main criticism is that only a very small part of the parser has been used in the experiment and that the number of train sentences is too small. It is not possible to say something about the possibility of probabilistic parsing on the basis of this experiment.

Is it possible to build a probabilistic parser based on a parser with more than 1000000 actions? Suppose the average sentence length is 15 words. Suppose the average number of applied grammar rules in a sentence of 15 words is 20. Then about 51000 sentences are needed to apply every action one time if all actions used in the parse trees of the 51000 sentences are different. To get real statistical information the train set has to consist of more than 10 times 51000 (analyzed) sentences. In practice a large number of applied actions will be the same. Thus if one wants statistical information about all actions of the parser the train set has to contain much more sentences. With a need for so many analyzed sentences it is better to try to reduce the size of the grammar.

11.3 a model for probabilistic parsing

11.3.1 look-aheads

Briscoe and Carroll use a small look-ahead of terminal symbols to reduce the ambiguity. In the view of Marcus one should not only use terminal symbols as look-ahead but nonterminal symbols as well. Using nonterminal and terminal symbols as look-ahead is more powerful than only using terminal symbols as look-ahead (in the case nonterminal look-aheads do not rewrite to ϵ). One nonterminal look-ahead may examine more than one terminal symbol. There is another advantage of constituent look-ahead for probabilistic parsing. A lot of nonterminal symbols can left-rewrite to the same terminal symbol. So with a reduce-reduce conflict or a shift-reduce conflict in a parser, a more significant decision can be taken on the basis of the next nonterminal than on the basis of the next terminal symbol. In the following example the difference between the power of terminal and constituent look-ahead can be seen.

The use of look-aheads will reduce the number of parsing conflicts. Sometimes it is possible to postpone parsing decisions if one or more look-aheads are used. With the following grammar:

- $ROOT \rightarrow S$
- $S \rightarrow A\ S$
- $S \rightarrow d$
- $A \rightarrow a$
- $A \rightarrow a\ S$

a parser without look-aheads has a shift-reduce conflict with sentence "ad" because sentence "add" has the same left symbol string but a different rewriting of symbol A (figure 11.1). After recognizing symbol a the parser has to reduce symbol A or to shift the next input symbol. The parser has to know what the following input symbols are, before it can take the correct decision.

Figure 11.1: parse trees for the sentences "add" and "ad"

With one Marcus-like look-ahead ($k = 2$) the parser can shift the first two input symbols *ad* and then it can take the correct decision on the basis of the existence of another input symbol *d*. The graph in figure 11.2 of the states with their transitions shows this. After a shift of symbol *a* and a shift of symbol *d* the parser can reduce rewriting $S \rightarrow d$ or it can shift the next input symbol. If there is not a next input symbol the parser can only reduce $S \rightarrow d$.

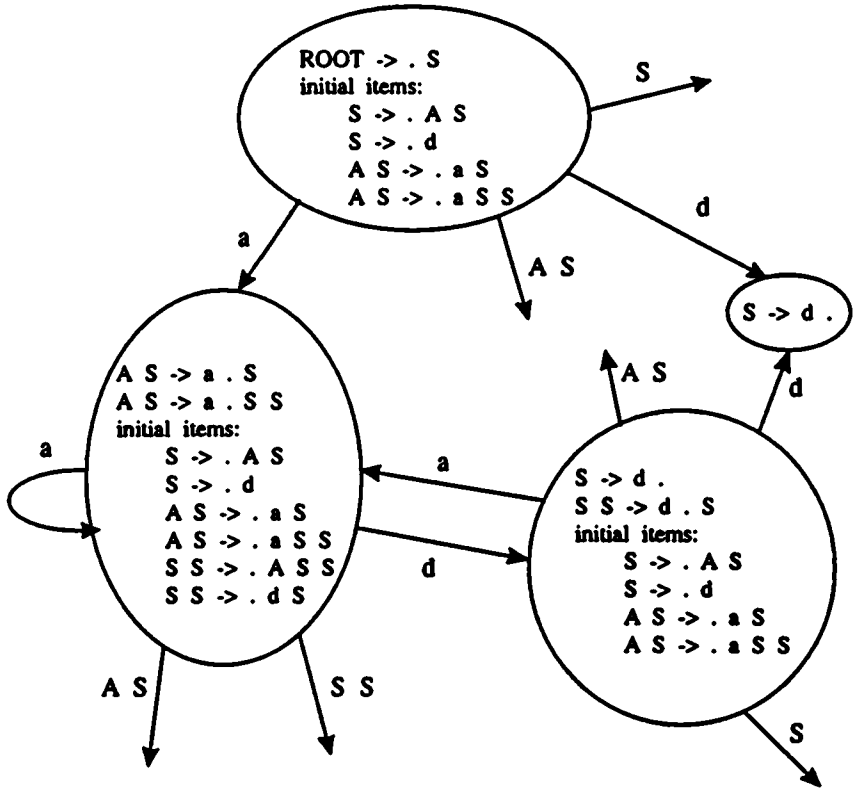


Figure 11.2: states with transitions

With terminal look-aheads (as with LR(k) parsing) one terminal look-ahead would be not enough to solve the ambiguity. After recognizing symbol *a* the parser had to decide between a shift and a reduce action. This decision depends on the third input symbol and with one terminal look-ahead the parser does not know this symbol.

11.3.2 subdividing reductions

Briscoe and Carroll use the state reached after a reduction to distinguish between different reduce actions. In this way a little bit context of the application of a grammar rule can be processed in the statistical information. With the look-aheads of the RAM parser, context of the application of a grammar rule is handled in a natural way. The following example shows the use of nonterminal look-aheads for subdividing reduce actions.

We have the following grammar:

$S \rightarrow A\ D$	$B \rightarrow a$
$S \rightarrow A\ E$	$C \rightarrow b$
$S \rightarrow B\ C$	$D \rightarrow b$
$A \rightarrow a$	$E \rightarrow b$

From root state $\{ROOT \rightarrow .\ S\}$ the parser makes a transition on symbol a. The resulting state is $\{A \rightarrow a\ ., B \rightarrow a\ .\}$ (figure 11.3). The parser has to reduce A or B. This state can also be reached from other states. So the statistical information of the two

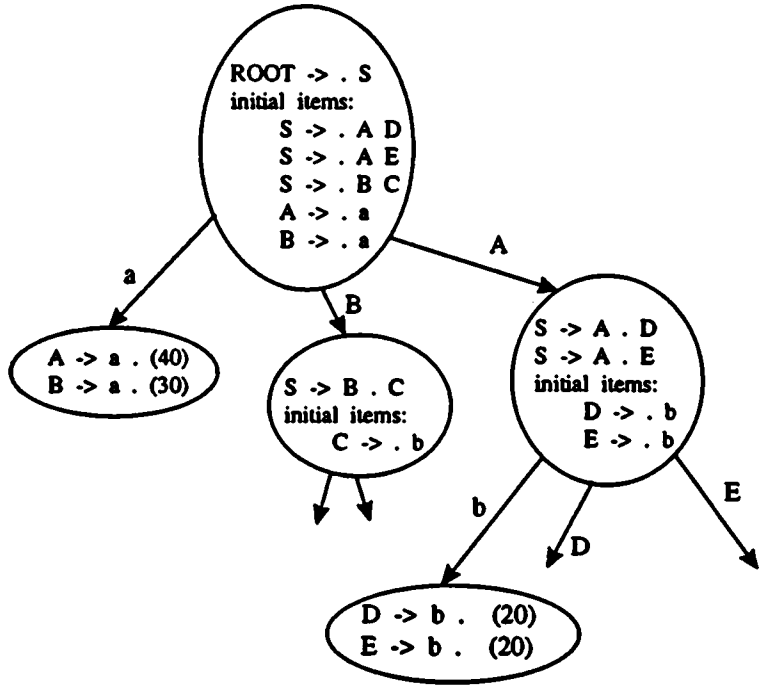


Figure 11.3: states without look-ahead, with statistical information

reduce actions of this state may have been influenced by transitions from other states than the root state. Using a look-ahead context of one look-ahead will overcome this problem a little bit (figure 11.4). The context information is not the state reached after reduction, but the symbol after the reduction symbol. With one look-ahead this reduction is implicit, it is represented in items like $A\ D \rightarrow a\ .\ D$. Using the state reached after reduction as context gives other information than constituent look-aheads. With the LALR(1) model, probably it will be needed to use extra context information, because of the little look-

ahead power of terminal symbols. If the parser has to make a choice between two different reductions, one terminal symbol does not distinguish these reductions very well. In the view of Marcus, constituent look-aheads represent more significant information. Different constituent look-aheads may left rewrite to the same terminal symbol, thus to the same terminal look-ahead.

There is another effect on reductions when using constituent look-aheads. Suppose there are no other states that have transitions to the states shown in figure 11.3. Statistical information says for example that a reduction of $A \rightarrow a.$ is made 40 times and a reduction of $B \rightarrow a.$ 30 times. If the parser chooses the most likely reduction a transition on symbol A will be made. The state that is reached in this way has items:

- $S \rightarrow A . D$
- $S \rightarrow A . E$

Suppose that according to the statistical information about this state the reductions of $D \rightarrow b.$ and $E \rightarrow b.$ occur equally often. So the relation 40:30 for reductions of A and B can be better specified. The frequencies for the rewritings AD, AE and BC of S are respectively 20, 20, 30. It could be better to choose the reduction of $B \rightarrow a..$ With one look-ahead the rewritings AD, AE and BC can be distinguished because a choice between a reduction $A \rightarrow a.$ and $B \rightarrow a.$ is postponed as can be seen in (figure 11.4):

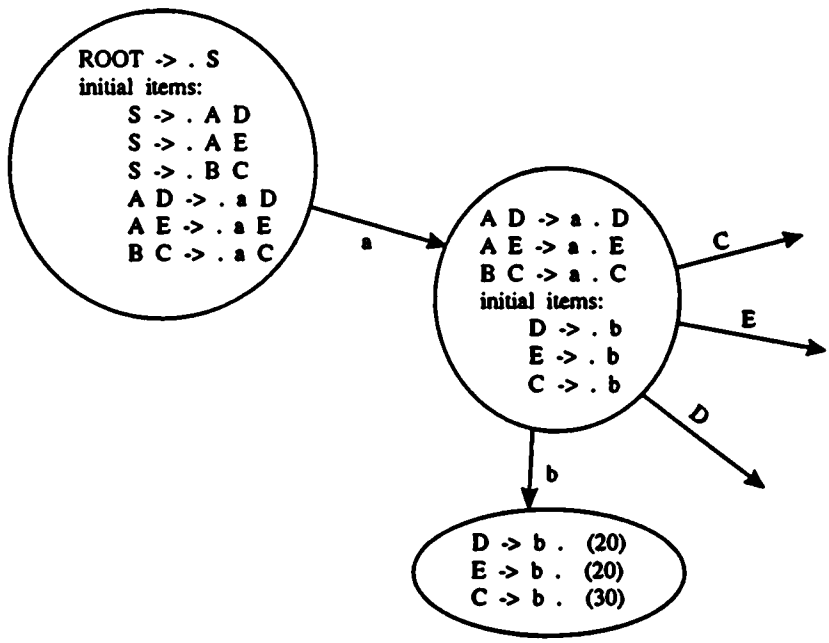


Figure 11.4: states with one look-ahead and statistical information

11.4 conclusions

For probabilistic context-sensitive parsing, a probabilistic LR parser seems a realistic model to work with. Distinguishing reductions is possible with terminal look-aheads

(LALR(k) parsing). Better (more significant) distinctions between reductions can be made with constituent look-aheads (RAM parsing).

Chapter 12

the quest for probabilistic parsing

12.1 introduction

The development of a probabilistic parser is a difficult task. Small experiments can be done. One cannot say that the results of a probabilistic parser of a large subset of a natural language will be worse (or better) than the results of such a small parser. Small subsets of a natural language do not cover all different common sentence styles, etcetera. Thus, experiments with small probabilistic parsers give at most rough indications about the usefulness of probabilistic parsing of large subsets of natural languages.

It is important to determine the factors that influence the success of probabilistic parsing. Otherwise, a lot of work may be useless. In our research, these factors have been determined. We recapitulate the results of our research in this chapter. Our research also results in a proposal for an experiment which gives a decisive answer about the usefulness of statistics in parsing of natural language in large domains.

12.2 statistical significance, grammars, parsers and corpora

As we have seen in chapter 11, it will be very difficult to get enough information needed for a probabilistic LR parser. For probabilistic parsing with use of (sub)trees, more information is needed to 'train' the parser. Hence, for the time being probabilistic parsing with use of (sub)trees is not manageable. At the other hand, probabilistic parsing with use of probabilistic context-free grammars seems unrealistic, because only the context information represented in a grammar rule is used. Probabilistic LR parsers offer a model with which a balance between manageability and the need for context information can be found. In the next three sections we will respectively look at grammars, parsers and corpora with respect to probabilistic LR parsing. In section 12.6 the relation between grammars, parsers and corpora is described.

12.3 grammars for probabilistic LR parsers

12.3.1 size of grammars

For LR-like parsing methods a context-free grammar has to be used. It is not difficult to write a context-free grammar with thousands of rules for a natural language. Such a grammar is difficult to manage. A very large grammar causes computational problems. The size of the parser will be (very) large. Generating such a parser will take a long time. More important is the problem with the statistical component. Collecting sufficient statistical information will be very problematic because a lot of analyzed sentences are needed. As we have shown in section 11.2.3 the size of the grammar used by Briscoe and Carroll creates a real problem for making the parser probabilistic.

It is realistic not to expect miracles from probabilistic parsing. May be the parser needs look-aheads to get useful statistical information. In the view of Marcus a parser needs a few constituents as look-ahead to parse deterministically. We believe likewise that a probabilistic parser needs a few constituents as look-ahead to be able to take decisions on the basis of significant statistics. The size of the parser grows rapidly when the number of look-aheads increases. For a grammar of the size of the grammar of Briscoe and Carroll it is impossible to use a parser with two or three look-aheads.

The size of the grammar has to be minimal[1]. At the same time, the grammar has to describe at least the basic part[2] of a large subset of a natural language. Syntactical constructions that are not in the basic part have to be handled by the robustness component of the parser and/or by a set of extra grammar rules. Writing a minimal grammar which describes at least the basic part of a natural language is difficult. The problem concerns the number of syntactical categories (how specific are these categories), the use of attributes and the manner of writing a context-free grammar. Because analyzed sentences are needed for training the parser one would expect that extracting a grammar from a corpus of analyzed sentences would be a useful way to get a grammar. As we have shown in chapter 8 this is not the case. A grammar for a probabilistic parser has to be written by linguists.

Different syntactical categories are used to distinguish between different language constructions. It is easy to see that a grammar with a dozen syntactical categories (e.g. S, NP, VP, PP, noun, verb, prep, det) that describes a big subset of a natural language extremely over-generates. But do we need 575 categories as with the grammar of Briscoe and Carroll? A grammar for a probabilistic parser must have enough categories to describe legal syntactical constructions and not (many) more. Information that is irrelevant for this purpose should not be encoded in categories but should be expressed with attributes. On the other hand, one should not introduce categories that are not strictly needed to prevent over-generation. If a grammar over-generates because of too few syntactical categories, some significant differences between syntactical categories are ignored because these categories have been merged into one category. This is not good for the language the grammar describes. It is also not good for the statistical component of the parser, because significant different cases are not distinguished.

[1] At CLIN 1991 in Amsterdam, Pereira suggested a minimal use of rules together with statistical information to be the only solution to the complexity problem of the computational linguistic.

[2] The basic part of a subset of a natural language is the set of common syntactical constructions in that subset.

Attributes can be used for determining features (non-syntactical information) but also for syntactical constructions (syntactical attributes). This has been done in the ROSETTA surface grammars as shown in a previous chapter. In this way the number of grammar rules can be reduced. For a grammar for a probabilistic parser this method is not useful. If attributes are used for syntactical purposes, statistical information has to be collected not only about shift and reduce actions, but about these attributes as well. That would be very complicated, may be it will be impossible. So all important syntactical information has to be expressed in syntactical categories.

Two techniques exist to reduce the number of grammar rules of a grammar without loosing the power of this grammar. The writers of a grammar can introduce auxiliary categories and/or they can use optional symbols. Both techniques have been described in chapter 9. In the next section we will see that the statistical significance is influenced by these reduction techniques.

12.3.2 statistical significance and grammars

For probabilistic parsers of natural language, small grammars are needed, otherwise the size of the train corpus has to be very large. As we have seen in a previous chapter, reduction of grammar size can reduce the significance of the statistical information that can be collected (section 10.4.2). The reduction of statistical significance is caused by a reduction in the amount of available context.

If auxiliary categories are used, the statistical significance will reduce. A sequence of symbols will be replaced by an auxiliary category. For example the rule

A → B C D E

could be replaced by the rules

A → B F E

F → C D

Because the rule F → C D can be used in other rules with the sequence C D in the right-hand-side, a reduction of the rule F → C D does not equal the recognition of symbols C D in the rule A → B C D E. Hence, it could be that the statistical information cannot express the probability that symbols C D reduce to F in the rule A → B F E.

Using optional symbols causes the same effect. An optional symbol category can be seen as an auxiliary category. But there is another negative effect. The rule

A → B C opt_E D F

represents the two rules

A → B C E D F

A → B C D F

It is not possible to make a distinction in the version with optional symbols between the possibility of the reduction of the rule A → B C D F and the rule A → B C E D F in a state of the parser. This problem can also occur with auxiliary categories if they may rewrite to different sequences of symbols.

12.3.3 conclusions

There is a trade-off between grammar size and statistical significance. The statistical significance reduces when the grammar size reduces without loosing power in describing sentence structures. The only way the grammar size can be reduced without loosing power in describing sentence structures, is to take grammar rules together. Thus with reducing the size of a grammar, different structures will be (partly) merged. It is not (always) possible to make a distinction in the statistics between structures that are merged. So the statistical significance will decrease.

The grammar has to describe parse trees. We have seen (section 10.2) that a grammar describing parts of parse trees causes big problems with the size of the analyzed corpus. Context-free grammars are better to work with, but a context-free grammar of a natural language may be still too large. So perhaps there have to be made some reductions on the grammar size. Using auxiliary categories that only rewrite to a single sequence of symbols is a good way to reduce the grammar size. If optional symbols are used, one should not have more than one or two optional symbols in a right-hand-side. Otherwise the statistical significance will strongly decrease.

12.4 probabilistic LR parsers

The context information created in LR(0) parsers highly depends on the form the grammar has been written. LR(0) parsers do not create a lot of context information of grammar rules. So (probably) it will be needed to use look-aheads to add context information. As we have seen in chapter 11, constituent look-aheads are more significant than terminal look-aheads. Thus LR parsers with constituent look-ahead can be better used for probabilistic parsing than LR parsers with terminal look-ahead. The recursive ascent Marcus parser is described in chapter 2. This parser is able to use a lot of context information. That is why the parser size will be very large if a few look-aheads are used. Another possibility is to use the Marcus LR parser, described in chapter 5. Look-aheads are added to a LR(0) parser, so the number of states does not depend on the number of look-aheads. A consequence is that the MaLR parser does not use as much context information as the RAM parser. The only advantage with respect to LALR(k) parsing is the significance of constituent look-aheads for making distinctions between reductions. Subdividing reductions, as described in section 11.3.2, is not possible with MaLR parsing.

The RAM parser is the most appropriate parser for probabilistic LR parsing. If the parser will be too large to handle, the MaLR parser can be used.

12.5 corpora

12.5.1 the need for a large analyzed corpus

The size of the corpus used to train a probabilistic LR parser depends on the number of different actions in the parser. It is hard to say in general how many parser actions will be created for a specific grammar. But for a large grammar the number of different shift and reduce actions will be high. So the parser has to be trained with a lot of sentences.

In the view of some people, the parser could be trained with unsupervised learning. It is not known how good the results are with this method. In practice, training with

unsupervised learning is a difficult task. For a very large set of sentences all parses have to be computed. Because the average sentence length will be more than ten words[3], the sentences will be (very) ambiguous, so there will be a very large set of parses. Because the result of unsupervised training is uncertain and because the result will be inferior to the result with supervised learning, it seems unattractive to try to create a probabilistic parser based on unsupervised learning.

With supervised training, a corpus of analyzed sentences has to be made. This will take a lot of time if a hundreds of thousands analyzed sentences are needed. An approximation of the time at least needed to make a useful analyzed corpus for an experiment with probabilistic parsing is described in the following section.

12.5.2 making a new analyzed corpus

It took one person/month to make the train corpus of Briscoe and Carroll (150 sentences). To get a more realistic approach of the costs of making an analyzed corpus, we asked the corpus linguists at the University of Nijmegen how much time they needed for analyzing sentences.[4]

The corpus linguists at the University of Nijmegen divide the analysis of sentences in three phases. First, a sentence is supplied with lexical tags. For the lexical disambiguation together with a little syntactical analysis, one person has to work ten days to analyze a sample of 20000 words. After lexical disambiguation, the sentences are parsed. Ambiguous parts of the parse trees of a sentence are merged so that in the selection phase of the correct parse tree not all the different trees are shown to the syntactical disambiguator. If the parses of a sentence cannot be computed within one hour CPU time (SUN3 system), the computation is stopped. With fiction (written prose) a lot of sentences can be analyzed in this way. For one sample of 20000 words, the parser needs about six nights to compute the parse trees. To select the correct analysis, a linguist needs ten days to examine one sample. It is very difficult to analyze sentences from scientific articles. The parser needs too much time, due to the long sentences (sometime more than 99 words). Sentences from articles in newspapers etcetera are also more difficult to handle in reasonable time. These data depends on the size of the grammar etcetera, so for other grammars than used in Nijmegen, it could take less time to analyze one sample of 20000 words.

Hence, when a new analyzed corpus has to be made for a large experiment with probabilistic parsers, a lot of work must be done. The parser of Briscoe and Carroll has more than 1,000,000 actions. Suppose, with a minimal grammar for probabilistic parsing, one has a parser with 500,000 actions. Suppose, on average the sentences exists of 20 words. Suppose, on average 20 grammar rules are used to make up a parse tree. Thus there are 20 reduce actions for one sentence and 20,000 reduce actions for one sample of 20,000 words. So at least 25 samples are needed to use every reduction one time. To get significant probabilities of reduce actions, much more samples are needed. Thus one linguist has to work many years before a large analyzed corpus has been made.

[3]Some parts of the LDB corpus have an average sentence length of more than 20 words.

[4]Thanks to Nelleke Oostdijk for her detailed information.

12.6 the trade-off between grammars, parsers and corpora

A grammar has to describe a large subset of a natural language. The grammar can be written in different formalisms. The context-free formalism seems one of the most natural approaches for probabilistic parsing (see section 12.3). The manner the grammar is written influences the size of the grammar and the number of structures that are written out in different grammar rules. This influences the size of the parser and the amount of context information that can be used. In general, larger amounts of context information are better, but with more context information a larger analyzed corpus is needed. So the size of the grammar and the number of look-aheads are restricted by the size of the analyzed corpus.

12.7 a proposal for a decisive experiment with probabilistic parsing

Small experiments with probabilistic parsing do not give us any certainty about the usefulness of statistics in parsing a large subset of a natural language (see 12.1). So we propose to stop with small experiments (except for small domains that exist in real life). We need a large experiment which can give us a decisive answer on the question whether probabilistic parsing is useful or not. A large experiment with use of the recursive ascent Marcus parser is appropriate for this goal.

For the experiment, we need a context-free grammar of a natural language. The grammar must be suitable for probabilistic parsing, so the aspects of grammar writing described in section 12.3 must be taken into account. As far as we know, such a context-free grammar does not exist. Nowadays, grammars are written in other formalisms than the (simple) context-free formalism. In addition, large grammars are rarely written.

An analyzed corpus is also needed for a large experiment. As has been written in section 12.5, a very large analyzed corpus will take a lot of work and thus a lot of money. May be existing corpora can be used, by transforming their analyses to the context-free grammar that will be used in the experiment.

For the parser, we prefer the recursive ascent Marcus parser[5]. Its advantages have been described in chapter 11. With a first experiment one can use one (constituent) look-ahead. If the probabilities of conflicting actions in the parser are not distinctive enough, more look-aheads can be used. Possibly, more look-aheads can be used only for those states that contain conflicting actions without significant different probabilities. With changeable look-ahead, it can be examined whether statistical information really solves a lot of conflicts or whether much more context information is needed. It can be seen whether it is practically realizable to get enough statistical information about the needed context information for successful probabilistic parsing.

[5]If it is impossible to use the RAM parser (because of the size of the parser), the Marcus LR parser can be used. This causes a large reduction of statistical significance.

Appendix A

complexity

grammar	$\mid V_N \mid$	$\mid V_T \mid$	$\#(G)$	$\mid G \mid$	k	#(items)	#(states)	$\frac{\log \frac{\#(items)k}{\mid G \mid}}{\log V_N}$
extracted	44	32	98	237	1	237	207	0
					2	1191	1300	0.61
optional	76	70	168	216	1	216	173	0
					2	935	941	0.50
					3	?	> 3200	?
Tomita II	13	9	43	90	1	90	65	0
					2	432	662	0.88
					3	1253	2116	1.46

The extracted grammar is extracted from 8 sentences out of the LDB-corpus. The optional grammar is a part of the surface grammar of the ROSETTA system. The Tomita II grammar is the second example grammar described in [Tomita, 1986].

Appendix B

implementation of the recursive ascent Marcus parser

B.1 overview

The implementation of the recursive ascent Marcus parser is not made with use of a functional programming language. Such an implementation would not be efficient. We use the compiler-generator system Elegant ([Augusteijn, 1990]) for our implementation. On the base of a grammar of context-free grammars, a syntax-checker of context-free grammars has been made by Elegant. This grammar of context-free (and attribute-free) grammars looks like:

```
Grammar      ::=  'TERMINAL' { Terminal }
                  'GRAMMAR' { GrammarRule } .
GrammarRule  ::=  Ident '->' { Symbol } '.' .
Symbol       ::=  Ident .
Terminal     ::=  Ident .
```

With use of the attribute evaluation system and functions written in the languages Elegant/CodeGen of the Elegant system, we have developed a program that has as input a context-free and attribute-free grammar. Data that represent the states of the related parser and their relations (the transitions), together with functions $[q]$, $\overline{[q]}$ and $goto(q, \delta)$ for every state q (written in the programming language C) are output of the program. Other functions needed for the Marcus recognizer that not depend on the grammar are also written in C. A Marcus recognizer with $k-1$ constituent look-aheads for a context-free and attribute-free grammar can be generated by compiling the C-code.

Parse trees in the format of the LDB-corpus are input of the recognizer. They look like:

```
( SF ( NO ( "HELENA" ) , VAPF ( "WAS" ) , VBKG ( "TRYING" ) , SN ( TO (
"TO" ) , VBKI ( "STOP" ) , VITG ( "SHIVERING." ) ) ) )
```

The recognizer will only use the terminal categories, so the actual input of the recognizer is

```
NO VAPF VBKG TO VBKI VITG
```

In a special mode, the recognizer can use the tree information to speed up the recognizing process (see section B.3).

B.2 a small reduction

In the implementation of the recognizer, we have made a small reduction in the number of computations. This reduction reduces the number of parse trees that are delivered. We describe this reduction in this section.

In the definition of *goto*

$$goto(q,\delta) = \{\gamma \rightarrow \alpha\lambda\delta.\beta \mid \gamma \rightarrow \alpha.\lambda\delta\beta \in (q \cup ini(q)) \wedge \delta = k{:}\delta\beta \wedge \lambda \rightarrow^* \epsilon\}$$

and in the definitions of the functions $\overline{[q]}$

$$\begin{aligned}\overline{[q]}(\delta,i) = {} & \{(\gamma \rightarrow \alpha.\lambda\delta\mu, j) \mid \gamma \rightarrow \alpha.\lambda\delta\mu \in q \wedge \lambda \rightarrow^* \epsilon \\ & \qquad \wedge (\gamma \rightarrow \alpha\lambda\delta.\mu, j) \in [goto(q,\delta)](i)\} \\ & \cup \{(\gamma \rightarrow \alpha.\beta, j) \mid \exists_{A\xi\lambda\delta\nu\ell}((\gamma \rightarrow \alpha.\beta, j) \in \overline{[q]}(A\xi,\ell) \\ & \qquad \wedge A\xi \rightarrow .\lambda X\nu\xi \in ini(q) \wedge k : X\nu\xi = \delta \wedge \lambda \rightarrow^* \epsilon \\ & \qquad \wedge (A\xi \rightarrow \lambda\delta.(X\nu\xi : k),\ell) \in [goto(q,\delta)](i))\}\end{aligned}$$

for an item of the form $\gamma \rightarrow \alpha.\lambda\delta\beta$ with $\lambda \rightarrow^* \epsilon$ following situation is possible:

$$\begin{aligned}&(\lambda_1\delta\mu\nu \equiv \lambda_2\delta\nu \equiv \lambda\delta\beta) \\ &\wedge\ \lambda_1 \rightarrow^* \epsilon \\ &\wedge\ \lambda_2 \rightarrow^* \epsilon \\ &\wedge\ |\lambda_2| > |\lambda_1|\end{aligned}$$

In the right-hand-side after the dot of the item the parser founds in more than one place a δ with a preceding ϵ-rewriting. If all the parse-trees for a sentence have to be delivered it is not possible to reduce the number of computations. But for robust parsing probably there is no fundamental difference between the trees:

It can be proved that in this case μ can always rewrite to the empty string. If the parser in the definitions of *goto* and $\overline{[q]}$ only looks for $\gamma \rightarrow \alpha.\lambda\delta\beta$, $\lambda \rightarrow^* \epsilon$ and $|\lambda|$ minimal, as in our implementation, only the parse tree with δ as left as possible in the right-hand-side after the dot will remain. The number of ambiguous trees of a sentence reduces, but the parser still can parse the same set of sentences as without this reduction.

B.3 recognizing with tree information

The recognizer can use the tree information of the input to speed up the recognition process in the case no look-aheads are used (with look-aheads it will be more difficult

to use this information). If the recognizer uses this information, for the input showed in figure B.1 the recognizer skips recognitions with for example the rewriting

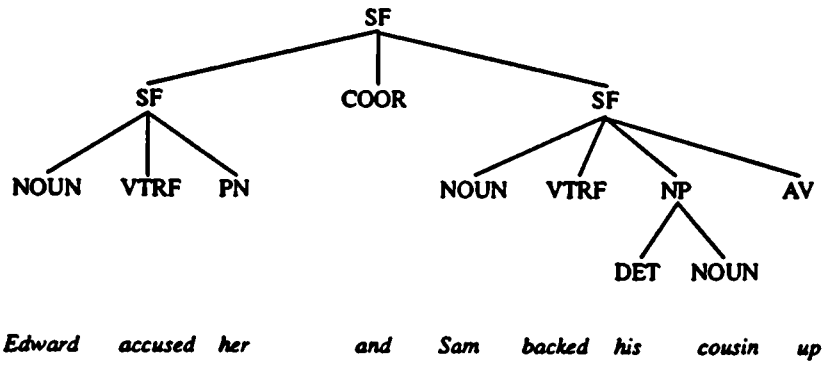


Figure B.1: input for the recognizer with tree information

NonTerminal $\rightarrow^*$ *accused her and Sam*

Supposition with this reduction is that the structure of the parse tree of a sentence with grammar A equals the structure of the parse tree of the same sentence with grammar B.

B.4 efficiency

It has not been tried to make an efficient implementation of the Marcus recognizer. Only the administration of items and lists of symbols has been made efficient, because this has a great influence on the number of computations. List of symbols are hashed on the first symbol of the lists. Thus a list of lists of symbols is connected with every symbol S (see figure B.2). Symbol S is the head of all these lists of symbols. In this way the

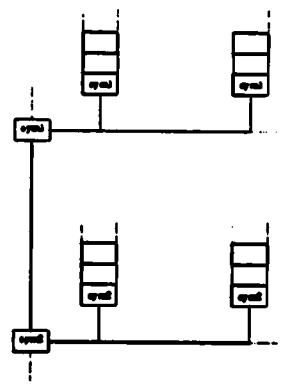


Figure B.2: symbol list hash administration

concatenation of a symbol with a list is easy to implement. Every list is unique. Hence, to equate two lists of symbols, only the pointers that point to these lists need to be compared.

Items are hashed in the same way.

Bibliography

[op den Akker, 1988] H.J.A. op den Akker. *Parsing Attribute Grammars.* Phd. Thesis. University of Twente.

[Augusteijn, 1990] L. Augusteijn. *The Elegant Compiler Generator System.* Attribute Grammars and their Applications. Eds. P. Deransart & M. Jourdan. Lecture Notes in Computer Science 461. Berlin. 238-254.

[Bod, 1992] R. Bod. *A Computational Model of Language Performance: Data Oriented Parsing.* COLING 1992. Nantes.

[Brill et al., 1990] E. Brill, D. Magerman, M. Marcus & B. Santorini. *Deducing Linguistic Structure from the Statistics of Large Corpora.* Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley. Pennsylvania. 275-281.

[Briscoe & Carroll, 1991] T. Briscoe & J. Carroll. *Generalised Probabilistic LR Parsing of Natural Language (Corpora) with Unification-based Grammars.* University of Cambridge, Technical Report No. 224, June 1991.

[Chitrao & Grishman, 1990] M.V. Chitrao & R. Grishman. *Statistical Parsing of Messages.* Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley. Pennsylvania. 263-266.

[Fujisaki, 1984] T. Fujisaki. *A Stochastic Approach to Sentence Parsing.* COLING 1984. California, July 1984. 16-19.

[Fujisaki et al., 1989] T. Fujisaki, F. Jelinek, J. Cocke, E. Black & T. Nishino. *A Probabilistic Method for Sentence Disambiguation.* Proceedings of the First International Workshop on Parsing Technologies. IWPT89. Pittsburgh, August 1989. 85-94.

[Gazdar et al., 1985] G. Gazdar, E. Klein, G.K. Pullum & I.A. Sag. *Generalized Phrase Structure Grammar.* Harvard University Press. Cambridge, Mass.

[Grishman et al., 1992] R. Grishman, C. Macleod & J. Sterling. *Evaluating Parsing Strategies Using Standardized Parse Files.* Proceedings

of the Third Conference on Applied Natural Language Processing. ACL. Trento, April 1992. 156-161.

[Halteren & Heuvel, 1990] H. van Halteren & T. van den Heuvel. *Linguistic Exploitation of Syntactic Databases, the use of the Nijmegen Linguistic DataBase program.* Amsterdam - Atlanta, GA 1990, the Netherlands.

[Heidorn et al., 1982] G.E. Heidorn, K. Jensen, L.A. Miller, R.J. Byrd & M.S. Chodorow. *The EPISTLE Text-critiquing System.* IBM Systems Journal, vol. 21, no. 3. 305-326.

[Horspool, 1991] R. Nigel Horspool. *Recursive Ascent-Descent Parsers.* Lecture Notes in Computer Science 477. Berlin. 1-10.

[Jelinek et al., 1991] F. Jelinek & J.D. Lafferty. *Computation of the Probability of Initial Substring Generation by Stochastic Context-Free Grammars.* Computational Linguistics, vol. 17, no. 3. 315-324.

[Johnson, 1989] M. Johnson. *The Computational Complexity of Tomita's Algorithm.* Proceedings of the First International Workshop on Parsing Technologies. IWPT89. Pittsburgh, August 1989. 203-207.

[Kruseman Aretz, 1988] F.E.J. Kruseman Aretz. *On a Recursive Ascent Parser.* Information Processing Letters 29, 201-206.

[Leermakers, 1991] M.C.J. Leermakers. *The Robust Parser Project.* Philips Research Laboratories. Internal Note.

[Leermakers, 1991a] M.C.J. Leermakers. *Recursive Ascent Parsing.* Proceedings of the First Twente Workshop on Language Technology: Tomita's Algorithm, Extensions and Applications. Memoranda Informatica 91-68. R. Heemels, A. Nijholt & K. Sikkel (eds.). 9-20.

[Leermakers, 1992] M.C.J. Leermakers. *Mathematics of Parsing.* To appear in 1992 or 1993.

[Leermakers, 1993] M.C.J. Leermakers. *Recursive Ascent Marcus Parsers.* Theoretical Computer Science 106. To appear.

[Magerman et al., 1991] D.M. Magerman & M.P. Marcus. *Pearl: A Probabilistic Chart Parser.* Proceedings of the Second Internatinal Workshop on Parsing Technologies. IWPT91. Cancun, February 1991. 193-199.

[Marcus, 1980] M.P. Marcus. *A Theory of Syntactic Recognition of Natural Language.* Cambridge MA: MIT Press.

[Ng & Tomita, 1991] S. Ng & M. Tomita. *Probabilistic LR Parsing for General Context-Free Grammars.* Proceedings of the Second International Workshop on Parsing Technologies. IWPT91. Cancun, February 1991. 154-163.

[Richardson et al., 1988] S.D. Richardson & L.C. Braden-Harder. *The Experience of Developing a Large-scale Natural Language Text Processing System: CRITIQUE.* Proceedings of the Second Conference on Applied Natural Language Processing. ACC. February 1988. 195-202.

[Robinson, 1982] J.J. Robinson. *DIAGRAM: A Grammar for Dialogues.* Communications of the ACM, vol. 25, no. 1. 27-47.

[Sager, 1981] N. Sager. *Natural Language Information Processing, a Computer Grammar of English and Its Applications.* Addison-Wesley Publishing Company, Inc. Massachusetts.

[Sharman et al., 1990] R.A. Sharman, F. Jelinek & R. Mercer. *Generating a Grammar for Statistical Training.* Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley. Pennsylvania. 267-274.

[Shieber, 1986] S.M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes 4. Stanford University.

[Tomita, 1986] M. Tomita. *Efficient Parsing for Natural Language, a fast algorithm for practical systems.* Kluwer Academic Publishers.

[Ukkonen, 1983] E. Ukkonen. *Upper Bounds on the Size of LR(k) Parsers* University of Helsinki, Department of Computer Science. Report C-1983-11.

[Wright et al., 1991] J. Wright & A. Wrigley. *Adaptive Probabilistic Generalized LR Parsing.* Proceedings of the Second Internatinal Workshop on Parsing Technologies. IWPT91. Cancun, February 1991. 100-109.