Error Recovery in a Natural-Language Parser

F.J. de Wilde

Department of Computer Science,
Vrije Universiteit, Amsterdam

1. Introduction

1.1. Outline of the project

The goal of the project was to create an error recovery module as a part of a much larger project currently being developed at The Computer Science Department of the *Vrije Universiteit* in Amsterdam. The subject of this larger project is to implement an **Automatic proof-reader** for any natural-language. The task of an automatic proof-reader is to detect errors in a piece of text of a natural language and then possibly automatically correct them.

So far, already some parts of the project have been written and currently we are able to define a grammar for any natural-language as well as a vocabulary and, given these definitions, test some input for syntactical correctness through the use of a natural-language parser. Also, a program exists, a spelling corrector, which tests separate words of the input for lexical correctness, for example to correct errors made at the typewriter. In the future, tests for contextual correctness may also be included. It is thus already possible to perform (syntactical) *error-detection* on a sentence through the use of the above parser and the definition of the natural-language. After having found a sentence syntactically incorrect, the current parser program just prints a message that it can not find a parsing for the sentence. It never points out what error was made in the sentence. For a proof-reading program to be automatic, we also need to be able to perform (syntactical) *error-correction* on an incorrect sentence. We want, having found a sentence incorrect, to specify where in the sentence a syntactical error was made and possibly physically correct the sentence. The error recovery module tries to correct errors on the syntactical level, but as we shall see later, can also provide the spelling corrector with powerful information. Fig. 1.1 gives an overview of the parts of the project which are already implemented or still in progress.

The justification for this project is quite obvious: correcting large texts by hand is slow and error-prone. Not to mention the fact that proof-reading texts can be boring to a non-interested reader. However, since at this moment we can only correct text up to and including the syntactical level (still with a lot of restrictions), the human proof-reader will be on the job for quite some time.

For now, we are only interested in sentences which are correct on the syntactical level and, as a second restriction, only syntactical correctness of one sentence at the time is tested. This means that a sentence like:

```
The triangle plays the violins.
```

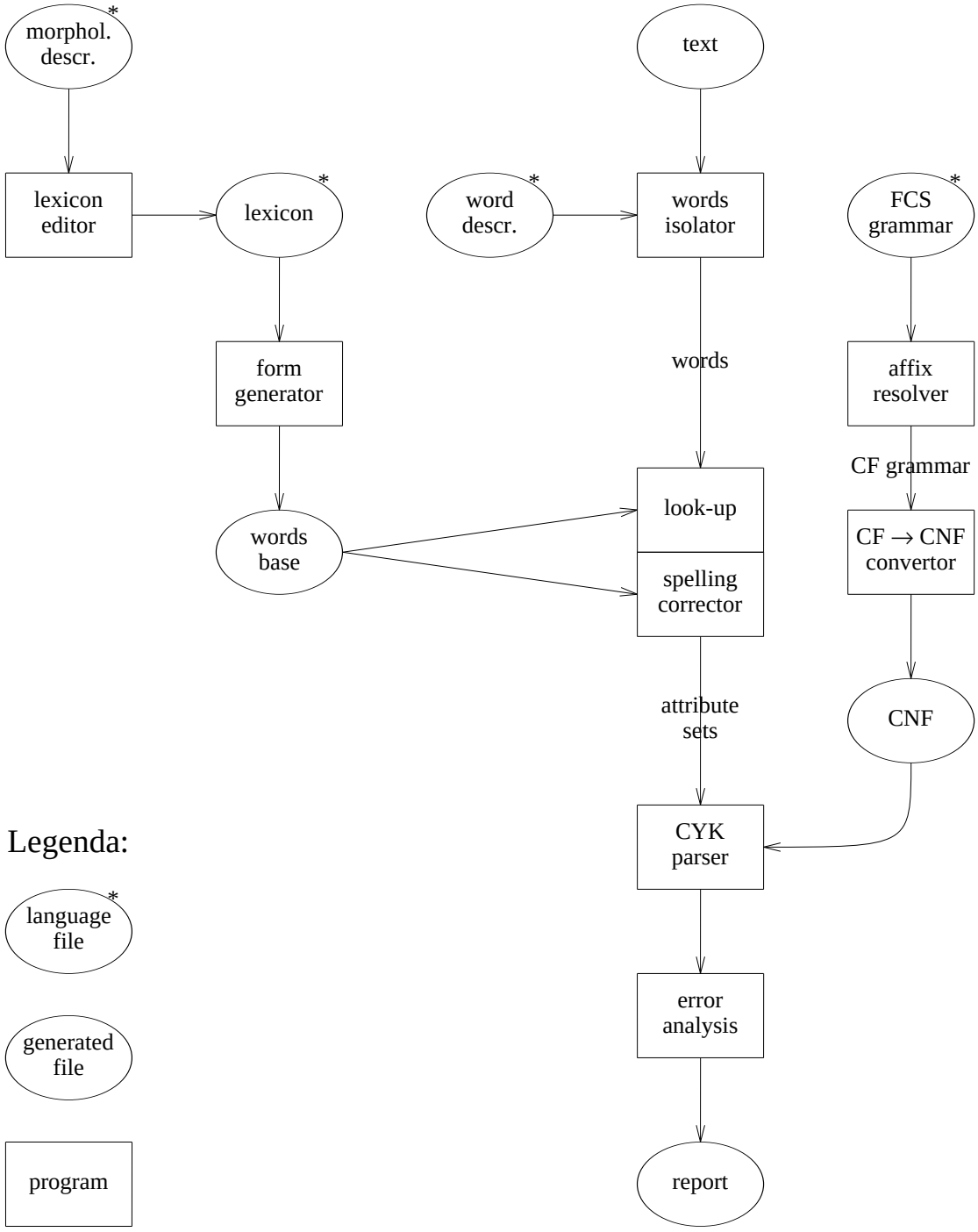
is a correct English sentence as far as we are concerned. We want our error recovery module to be able to correct sentences which lack this syntactical correctness. For example, consider the incorrect sentence:

```
The triangle the violins.
```

We would like to see that the program signals the absence of a verb after the word triangle or changes the second "the" in the sentence to an appropriate verb (e.g. "likes" in the case where the triangle is used to eat violins for breakfast).

In general we may correct a sentence by deleting, replacing or inserting one or more

Automatic Proof-reader



FCS: Finite Context-Sensitive
CF: Context-Free
CNF: Chomsky Normal Form

Figure 1.1: *Overview of the project (by Dick Grune).*

words. The problem is that we want to find the smallest of the above corrections of a sentence that will turn an incorrect sentence into a correct one, and then list all possibilities. At this moment it is still up to the user to pick out a corrected sentence and it is impossible that the writer of a corrected piece of text can ever be ruled out. As an example, consider once again the sentence that was given above. Obviously, there are many verbs that the second "the" may be changed into to form a correct sentence. The program will never be sure about the intentions of the user so the best thing it can do here is to mention to the user that he should change the second "the" into a transitive verb.

This example shows that the best we can do is to generate types of words in a correction rather than a single word, unless there is only one possibility. Fortunately the whole definition of the grammar for the natural language as well as the vocabulary hinge on types of words, as we shall see in the next section, where we will describe the mechanism for defining a language in more detail.

1.2. Some details of the natural-language definition

In Fig. 1.1 the right branch of the diagram illustrates the way a grammar for a natural-language is processed. The definition of the natural-language is given by a user-supplied Two-Level or Affix grammar which allows limited context-sensitivity to be defined through the use of affixes. That is why it is marked FCS (Finite Context-Sensitive grammar). For a complete description of the affix grammar used (and for more information about the natural-language parser in general) we refer to the report of [Altena and Ott de Vries 87]. In fact, we strongly advise to read their paper first, since the error recovery module is an extension to their implementation of the CYK-parser(Cocke, Younger and Kasami [Younger 67]). Here we restrict ourselves to a general description of it.

The need to be able to define some level of context-sensitivity in the definition of a natural-language can be shown by the following example of a correct English sentence:

```
John eats peanuts.
```

The form of the verb depends on the subject of this sentence. We will not go further through the details of how this context-sensitivity is properly defined in a definition for a natural-language; rather we will describe the general form of the grammar. If context-sensitivity is ignored, a grammar defining the English language may contain rules like:

```
Sentence:
    Subject <verb> Object <dot>

Subject:
    <noun>
Object:
    <noun>
```

That is, instead of identifying the form of a sentence by distinct words, we define its form by a certain order of **types** of words, which we call **attributes**. The form of a sentence derived

from this grammar is:

```
Sentence:
    <noun> <verb> <noun> <dot>
```

The given affix-grammar is processed and rewritten to an equivalent Context-Free Grammar(CFG), which then is transformed to a CFG in Chomsky Normal-Form(CNF), being the suitable form we need in the core of the natural-language parser, which is formed by a CYK-parser. The CFG conserves the way a certain sentence is produced from the start-symbol. Also, the resulting CFG in CNF carries information about the original structure of the productions in the CFG from which it was derived. What is important to us is that a definition of the grammar of a natural language consists of rules that are build of types of words and that each such grammar can be rewritten to a CFG. Thus, instead of a grammar which generates distinct words, its terminals consist of attributes. The definitions of a CFG will be repeated at the start of chapter 2, as well as the definition of CNF.

Apart from the grammar, the natural-language definition also consists of a vocabulary of legal words in this language. Each of this words carrys a set of attributes describing the types belonging to this specific word. For example, the English word "hand" may have two attributes:

```
hand : <verb>, <noun>
```

For each legal word defined in the vocabulary, there is a set of attributes describing the functions this word may fulfill in a sentence. It is up to the CYK-parsing algorithm (also to be described in chapter 2) to find a sentence derived from the CFG which matches the attributes of a given line of input. The derivation from the start-symbol and the attributes of a certain sentence are shown in Fig. 1.2. In this case a match of the sentence exists.

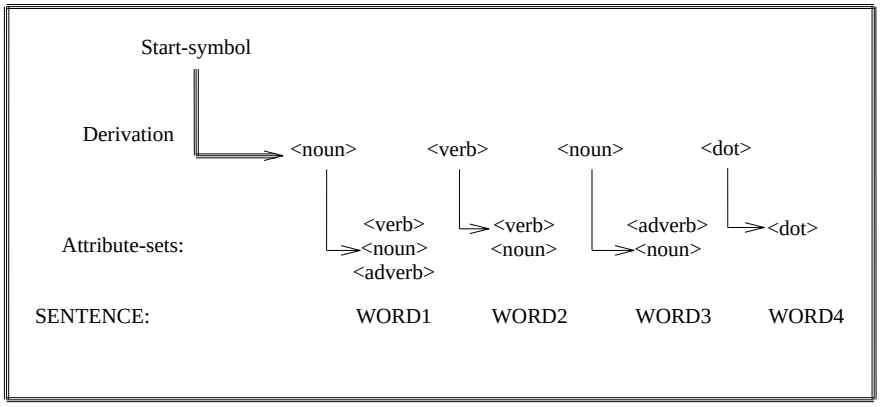


Figure 1.2: *Matching of attributes of words in a sentence.*

The parser will then, as a second step, list the productions used to match such a sentence in a format which gives a clear view of the way the sentence was derived (it produces a **parsing**). However, we do not necessarily need such a listing in proofreading a piece of text, because we do not have to make changes in a correct sentence.

In a natural language, unlike in most programming languages, some sentences are *ambiguous*, which means they can be parsed in more than one way. An example of an (syntactically) ambiguous English sentence is:

```
The green flies like birds.
```

which could either denote the fact that the green flies around like birds do or that a swarm of green flies are particularly fond of birds. The CYK parsing algorithm can deal with any CFG (a lot of parsers can only deal with CFG's that are restricted in some way), so in particular ambiguous sentences are no problem: the parser is capable of matching every legal sentence with the attributes of the input.

Clearly, the set of attributes generated by the grammar must be the same as the one used for defining attributes for words in the vocabulary. All of this is not of our concern, since programs exist that take care of this. The error recovery module can also safely assume that every word in the input is a legal word in the natural-language, for this should have been taken care of by the spelling corrector program, which in the future processes a sentence before syntactical correctness of the sentence is tested. Only if no match of the attributes of the words in a sentence is found, we should use the definition of the grammar to change the attributes of some words in a sentence, or delete or insert a new set of attributes between two words, while keeping in mind we should use as few corrections as possible.

The algorithm which is capable of correcting sentences is the main subject of this paper. In fact it is an extension to the CYK-parsing algorithm and all of its details will be described in chapter 4. Chapter 3 contains some solutions to the error recovery problem which are shown to be less suitable, as well as a justification for the error recovery algorithm actually used. Chapter 5 lists some modifications to the program, which could be implemented in the future.

2. Context-free grammars and the CYK-algorithm

2.1. Definition of CFG and CNF

A context-free grammar **G** is a 4-tuple **(N, T, P, S)** where:

- **N** is a set of *nonterminal* symbols, **A, B, ...** .
- **T** is a set of *terminal* symbols, **a, b, ...**, disjunct from **N**, which form an alphabet Σ.
- **P** is a set of *productions*, **p, q, ...** of the form $\mathbf{A}\rightarrow\gamma$, with $\mathbf{A}\in\mathbf{N}$ and γ a string consisting of elements of **T** and **N**.
- **S** is a special element of **N**, called the *start-symbol*.

This definitions together define a grammar which can be used to generate a certain class of languages, called the *context-free* languages (see [Hopcroft & Ullman 79], the part on the Chomsky-hierarchy).

Now this formalism can be used to generate a language as follows: Let α, β and γ be strings of grammar symbols that is, strings consisting of elements of **N** and **T**. We say $\alpha\Rightarrow\beta$, if there exists a nonterminal **A**, which is in α, and a production $\mathbf{A}\rightarrow\gamma$ and the string β is equal to α with one occurrence of **A** replaced by the left-hand side of the production. We call this one *step*. The process could be repeated for β if it still has nonterminals left, otherwise we end up with a string consisting of terminal symbols only. We also define $\alpha\Rightarrow*\beta$ to denote that the string of grammar symbols β can be derived from α by zero or more steps.

The *language* **L(G)** defined by the CFG **G** is defined as:

$$\mathbf{L(G)} = \{w \mid w\in \Sigma^* \wedge \mathbf{S}\Rightarrow*w\}$$

Thus **L(G)** consists of strings of terminals that can be derived from the start-symbol by zero or more steps. It is also legal for a production to generate the empty word, ε. The production is of the form $\mathbf{A}\rightarrow\varepsilon$ and in this case it is also possible that **L(G)** contains the empty word.

It has been proven (see [Hopcroft & Ullmann 79]) that any CFG **G** that does not generate the empty word can be rewritten to an equivalent CFG in Chomsky Normal-Form (CNF), in which **P** only contains productions of the following form:

- Either the production is of the form: $\mathbf{A}\rightarrow\mathbf{BC}$, with **A**, **B** and $\mathbf{C}\in\mathbf{N}$,
- Or the production has the form: $\mathbf{A}\rightarrow\mathbf{a}$, with $\mathbf{A}\in\mathbf{N}$ and $\mathbf{a}\in\mathbf{T}$.

If a production is of the first type, it is called a *nonterminal Chomsky production*, else it is called a *terminal Chomsky production*. The set of productions of a CFG in CNF is split in two by this discrimination. From now on we refer to the set of terminal Chomsky productions as $\mathbf{P_T}$ and the set of nonterminal Chomksy productions is denoted by $\mathbf{P_N}$. The CYK-parsing algorithm uses this rewritten form of a CFG to test whether a given sentence is in the language of the given grammar and to generate a parsing of it.

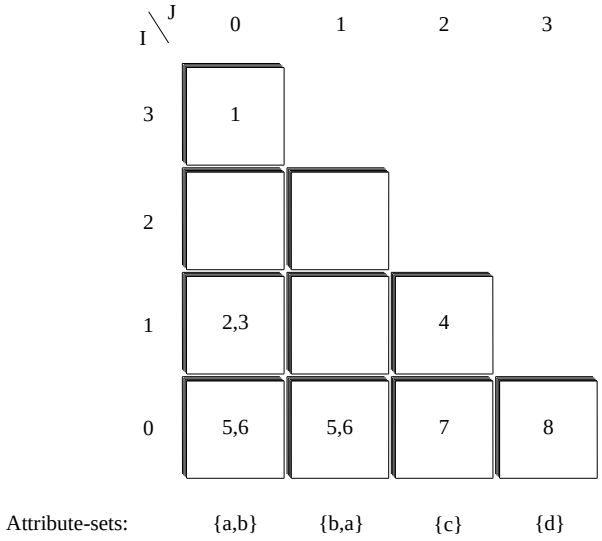
We repeated this definitions here because, as stated earlier, the natural language

definition inputed to the CYK-parser is rewritten to a CFG (and then to CNF) first. The terminal symbols of the grammar in CNF are attributes of the natural-language definition, so the words generated by this grammar are sentences constituted of attributes. We are now ready to describe the CYK-algorithm.

2.2. The CYK-parser

2.2.1. Filling the P-triangle

The first thing that is being done, when a sentence of input is given in the natural-language is the evaluation of each word's attributes. Assume the sentence consists of *n* words, then we end up with *n* attribute-sets. The actual algorithm is presented in the form given by [Hopcroft &Ullman 79]. To test whether a matching of this attribute-sets exists, a triangular structure of size $n \times n$ is used, as shown in Fig. 2.1 for the case of $n=4$.



PRODUCTIONS OF G_1:

1: S→EF

2: E→AB

3: E→BA

4: F→CD

5: A→a

6: B→b

7: C→c

8: D→d

Figure 2.1: *The P-triangle, filled for* G_1.

The productions of CFG $\mathbf{G_1}$=({S,A,B,C,D,E,F},{a,b,c,d},P,S) are defined in the figure and for convenience every production is numbered. Each cell in the triangle is indexed, with (0,0) being the index of the bottom-left cell and (*n*-1,0) being the index of the top-left cell. Each cell can contain a set, consisting of zero or more CNF productions of the natural-language definition. That is why we will refer to it as the P-triangle from now on. Beneath the bottom-row the sets of attributes of words are written in the order they appear in the sentence. From here we refer to the attribute-sets of the words in the sentence by ATT(0), ATT(1), ... , ATT(*n*-1), where ATT(0) corresponds to the attribute-set of the first word,

ATT(1) to the second and so on.

The cells of the triangle are filled by the algorithm below:

(1) For each cell j in the zeroth row, with its attribute-set, ATT(j), corresponding to (j+1)-th word in the original sentence, do the following:
If there is an attribute **a** in the attribute-set, and there exists a $\mathbf{p} \in \mathbf{P_T}$, such that the attribute on its right-hand side is **a**, then put this production in the cell.

(2) For i = 1 to n-1 ,
For j = 0 to n-i-1,
Let C(i, j) be a certain cell of the triangle.
For k = 0 to i - 1,
Let $\mathbf{p} \in \mathbf{P_N}$ be of the form $\mathbf{A} \rightarrow \mathbf{BC}$. If there is a production **q** in C(k,j) whose left-hand side is **A** and there is a production **r** in C(i-k-1,j+k+1) whose left-hand side is **B**, then add **p** to C(i,j).

Stated less formally, when a production **p** is added to C(i,j), we have matched the attribute-sets of a subsentence of length i starting at position j (call these attribute-sets of the subsentence l(i,j), a list which preserves the order of the sets ATT(j),ATT(j+1),...,ATT(j+i-1)) in the original sentence by some derivation from the left-hand side of production **p**. We say $\mathbf{p} \Rightarrow_{match} l(i,j)$ if this is the case. We find such productions by looking at all the productions **q** in C(0,j) and all productions **r** in C(i-1,j+1). Every two productions, one from each of these cells, deliver us two left-hand sides, say **B** and **C**. If there exists a production **p** of the form $\mathbf{A} \rightarrow \mathbf{BC}$, then we know that $\mathbf{p} \Rightarrow_{match} l(i,j)$ and **p** is added to C(i,j). We then repeat this for C(1,j) and C(i-2,j+2), each containing productions capable of matching sentences of length 2 starting at position j and sentences of length i-2 starting at position j+2 etc..

After termination of the algorithm, C(n-1,0) contains all productions, whose nonterminal on the left-hand side can derive a matching of the attribute-sets of the whole sentence. Thus for every production **p** in C(n-1,0) it holds that $\mathbf{p} \Rightarrow_{match} l(n,0)$. If there exists a production in C(n-1,0) with the start-symbol **S** on the left-hand side, then we find the original sentence correct (in syntactical sense), since there exists some derivation from the start-symbol in the grammar that matches the attribute-sets of the input sentence.

Step (1) of the algorithm takes O(n) operations. In step (2), i, j and k are all bounded by n, so the complexity of the algorithm is O(n^3). Actually, we should also account for the search through the productions in step (2), but this search is bounded by $|\mathbf{P}|^2$. This boundary can be achieved by filling a table of $|\mathbf{N}| \times |\mathbf{N}|$, indexed by each element of **N**, with productions of $\mathbf{P_N}$ at each entry (**A**,**B**), whose right-hand side consists of **AB**. Since every cell in the triangle can contain at most $|\mathbf{P}|$ productions, and since every production pair from two cells has to be considered once, the boundary is achieved. We will not take these steps in account in the complexity of the algorithm, because $|\mathbf{P}|$ is constant throughout each run of the algorithm with the same grammar. The example in Fig.2.1 illustrates the contents of the P-triangle after termination of the algorithm, for the given grammar and attribute-sets of a certain input. The productions are identified by a number for convenience.

In the actual implementation of the algorithm, an optimization is implemented, which uses an N-triangle, filled with elements of **N**, which are the left-hand side of productions of

the corresponding cell in the P-triangle. To see why an optimization is achieved, consider cell C(0,1) in Fig. 2.1. It now contains 2 productions, which have the same left-hand side. When filling cell C(0,3), we test twice, while no new production is added, other than 1 (remember that the cells contains sets, so no element appears twice). If we use the N-triangle to fill cell C(3,0), only one comparison is needed, because only nonterminal **E** will appear in cell C(1,0). The resulting N-triangle is shown in Fig.2.2.

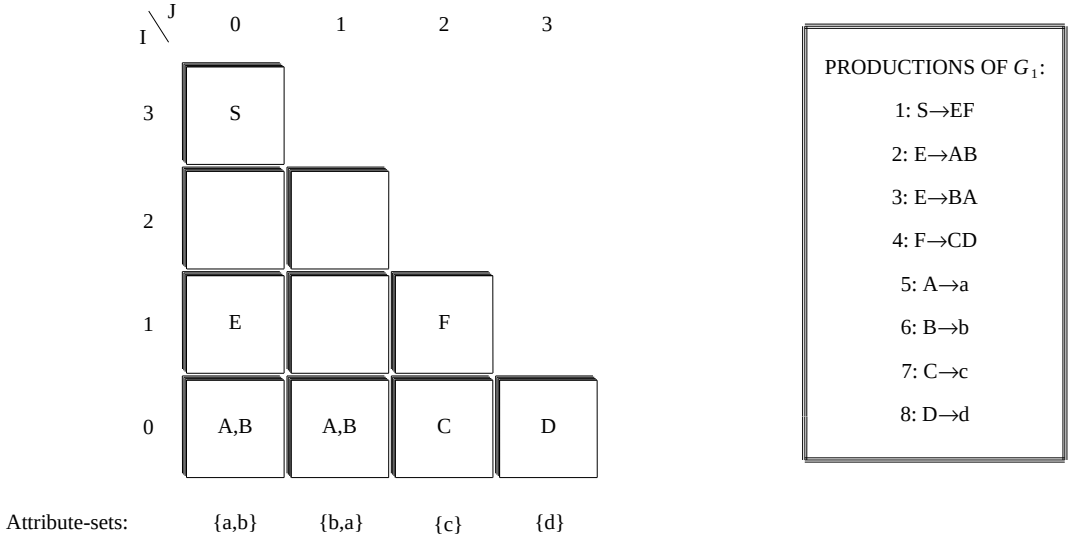


Figure 2.2: *The N-triangle, filled for* G_1.

This optimization is also used in the recovery algorithm. For the grammar used in testing the algorithm the ratio of productions and left-handsides is roughly 3/1. So the lookup time is about nine times shorter than in the original algorithm.

2.2.2. Parsing of a sentence in the natural language

By a *parsing* of a sentence (word) in a language **L(G)** of a CFG **G**, we mean a specification of the way this sentence was derived from the start-symbol by listing all productions used in some well defined order. In this context, we first define a *leftmost derivation* of some sentence by a CFG, if for two strings of grammar-symbols, α and β, which appear in this derivation as $\alpha \Rightarrow \beta$, always the leftmost nonterminal in α is rewritten by an appropriate production. The language **L(G)** is said to be *ambiguous* if there exists a sentence in **L(G)**, which has more than one leftmost derivation.

The CYK-parser produces all leftmost derivations of a matching of a sentence in the natural-language. Its output is a stack of productions, filled in the order each production was

used to derive the matching. This order is extracted from the P-triangle, by running the CYK-algorithm in the opposite direction. There can be several derivations due to two reasons:

(1) There is more than one sentence, derived from the grammar, which matches the attribute-sets.

(2) A sentence derived from the grammar that matches the attribute-sets has more than one leftmost derivation. In this last case the sentence in the natural language is ambiguous in syntactical sense.

We will not describe the algorithm, because it is already documented in [Altena & Ott de Vries 87] and because no parsing will be generated for a corrected sentence. What is important at this point is that if we have a stack of productions, we can feed it to a module, called *WegTerug*, part of the CYK-parser, which will produce a parsing of the sentence in the original CFG from which the sentence was derived.

3. Considerations in error recovery in the CYK-parser

3.1. Corrections and constraints on error recovery

3.1.1. Types of corrections

There are three types of corrections that can be made to an incorrect sentence in the natural-language:

(1) Deleting a word of the input sentence.

(2) Inserting some word in the sentence.

(3) Replacing a word in the sentence with another word. Actually this is equal to an application of (1) and (2), which would then count for two errors, however, where the replacement counts for one.

We are interested in developing an algorithm which corrects sentences in the sense described by these three elementary operations and which can be implemented as a smooth extension to the CYK-algorithm.

So far, not much work is done on error recovery in general-CFG parsers. There are some articles about error recovery in this kind of parsers, for example, [Peterson &Aho 72] and [Lyon 74], but they describe error recovery in the Earley-parser. This parser was not used in the proofreader-project because of its complexity in the first place, so we cannot directly use any of these solutions. However, they all make the above distinction between corrections and, as such, give some idea of the problems encountered.

3.1.2. Constraints on the algorithm

We needed to invent an algorithm on our own, which matched the following constraints:

(a) The algorithm should always find the minimum number of corrections necessary to correct a given incorrect sentence.

(b) The algorithm should have an acceptable computational complexity for a given sentence of input of length n. We mean by this that we do not want an algorithm of $O(a^n)$ or of $O(n^b)$, with $a > 1$ and b too large.

(c) The algorithm should make use of the structure of the CYK-algorithm, without imposing extra time on the parsing of correct sentences. This means that we do not want the error recovery extension to the parser to slow down the process of parsing, if it will turn out that there is nothing to recover. We would like to see that the recovery-algorithm

uses the structures created in the P-triangle and N-triangle in some way, because these already provide some information about nonterminal symbols and productions, which are able to derive part of the sentence.

In the next section, we will describe two approaches, which looked easy to implement and smart respectively, but which did not match the constraints in the end.

3.2. Attempts that failed

3.2.1. Changing the input sentence

As a first attempt in trying to correct a sentence, we use a built-in feature of the CYK-parser. Whenever the parser spots the word "*" in the input sentence it assigns to this special word's attribute-set every attribute that exists in the definition of the natural-language. This feature could, for example, be used to find out how many (types of) sentences exist in the language of a certain length. In order to test how many types of sentences of length 4 can be matched, just feed the parser with the sentence:

```
Sentence:    "*"    "*"    "*"    "*"
```

We use it to correct sentences (of a given length n) the following way:
Let the *length* of a correction of a sentence be defined by the number of elementary operations applied to a sentence. Given a sentence for which no matching is found, delete each word from the sentence once, rerun the parser with this new input, and see if a correct sentence is formed. Add back the word, delete the next one and try again. This generates corrections of length 1 of the "delete"-type. Next, replace each word once by the special word "*", and rerun the algorithm with this changed sentence and see if a matching exists. Add back the word and replace the next one. This generates corrections of length one of the "replace"-type. At last, fill in the word "*" before the first word, rerun the algorithm, delete the "*", fill it in before the second word etc.. This generates corrections of length one of the "insertion"-type. The procedure is illustrated in Fig. 3.1.

We could repeat the procedure for corrections of greater length, by looking at every possibility of mixing several types of corrections at certain locations in the sentences (Note that we should not put a "delete"-correction and an "insert"-correction next to each other).

The complexity of each single run is $O(n^3)$. If we would only consider corrections of the insertion type, then, by the time we try to correct the sentence by inserting, say, c corrections, we would have to make a run for each possibility of inserting c "*" 's in the sentence. Because there are $O(n^c)$ possibilities of doing so, the algorithm would already be of $O(n^{3+c})$, which is a quite undesirable computational complexity for an algorithm. The number of runs would only increase, if we should mix different types of errors as well, so this algorithm is rejected, because it severly violates constraint (b).

Original sentence: $word_1\ word_2\ word_3\ ...\ word_{n-1}\ word_n$

Sentence changed

by one deletion:

1: $word_2\ word_3\ ...\ word_{n-1}\ word_n$

2: $word_1\ word_3\ ...\ word_{n-1}\ word_n$

...

n: $word_1\ word_2\ word_3\ ...\ word_{n-1}$

by one replacement:

1: "*" $word_2\ word_3\ ...\ word_{n-1}\ word_n$

2: $word_1$ "*" $word_3\ ...\ word_{n-1}\ word_n$

...

n: $word_1\ word_2\ word_3\ ...\ word_{n-1}$ "*"

by one insertion:

0: "*" $word_1\ word_2\ word_3\ ...\ word_{n-1}\ word_n$

1: $word_1$ "*" $word_2\ word_3\ ...\ word_{n-1}\ word_n$

...

n: $word_1\ word_2\ word_3\ ...\ word_{n-1}$ "*" $word_n$

n+1: $word_1\ word_2\ word_3\ ...\ word_{n-1}\ word_n$ "*"

Figure 3.1: *Simple error recovery on a sentence of length n.*

3.2.2. Conserving subsentences

After the parsing of a sentence has failed, we could look at the productions available in the P-triangle, to search for productions, whose left-hand side could match a large part of the input sentence. Formally, we are looking at productions **p**, such that $\mathbf{p} \Rightarrow_{match} l(i,j)$, for a relatively large *i*. We find such productions in cell C(*i,j*). We could define some sort of algorithm, which corrects parts of the sentence that are not included in this particular subsentence, "covered" by production **p**. In effect, we try to conserve the subsentence.

The reason why this could turn out handy, is shown in Fig. 3.2(a). The figure shows the contents of the P-triangle after a run of the algorithm with an example grammar G_2 and a given list of sets of attributes. Cell C(2,1) contains a production with the start-symbol on the left-hand side. If we maintain the list of attribute-sets which starts at position 1, we would have a quick success in finding a correction of length 1, by deleting the first set of attributes, ATT(0), which effectively means deleting the first word from the input. The resulting P-triangle, with this shorter input, is shown in Fig. 3.2.(b). We now find production 1 (**S**→**BE**) in the top cell, so this sentence is corrected well.

Already we would have to figure out in some way if this is the only correction of the given length that is possible. There might well be another type of correction, which also corrects the sentence by conserving our "correct" part of the sentence. It can, however, very well be the case, that a correction has to be made in this "correct" part. An example is given in Fig. 3.3(a) for the grammar G_3 and a given list of sets of attributes.

Here, one obvious way to correct the sentence is to delete ATT(2) (corresponding to the third word in the input sentence). The situation is shown in Fig. 3.3(b). With this attribute-set deleted, we find that the left-hand side of production 3 (**F**→**BD**) can now be used to match the attribute-sets ({b},{d}) and, as a result of that, the left-hand side of production 1 (**S**→**AF**), which is **S**, can be used to match all sets. So an appropriate correction has been made.

This example shows that we will have to do more than just look at large portions of the sentence, for which we have some production with or without a left-hand side equal to the start-symbol that can match a relative large part of the attribute-sets of the sentence. Moreover, suppose we have a sentence of length *n*, which is first processed in the usual way by the CYK-algorithm and for which is found a production of the type above for a large subsentence of length, say, *n*-1. If errors occur in a sentence with the same probability for each word, regardless whether it is part of such a large "correct" part (which we think they do), then the probability that the error occurred in this part is also large. This argument even contradicts the conservation of large portions.

The following example shows that a sentence that is almost correct, except for one correction, does not have to generate productions that match large parts of the attribute-sets of a sentence in the P-triangle:

Consider the example in Fig. 3.4(a), where for the given grammar G_4 and the given list of attribute-sets of a sentence of length 4, a correct parsing is found. If we were given an original sentence, which consisted only of the first three words of the first sentence, then we know now that we could correct it, by adding a fourth set of attributes consisting of element {d}. However, Fig 3.4(b) shows us that we will not find productions in the P-triangle for this shorter sentence, whose left-hand sides match more than one attribute-set each. This shows

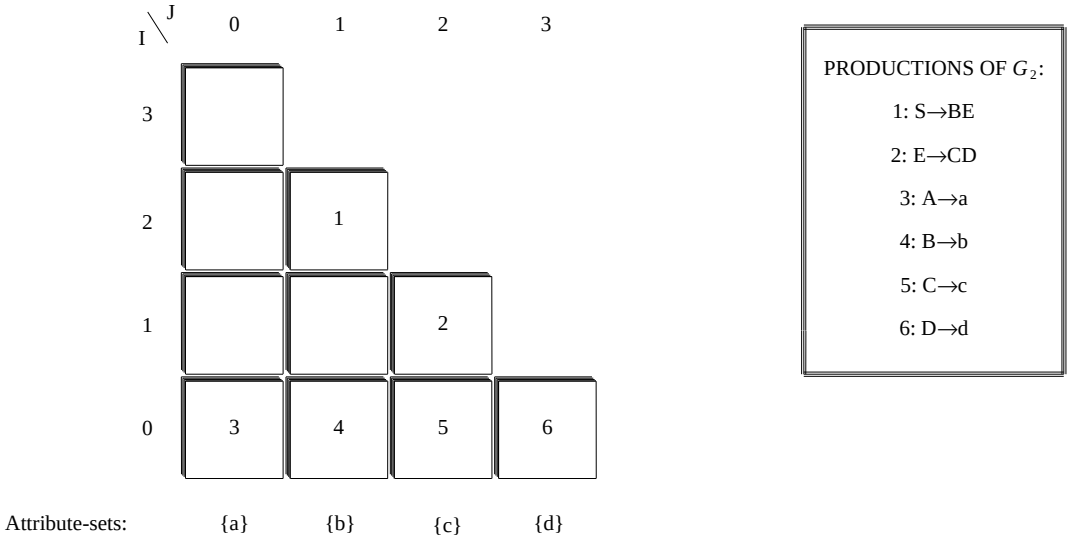


Figure 3.2(a): *P-triangle for grammar* G_2.

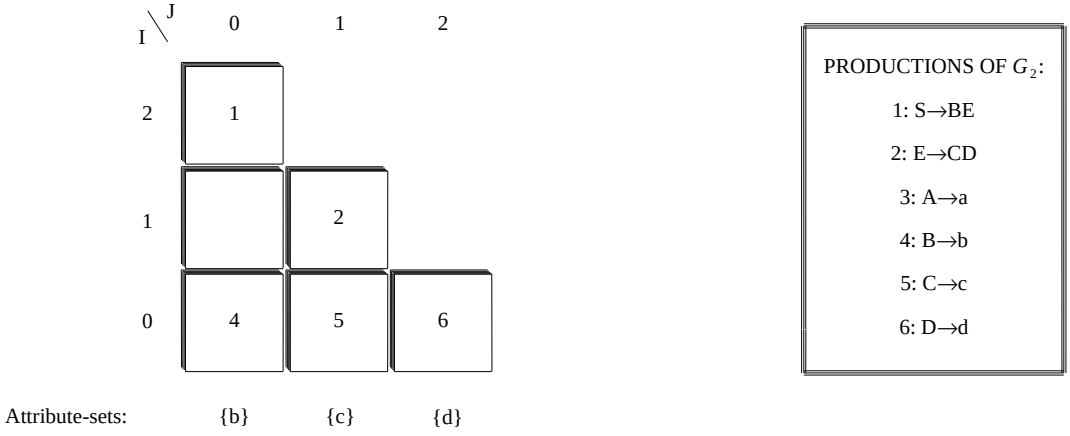
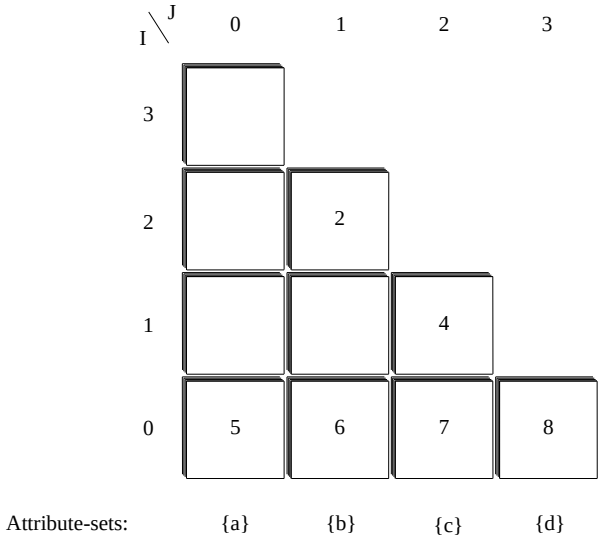


Figure 3.2(b): *P-triangle after deletion of ATT(0).*

that there is more to do to correct a sentence, than just conserving large "correct" portions. The algorithm could violate constraint (a). So due to reasons of incompleteness, an algorithm based on this idea is rejected as well.



PRODUCTIONS OF G_3:

1: S→AF

2: E→BG

3: F→BD

4: G→CD

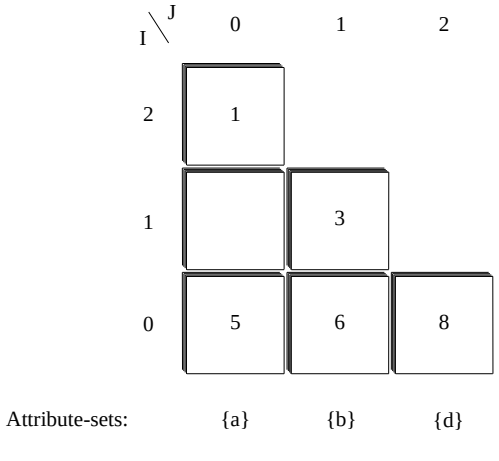
5: A→a

6: B→b

7: C→c

8: D→d

Figure 3.3(a): *Correction required in "correct" part of the sentence.*



PRODUCTIONS OF G_3:

1: S→AF

2: E→BG

3: F→BD

4: G→CD

5: A→a

6: B→b

7: C→c

8: D→d

Figure 3.3(b): *The resulting P-triangle after deletion of the third word.*

3.3. Conclusion

The two examples above really show that we have to look at every possible correction that can be made to an incorrect sentence, simultaneously. The algorithm described in chapter 4 does so, by generating corrections of higher order (greater length) successively.

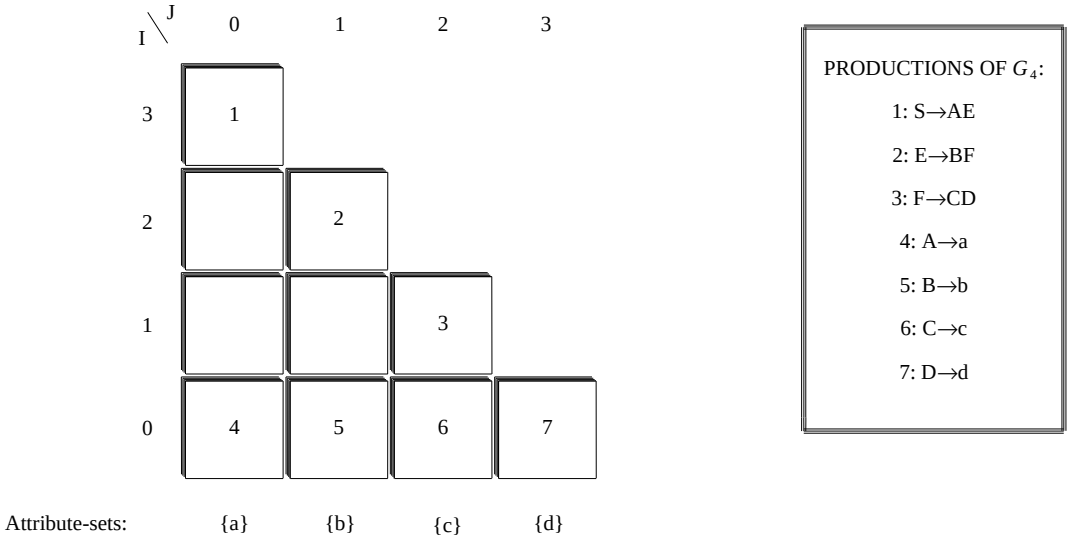


Figure 3.4(a): *The P-triangle for a sentence of length 4.*

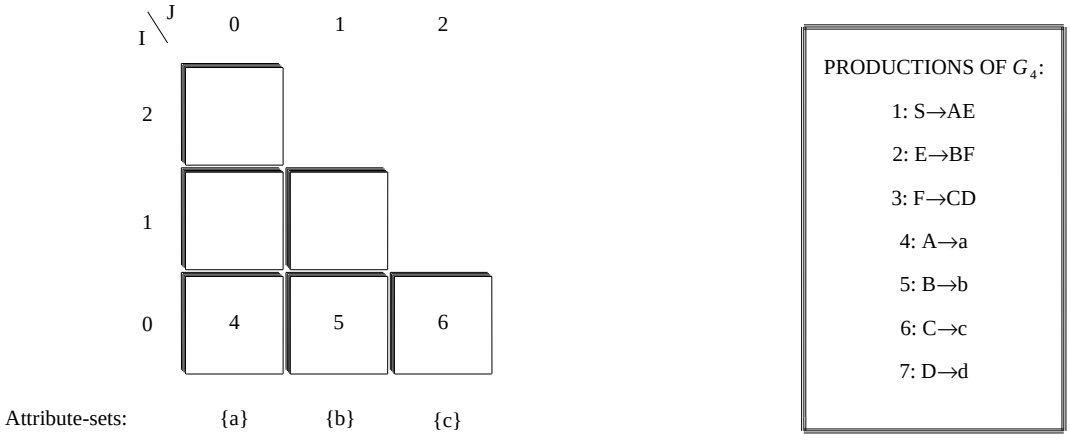


Figure 3.4(b): *The P-triangle of the original sentence of length 3.*

Also the algorithm is of polynomial complexity depending on the length of input. But most important of all, its operations are done entirely in terms of the CYK-algorithm and the corresponding structure of the P-triangle and N-triangle. It also uses the structures found in the first run of the program without slowing down this run in some way, so one could say that

error recovery already starts at the time we are just testing, whether a sentence is correct or not.

4. Error recovery

4.1. Formal definition

4.1.1. Definitions for attribute-sets

Before we describe the general algorithm, first some definitions will be given.
We already defined l(i,j) to be the list of i attribute-sets, starting at position j in the list of all attribute-sets, which belong to a given input sentence having the same order of the words in the input sentence, from which they were derived. Also, for a production **p**, we wrote $\mathbf{p} \Rightarrow_{match} l(i,j)$, if there exists a matching of the attribute-sets l(i,j), derived from the left-hand side of production **p**. We also define ATTRIB to be the set of attributes consisting of all attributes in the natural-language definition and use X, Y, Z, ... to denote some subset of ATTRIB. We will call the empty set of attributes $ATT_{\varnothing}$. Let, finally, Λ(ATTRIB) be the set of all lists of subsets of ATTRIB.

4.1.2. Correction-operators and chains

We now introduce *correction-operators*.
A correction-operator, $OP_1(k,X)$: $\Lambda(ATTRIB) \rightarrow \Lambda(ATTRIB)$, with *index* k and attribute-set X is one of four types of operators that can be applied to a list of attribute-sets l(i,j), defined below:

(1) The *empty*-operator $\varepsilon(k, ATT_{\varnothing})$. We will write it as $\varepsilon()$, since it has dummy arguments. When applied to a list of attribute-sets, l(i,j), it holds that

$$\varepsilon()(l(i,j)) = l(i,j).$$

(2) The *deletion*-operator, $Del(k,ATT_{\varnothing})$. We will write it as Del(k), since it second argument is a dummy. When applied to a list of attribute-sets

$$l(i,j) = (ATT(j), \ldots, ATT(k\text{-}1), ATT(k), ATT(k\text{+}1), \ldots, ATT(j\text{+}i\text{-}1)),$$

it holds that

$$Del(k)(l(i,j)) = (ATT(j), \ldots, ATT(k\text{-}1), ATT(k\text{+}1), \ldots, ATT(j\text{+}i\text{-}1))$$

with $(j \le k \le j+i-1)$.

(3) The *replacement*-operator, Repl(k,X). When applied to a list of attribute-sets

$l(i,j) = (ATT(j), \ ... , ATT(k\text{-}1), ATT(k), ATT(k\text{+}1), ... , ATT(j\text{+}i\text{-}1))$,

it holds that

$Repl(k,X)(l(i,j)) = (ATT(j), ... , ATT(k\text{-}1), X, ATT(k\text{+}1), ... , ATT(j\text{+}i\text{-}1))$

with $(j \leq k \leq j+i-1)$.

(4) The *insertion*-operator, Ins(*k*,X). When applied to a list of attribute-sets

$l(i,j) = (ATT(j), \ ... , ATT(k\text{-}1), ATT(k), ... , ATT(j\text{+}i\text{-}1))$,

it holds that

$Ins(k,X)(l(i,j)) = (ATT(j), ... , ATT(k\text{-}1), X, ATT(k), ... , ATT(j\text{+}i\text{-}1))$

with $(j \leq k \leq j+i)$.

Several operators can also be chained by writing:

$OP_1 * OP_2 * OP_3 * ... * OP_n \ (n \geq 1)$

which is equivalent to first applying operator OP_n to a given list of sets of attributes, then OP_{n-1} ..., until all operators in right to left order have been applied.

We will denote such *chains* by π, ρ, σ, ... Note that π could also consist of a single operator (because $OP_1 * \varepsilon() = OP_1$). Therefore we will refer to such chains by calling them operators.

One should realize that the indices of the operators only refer to elements of the list of attributes l(*i*,*j*). This means that the list of sets of attributes that is produced as a result of applying one operator still carries the same indices ATT(*i*). An example should make this point clear. Suppose we have the chain:

Repl(0,X) * Ins(0, Y)

If we apply this operator to

$l(i,0) = (ATT(0), ATT(1), ... , ATT(i\text{-}1))$

The result will be

(Y, X, ATT(1), ... , ATT(*i*-1))

instead of

(X, ATT(0), ATT(1), .. , ATT(*i*-1)).

Thus every operator is assigned to a specific element of the original list of attribute-sets by its index. This puts some restrictions on the chain of operators.

4.1.3. Constraints on chains

First, it is illegal to have an operator Del(*k*) twice in a chain, since every attribute-set can only be deleted once. For a similar reason it is pointless to put Del(*k*) and Repl(*k*) in one chain. It makes no sense to first replace an attribute-set if it is deleted anyway and also we can not replace an already deleted set. Therefore, this will be considered illegal from now on. To put Repl(*k*,X) and Repl(*k*,Y) in a chain is forbidden as well. It is also illegal to have a Del(*k*)-operator and an Ins(*k*,X) operator in a chain, since we then make two corrections to a list of attribute sets, while one, namely Repl(*k*,X) would do the job. For the same reason Del(*k*-1) and Ins(*k*,X) are forbidden as well in one chain. Note that declaring combinations of operators illegal, is only done, when there are "too many" operators in a chain. Each illegal combination could have been rewritten by a chain consisting of fewer operators, but having the same effect on the list of attribute-sets it was applied to.

A last constraint on chains is that we will always consider it "ordered" in the following way:
All indices of the operators will appear in a chain with a non-decreasing value from left to right. Since a chain can have several insertion-operators, Ins(*k*,X), Ins(*k*,Y), ... , and possibly a replace-operator, Repl(*k*,Z), we will demand that the insertion operators appear left of the replacement-operator in a chain. Note that the order in which insertion-operators Ins(*k*,X) are given in a chain is important, since they are evaluated right to left.

The *length* of a chain π, $|\pi|$ is given by the number of its constituting operators, where we do not count the empty operator.
By the argumentation given above, ruling out the illegal chain does not limit the possibility to correct a list of attribute-sets to any other list of attribute-sets by a minimum length chain.

In the next sections, we will use the structure of the P-triangle again and fill it with productions. In particular, productions will turn out to be indexed by some chain of correction-operators. Whenever we fill in the production $\mathbf{p}_\pi$, for a production **p** and some chain π, in some cell C(*i*,*j*) in the P-triangle used, we mean that $\mathbf{p} \Rightarrow_{match} \pi(l(i+1,j))$. Informally we mean that from the left-hand side of production **p** a match of l(*i*+1,*j*), changed according to operator π, can be derived.

4.2. The error-recovery algorithm

4.2.1. General idea

The general idea of the algorithm is to simulate the CYK-algorithm for the given sets of attribute lists of the original input corrected by some increasing number of correction-operators. We start off with the P-triangle as it was left by the very first run of the algorithm

where we did not find a matching from the start-symbol for the attribute-sets. We will fill each cell C(*i*,*j*) of the P-triangle with productions **p**, from whose left-hand side there exists a derivation, which can match the list of attribute-sets l(*i*-1,*j*) corrected by some chain of operators π. To distinguish the production **p**, used in this way, from an ordinary use of **p** we write it as $\mathbf{p}_{\pi}$. Note that the productions **q** already present after the first run can be written as $\mathbf{q}_{\varepsilon()}$ following the formalism. We then try to combine such productions, which define a correction to a subsentence, and their respective chain of corrections the way it is normally done in the CYK-algorithm to form larger corrected subsentences.

We will assume that there is some predefined maximum to the number of corrections we are allowed to make to a list of sets of attributes defined by some input sentence in the natural-language. This is not a strange assumption, since we should keep the program from making too many corrections for at least three good reasons:

(1) If we need to make too many corrections, it might well be the case, that the input sentence was complete nonsense in the first place.

(2) If we make too many corrections, the probability decreases, that the corrected sentence will have anything to do with the original intentions of the user who wrote it.

(3) In practice, it turns out that there are often numerous ways to correct a sentence using a large number of corrections. It is hardly useful to know that a sentence can be corrected in e.g. 20,000 ways since it will be very hard to pick out a decent correction.

Therefore the maximum number of corrections to a sentence of length *n* will be set to *n*, which is already large, but might be appropriate to short sentence of e.g. one word in which one word causes incorrectness.

4.2.2. General from

The general form of the algorithm is as given below. We refine its details in later sections.

Let *maxcorrections* denote the maximum number of corrections allowed to an input sentence. Let *n* be the length of the sentence. Given the P-triangle after the first run of the CYK-algorithm, do the following for *corrlevel* = 1 to *maxcorrections*:

(1) If *corrlevel* = 1, introduce "basic" replacement-corrections in the bottom-row of the P-triangle. This procedure will be described in section 4.2.3.

(2) Introduce "basic" deletion-corrections for the current number of corrections made. This procedure will be described in section 4.2.4.

(3) Generate insertions-corrections for the bottom-row elements of the P-triangle. This procedure will be described in section 4.2.5.

(4) We now visit every cell in the triangle from row 1 to row n-1 in the usual manner, but this time we add productions with a chain of operators, which describe the way the list of attribute-sets, "covered" by this cell, should be corrected in order for the left-hand side of this production to be able to produce a match of this corrected list of attribute-sets.

For i = 1 to n - 1,
For j = 0 to n - i - 1,
consider cell C(i,j) of the P-triangle.
For k = 0 to i - 1,

For l = 0 to *corrlevel*,
Let $\mathbf{p} \in \mathbf{P_N}$ be of the form $\mathbf{A} \rightarrow \mathbf{BC}$. If there is a production $\mathbf{q}_\rho$ in cell C(k,j), whose left-hand side is **B** and whose added chain of correction-operators is ρ, with $|\rho| = l$ and there is a production $\mathbf{r}_\sigma$ in C(i-k-1,j+k+1), whose left-hand side is **B** and whose added chain of correction-operators is σ, with $|\sigma| =$ *corrlevel* - l, then add $\mathbf{p}_\pi$ to cell C(i,j), if it does not violate Rule (*). Rule (*) is a logical optimization rule to the algorithm. It means that we will not add $\mathbf{p}_\pi$, with **p** having left-handside **A**, if there already exists an extended production $\mathbf{q}_\rho$ in C(i,j), also with left-handside **A**, and it holds that $|\rho| < |\pi|$. The rule is correct, because it is pointless to introduce the fact that from **A** a matching can be derived for some subsentence corrected by $|\pi|$ corrections, if we already could match it by a smaller amount $|\rho|$.

After this has been done for cell C(i,j), generate insertion-corrections for it, as described in section 4.2.5.

If there are productions in cell C(n-1,0) with the start-symbol **S** on the left-hand side, then interrupt the main-loop. We have found at least one appropriate correction to the input sentence of length *corrlevel*.

During each sweep of the algorithm for a value of *corrlevel*, we always add productions to each cell that have a chain of operators of length *corrlevel*. We will see that introduction of basic corrections and insertions does not violate this. Fig. 4.1 illustrates the global way the P-triangle is filled, when *corrlevel* = 2, assuming there are some productions with corrections π and ρ of length one already available in the bottom row (introducing basic corrections and insertions is not done). We will now describe the introduction of the basic corrections. The reason for their introduction will be described in section 4.2.6, where we will outline the validity of the algorithm.

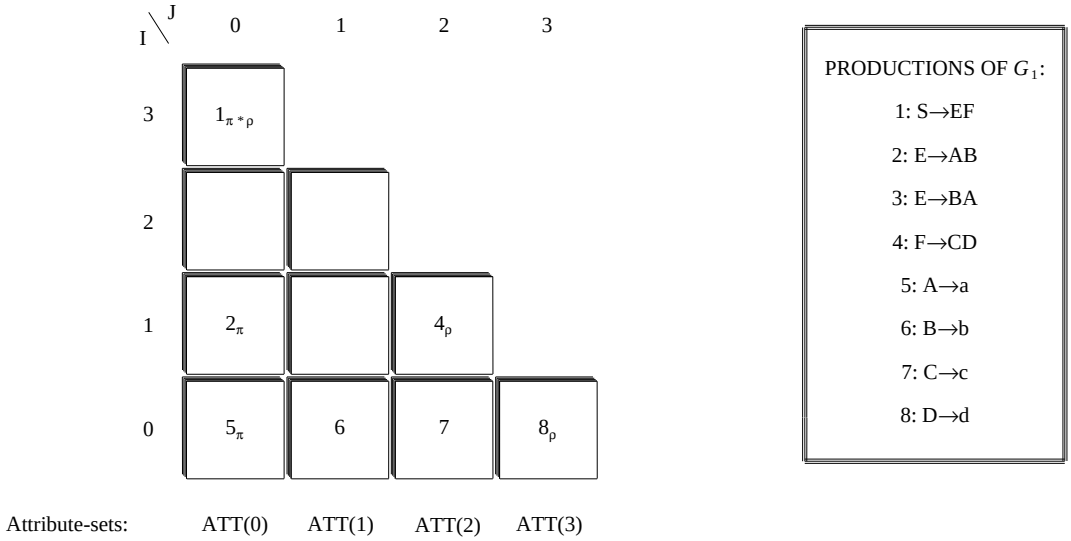


Figure 4.1: *Essence of the algorithm in simplified form.*

4.2.3. Basic replacement-corrections

Given the value 1 of the variable *corrlevel* in the program, we add the following productions to the bottom-row cells in the P-triangle:

Consider ATT(*j*). Define ATT(j)$_{comp}$ to be ATTRIB - ATT(*j*).
For each $\mathbf{p} \in \mathbf{P_T}$, being of the form $\mathbf{A} \rightarrow \mathbf{a}$, such that $\mathbf{a} \in$ ATT(*j*)$_{comp}$, *add* $\mathbf{p}_{\text{Repl}(i,\{a\})}$ to C(0,*j*), if it does not violate Rule (*).

We thus add every terminal Chomsky production that could match the complement of ATT(*j*), with respect to the set ATTRIB. By adding the correction operator, we have registered the fact that we made a correction of length one to derive a matching for this new set of attributes. The operation is shown in Fig. 4.2 for a given cell in the bottom-row. The complexity of this operation is clearly O(*n*).

4.2.4. Basic deletion-corrections

Given a value $i \geq 1$ of the variable *corrlevel*, add the following productions to each cell C(*i*,*j*) in the *i*-th row of the P-triangle, with the restriction imposed by Rule (*):

If there is a production $\mathbf{p} \in \mathbf{P_T}$ and **p** appears in C(0,*j*), then add a production $\mathbf{p}_{\pi}$ to C(*i*,*j*), where π consists of

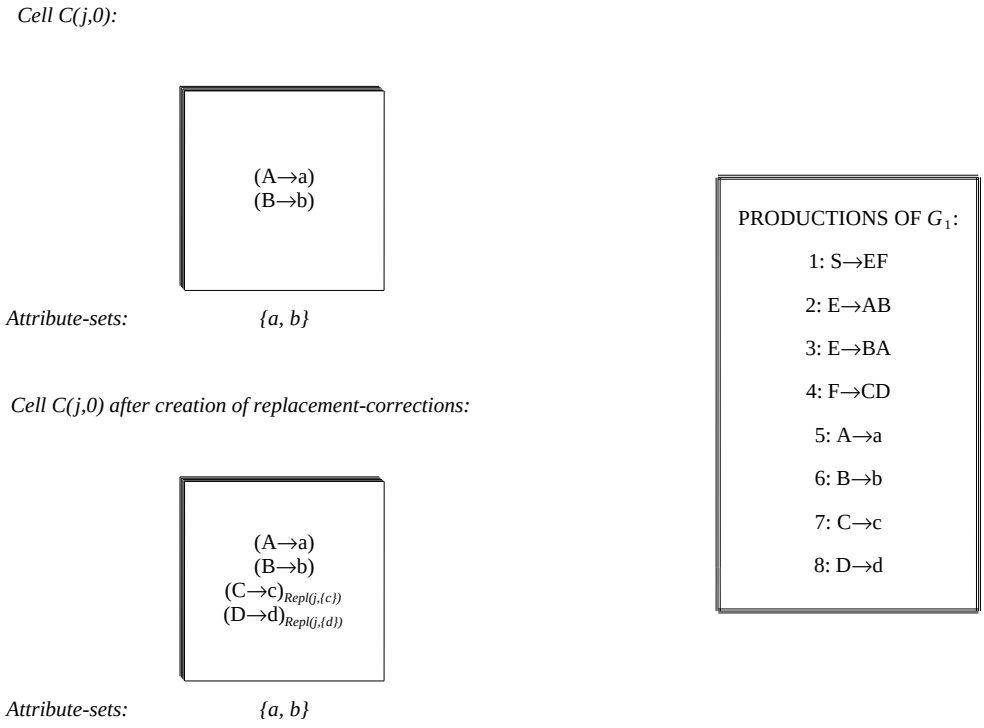


Figure 4.2: *Basic replacement-correction for cell C(0,j).*

(1) Del(j+1) * Del(j+2) * ... * Del(j + i-1)

Notice that deletion of the first attribute-set has not been considered so far. These deletions are accounted for in the following step.

If $j = 0$ then do the following for $k = 1$ to i:
If there is a production $\mathbf{p} \in \mathbf{P_N}$ in cell C(0,k) then add production $\mathbf{p}_\pi$ to cell C(i,0), where π consists of

(2) Del(0) * ... * Del(k-1) * Del(k+1) * ... * Del(i)

Note that each production has a chain of operators added equal in length to *corrlevel*. The general procedure is illustrated in Fig. 4.3 for an example grammar $\mathbf{G_1}$ with a list of three attribute-sets.

If *corrlevel* > 1 we also do the following:
Repeat the above operation, done for terminal Chomsky productions, also for terminal Chomsky productions present in the bottom-row of the P-triangle which carry one replacement correction, but we act as if the variable *corrlevel* carried a value one less than it in fact has. This way, we make sure that the length of the chain of operators added to each production is

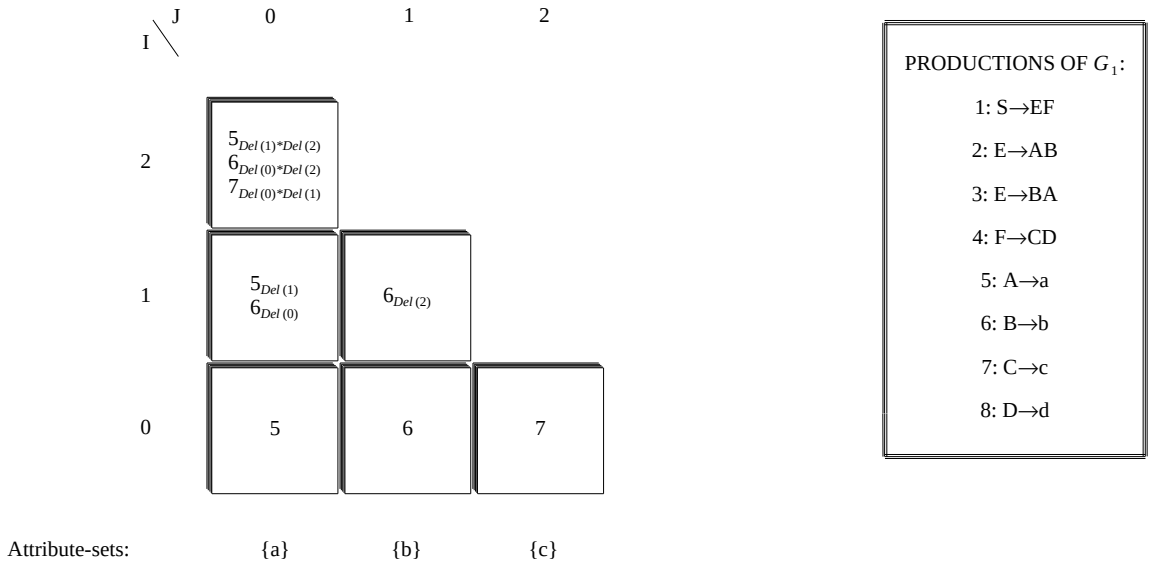


Figure 4.3: *Introduction of basic deletion-corrections.*

equal to the variable *corrlevel*.

The corrections produced in this way simulate the matching of a list of attribute-sets of length *i* (or *i*-1 in the case of the use of a terminal Chomsky production which carries a replacement operator) of which *i*-1 (*i*-2) sets are deleted. We will see that the productions added above are sufficient to catch any correction of a sentence done by deleting part of it.

4.2.5. Creating insertion-corrections

The creation of corrections resulting from inserting a set of attributes is done using an *insertion array*. The array consists of cells capable of holding productions like the cells in the P-triangle. We will refer to those cells by I(0), I(1), ... The first element of the array contains all productions whose left-hand side can match a single attribute. In practice, this means that it is filled with every terminal Chomsky production, $\mathbf{A}\rightarrow\mathbf{a}$, with an operator Ins(*undefined*, {a}). The reason of the index being undefined at this moment will become clear in a moment. Having filled the first element I(0), the second element of the insertion array is filled by considering every production $\mathbf{p}\in\mathbf{P_N}$ of the form $\mathbf{A}\rightarrow\mathbf{BC}$ such that there exists productions in $\mathbf{q}_\rho$ in I(0) whose left-hand side is **B** and productions $\mathbf{r}_\sigma$ in I(0) whose left-hand side is **C**. We then add a production $\mathbf{p}_\pi$ to I(1) with $\pi = \rho * \sigma$.

In general element I(*k*) of the insertion array contains all productions from whose left-hand side a matching can be derived for a list of attribute-sets, all equal to ATTRIB, of length *k*. Each array element is created once and remains valid for every input-sentence given to the parser. We only need to create the *k*-th element of the insertion array as soon as the variable

corrlevel reaches value *k*. The algorithm for filling the *k*-th element ($k > 1$) of the insertion array is as follows:

> For $i = 1$ to k-1,
>
> Let there be a production $\mathbf{p} \in \mathbf{P_N}$ of the form $\mathbf{A} \rightarrow \mathbf{BC}$. If there is a production $\mathbf{q}_\rho$ in I(i) such that its left-hand side is **B** and there is a production $\mathbf{r}_\sigma$ in I(k-i) such that its left-hand side is **C** and if there does not exist a production in an element of the insertion array, whose left-hand side is **A**, then add $\mathbf{p}_\pi$ to I(k), where $\pi = \rho * \sigma$.

The last condition prevents the addition of a production to I(k), whose left-hand side was already found able to match a shorter list of attribute-sets and is valid in the same sense Rule (*) is: we never create a larger correction than necessary. Note that every operator (with, so far, undefined index) in a chain of operators, being added to a production, only has attribute-sets consisting of one attribute. If the number of corrections is bounded by the length of the input-sentence, *n*, which is quite a large upper-bound in practice, the number of elements of the insertion-array will also be bounded by *n*. The number of operations to fill all the elements will then be bounded by $O(n^2)$. The procedure is illustrated in Fig. 4.4 for an example grammar $\mathbf{G}_5$ for the first three elements of the insertion array. Note that production 1 is not added to I(2) due to the optimization rule.

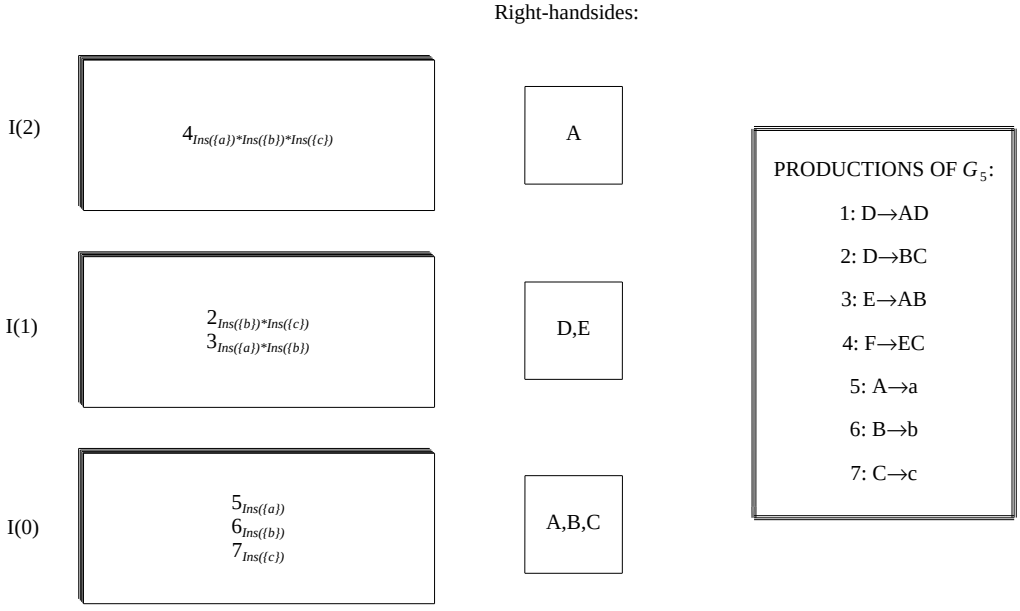


Figure 4.4: *Example of the way the insertion-array is filled.*

Now having introduced the insertion array, we can give the algorithm for creating insertion-corrections in the P-triangle during the run of the main algorithm. The idea is to "hang" inserted sets of attributes onto some cell of the P-triangle, which contains productions, whose left-hand sides have already been found capable of matching some part of the list of attribute-sets. The insertions are done left and right of the partial list of attribute-sets.

The following algorithm creates insertion-corrections for every cell in the P-triangle:

> Let C(*i,j*) be a cell of the P-triangle considered at the moment the productions are added to it as described by the main algorithm, let *k* be the value of the variable *corrlevel*. By the above algorithm for creating the insertion array, all elements of it up to and including element *k* have already been produced.
>
> For $m = 1$ to k,
>
> Let there be a production $\mathbf{p} \in \mathbf{P_N}$ of the form $\mathbf{A} \rightarrow \mathbf{BC}$ and let there be a production $\mathbf{q}_\rho \in$ I(*m*) with left-hand side **B** and a production $\mathbf{r}_\sigma$ in the cell C(*i,j*) with left-hand side **C** and $|\sigma| = k - m$. Consider the undefined indices of ρ to be set to *j*. If $\pi = \rho * \sigma$ defines a legal chain and if Rule (*) is not violated, then add $\mathbf{p}_\pi$ to C(*i,j*).

This operation defines insertion of attribute-sets left to the portion of attribute-sets covered by cell C(*i,j*). A similar operation is done for insertions at the right:

> For $m = 1$ to k,
>
> Let there be a production $\mathbf{p} \in \mathbf{P_N}$ of the form $\mathbf{A} \rightarrow \mathbf{BC}$ and let there be a production $\mathbf{q}_\rho \in$ I(*m*) with left-hand side **C** and a production $\mathbf{r}_\sigma$ in the cell C(*i,j*) with left-hand side **B** and $|\sigma| = k - m$. Consider the undefined indices of ρ to be set to $i + j$. If $\pi = \rho * \sigma$ defines a legal chain and if Rule (*) is not violated, then add $\mathbf{p}_\pi$ to C(*i,j*).

The complexity of each of these operations is O(*n*), if we assume that *k* is bounded by the length of the input-sentence.

The insertion array can be viewed as an extended set of cells virtually present in the P-triangle, between every two original attribute-sets and at the start and end of the whole list of attribute-sets.

4.3. Validity of the algorithm

4.3.1. Outline of the proof

The validity of the algorithm will be proven by construction. We will show that whenever a production $\mathbf{p}_\pi$ is added to the top-cell in the algorithm with the start-symbol on its left-handside there exists a shotest correction π to the original sentence.

4.3.2. Proof of validity

Whenever a production $\mathbf{p}_\pi$ appears in cell C(*n*-1,0) after one sweep of the algorithm for a given value of variable *corrlevel* which has the start-symbol on the left-hand side, then, by the algorithm, there exists some derivation from the start-symbol that matches π(l(*n*,0)).

We now want to prove the converse:

if there exists some shortest chain π of corrections to the list of attribute-sets l(n,0) then it will appear in cell C(n-1,0) after the sweep of the algorithm in which *corrlevel* is equal to π added to some production **q** having the start-symbol on the left-hand side.
Observe that such a production will only be added to cell C(n-1,0) when *corrlevel* has the value π. This is inherent to the algorithm since during any sweep only chains are added to productions which have a length equal to *corrlevel*.

Assume there exists some shortest chain of corrections ρ to an original set of attribute-sets. This means that there exists a leftmost derivation in the grammar, $\mathbf{S} \Rightarrow^* \rho(l(n,0))$, matching the attribute-sets. To prove our statement we will first look at the ways ρ can be built up.

(1) ρ may contain an operator

Repl(j,some_attribute).

Virtually this means that in the derivation there appears a terminal production related to the attribute-set of the word in the sentence indexed by j. In the algorithm the same production will be added to cell C(0,j) in step (1), covering the same portion of the sentence having added the same operator.

(2a) ρ may contain a chain of operators

Del(j)* .. *Del(j+k)

such that it does not contain Del(j-1) and Del(j+k+1) and $j > 0$. This means that in the derivation there virtually exists a terminal production related to the attribute-set of the word indexed by $j - 1$ covering the attribute-sets indexed by $j - 1$, ... , $j + k$ of the original sentence. In the algorithm this production will be added in step (2) to cell C(k,j-1) covering the same portion of the sentence having added the same chain.

(2b) ρ may contain a chain of operators

Del(0)* ... *Del(j-1)*Del(j+1)* ... *Del(j+k)

such that it does not contain Del(j) and Del(j+k+1). This means that in the derivation there virtually exists a terminal production related to the attribute-set of the word indexed by l covering the attribute-sets indexed by 0, ... , $j + k$ of the original sentence. In the algorithm this production will be added in step (2) to cell C(k,0) covering the same portion of the sentence having added the same chain.

(2c) ρ may contain a chain of operators

Repl(j-1,some_attribute)*Del(j)* .. *Del(j+k)

such that it does not contain Del(j+k+1) and $j > 0$. This means that in the derivation there virtually exists a terminal production related to the attribute-set of the word indexed by $j - 1$ covering the attribute-sets indexed by $j - 1$, ... , $j + k$ of the original sentence. In the

algorithm this production will be added in step (2) where we extend deletions to replaced attribute-sets to cell C(k,j-1) covering the same portion of the sentence having added the same chain.

(2d) ρ may contain a chain of operators

Del(0)* ... *Del(j-1)*Repl(j,some_attribute)* Del(j+1)* ... *Del(j+k)

such that it does not contain Del(j+k+1). This means that in the derivation there virtually exists a terminal production related to the attribute-set of the word indexed by l covering the attribute-sets indexed by 0, ... , $j + k$ of the original sentence. In the algorithm this production will be added in step (2) where we extend deletions to replaced attributes sets to cell C(k,0) covering the same portion of the sentence having added the same chain.

(3) ρ may contain an operator

Ins(j,some_attribute).

This means that in the derivation there exists a terminal production that derives an attribute inserted between two original attribute-sets or on one of the ends of the list of attribute-sets. Each of the adjacent attribute sets cannot be deleted or there would be a shorter correction chain. In the recovery algorithm, the insertion-array, containing terminal productions in the 0-th element is virtually present at the same locations. In particular, the production from the derivation will be present in the insertion-array with the right operator added (however no index is added yet).

So far, it is shown that any terminal production in a derivation associated to a shortest correction ρ of the whole sentence virtually covering some part of the original sentence is also present in the P-triangle in cells covering that same part. Also the operator added to this production is the same (this also holds for terminal productions that appear in the derivation deriving a correct part of the sentence).

In the derivation there must appear at least one nonterminal production having two adjacent terminal productions appearing in the derivation as "descendants". Each such production derives a part of the corrected sentence associated with each of the two terminal productions. We can think of these productions having added the concatenation of the two chains (virtually) present at each of its terminal descendants.

By the algorithm, step (3) and step (4), this nonterminal production will be added to the cell covering the part of the original sentence to which this chain is applied having added the same chain of a length smaller or equal to ρ. Whenever one of the terminal productions used to add this nonterminal production was present in the insertion-array, its chain will now get the right index for the inserted attribute. Whenever both terminal productions were present in the insertion-array, the nonterminal production can be considered present in the insertion-array at a higher level still having to wait for assignment of its index.

As a next step it is possible to find nonterminal productions in the derivation having two adjacent terminal or nonterminal productions already found to be deriving part of the corrected sentence with an virtual correction chain added as its descendants. Such a

production derives a larger part of the corrected sentence and it will, by the algorithm steps (3) and (4), be present in the P-triangle with an appropriate correction chain as well at a cell covering the part of the original sentence that was corrected by the chain.

Eventually we end up at the first production used in the derivation with the start-symbol on the left-handside. Since this production derives the whole corrected sentence, by the argumentation above, we find it in he top cell of the P-triangle as well, having added ρ. Note that it can not be on the insertion-array or we would have to have deleted the whole original sentence which would create a chain with an adjacent insertion and deletion.

4.4. Actual implementation of the algorithm

4.4.1. The procedure Recover()

In the actual implementation of the algorithm we do not directly store productions linked to a correction chain in the P-triangle or N-triangle. If we would link a correction chain to every production or nonterminal added we would not only quickly run out of memory, but there would also be a lot of computational overhead. Besides that, when adding a production to a cell, we would have to test whether it was already there with the same correction chain added and if not, add it several times with disjunct correction chains. Rather, the triangles are viewed as being extended into a new dimension. At the zero level productions and nonterminals are stored from which correct pieces of the sentence can be derived. In fact they derive these pieces by introducing no corrections. At level *k* in the cells in the triangles productions and nonterminals are stored from which it is possible to derive (sub)sentences corrected by *k* corrections.

When the filling of the triangles terminates the level *corrlevel* at cell C(*n*-1,0) contains at least one production with the start-symbol on the left-handside. It is now possible to trace back on the triangles every way this production was added to the cell until we end up at terminal Chomsky productions. Backtracing can extend over the insertion-array since it was also used in building up the triangles. Whenever a terminal production is encountered its place in the triangle defines the correction chain linked to it implicitly.

We will first present the algorithm and then explain its details. We use a stacktype capable of storing a production, its location in the triangles or on the insertion-array and a number denoting the number of corrections to a (sub)sentence still to be identified. Also a global array *Corrbuf* [] is used to store a chain of corrections.

For each production **p** present at level *corrlevel* in cell C(*n*-1,0) after termination of the filling, having the start-symbol on the left-handside, do the following:

Push(*stack*, **p**, C(*n*-1,0), *corrlevel*);

Then call a procedure:

Recover(0, *stack*);

The definition of Recover():

```
Recover(index, stack)
{
    Pop(stack) and set the variables prod, loc and corrs_to_go to the popped element;
    corrs_to_go denotes the level in the triangles;
    If prod∈ P_T then
    {
        Identify all correction chains (of length corrs_to_go) associated to this pro-
        duction as explained below;
        For each such chain do:
        {
            Assign Corrtab[index] to Corrtab[index+corrs_to_go−1] to the chain;
            If stack=EMPTY then
                Test the chain defined by Corrtab[] on uniqueness; if so, print it
                and store it in a global table;
            Else
                Recover(index+corrs_to_go, stack);
        }
    }
    Else /* prod∈ P_N */ {
        For every pair q and r of productions used by the filling algorithm to create
        prod at level corrs_to_go at loc, let loc_left and corrs_left be the location and
        the level of q and let loc_right and corrs_right be the location and the level of
        r;
        The locations can be on the insertion array;
        {
            Push(stack, r, loc_right, corrs_right);
            Push(stack, q, loc_left, corrs_left);
            Recover(index, stack);
        }
    }
}
```

The stack is used to remember where after the "split" in the second part of the procedure to go on recovering to the right when the whole part of the sentence to the left has been traced. The correction chain is built up by concatenating several subchains correcting parts of the sentence.

4.4.2. Identification of the chains of terminal productions

Identification of the chains associated to the terminal productions in the first part of the procedure can be done by considering the location and the level of the production.

First, if the location is on the insertion-array, obviously an insertion correction is

created. Its index is contained in the location and its attribute is defined by the right-handside of the production.

Now suppose the terminal production is not on the insertion-array and let the cell considered be C(i,j) at level k. The value of k can be i (with $i > 0$) or $i+1$ (without restrictions) or we would have an incorrect number of corrections.

If $k = i$ then we create a correction chain consisting totally of k deletion-corrections:

Del(j+1)* ... *Del(j+k).

Possibly, j can be 0 in which case also the possibilities of the first attribute-set being deleted have to be considered which works equivalently. We now look if the production exists in the the cells C(0,1) ... C(0,k). If this is true for a particular cell, say C(0,m) we generate the chain:

Del(0)* ... *Del(m-1)*Del(m+1) ... *Del(k).

If $k = i+1$ then the j-th attribute set must be a replacement-correction. The resulting chain of corrections will be of the form:

Repl(j, right_hand_side)*Del(j+1)* ... *Del(j+k-1).

In the last expression the Del()-operators are omitted if $k = 1$. Deletion of the first attribute set is generated in the same way as the example given above.

This way the terminal productions in a derivation for the corrected sentence define the chain that corrects the whole sentence. A concatenation of these corrections will yield a legal chain by the fact that the indices of the operators are well-ordered and by the fact that the mere presence of a production in a cell at some level implicates it was created by the filling algorithm and thus minimal. So no "adjacent" deletions and insertions exist.

4.5. Complexity of the algorithm

4.5.1. Computational complexity

Assuming the number of corrections allowed to a sentence, is proportional to its length, which means that the value of the variable *corrlevel* is of O(n), then step (1), (2) and (3) of the algorithm are all O(n). In step (4) i, j and k are all bounded by n. The variable l is never greater than *corrlevel*, so it is also bounded by n. In all, step(4) takes O(n^4) operations and, since it is repeated at most *corrlevel* times the complexity of the whole algorithm is O(n^5). This may seem large, but due to a large number of optimizations in the actual implementation the algorithm is still reasonably fast. We have to perform more operations, of course, than in a plain non-correcting CYK-algorithm, but it is still a lot better than the solution given in section 3.2.1. Also, if there exists at least one correct sentence of some length in the natural language, then the algorithm will, if the number of corrections is unbounded, eventually find

enough corrections which turn the incorrect list of attributes of an input-sentence to a list of attribute-sets that is correct. Because we can safely assume that there will be at least one finite sentence in a natural language (otherwise there is no use of the language), the algorithm will halt for every input.

4.5.2. Further characteristics

The number of productions present in the CFG also has some impact on processing time. However, the algorithms performance tends to be influenced mostly by the number of terminal Chomsky productions. The generation of replacement-corrections and insertion-corrections is responsible for this: there are a large number of terminal productions present in the bottom rows of the P-triangle. Tests have shown that the upper part of the triangle containing nonterminal productions is usually *sparsely* filled. Because the number of terminal symbols in a natural language definition is small, the number of terminal productions is also relatively small compared to the number of nonterminal productions. Therefore the size of the grammar will not increase execution time dramatically.

4.6. Format of the output

The output contains some control information preceded by a '#' and all the unique chains found for the sentence. the format is as follows:

corrections: *corrlevel*

$Chain_1$

$Chain_2$

$Chain_{number\_of\_unique\_chains}$

chains: *number_of_unique_chains*

The output printed in this way can, in the future, be processed further, most probably through the use of user-supplied plausibility data. More likely however, this extension will be incorporated in the program itself. Therefore all (unique) correction chains of the sentence are stored in a dynamic lexicographically sorted array named *Corrtab*[] from which an easy access to each correction is possible. This array is built up while backtracing on the triangles and is used for checking each correction chain found on uniqueness. If it is not already in the array, it is added, else it is omitted from the output.

5. Proposed further extensions

5.1. Optimization by comparing equal lists of attribute-sets

If we compare the contents of two cells in the P-triangle, C(i,j) and C(i,k), which are in the same row i, and it holds that (ATT(j), ATT(j+1), ... , ATT(j+i-1) = (ATT(k), ATT(k+1), ... , ATT(k+i-1)), then they will contain the same productions. This fact can be used to optimize memory usage and computation time when running the CYK-algorithm.

If we consider the P-triangle to be defined by a triangular array of pointers, P(i,j), to some portion of data, the actual cells C(i,j), where information about the productions is stored, we could optimize the CYK-algorithm by preprocessing these pointers in the following way:

Assign to each pointer a flag, initially undefined. Let n be the length of the input sentence.

(1) For i = 0 to n - 2,
For j = i + 1 to n - 1,
If ATT(i) = ATT(j) then Assign P(0,j) = P(0,i). Set the flag of P(0,j) to i, if the flag of P(0,i) is undefined, else set the flag of P(0,j) to the flag of P(0,i).

(2) For i = 1 to n - 2,
For j = 0 to n - i - 2,
For k = j + 1 to n - i - 1,
If P(i-1,k) = P(i-1,j) and ATT(i+k) = ATT(i+j) then Assign P(i,k) = P(i,j). If the flag of P(i,j) is undefined, set the flag of P(i,k) to j, else set the flag of P(i,k) to the flag of P(i,j).

In essence, what we are doing in step (1) is testing each pair of attribute-sets on equality. If they are equal, the pointer of the highest numbered element of the bottom-row is set to the pointer of the lowest numbered element. The flag of the highest numbered element is set to the index of the lowest numbered element. If the flag of the lowest numbered element was already defined, it means that its pointer was already set to yet a lower numbered element in the row. In this case the flag of the highest numbered element is set to the index of this yet lower numbered element.

In the second step a similar assignment is done for the higher numbered rows. To test whether the list of attribute-sets corresponding to each pair of pointers in row i are the same, we test if the two pointers one row below with the same index. If so, we know that their list of attribute-sets match up to the last element. These two last elements are compared, and, if found equal, flags and pointers are changed accordingly.

When we run the CYK-algorithm, whenever we encounter an element in the P-triangle, described by a pointer to appropriate data, whose flag has been set, we do not have to consider it, since its pointer has been set to a portion of data, containing all necessary productions. This optimization can also be used in the CYK-error-recovery algorithm, but we have

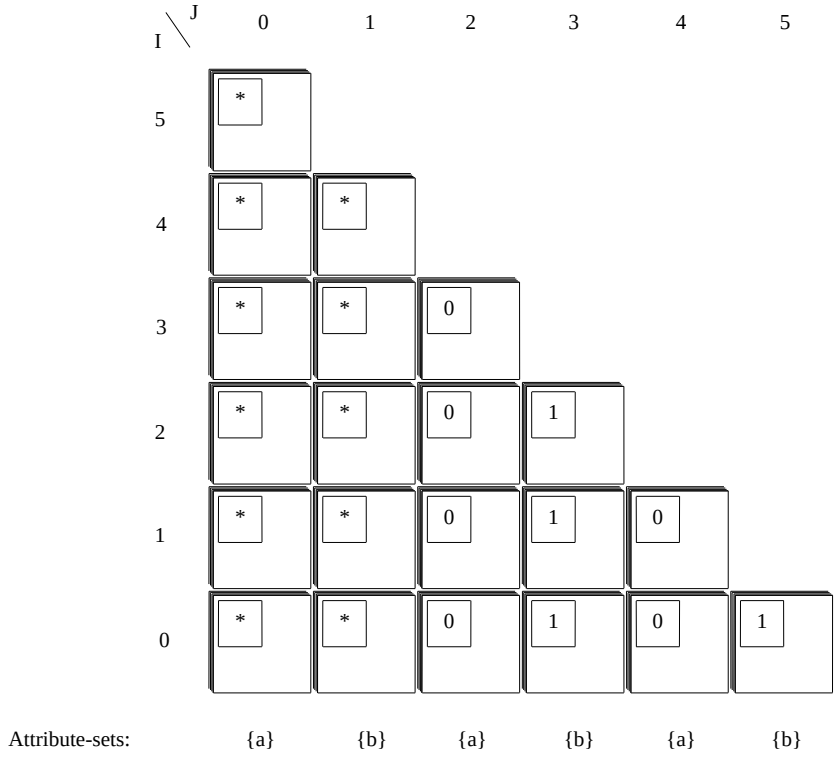


Figure 5.1: *Preprocessing done for a sentence of length 6.*

to do the following, if we use a production from some data, pointed to by a pointer whose flag has been set. Assume we use a production extracted from the data pointed to by pointer P(*i*,*j*). Assume its flag is set to *k* (*k* < *j*). Then if the production is $\mathbf{p}_\pi$, simulate the indices of the constituting operators of π to be adjusted by *j* - *k*. The algorithm is illustrated in Fig. 5.1 for attribute-sets ({a},{b},{a},{b},{a},{b}). The "*" in each cell, means that the pointer for this cell refers to original data. The numbers in each cell are the flags set to the specified elements in the same row.

Step (1) of the algorithm is $O(n^2)$. In step (2) *i*, *j* and *k* are all bounded by *n*, so this is $O(n^3)$. The preprocessing does not increase the complexity of the CYK-algorithm, but doing the preprocessing before the very first run of the algorithm will increase the number of computational operations. We could however also decide to run the program at the point, where the error recovery algorithm is invoked, without slowing down error detection.

Of course, the preprocessing is useless, if there are no equal subsentences. Therefore, not much is gained, if we would run it on short sentences in the natural-language, because there is a small change that a word will appear twice. Even most large subsentences, such as the one before this one, only the words "is" and "a" appear twice (there are four comma's though). It could well be possible that the optimization applies to some languages more than to others. For example, in the Indonesian language every plural is formed by repeating the

noun: "babi babi" = "pigs".

5.2. Providing context information to the spelling corrector

As mentioned earlier, a spelling corrector will be introduced as part of the proof-reader program in the future. All details about it are documented in [van Eerden & Gouweleeuw 89]. It has a large variety of ways to correct the input sentence at the lexical level. Because it only looks to correct one word of the sentence at the time, it cannot use the context in which an incorrect word appears. One way in the spelling corrector to correct words is to assume errors in words are made at the typewriter and try to replace letters, appearing in the word, by letters that are physically near to them at the typewriter. For example, the incorrect word "dpg", might be corrected to "dog", since the "p" and "o" are next to each other on the typewriter-keyboard. On the syntactical level the word "dog" is a noun (its attribute-set consists of "noun"-like attributes). Now if the misspelled word appeared originally in the sentence:

```
I dpg a hole.
```

then, if the error-recovery module was fed the sentence:

```
I dog a hole.
```

it would find a correction of length one, by replacing the "noun"-like attribute set by a "verb"-like attribute. Its output would effectively be:

Rep(1, "verb_like_attribute")

(remember that the second word is referred to by index 1). If there will be some way in the future to feed this information back to the spelling corrector, it would have a major clue to what the word should have been. It could look up its vocabulary and find a verb "dig", differing only one letter from the incorrect word. An even more advanced example can be given if we are considering the sentence:

```
I see a hand some man.
```

Now the spelling corrector does not find an incorrect word. The error-recovery algorithm might correct the sentence by replacing the attributes of the words "hand" and "some" by an "adjective"-like attribute. If the spelling corrector would be very smart, it should deduct, that a most probable way to correct the sentence is to delete the space between "hand" an "some".

ACKNOWLEDGEMENT

Thanks to the originator of the Automatic Proof-reader, Dick Grune, for encouragement, optimism, approval but, most of all, patience.

REFERENCES

[Altena & Ott de vries 87] Altena, T. & Ott de Vries, T., *Een parser voor natuurlijke taal, Vrije Unversiteit, 1987, (master-thesis).*

[van Eerden & Gouweleeuw 89]van Eerden, F. & Gouweleeuw, R., *A learning user-dependent spelling corrector, Vrije Unversiteit, 1989, (master-thesis).*

[Hopcroft & Ullman 79] Hopcroft, J. E. & Ullman, J. D., *Introduction to Automata Theory, Languages and Computation,* Addison-Wesley Publ. Comp.

[Lyon 74] Lyon, G., *Syntax-directed least-errors analysis for context-free languages: a practical approach,* Commun. ACM, Vol. 17, pp. 3-14.

[Peterson & Aho 72] Aho, A.V. & Peterson T.G., *A minimum-distance error-correcting parser for context-free languages,* SIAM J. Comput. Vol. 1, pp. 305-312.

[Younger 67] Younger, D. H., *Recognition and parsing of context-free languages in time* n^3, Information and Control 10:2, pp. 189-208.

APPENDIX A

We present some example incorrect sentences in a dutch grammar and their possible correction chains generated by the error-recovery module.

> jij geef.

corrections: 1

Repl(1, zww_geenM_geenL_tweede_enkelvoud)

Repl(0, persoonlijkvoornaamwoord_eerste_enkelvoud_nominatief)

chains: 2

> jij wacht op de jongens maar ik wacht op de meisjes.

corrections: 1

Repl(5, "en")

Ins(5, "komma")

chains: 2

> ik en jou de hele snelle trein en jij geeft mij treinen en.

corrections: 2

Repl(1, zww_Mvoor_Lv_eerste_enkelvoud) *
Del(12)

Repl(1, zww_Mvoor_Lv_eerste_enkelvoud) *
Repl(12, bijwoord)

Repl(1, zww_Maan_Lv_eerste_enkelvoud) *
Del(12)

Repl(1, zww_Maan_Lv_eerste_enkelvoud) *
Repl(12, bijwoord)

chains: 4

> ik en jou treinen jij geeft hem treinen hij geeft mij treinen.

corrections: 3

Repl(1, zww_geenM_Lv_eerste_enkelvoud) *
Repl(3, "en") *
Repl(7, "en")

Repl(1, zww_geenM_Lv_eerste_enkelvoud) *
Repl(3, "en") *
Ins(8, "en")

Repl(1, zww_Mvoor_Lv_eerste_enkelvoud) *
Ins(4, "en") *
Repl(7, "en")

Repl(1, zww_Mvoor_Lv_eerste_enkelvoud) *
Ins(4, "en") *
Ins(8, "en")

Repl(1, zww_Maan_Lv_eerste_enkelvoud) *
Ins(4, "en") *
Repl(7, "en")

Repl(1, zww_Maan_Lv_eerste_enkelvoud) *
Ins(4, "en") *
Ins(8, "en")

chains: 6

>

APPENDIX B

In this appendix the performance of the algorithm for various sentence lengths and corrections is given. The algorithm was implemented in C and performance was measured on a SUN-4, SPARC-station, running at about 16 Mips.

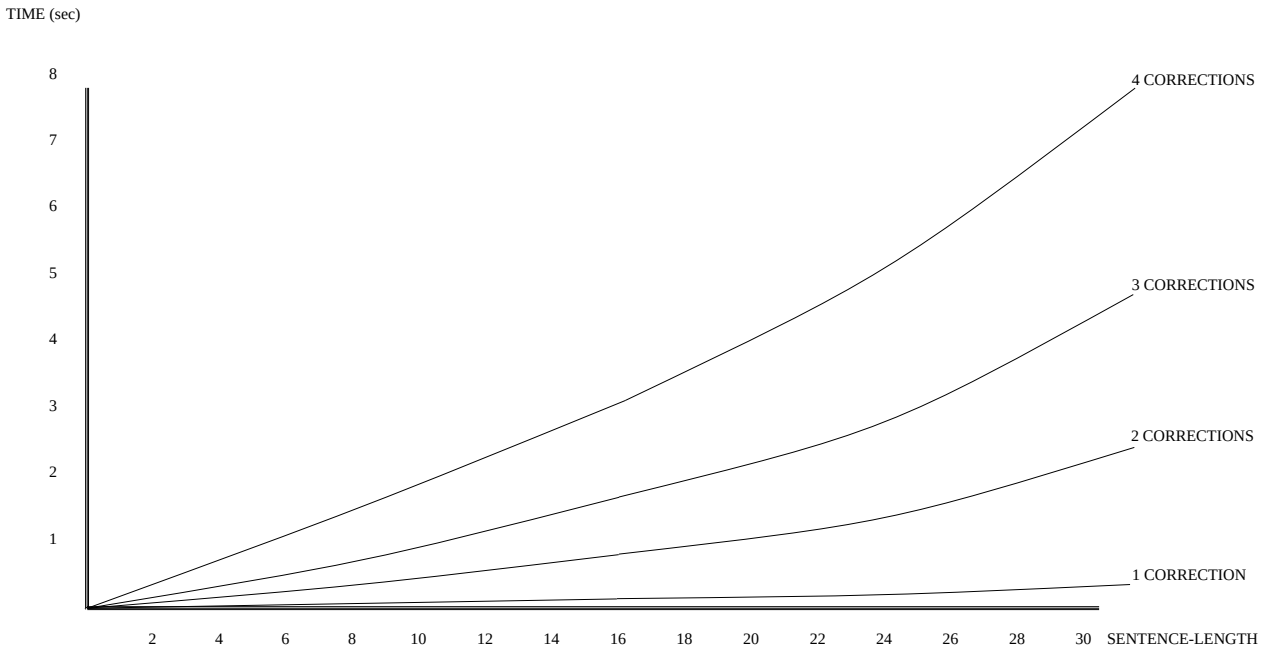


Figure B1:*Performance of the algorithm.*

The graphs show the average time consumption for sentences of a given length. Rather than giving the exact (somewhat spiked) performance graphs, the results were smoothened in order to highlight the global performance. Clearly the algorithm starts being slower as the sentence length grows and more than three corrections have to be made in a long sentence. For one correction, however, it remains very fast. Time consumption also depends on the number of chains that can correct the sentence, but since the backtracing algorithm is very fast this does not change the performance spectacularly.