

# Are Efficient Natural Language Parsers Robust?

Kyongho Min & William H. Wilson  
School of Computer Science & Engineering  
University of New South Wales  
Sydney NSW 2052 Australia  
Email: {min, billw}@cse.unsw.edu.au

## Abstract

This paper discusses the robustness of four efficient syntactic error-correcting parsing algorithms that are based on chart parsing with a context-free grammar. In this context, by *robust* we mean able to correct detectable syntactic errors. We implemented four versions of a bottom-up error-correcting chart parser: a basic bottom-up chart parser, and chart parsers employing selectivity, top-down filtering, and a combination of selectivity and a top-down filtering. The combined selectivity and top-down filtering parser was the most efficient. However, this parser failed to correctly repair more sentences than the other parsers, failing on 18 out of 119 ill-formed sentences, compared to no failures for the basic bottom-up chart parser. This paper examines trade-offs between parsing efficiency and robustness at the syntactic level.

*Keywords:* robust parsing, ill-formed text, selectivity, top-down filtering, spelling correction.

## 1. Introduction

This paper discusses the robustness of efficient syntactic parsing algorithms based on chart parsing with a context-free grammar. In the 1970s, efficient CFG-based parsing algorithms that employ selectivity (Aho & Johnson, 1974; DeRemer, 1971), top-down filtering (Pratt, 1975), or the combination of selectivity and top-down filtering (Pratt, 1975), were applied to natural language parsing. These systems were implemented for parsing *well-formed* sentences. Mickunas & Modry (1978) described automatic error recovery using an LR parser. Their recovery parser used two phases: an error detection phase (the *condensation phase*) and an error correction phase (the *correction phase*) using a single LR parser for processing programming language texts.

In the domain of chart parsing, the efficiency of a parser can be estimated by the number of chart objects (i.e. active or inactive arcs) it produces (Wiren, 1987). However, if a text includes an error, a parser employing an algorithm that is efficient in this sense may not produce sufficient information to allow recovery of the ill-formed sentence.

When parsing a sentence, a single parser can handle both a well-formed and an ill-formed sentence (Weischedel & Sondheimer (1983) called this *one-stage error recovery*). Our system employs two parsers: a parser for parsing well-formed sentences and a second parser for recovering from ill-formedness. The first parser produces a chart even with ill-formed sentences. This is called *two-stage error recovery* (Mellish, 1989).

Following Mellish's approach (1989), Kato (1994) applies eight rules to error detection and gives different priorities to each rule, using two parameters. Our system employs both top-down expectation and bottom-up satisfaction, like Mellish, but avoids using a complex selection scheme to which of six rules to use in a given situation. In section 2, we shall describe our four chart parsing algorithms. We tested these with 5 types of errors (replacement of an unknown/known word, insertion of an unknown/known word, and deletion of a word) in sentences of length 3, 5, 7, and 11. In section 3, we describe how to detect and correct errors at both lexical and syntactic levels. In section 4, we describe test results with the four parser versions. In the final section, we present conclusions and problems.

## 2. Implementation of Efficiency-Enhanced Chart Parsers

Our basic parser WFSCP (Well-Formed Sentence Chart Parser) is a bottom-up chart-based parsing algorithm. We have implemented four versions of WFSCP: (i) WFSCP<sub>1</sub>, a plain

version of bottom-up chart parser; (ii) WFSCP<sub>2</sub>, which incorporates selectivity; (iii) WFSCP<sub>3</sub>, which incorporates top-down filtering; and (iv) WFSCP<sub>4</sub>, which has both selectivity and top-down filtering. When parsing a sentence (whether well-formed or ill-formed), all four versions incorporate the same set of techniques to reduce chart size by avoiding useless arc creation and extension. Wiren (1987) showed that the combination of top-down filtering & selectivity (cf. LCK<sub>St</sub>), similar to our WFSCP<sub>4</sub>, is the most efficient of the chart parsers that he studied. The "No Error" row of Table 1 partially confirms Wiren's finding.

| <i>Error Type</i>    | <i># of Sentences</i> | <i>WFSCP<sub>1</sub></i> | <i>WFSCP<sub>2</sub></i> | <i>WFSCP<sub>3</sub></i> | <i>WFSCP<sub>4</sub></i> |
|----------------------|-----------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| No Error             | 4                     | 688                      | 616                      | 494                      | 480                      |
| Unknown Word Errors  | 56                    | 16487                    | 15567                    | 13562                    | 12947                    |
| Known Word Errors    | 44                    | 15342                    | 12540                    | 11268                    | 10277                    |
| Word Deletion Errors | 19                    | 6466                     | 4466                     | 3501                     | 3138                     |
| Total                | 123                   | 38983                    | 33189                    | 28855                    | 26842                    |
| Average              | –                     | 317                      | 270                      | 235                      | 218                      |

**Table 1.** Total number of chart items (= active/inactive arc + need-arcs + repaired constituents) generated by WFSCP+IFSCP for 123 sentences (Time taken is proportional to number of chart items: 57 chart items are produced per second on a Mac IIsi.).

When an ill-formed sentence includes a misspelt word, then the pure versions of both selectivity and top-down filtering parsers are blocked after that word. Thus in this case, we use modified versions of the algorithm. In the case of selectivity parsing, the expected lexical constituent requirements (selective constituent constraint) are as described below:

Normally with selectivity, only the "expected" preterminal categories can be involved in rule invocation process. When the selectivity-based parser encounters a known word, it normally intersects the lexical categories of the word as per the lexicon with the lexical categories expected by the parser at this point, and builds constituents for each category in this intersection. After "skipping over" an unknown word, it has no expected categories, so it must rely just on the lexicon to decide which lexical constituents to build.

Second, in top-down filtering parsing, the expected phrasal constituent requirements (filtering constituent constraint) are modified by a specific rule invocation phase.

In top-down filtering parsing, only a rule whose LHS (Left-Hand Side) is a member of "expected" phrasal constituents (e.g. filtering constituents) is invoked and builds constituents for each category of an input string. This is called the filtering constituent constraint. When the parser encounters an unknown word, the word is skipped over and the lexical category information for the next word, taken from the lexicon, is used to decide which rule to invoke.

For example, given the partial string " 0 a 1 *bif* 2 boy 3", the selectivity version, WFSCP<sub>2</sub>, is blocked at 2 because "bif" is an unknown word. In this case, no expected lexical constituents are generated. Thus the parser builds constituents using the lexical information for "boy" without applying any selective constituent constraint. When parsing the same string with WFSCP<sub>3</sub> (top-down filtering), the parser invokes relevant rules using the lexical information for "boy" without applying any filtering constituent constraint.

### 3. Error Correction and Detection

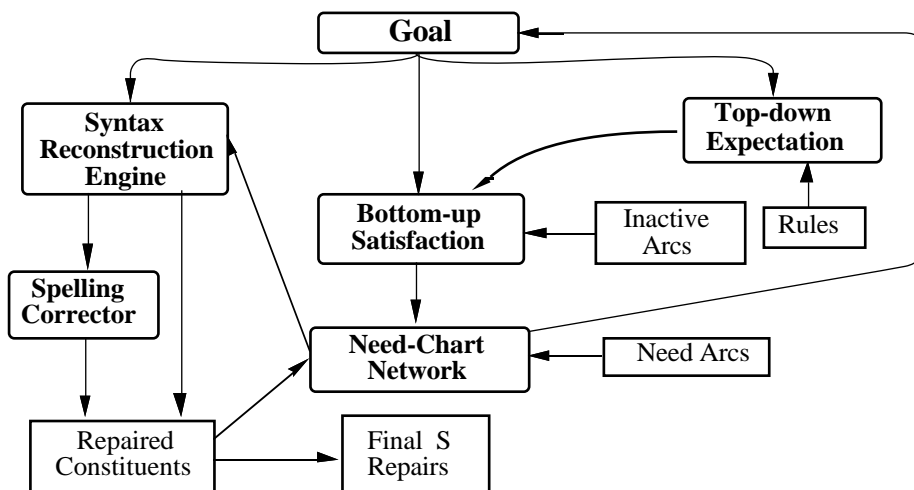
We detect and correct syntactic errors using a system component called IFSCP (Ill-Formed Sentence Chart Parser) described by Min & Wilson (1994), together with a spelling correction module. If WFSCP can not produce any constituents of type S covering the whole sentence, then the sentence is classed ill-formed and the system invokes another parser, IFSCP, to locate and repair the error. Thus our system performs *two-stage error recovery* (Mellish, 1989).

Unlike the different versions of WFSCP described above, the system employs only one version of IFSCP. The IFSCP repairs an ill-formed sentence in three phases: *goal-generation* and *top-down expectation*, *bottom-up satisfaction* and production of a need-chart network, and *syntax*

*reconstruction* by retracing the need-chart network plus *spelling correction* (see Fig. 1) (Min & Wilson, 1994). The first two phases are for error detection, the last phase is for correction.

### 3.1 Syntactic error detection and correction

If a sentence is recognised as ill-formed, then the system hypothesises that the sentence is well-formed with a single error at the syntactic level. The system starts by generating an initial goal for the whole sentence (an S node), viewed as a local tree which must include a syntactic error. After a goal is generated, the goal is expanded to localise the error in a sub- local tree by invoking rules. This is called the *topdown expectation* phase. After expanding a goal, either a leftmost or rightmost constituent of the expanded goal is sought for, using inactive constituents left behind by WFSCP, and a *need-arc* is produced using information from both the expanded goal and found constituents. The need-arc includes information about which constituents are found and which are not (i.e. needed constituents) between two positions in an input string, together with its parent need-arc, that is, a previous need-arc involved in the production of this need-arc, and which allows access to the history of derivation of this need-arc from the first goal S). This is called a *bottom-up satisfaction* phase.



**Figure 1.** The Structure of IFSCP.

When an error is detected, whether during a goal generation phase or a bottom-up satisfaction phase, the *syntax reconstruction phase* is invoked to correct the error. When a repair is suggested, a **deviance note** is made: this describes the details of the error correction, and the parent data structure (a need-arc), which produced the repair, is used to continue retracing the need-chart network up to the initial goal node S.

Our algorithm for error detection and correction is as follows:

**to** process the sentence

**perform** basic chart parser;

**if** the input sentence cannot be parsed by the basic chart parser

**then** it is deemed ill-formed, and a first goal, `goal1` of type S, is generated;

`goal1` is stored in the agenda variable, `*goal*`;

**perform** Detect and Correct

**end if**

**end** process

**to** Detect and Correct

*goal generation phase:*

**note:** the structure `*goal*` is the set of constituents needed to achieve error correction

**if** first item in `*goal*` satisfies the condition for a substitution or deletion error,

i.e. \*goal\*'s ps is either 0 (substitution) or -1 (deletion),  
**then goto** *preterminal error correction* **end if**  
**if** \*goal\* is a single constituent (e.g. (NP))  
**then** generate a goal (e.g. by expanding a rule for NP) to further localize the errors  
add it to \*goal\*  
**else goto** *bottomup satisfaction phase* **end if**  
*top-down expectation phase:*  
**while** there \*goal\* is not empty **do**  
get a goal from \*goal\*  
**perform** expanding the goal by invoking corresponding rules  
**if** the goal's label is the same as the LHS of a rule (e.g. R1 for goal1)  
**then** invoke all rules and expand the goal (e.g. T1) and  
**goto** *bottomup satisfaction* with invoked rules recursively  
**end if**  
**end while**  
*bottom-up satisfaction phase:*  
**let** needed constituents be from either *topdown expectation phase* or *goal generation phase*  
**perform** search for the leftmost or rightmost constituents of needed constituents from inactive arcs  
**if** the leftmost or rightmost constituents are found  
**then goto** *need-chart network phase* with found constituents and needed constituents  
**end if**  
**if** needed constituent is a single lexical constituent and no inactive arc is found **or**  
needed constituent is a single phrasal constituent with MEL=1 (see Fig. 2) and no inactive arc is found  
**then goto** *goal generation phase* with the needed constituent  
**else** discard the needed constituents because it is not satisfied *bottomup satisfaction phase*.  
**end if**  
*need-chart network phase:*  
make a need-arc (like narc1 in the example) to add to the need-chart network  
**if** needed constituents are left (like narc1 in the example)  
**then goto** *goal generation phase*  
**else if** needed constituent is empty and the need-arc covers an extra word (e.g. from *i* to *i+1*, like narc4 in Fig. 2).  
**then perform** Correct Preterminal Errors to repair an error of an extra word added  
**end if**  
**end while**  
**end** Detect and Correct  
**to** Correct Preterminal Errors  
**perform** *bottommost error correction*  
**case1** replacement error (needed constituent is a single lexical constituent, penalty score 0)  
**if** a goal from *goal generation phase* satisfies substitution error  
**then** generate a new correct constituent using the goal's information  
**case2** deletion error (needed constituent is a single lexical constituent, penalty score -1)  
**if** a goal from *goal generation phase* satisfies deletion error  
**then** generate a new correct constituent using the goal's information  
**case3** addition error (the need-arc's needed constituent is empty and its penalty score is 1)  
**if** a need-arc from *need-chart network phase* satisfies addition error  
**then** generate a new correct constituent using the need-arc's information  
store the correct constituents into the reconstruction agenda, \*stack\*  
**end** Correct Preterminal Errors

**to Reconstruct Syntax**

**note** the goal of this section is *reconstruction of the S constituent*

**while** \*stack\* is not empty

pop a repaired constituent from \*stack\*

get the parent need-arc (parc) of the repaired constituents

**perform** Retrace Need-Arcs(repaired constituent, parent need-arc)

**end while**

**end Reconstruct Syntax**

**to Retrace Need-Arcs(repaired constituent, parent need-arc)**

generate a new repaired constituent using the parent need-arc and the repaired constituent

**unless** the parent need-arc has no parent **do**

**perform** Retrace Need-Arcs(newly repaired constituent, parent of the parent need-arc)

**end unless**

**end Retrace Need-Arcs**

**An Example**

0 the 1 bif 2 boy 3 has 4 a 5 dog 6

(1) *Context-free grammar*

R1: S → NP VP (\*mel = 2)

R2: NP → PRON (mel = 1)

R3: NP → DET NOUN (mel = 2)

R4: NP → DET ADJP NOUN (mel = 3)

R5: VP → VERB (mel = 1)

R6: VP → VERB NP (mel = 2)

R7: VP → BE NP (mel = 2)

R8: ADJP → ADV ADJ (mel = 2)

R9: ADJP → ADJ (mel = 1)

\*mel (minimal extension length) describes minimal number of preterminal categories which are necessary for the production of the rules LHS category. For example, the mel of S is 2, because of examples like *I go*.

(2) *Goal Generation Phase*

goal1: constituent S needs from 0 to 6 (\*ps = 4, \*\*parc = nil)

goal2: constituent NP needs from 0 to 3 (ps = 2, parc=narc1)

goal3: constituent ADJP needs from 1 to 2 (ps=0, parc=narc5)

goal4: constituent AdJ needs from 1 to 2 (ps=0, parc=narc5)

\*ps (penalty score of this goal) = total number of strings covered with the goal (6 strings = to (6) - from (0)) minus the mel (2) of goal's label.

\*\*parc (parent narc) = when this goal was generated, which narc is involved in the goal's generation.

(3) *Topdown Expectation*

T1: label=S, (NP VP) needs from 0 to 6, (parc= nil)..... from goal1 + R1

T2: label=NP, (PRON) needs from 0 to 3, (parc=narc1) ..... R2 + goal2

T3: label=NP, (DET NOUN) needs from 0 to 3, (parc=narc1) ..... R3 + goal2

T4: label=NP, (DET ADJP NOUN) needs from 0 to 3, (parc=narc1) ... R4 + goal2

T5: label=nil, (NOUN) needs from 1 to 3, (parc=narc2) ..... narc2

T6: label=nil, (ADJP NOUN) needs from 1 to 3, (parc=narc3) ..... narc3

T7: label=ADJP, (ADV ADJ) needs from 1 to 2, (parc=narc5) ..... R8 + goal3

T8: label=ADJP, (ADJ) needs from 1 to 2, (parc=narc5) ..... R9 + goal3

(4) *Bottom-up satisfaction & need-chart network*

B1: VP3 (“has a dog”) found from 3 to 6 (for T1)

B2: DET1 (“the”) found from 0 to 1 (for T2 T3 T4)

B3: NOUN2 (“boy”) found from 2 to 3 (for T5 T6)

B4: no constituent for ADJ or ADV is found from 1 to 2

|  |                    |
|--|--------------------|
| <i>(5) Need-Chart-Network</i>  |                    |
| narc1: (S → VP3 * NP) needs from 0 to 3, parc=nil .....  | T1 + B1            |
| (narc1 means that VP3 is found from the rhs of a local tree S → (NP VP) and unfound constituent NP needs from 0 to 3).             |                    |
| narc2: (NP → DET1 * NOUN) needs from 1 to 3, parc=narc1 .....  | T3 + B2            |
| narc3: (NP → DET1 * ADJP NOUN) needs from 1 to 3, parc=narc1 ...   | T4 + B2            |
| narc4 (NP →DET1 NOUN2 * NIL) needs from 1 to 2, parc=narc1 .....   | T5 + B3            |
| narc5 (NP →DET1 NOUN2 * ADJP) needs from 1 to 2, parc=narc1 ...  | T5 + B3            |
| narc6:(ADJP → *ADJ) needs from 1 to 2, parc=narc5 .....  | T8 + B4            |
| <i>(6) Syntax Reconstruction</i>   |                    |
| NP3→ (DET1 NOUN3) is produced from 0 to 3, parc=narc1<br>(delete a word "bif" from 1 to 2) .....                                   | from narc4         |
| S1→ (NP3 VP3) is produced from 0 to 6 .....  | from NP3 + narc1   |
| ADJ2 → (ADJ "bif") is replaced from 1 to 2, parc=narc6<br>(replaced a word "bif" from 1 to 2 into a preterminal category ADJ.).... | from goal4         |
| ADJP3 → (ADJ2) is produced from 1 to 2, .....  | from ADJ2 + narc6  |
| NP4 → (DET1 ADJP3 NOUN2) is produced from 0 to 3 .....   | from ADJP3 + narc5 |
| S2 → (NP4 VP3) is produced from 0 to 6 .....   | from NP4 + narc1   |

Figure 2. Example data structures during Error Correction and Detection.

### 3.2 Spelling Detection and Correction

Vosse (1992) sought to detect and correct a misspelt word error at the morpho-syntactic level. If a parser recognises a word as misspelt, his system invokes spelling correction first and suggests the best correction without syntactic information. If the best correction does not produce a parse tree (S), syntactic recovery by top-down parsing is invoked to correct the misspelt word again. In our system the spelling-corrector is invoked, with syntactic information, only after a substitution error is corrected by the syntax reconstruction engine.

If the type of a local error detected by IFSCP is substitution of a word (e.g. either a legal word or a misspelt word), then the **spelling-corrector** subsystem is invoked to correct the spelling of the word. The spelling correction algorithm is based on Damerau's study (1964) and employs a strategy of dictionary lookup, during which syntactic information is obtained. It retrieves all words obtainable from the erroneous word by single letter substitution/addition/deletion, or by transposition of two adjacent letters, and that satisfy the known syntactic constraints. For each such word, a repaired constituent is created, and the repair is pursued up to the full sentence level. As there are often many corrections for short words, this can lead to large numbers of syntax trees.

## 4. Results of Four Chart Parsers

We tested four various versions of bottom-up chart parsers: (i) WFSCP<sub>1</sub> (a normal bottom-up chart parser without a particular rule invocation strategy), (ii) WFSCP<sub>2</sub> (selectivity), (iii) WFSCP<sub>3</sub> (top-down filtering), and (iv) WFSCP<sub>4</sub> (selectivity plus top-down filtering). The algorithms were coded in Macintosh Common Lisp 2.0, and testing was done on a Macintosh IIsi with 9 MB of memory. We tested 4 different lengths of sentences (3, 5, 7, and 11) and 5 different error types, with a grammar of 210 context-free rules designed to parse a simple declarative sentence with no conjunctions, passivisation, or relative clauses. Each basic test sentence had a single error introduced into it in all possible locations and of each of the five error types, giving rise to large numbers of test sentence from each basic sentence. An error

was deemed to be correctly repaired if one of the repairs found (if any) consisted of reversing the transformation used to create the error - e.g. deleting a word that had been inserted.

In the case of test sentences with an unknown word error, WFSCP naturally recognised the sentences as ill-formed. However, some test sentences with other types of error are parsed as well-formed although they may be ill-formed at the semantic level. For example, if the word *big* is deleted from the string *a big boy*, then the error cannot be detected. The five error types we tested were: substitution of an unknown word (26 sentences), addition of an unknown word (30 sentences), substitution of a known word (26 sentences), addition of a known word (30 sentences), and deletion of a word (26 sentences). This made a total of 138 sentences, which were fed to each of WFSCP<sub>1-4</sub>. Nineteen of the 138 sentences (14%) were parsed as well-formed (Table 3). Eight of 19 sentences arise from addition of a known word, seven from substitution of a known word, and four from deletion of a word.

With unknown word errors, 3 of 224 sentences across four parser versions are not repaired because of insufficient syntactic information for recovery. With the other types of error, 13 of 328 sentences (4%) are not repaired. All unknown word errors that are repaired at all are repaired correctly, but with other types of error, 31 of 328 sentences (9%) are not repaired correctly. Most of the incorrect repairs (28 of the 31) are to sentences produced by deletion of a word (see Table 2).

| Error type              | Number of testing sentences | Number of errors detected | Number of unrepaired sentences | Number of correct repairs | Number of incorrect repairs |
|-------------------------|-----------------------------|---------------------------|--------------------------------|---------------------------|-----------------------------|
| Substitute Unknown Word | 104                         | 104(100%)                 | 2(2%)                          | 102(98%)                  | 0                           |
| Add Unknown Word        | 120                         | 120(100%)                 | 1(1%)                          | 119(99%)                  | 0                           |
| Substitute Known Word   | 104                         | 88(85%)                   | 3(3%)                          | 83(80%)                   | 2(2%)                       |
| Add Known Word          | 120                         | 88(73%)                   | 4(3%)                          | 83(69%)                   | 1(1%)                       |
| Delete a Word           | 104                         | 76(73%)                   | 6(6%)                          | 42(40%)                   | 28(27%)                     |

**Table 2.** Result summaries for the five error types.

The charts produced by WFSCP<sub>1-4</sub> contain different information, and so IFSCP performs differently with them as input. With the chart produced by WFSCP<sub>2</sub>, 4 out of 138 test sentences cannot be repaired, and a further 9 are repaired incorrectly. For WFSCP<sub>3</sub>, the figures are 5 out of 138 sentences unrepaired, and a further 11 incorrectly repaired. For WFSCP<sub>4</sub>, 7 of 138 sentences cannot be repaired, and again a further 11 are incorrectly repaired (see Table 3).

| Version            | Number of testing sentences | Number of errors detected | Number of unrepaired sentences | Number of correct repairs | Number of incorrect repairs |
|--------------------|-----------------------------|---------------------------|--------------------------------|---------------------------|-----------------------------|
| WFSCP <sub>1</sub> | 138                         | 119 (86%)                 | 0                              | 119 (86%)                 | 0                           |
| WFSCP <sub>2</sub> | 138                         | 119 (86%)                 | 4 (3%)                         | 106 (77%)                 | 9 (6%)                      |
| WFSCP <sub>3</sub> | 138                         | 119 (86%)                 | 5 (3%)                         | 103 (75%)                 | 11 (8%)                     |
| WFSCP <sub>4</sub> | 138                         | 119 (86%)                 | 7 (5%)                         | 101 (73%)                 | 11 (8%)                     |

**Table 3.** Result summaries for the four parser versions.

In terms of the number of items in the charts, WFSCP<sub>4</sub> is the most efficient of the four versions (see Table 1). However, 18 of 138 sentences (13%), were not repaired or incorrectly repaired using the chart produced by WFSCP<sub>4</sub> and 19 are undetected. The least efficient version, WFSCP<sub>1</sub>, gives the largest number of repaired sentences: it correctly repairs all testing sentences except for those whose errors are undetectable (19 of 138 sentences (14%)). WFSCP<sub>2</sub> doesn't repair or incorrectly repairs another 13 of the 138 sentences (9%), while WFSCP<sub>3</sub> doesn't repair or incorrectly repairs another 16 of the 138 sentences (11%) (see Table 3).

## 5. Conclusions

The purpose of efficiency-enhanced parsers (e.g. lookahead, top-down filtering) is to generate only necessary syntactic information so that fewer useless constituents and arcs are produced. However, when the domain of the parser includes ill-formed sentences, we found that the more efficient parsing algorithms cannot handle them as robustly as a more basic parser. This is because, with our two-stage error-correction strategy, the second-stage parser (13% of our test cases) gains benefits from the extra information in a "less efficient" chart.

*Critical appraisal of results:* Our results are to some extent dependent on the test sentence and grammar used. If our grammar were expanded, then the results with the various systems would change. Further, our system did not prevent the invocation of useless rules during top-down expectation. If a finer rule invocation scheme were employed (cf. Wood, 1982), then the number of false corrections might be reduced. Our system also employed a spelling-correction module: if the size of lexicon were increased, then the number of repaired local trees and top-level trees would be increased too, to some extent. There is an argument for also using non-synthetic data for testing the parsers, as "real-life" ill-formed sentences will give a more practical evaluation of parser performance (most errors seem to be substitutions of unknown words, real-life data would also cover a wider range of syntactic configurations). On the other hand, synthetic data allow us to torture-test the parsers by ensuring errors occur in all parts of the sentence and by providing a balanced sample of error-types - e.g. plenty of the rather rare "add known word" errors, in all positions. Finally, a type-specific one-stage error recovery scheme might be employed for unknown word errors using selectivity algorithm (Vosse, 1992).

## References

- [1] Aho, A. & Johnson S. (1974). LR Parsing. *Computing Survey*, **6**(2): 99-124.
- [2] Damerau, F. (1964). A Technique for Computer Detection and Correction of Spelling Errors. *CACM*, **7**(3): 171-176.
- [3] DeRemer, F. (1971). Simple LR(k) Grammars. *CACM*, **14**(7): 453-460.
- [4] Kato, T. (1994). Yet Another Chart-Based Technique for Parsing Ill-formed Input. *Proceedings, Fourth Conference on Applied Natural Language Processing*, 107-112.
- [5] Mellish, C. (1989). Some Chart-based Techniques for Parsing Ill-formed Input. *ACL Proceedings, 27th Annual Meeting*, 102-109.
- [6] Mickunas, D. & Modry, J. (1978). Automatic Error Recovery For LR Parsers. *CACM*, **21**(6): 459-465.
- [7] Min, K. & Wilson, W. (1994). Chart Parser for Ill-formed Input Sentences. *Language Teaching and Research*, 23, 141-154, Language Research Center, Chonnam National University: Kwangju.
- [8] Pratt, V. (1975). LINGOL- A Progress Report. *IJCAI-75*, **1**: 422-428.
- [9] Vosse, T. (1992). Detecting and Correcting Morpho-syntactic Errors in Real Texts. *ACL Proceedings, Third Conference on Applied Natural Language Processing*, 111-118.
- [10] Weischedel, R. & Sondheimer, N. (1983). Meta-rules as a Basis for Processing Ill-formed Input. *AJCL*, **9**(3-4): 161-177.
- [11] Wren, M. (1987). A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing. *ACL Proceedings, third European Conference*, 226-223.
- [12] Wood, W. (1982). Optimal Searching Strategies for Speech Understanding Control. *Artificial Intelligence*, **18**: 295-326.