TTP: A FAST AND ROBUST PARSER FOR NATURAL LANGUAGE

TOMEK STRZALKOWSKI

Courant Institute of Mathematical Sciences
New York University
715 Broadway, rm 704
New York, NY 10003
tomek@cs.nyu.edu

ABSTRACT

In this paper we describe TTP, a fast and robust natural language parser which can analyze written text and generate regularized parse structures for sentences and phrases at the speed of approximately 0.5 sec/sentence, or 44 word per second. The parser is based on a wide coverage grammar for English, developed by the New York University's Linguistic String Project, and it uses the machine-readable version of the Oxford Advanced Learner's Dictionary as a source of its basic vocabulary. The parser operates on stochastically tagged text, and contains a powerful skip-and-fit recovery mechanism that allows it to deal with extra-grammatical input and to operate effectively under a severe time pressure. Empirical experiments, testing parser's speed and accuracy, were performed on several collections: a collection of technical abstracts (CACM-3204), a corpus of news messages (MUC-3), a selection from ACM Computer Library database, and a collection of Wall Street Journal articles, approximately 50 million words in total.

1. INTRODUCTION

Recently, there has been a growing demand for fast and reliable natural language processing tools, capable of performing reasonably accurate syntactic analysis of large volumes of text within an acceptable time. A full sentential parser that produces complete analysis of input, may be considered reasonably fast if the average parsing time per sentence falls anywhere between 2 and 10 seconds. A large volume of text, perhaps a gigabyte or more, would contain as many as 7 million sentences. At the speed of say, 6 sec/sentence, this much text would require well over a year to parse. While 7 million sentences is a lot of text, this much may easily be contained in a fair-sized text database. Therefore, the parsing speed would have to be increased by at least a factor of 10 to make such a task manageable.

In this paper we describe a fast and robust natural language parser that can analyze written text and generate regularized parse structures at a speed of below 1 second per sentence. In the experiments conducted on variety of natural langauge texts, including technical prose, news messages, and newspaper articles, the average parsing time varied between 0.4 sec/sentence and 0.7 sec/sentence, or between 1600 and 2600 words per minute, as we tried to find an acceptable compromise between parser's speed and precision.[1]

It has long been assumed that in order to gain speed, one may have to trade in some of the parser's accuracy. For example, we may have to settle for partial parsing that would recognize only selected grammatical structures (e.g. noun phrases; Ruge et al., 1991), or would avoid making difficult decisions (e.g. pp-attachment; Hindle, 1983). Much of the overhead and inefficiency comes from the fact that the lexical and structural ambiguity of natural language input can only be dealt with using limited context information available to the parser. Partial parsing techniques have been used with a considerable success in processing large volumes of text, for example AT&T's Fidditch (Hindle and Rooth, 1991) parsed 13 million words of Associated Press news messages, while MIT's parser (de Marcken, 1990) was used to process the 1 million word Lancaster/Oslo/Bergen (LOB) corpus. In both cases, the parsers were designed to do partial processing only, that is, they would never attempt a complete analysis of certain constructions, such as the attachment of pp-adjuncts, subordinate clauses, or coordinations. This kind of partial analysis may be sufficient in some applications because of a relatively high precision of identifying correct syntactic dependencies.[2] However, the ratio at which these dependencies are identified (that is, the recall level) isn't sufficiently high due to the inherently partial character of the parsing process. The low recall means that many of the important dependencies are lost in parsing, and

[1] These results were obtained on a 21 MIPS SparcStation ELC. The experiments were performed within an information retrieval system so that the final recall and precision statistics were used to measure effectiveness of the parser.

[2] Hindle and Rooth (1991) and Church and Hanks (1990) used partial parses generated by Fidditch to study word co-occurrence patterns in syntactic contexts.

therefore partial parsing may not be suitable in applications such as information extraction or document retrieval.

The alternative is to create a parser that would attempt to produce a complete parse, and would resort to partial or approximate analysis only under exceptional conditions such as an extra-grammatical input or a severe time pressure. Encountering a construction that it couldn't handle, the parser would first try to produce an approximate analysis of the difficult fragment, and then resume normal processing for the rest of the input. The outcome is a kind of "fitted" parse, reflecting a compromise between the actual input and grammar-encoded preferences (imposed, mainly, in rule ordering).[3]

2. SKIP-AND-FIT RECOVERY IN PARSING

A robust parser must deal efficiently with difficult input, whether it is an extra-grammatical string, or a string whose complete analysis could be considered too costly. Frequently, these two situations are not distinguishable, especially for long and complex sentences found in free running text. The parser must be able to analyze such strings quickly and produce at least partial structures, imposing preferences when necessary, and even removing or inserting small input fragments, if the data-driven processing falters. For example, in the following sentence,

> The method is illustrated by the automatic construction of both recursive and iterative programs operating on natural numbers, lists, and trees, in order to construct a program satisfying certain specifications *a theorem induced by those specifications is proved,* and the desired program is extracted from the proof.

the italicized part is likely to cause additional complications in parsing this lengthy string, and the parser may be better off ignoring the fragment altogether. To do so successfully, the parser must close the constituent which is being currently parsed, and possibly a few of its parent constituents, removing corresponding productions from further consideration, until an appropriate production is reactivated. The parser then jumps over the intervening material so as to restart processing of the remainder of the sentence using the newly reactivated production. In the example at hand, suppose that the parser has just read the word *specifications* and is looking at the following article *a*. Rather than continuing at the present level, the parser reduces the phrase *a program satisfying certain specifications* to *NP*, and then forces further reductions: $SI \rightarrow$ *to V NP*; $SA \rightarrow SI$; $S \rightarrow NP\ V\ NP\ SA$, until production $S \rightarrow S$ *and* S is reached.[4] Subsequently, the parser skips input to find *and*, then resumes normal processing.

As may be expected, this kind of action involves a great deal of indeterminacy which, in case of natural language strings, is compounded by the high degree of lexical ambiguity. If the purpose of this skip-and-fit technique is to get the parser smoothly through even the most complex strings, the amount of additional backtracking caused by the lexical level ambiguity is certain to defeat it. Without lexical disambiguation of input, the parser's performance will deteriorate, even if the skipping is limited only to certain types of adverbial adjuncts. The most common cases of lexical ambiguity are those of a plural noun (nns) vs. a singular verb (vbz), a singular noun (nn) vs. a plural or infinitive verb (vbp,vb), and a past tense verb (vbd) vs. a past participle (vbn), as illustrated in the following example.

> The notation used (*vbn or vbd?*) explicitly associates (*nns or vbz?*) a data structure (*vb or nn*) shared (*vbn or vbd?*) by concurrent processes (*nns or vbz?*) with operations defined (*vbn or vbd?*) on it.

3. PART OF SPEECH TAGGER

One way of dealing with lexical ambiguity is to use a tagger to preprocess the input marking each word with a tags that indicates its syntactic categorization: a part of speech with selected morphological features such as number, tense, mode, case and degree. The following are tagged sentences from the CACM-3204 collection: [5]

> The(*dt*) paper(*nn*) presents(*vbz*) a(*dt*) proposal(*nn*) for(*in*) structured(*vbn*) representation(*nn*) of(*in*) multiprogramming(*vbg*) in(*in*) a(*dt*) high(*jj*) level(*nn*) language(*nn*) .(*per*)
>
> The(*dt*) notation(*nn*) used(*vbn*) explicitly(*rb*) associates(*vbz*) a(*dt*) data(*nns*) structure(*nn*) shared(*vbn*) by(*in*) concurrent(*jj*) processes(*nns*) with(*in*) operations(*nns*) defined(*vbn*) on(*in*) it(*pp*) .(*per*)

The tags are understood as follows: dt - determiner, nn - singular noun, nns - plural noun, in - preposition, jj - adjective, vbz - verb in present tense third person

[3] The idea of parse "fitting" was partly inspired by the IBM parser (Jensen et al., 1983), as well as by the standard error recovery techniques used in shift-reduce parsing.

[4] The decision to force a reduction rather than to back up could be triggered by various means. In case of TTP parser, it is always induced by the time-out signal.

[5] Tagged using the 35-tag Penn Treebank Tagset created at the University of Pennsylvania.

singular, to - particle "to", vbg - present participle, vbn - past participle, vbd - past tense verb, vb - infinitive verb, cc - coordinate conjunction.

Tagging of the input text substantially reduces the search space of a top-down parser since it resolves most of the lexical level ambiguities. In the examples above, tagging of *presents* as "vbz" in the first sentence cuts off a potentially long and costly "garden path" with *presents* as a plural noun followed by a headless relative clause starting with *(that) a proposal* .... In the second sentence, tagging resolves ambiguity of *used* (vbn vs. vbd), and *associates* (vbz vs. nns). Perhaps more importantly, elimination of word-level lexical ambiguity allows the parser to make projection about the input which is yet to be parsed, using a simple lookahead; in particular, phrase boundaries can be determined with a degree of confidence (Church, 1988). This latter property is critical for implementing skip-and-fit recovery technique outlined in the previous section.

Tagging of input also helps to reduce the number of parse structures that can be assigned to a sentence, decreases the demand for consulting of the dictionary, and simplifies dealing with unknown words. Since every item in the sentence is assigned a tag, so are the words for which we have no entry in the lexicon. Many of these words will be tagged as "np" (proper noun), however, the surrounding tags may force other selections. In the following example, *chinese*, which does not appear in the dictionary, is tagged as "jj":[6]

this(*dt*) paper(*nn*) dates(*vbz*) back(*rb*) the(*dt*) genesis(*nn*) of(*in*) binary(*jj*) conception(*nn*) circa(*in*) 5000(*cd*) years(*nns*) ago(*rb*) ,(*com*) as(*rb*) derived(*vbn*) by(*in*) the(*dt*) chinese(*jj*) ancients(*nns*) .(*per*)

We use a stochastic tagger to process the input text prior to parsing. The tagger is based upon a bigram model; it selects most likely tag for a word given co-occurrence probabilities computed from a small training set.[7]

4. PARSING WITH TTP PARSER

TTP (Tagged Text Parser) is a top down English parser specifically designed for fast, reliable processing of large amounts of text.

TTP is based on the Linguistic String Grammar developed by Sager (1981). Written in Quintus Prolog, the parser currently encompasses more than 400 grammar productions.[8] TTP produces a regularized representation of each parsed sentence that reflects the sentence's logical structure. This representation may differ considerably from a standard parse tree, in that the constituents get moved around (e.g., de-passivization, de-dativization), and the phrases are organized recursively around their head elements. An important novel feature of TTP parser is that it is equipped with a time-out mechanism that allows for fast closing of more difficult sub-constituents after a preset amount of time has elapsed without producing a parse. Although a complete analysis is attempted for each sentence, the parser may occasionally ignore fragments of input to resume "normal" processing after skipping a few words. These fragments are later analyzed separately and attached as incomplete constituents to the main parse tree.

As the parsing proceeds, each sentence receives a new slot of time during which its parse is to be returned. The amount of time allotted to any particular sentence can be regulated to obtain an acceptable compromise between parser's speed and precision. In our experiments we found that 0.5 sec/sentence time slot was appropriate for the CACM abstracts, while 0.7 sec/sentence was more appropriate for generally longer sentences in MUC-3 articles.[9] The actual length of the time interval allotted to any one sentence may depend on this sentence's length in words, although this dependency need not be linear. Such adjustments will have only limited impact on the parser's speed, but they may affect the quality of produced parse trees. Unfortunately, there is no obvious way to evaluate quality of parsing except by using its results to attain some measurable ends. We used the parsed CACM collection to generate domain-specific word correlations for query processing in an information retrieval system, and the results were satisfactory. For other applications, such as information extraction and deep understanding, a more accurate analysis may be required.[10]

[6] We use the machine readable version of the Oxford Advanced Learner's Dictionary (OALD).

[7] The program, supplied to us by Bolt Beranek and Newman, operates in two alternative modes, either selecting a single most likely tag for each word (best-tag option, the one we use at present), or supplying a short ranked list of alternatives (Meteer et al., 1991).

[8] See (Strzalkowski, 1990) for Prolog implementation details.

[9] Giving the parser more time per sentence doesn't always mean that a better (more accurate) parse will be obtained. For complex or extra-grammatical structures we are likely to be better off if we do not allow the parser wander around for too long: the most likely interpretation of an unexpected input is probably the one generated early (the grammar rule ordering enforces some preferences).

[10] A qualitative method for parser evaluation has been proposed in (Harrison et al., 1991), and it may be used to make a relative comparison of parser's accuracy. What is not clear is how accurate a parser needs to be for any particular application.

Initially, a full analysis of each sentence is attempted. If a parse is not returned before the allotted time elapses, the parser enters the time-out mode. From this point on, the parser is permitted to skip portions of input to reach a starter terminal for the next constituent to be parsed, and closing the currently open one (or ones) with whatever partial representation has been generated thus far. The result is an approximate partial parse, which shows the overall structure of the sentence, from which some of the constituents may be missing. The fragments skipped in the first pass are not thrown out, instead they are analyzed by a simple phrasal post-processor that looks for noun phrases and relative clauses and then attaches the recovered material to the main parse structure.

The time-out mechanism is implemented using a straightforward parameter passing and is at present limited to only a subset of nonterminals used by the grammar. Suppose that X is such a nonterminal, and that it appears on the right-hand side of a production $S \rightarrow X\ Y\ Z$. The set of "starters" is computed for Y, which consists of the word tags that can occur as the left-most constituent of Y. This set is passed as a parameter while the parser attempts to recognize X in the input. If X is recognized successfully within a preset time, then the parser proceeds to parse a Y, and nothing else happens. On the other hand, if the parser cannot determine whether there is an X in the input or not, that is, it neither succeeds nor fails in parsing X before being timed out, the unfinished X constituent is closed with a partial parse, and the parser is restarted at the closest element from the starters set for Y that can be found in the remainder of the input. If Y rewrites to an empty string, the starters for Z to the right of Y are added to the starters for Y and both sets are passed as a parameter to X. As an example consider the following clauses in the TTP parser:[11]

```
sentence(P) :- assertion([],P).
assertion(SR,P) :-
        clause(SR,P1),s_coord(SR,P1,P).
 ...
clause(SR,P) :-
        sa([pdt,dt,cd,pp,ppS,jj,jjr,
            jjs,nn,nns,np,nps],PA1),
        subject([vbd,vbz,vbp],Tail,P1),
        verbphrase(SR,Tail,P1,PA1,P),
        subtail(Tail).
 ...
thats(SR,P) :-
        that,assertion(SR,P).
```

In the *clause* production above, a (finite) clause rewrites into an (optional) sentence adjunct (SA), a subject, a verbphrase and subject's right adjunct (SUBTAIL, also optional). With the exception of *subtail*, each predicate has a parameter that specifies the list of "starter" tags for restarting the parser, should the evaluation of this predicate exceed the allotted portion of time. Thus, in case *sa* is aborted before its evaluation is complete, the parser will jump over some elements of the unparsed portion of the input looking for a word that could begin a subject phrase (either a predeterminer, a determiner, a count word, a pronoun, an adjective, a noun, or a proper name). Likewise, when *subject* is timed out, the parser will restart with *verbphrase* at either vbz, vbd or vbp (finite forms of a verb). Note that if *verbphrase* is timed out, then *subtail* will be ignored, both *verbphrase* and *clause* will be closed, and the parser will restart at an element of set *SR* passed down to *clause* from *assertion*. Note also that in the top-level production for a sentence the starter set for *assertion* is initialized to be empty: if the failure occurs at this level, no continuation is possible.

When a non-terminal is timed out and the parser jumps over a non-zero length fragment of input, it is assumed that the skipped part was some subconstituent of the closed non-terminal. Accordingly, a place holder is left in the parse structure under the node dominated by this non-terminal, which will be later filled by some nominal material recovered from the fragment. The examples given in the Appendix show approximate parse structures generated by TTP.

There are a few caveats in the skip-and-fit parsing strategy just outlined which warrant further explanation. In particular, the following problems must be resolved to assure parser's effectiveness: how to select starter tags for non-terminals, how to select non-terminals at which to place the starter tags, and finally how to select non-terminals at which input skipping can occur.

Obviously some tags are more likely to occur at the left-most position of a constituent than others. Theoretically, a subject phrase can start with a word tagged with any element from the following list: pdt, dt, cd, jj, jjr, jjs, pp, ppS, nn, nns, np, nps, vbg, vbn, rb, in.[12] In practice, however, we may select only a subset of these, as shown in the *clause* production above. Although we now risk missing the left-hand boundary of subject phrases in some sentences, while skipping an adjunct to their left, most cases are still covered and the chances of making a serious misinterpretation of

[11] The clauses are slightly simplified, and some arguments are removed for expository reasons.

[12] This list is not complete. In addition to the tags explained before: pdt - predeterminer, jjr - comparative adjective, jjs - superlative adjective, pp - pronoun, ppS - genitive, np, nps - proper noun, rb - adverb.

input are significantly lower.

We also need to decide on how input skipping is to be done. In a most straightforward design, when a nonterminal *X* is timed-out, the parser would skip input until it has reached a starter element of a nonterminal *Y* adjacent to *X* from the right, according to the top-down predictions.[13] On the other hand, certain adjunct phrases may be of little interest, possibly because of their typically low information contents, and we may choose to ignore them altogether. Therefore, if *X* is timed out, and *Y* is a low contents adjunct phrase, we can make the parser to jump right to the next nonterminal *Z*. In the *clause* production discussed before, *subtail* is skipped over if *verbphrase* is timed out.[14]

Finally, it is not an entirely trivial task to select non-terminals at which the input skipping can occur. If wrong non-terminals are chosen the parser may generate rather uninteresting structures that would be next to useless, or it may become trapped in inadvertently created dead ends, hopelessly trying to fit the parse. Consider, for example, the following sentence, taken from MUC-3 corpus of news messages:

> HONDURAN NATIONAL POLICE ON MONDAY PRESENTED TO THE PRESS HONDURAN JUAN BAUTISTA NUNEZ AMADOR AND NICARAGUAN LUIS FERNANDO ORDONEZ REYES, WHO TOLD REPORTERS THAT COMMANDER AURELIANO WAS ASSASSINATED ON ORDERS FROM JOSE DE JESUS PENA, THE NICARAGUAN EMBASSY CHIEF OF SECURITY.

After reaching the verb PRESENTED, the parser consults the lexicon and finds that one of the possible subcategorizations of this verb is [pnn,to], that is, its object string can be a prepositional phrase with 'to' followed by a noun phrase. The parser thus begins to look for a prepositional phrase starting at "TO THE PRESS ...", but unfortunately misses the end of the phrase at PRESS (the following word is tagged as a noun), and continues until reaching the end of sentence. At this point it realizes that it went too far (there is no noun phrase left), and starts backing up. Before the parser has a chance to back up to the word PRESS and correct the early mistake, however, the time-out mode is turned on, and instead of abandoning the current analysis, the parser now tries hard to fix it by skipping varying portions of input. This may take a considerable amount time if the skip points are badly placed. On the other hand, we wouldn't like to allow an easy exit by accepting an empty noun phrase at the end of the sentence![15]

One of the essential properties of the input skipping mechanism is its flexibility to jump over varying-size chunks of the input string. The goal is to fit the input with a closest matching parse structure while leaving the minimum number of words unaccounted for. In TTP, the skipping mechanism is implemented by adding extra productions for selected nonterminals, and these are always tried first whenever the nonterminal is to be expanded. We illustrate this with *rn* productions covering right adjuncts to a noun.

```
rn(SR,P) :-
    timed_out,!,
    skip(SR),store(P).
rn(_,[]) :-
    la([[pdt,dt,vbz,vbp,vbd,
         md,com,sem,per]]),
    \+la([[com],[wdt,wp,wps]]).
rn(SR,P) :- rn1(SR,P).
```

In the *rn* predicate, *SR* is the list of starter tags and *P* is the parse tree fragment. The first production checks if the time-out mode has already been entered, in which case the input is skipped until a starter tag is found, while the skipped words are stored into *P* to be analyzed later in the parser's second pass. Note that in this case all other *rn* productions are cut off; however, should the first skip-and-fit attempt fail to lead to a successful parse, backtracking may eventually force predicate *skip(SR)* to evaluate again and make a longer leap. In a top-down left to right parser, each input skipping location becomes potentially a multiple backtracking point which needs to be controlled in order to avoid a combinatorial explosion of possibilities. This is accomplished by supplementing top-down predictions with bottom-up, data-driven fragmentation of input, and a limited lookahead. For example, in the second of the *rn* productions above, a right adjunct to a noun can be considered empty if the item following the noun is either a period, a semicolon, a comma, or a word tagged as pdt, dt, vbz, vbp, vbd, or md, but not a comma followed by a relative pronoun.[16]

[13] Note that the top-down predictions are crucial for the skipping parser, whether the parser's processing is top-down or bottom-up.

[14] *subtail* is the remainder of a discontinued subject phrase.

[15] In the present implementation, when the skipping mode is entered, it will stay on for the balance of the first pass in parsing of the current sentence. This way, one skip-and-fit attempt may lead to another before any backtracking is considered. An alternative is to do time-out on a nonterminal by nonterminal basis, that is, to time out processing of selected nonterminals only and then resume regular parsing. This design leads to a far more complex implementation and somewhat inferior performance, but it might be worth considering in the future.

[16] md - modal verb; vbp - plural verb; wdt, wp, wps - relative pronouns.

5. ROBUSTNESS

TTP is a robust parser and it will process nearly every sentence or phrase, provided the latter is reasonably correctly tagged. [17] The parser robustness is further increased by allowing for a gradual degradation of its performance rather than an outright failure in the face of an unexpected input. Each sentence or phrase is attempted to be analyzed in up to four ways: (1) as a sentence, (2) as a noun phrase or a preposition phrase with a right adjunct(s), (3) as a gerundive clause, and if all these fail, (4) as a series of simple noun phrases, with each of these attempts allotted a fresh time slice.[18] The purpose of this extension is to accommodate some infrequent but still important constructions, such as titles, itemizations, and lists.

6. DISCUSSION

In this paper we described TTP, a fast and robust parser for natural language. In the experiments conducted with various text collections of more that 50 million words the average parsing speed recorded was approx. 0.5 sec/sentence. For example, the total time spent on parsing the CACM-3204 collection was less than 1.5 hours. In other words, TTP can process 100,000 words in approximately 45 minutes, and it could parse a gigabyte of text (approx. 150 million words) in about 40 days, on a 21 MIPS computer.

The parser is based on a wide coverage grammar for English, and it contains a powerful skip-and-fit recovery mechanism that allows it to deal with unexpected input and to perform effectively under a severe time pressure. Prior to parsing, the input text is tagged with a stochastic tagger that assigns part-of-speech labels to every word, thus resolving lexical level ambiguity.

TTP has been used as front-end of a natural language processing component to a traditional document-based information retrieval system (Strzalkowski and Vauthey, 1992). The parse structures were further analyzed to extract word and phrase dependency relations which were in turn used as input to various statistical and indexing processes. The results obtained were generally satisfactory: an improvement in both recall and precision of document retrieval have been observed. At present, we are also conducting experiments with large corpora of technical computer science texts in order to extract domain-specific conceptual taxonomies for an even greater gain in retrieval effectiveness.

[17] Some sentences (1 in 5000) may still fail to parse if tagging errors are compounded in an unexpected way.

[18] Although parsing of some sentences may now approach four times the allotted time limit, we noted that the average parsing time per sentence at 0.745 sec. is only slightly above the time-out limit.

7. ACKNOWLEDGEMENTS

We wish to thank Ralph Weischedel and Heidi Fox of BBN for assisting in the use of the part of speech tagger. ACM has generously provided us with the Computer Library text database. This paper is based upon work supported by the Defense Advanced Research Project Agency under Contract N00014-90-J-1851 from the Office of Naval Research, the National Science Foundation under Grant IRI-89-02304, and by the Canadian Institute for Robotics and Intelligent Systems (IRIS).

8. REFERENCES

Church, Kenneth Ward. 1988. "A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text." Proceedings of the Second Conference on Applied Natural Language Processing, pp. 136-143.

Church, Kenneth Ward and Patrick Hanks. 1990. "Word association norms, mutual information, and lexicography." *Computational Linguistics*, 16(1), MIT Press, pp. 22-29.

De Marcken, Carl G. 1990. "Parsing the LOB corpus." Proceedings of the 28th Meeting of the ACL, Pittsburgh, PA. pp. 243-251.

Harrison, Philip, et al. 1991. "Evaluating Syntax Performance of Parser/Grammars of English." Natural Language Processing Systems Evaluation Workshop, Berkeley, CA. pp. 71-78.

Hindle, Donald. 1983. "User manual of Fidditch, a deterministic parser." Naval Research Laboratory Technical Memorandum 7590-142.

Hindle, Donald and Mats Rooth. 1991. "Structural Ambiguity and Lexical Relations." Proceedings of the 29th Meeting of the ACL, Berkeley, CA. pp. 229-236.

Jensen, K., G.E. Heidorn, L.A. Miller, and Y. Ravin. 1983. "Parse fitting and prose fixing: Getting a hold of ill-formedness." *Computational Linguistics*, 9(3-4), pp. 147-161.

Meteer, Marie, Richard Schwartz, and Ralph Weischedel. 1991. "Studies in Part of Speech Labelling." Proceedings of the 4th DARPA Speech and Natural Language Workshop, Morgan-Kaufman, San Mateo, CA. pp. 331-336.

Ruge, Gerda, Christoph Schwarz, Amy J. Warner. 1991. "Effectiveness and Efficiency in Natural Language Processing for Large Amounts of Text." *Journal of the ASIS*, 42(6), pp. 450-456.

Sager, Naomi. 1981. *Natural Language Information Processing*. Addison-Wesley.

Strzalkowski, Tomek. 1990. "Reversible logic grammars for natural language parsing and

generation." *Computational Intelligence*, 6(3), NRC Canada, pp. 145-171.

Strzalkowski, Tomek and Barbara Vauthey. 1992. "Information Retrieval Using Robust Natural Language Processing." Proceedings of the 30th Annual Meeting of the ACL, Newark, Delaware, June 28 - July 2.

APPENDIX: Sample parses

A few examples of non-standard output generated by TTP are shown in Figures 1 to 3. In Figure 1, TTP has failed to find the main verb and it had to jump over much of the last phrase *such as the LR(k) grammars*, partly due to an improper tokenization of *LR(k)* (note *skipped* nodes indicating the material ignored in the first pass). In Figure 2, the parser has initially assumed that the conjunction in the sentence has the narrow scope, then it realized that something went wrong but, apparently, there was no time left to back up. Note, however, that little has been lost: a complete structure of the second half of this sentence following the conjuction *and* is easily recovered from the parse tree (*var* points up to the dominating *np*). Occasionally, sentences may come out substantially truncated, as shown in Figure 3, where *although* has been mistagged as a preposition.

SENTENCE:
The problem of determining whether an arbitrary context-free grammar is a member of some easily parsed subclass of grammars such as the LR(k) grammars is considered.
APPROXIMATE PARSE:

```
[[verb,[]],[subject,[np,[n,problem],[t_pos,the],
      [of,[[verb,[determine]],[subject,anyone],
          [object,[[verb,[be]],
              [subject,[np,[n,grammar],[t_pos,an],
                  [adj,[arbitrary]],[adj,[context_free]]]],
              [object,[np,[n,member],[t_pos,a],
                  [of,[np,[n,subclass],[t_pos,some],
                      [a_pos_v,
                          [[verb,[parse,[adv,easily]]],
                              [subject,anyone],
                              [object,pro]]],
                      [of,[np,[n,grammar],
                          [rn_wh,[[verb,[such]],
                              [subject,var]]]]]]]]]],
          [sub_ord,[as,[[verb,[be]],[subject,pro],
              [object,[np,[n,k],[t_pos,the],
                  [adj,["lr(")]]]]]],
          [skipped,[[np,[n,grammar]]]]]]],
    [skipped,[[is],[wh_rel,[[verb,[consider]],
              [subject,anyone],[object,var]]]]]].
```

Figure 1.

SENTENCE:
The TX-2 computer at MIT Lincoln Laboratory was used for the implementation of such a system and the characteristics of this implementation are reported.
APPROXIMATE PARSE:

```
[[be],[[verb,[use]],
      [subject,anyone],
      [object,[np,[n,computer],[t_pos,the],[adj,[tx_2]]]],
      [for,[and,
          [np,[n,implementation],[t_pos,the],
              [of,[np,[n,system],[t_pos,[such,a]]]]],
          [np,[n,characteristics],[t_pos,the],
              [of,[np,[n,implementation],[t_pos,this]]],
              [skipped,
                  [[are],[wh_rel,[[verb,[report]],
                          [subject,anyone],
                          [object,var]]]]]]]],
      [at,[np,[n,laboratory],[adj,[mit]],
          [n_pos,[np,[n,lincoln]]]]]].
```

Figure 2.

SENTENCE:
In principle, the system can deal with any orthography, although at present it is limited to 4000 Chinese characters and some mathematical symbols.
APPROXIMATE PARSE:

```
[[can_aux],[[verb,[deal]],
      [subject,[np,[n,system],[t_pos,the]]],
      [sub_ord,[with,[[verb,[limit]],
              [subject,anyone],
              [object,[skipped,
                      [[np,[n,orthography],[t_pos,any]],
                       [comS,although,at],
                       [np,[n,present]],
                       [np,[n,it]],
                       [is]]]],
              [to,[np,[n,character],
                  [count,[4000]],
                  [a_pos,[chinese]],
                  [skipped,
                      [[and],
                       [np,[n,symbol],[t_pos,some],
                           [adj,[mathematical]]]]]]]]]],
      [in,[np,[n,principle]]]].
```

Figure 3.