Extending A Broad-Coverage Parser for a General NLP Toolkit

Hassan Alam, Hua Cheng, Rachmat Hartono, Aman Kumar, Paul Llido, Crystal Nakatsu, Fuad Rahman, Yuliya Tarnikova, Timotius Tjahjadi and Che Wilcox

BCL Technologies Inc.
Santa Clara, CA 95050 U.S.A.
fuad@bcltechnologies.com

Abstract

With the rapid growth of real world applications for NLP systems, there is a genuine demand for a general toolkit from which programmers with no linguistic knowledge can build specific NLP systems. Such a toolkit should have a parser that is general enough to be used across domains, and yet accurate enough for each specific application. In this paper, we describe a parser that extends a broad-coverage parser, Minipar (Lin, 2001), with an adaptable shallow parser so as to achieve both generality and accuracy in handling domain specific NL problems. We test this parser on our corpus and the results show that the accuracy is significantly higher than a system that uses Minipar alone.

1 Introduction

With the improvement of natural language processing (NLP) techniques, domains for NLP systems, especially those handling speech input, are rapidly growing. However, most computer programmers do not have enough linguistic knowledge to develop NLP systems. There is a genuine demand for a general toolkit from which programmers with no linguistic knowledge can rapidly build NLP systems that handle domain specific problems more accurately (Alam, 2000). The toolkit will allow programmers to generate natural language front ends for new and existing applications using, for example, a program-through-example method. In this methodology, the programmer will specify a set of sample input sentences or a domain corpus for each task. The toolkit will then organize the sentences by similarity and generate a large set of syntactic variations of a given sentence. It will also generate the code that takes a user's natural language request and executes a command on an application. Currently this is an active research area, and the Advanced Technology Program (ATP) of the National Institute of Standards and Technology (NIST) is funding part of the work.

In order to handle natural language input, an NLP toolkit must have a parser that maps a sentence string to a syntactic structure. The parser must be both general and accurate. It has to be general because programmers from different domains will use the toolkit to generate their specific parsers. It has to be accurate because the toolkit targets commercial domains, which usually require high accuracy. The accuracy of the parser directly affects the accuracy of the generated NL interface. In the program-through-example approach, the toolkit should convert the example sentences into semantic representations so as to capture their meanings. In a real world application, this process will involve a large quantity of data. If the programmers have to check each syntactic or semantic form by hand in order to decide if the corresponding sentence is parsed correctly, they are likely to be overwhelmed by the workload imposed by the large number of sentences, not to mention that they do not have the necessary linguistic knowledge to do this. Therefore the toolkit should have a broad-coverage parser that has the accuracy of a parser designed specifically for a domain.

One solution is to use an existing parser with relatively high accuracy. Using existing parsers such as (Charniak, 2000; Collins, 1999) would eliminate the need to build a parser from scratch. However, there are two problems with such an approach. First, many parsers claim high precision in terms of the number of correctly parsed syntactic relations

rather than sentences, whereas in commercial applications, the users are often concerned with the number of complete sentences that are parsed correctly. The precision might drop considerably using this standard. In addition, although many parsers are domain independent, they actually perform much better in the domains they are trained on or implemented in. Therefore, relying solely on a general parser would not satisfy the accuracy needs for a particular domain.

Second, since each domain has its own problems, which cannot be foreseen in the design of the toolkit, customization of the parser might be needed. Unfortunately, using an existing parser does not normally allow this option. One solution is to build another parser on top of the general parser that can be customized to address domain specific parsing problems such as ungrammatical sentences. This domain specific parser can be built relatively fast because it only needs to handle a small set of natural language phenomena. In this way, the toolkit will have a parser that covers wider applications and in the mean time can be customized to handle domain specific phenomena with high accuracy. In this paper we adopt this methodology.

The paper is organized into 6 sections. In Section 2, we briefly describe the NLP toolkit for which the parser is proposed and implemented. Section 3 introduces Minipar, the broad-coverage parser we choose for our toolkit, and the problems this parser has when parsing a corpus we collected in an IT domain. In Section 4, we present the design of the shallow parser and its disadvantages. We describe how we combine the strength of the two parsers and the testing result in Section 5. Finally, in Section 6, we draw conclusions and propose some future work.

2 NLP Toolkit

In the previous section, we mentioned a Natural Language Processing Toolkit (NLPTK) that allows programmers with no linguistic knowledge to rapidly develop natural language user interfaces for their applications. The toolkit should incorporate the major components of an NLP system, such as a spell checker, a parser and a semantic representation generator. Using the toolkit, a software engineer will be able to create a system that incorporates complex NLP techniques such as syntactic parsing and semantic understanding.

In order to provide NL control to an application, the NLPTK needs to generate semantic representations for input sentences. We refer to each of these semantic forms as a *frame*, which is basically a predicate-argument representation of a sentence.

The NLPTK is implemented using the following steps:

1. NLPTK begins to create an NLP front end by generating semantic representations of sample input sentences provided by the programmer.
2. These representations are expanded using synonym sets and stored in a Semantic Frame Table (SFT), which becomes a comprehensive database of all the possible commands a user could request the system to do.
3. The toolkit then creates methods for attaching the NLP front end to the back end applications.
4. When the NLP front end is released, a user may enter an NL sentence, which is translated into a semantic frame by the system. The SFT is then searched for an equivalent frame. If a match is found, the action or command linked to this frame is executed.

In order to generate semantic representations in Step 1, the parser has to parse the input sentences into syntactic trees. During the process of building an NLP system, the programmer needs to customize the parser of the toolkit for their specific domain. For example, the toolkit provides an interface to highlight the domain specific words that are not in the lexicon. The toolkit then asks the programmer for information that helps the system insert the correct lexical item into the lexicon. The NLPTK development team must handle complicated customizations for the programmer. For example, we might need to change the rules behind the domain specific parser to handle certain natural language input. In Step 4, when the programmer finishes building an NLP application, the system will implement a domain specific parser. The toolkit has been completely implemented and tested.

We use a corpus of email messages from our customers for developing the system. These emails contain questions, comments and general inquiries regarding our document-conversion products. We modified the raw email programmatically to delete the attachments, HTML tags, headers and sender information. In addition, we manually deleted salutations, greetings and any information not directly related to customer support. The corpus contains around 34,640 lines and 170,000 words. We constantly update it with new emails from our customers.

From this corpus, we created a test corpus of 1000 inquiries to test existing broad-coverage parsers and the parser of the toolkit.

3 Minipar in NLPTK

We choose to use Minipar (Lin, 2001), a widely known parser in commercial domains, as the general parser of NLPTK. It is worth pointing out that our methodology does not depend on any individual parser, and we can use any other available parser.

3.1 Introduction to Minipar

Minipar is a principle-based, broad-coverage parser for English (Lin, 2001). It represents its grammar as a network of nodes and links, where the nodes represent grammatical categories and the links represent types of dependency relationships. The grammar is manually constructed, based on the Minimalist Program (Chomsky, 1995).

Minipar constructs all possible parses of an input sentence. It makes use of the frequency counts of the grammatical dependency relationships extracted by a collocation extractor (Lin, 1998b) from a 1GB corpus parsed with Minipar to resolve syntactic ambiguities and rank candidate parse trees. The dependency tree with the highest ranking is returned as the parse of the sentence.

The Minipar lexicon contains about 130,000 entries, derived from WordNet (Fellbaum, 1998) with additional proper names. The lexicon entry of a word lists all possible parts of speech of the word and its subcategorization frames (if any).

Minipar achieves about 88% precision and 80% recall with respect to dependency relationships (Lin, 1998a), evaluated on the SUSANNE corpus (Sampson, 1995), a subset of the Brown Corpus of American English.

3.2 Disadvantages of Minipar

In order to see how well Minipar performs in our domain, we tested it on 584 sentences from our corpus. Instead of checking the parse trees, we checked the frames corresponding to the sentences, since the accuracy of the frames is what we are most concerned with. If any part of a frame was wrong, we treated it as an error of the module that contributed to the error. We counted all the errors caused by Minipar and its accuracy in terms of correctly parsed sentences is 77.6%. Note that the accuracy is actually lower because later processes fix some errors in order to generate correct frames.

The majority of Minipar errors fall in the following categories:

1. Tagging errors: some nouns are mis-tagged as verbs. For example, in *Can I get a copy of the batch product guide?*, *guide* is tagged as a verb.
2. Attachment errors: some prepositional phrases (PP) that should be attached to their immediate preceding nouns are attached to the verbs. For example, in *Can Drake convert the PDF documents in Japanese?*, *in Japanese* is attached to *convert*.
3. Missing lexical entries: some domain specific words such as *download* and their usages are not in the Minipar lexicon. This introduces parsing errors because such words are tagged as nouns by default.
4. Inability to handle ungrammatical sentences: in a real world application, it is unrealistic to expect the user to enter only grammatical sentences. Although Minipar still produces a syntactic tree for an ungrammatical sentence, the tree is ill formed and cannot be used to extract the semantic information being expressed.

In addition, Minipar, like other broad-coverage parsers, cannot be adapted to specific applications. Its accuracy does not satisfy the needs of our toolkit. We have to build another parser on top of Minipar to enable domain specific customizations to increase the parsing accuracy.

4 The Shallow Parser

Our NLPTK maps input sentences to action requests. In order to perform an accurate mapping the toolkit needs to get information such as the sentence type, the main predicate, the arguments of the predicate, and the modifications of the predicate and arguments from a sentence. In other words, it mostly needs local dependency relationships. Therefore we decided to build a shallow parser instead of a full parser. A parser that captures the most frequent verb argument structures in a domain can be built relatively fast. It takes less space, which can be an important issue for certain applications. For example, when building an NLP system for a handheld platform, a light parser is needed because the memory cannot accommodate a full parser.

4.1 Introduction

We built a KWIC (keyword in context) verb shallow parser. It captures only verb predicates with their arguments, verb argument modifiers and verb adjuncts in a sentence. The resulting trees contain local and subjacent dependencies between these elements.

The shallow parser depends on three levels of information processing: the verb list, subcategorization (in short, subcat) and syntactic rules. The verb subcat system is derived from Levin's taxonomy of verbs and their classes (Levin, 1993). We have 24 verb files containing 3200 verbs, which include all the Levin verbs and the most frequent verbs in our corpus. A verb is indexed to one or more subcat files and each file represents a particular alternation semantico-syntactic sense. We have 272 syntactic subcat files derived from the Levin verb semantic classes. The syntactic rules are marked for argument types and constituency, using the Penn Treebank tagset (Marcus, 1993). They contain both generalized rules, e.g., *.../NN*, and specified rules, e.g., *purchase/VBP*. An example subcat rule for the verb *purchase* looks like this: *.../DT .../JJ .../NN*, *.../DT .../NN from/RP .../NN for/RP .../NN*. The first element says that *purchase* takes an NP argument, and the second says that it takes an NP argument and two PP adjuncts.

We also encoded specific PP head class information based on the WordNet concepts in the rules for some attachment disambiguation.

The shallow parser works like this: it first tags an incoming sentence with Brill tagger (Brill, 1995) and matches verbs in the tagged sentence with the verb list. If a match is found, the parser will open the subcat files indexed to that verb and gather all the syntactic rules in these specific subcat files. It then matches the verb arguments with these syntactic rules and outputs the results into a tree. The parser can control over-generation for any verb because the syntactic structures are limited to that particular verb's syntactic structure set from the Levin classes.

4.2 Disadvantages of Shallow Parser

The disadvantages of the shallow parser are mainly due to its simplified design, including:

1. It cannot handle sentences whose main verb is *be* or phrasal sentences without a verb because the shallow parser mainly targets command-and-control verb argument structures.
2. It cannot handle structures that appear before the verb. Subjects will not appear in the parse tree even though it might contain important information.
3. It cannot detect sentence type, for example, whether a sentence is a question or a request.
4. It cannot handle negative or passive sentences.

We tested the shallow parser on 500 sentences from our corpus and compared the results with the output of Minipar. We separated the sentences into five sets of 100 sentences. After running the parser on each set, we fixed the problems that we could identify. This was our process of training the parser. Table 1 shows the data obtained from one such cycle. Since the shallow parser cannot handle sentences with the main verb *be*, these sentences are excluded from the statistics. So the test set actually contains 85 sentences.

In Table 1, the first column and the first row show the statistics for the shallow parser and Minipar respectively. The upper half of the table is for the unseen data, where 55.3% of the sentences are parsed correctly and 11.8% incorrectly (judged by humans) by both parsers. 18.9% of the sentences are parsed correctly by Minipar, but incorrectly by the shallow parser, and 14.1% vise versa. The lower half of the table shows the result after

fixing some shallow parser problems, for example, adding a new syntactic rule. The accuracy of the parser is significantly improved, from 69.4% to 81.2%. This shows the importance of adaptation to specific domain needs, and that in our domain, the shallow parser outperforms Minipar.

SP/MP	Correct (74.1%)	Wrong (25.9%)
Correct (69.4%)	47 (55.3%)	12 (14.1%)
Wrong (30.6%)	16 (18.9%)	10 (11.8%)
SP/MP	Correct (74.1%)	Wrong (25.9%)
Correct (81.2%)	53 (62.4%)	16 (18.8%)
Wrong (18.8%)	10 (11.8%)	6 (7.1%)

Table 1: Comparison of the shallow parser with Minipar on 85 sentences

The parsers do not perform equally well on all sets of sentences. For some sets, the accuracies of Minipar and the shallow parser drop to 60.9% and 67.8% respectively.

5 Extending Minipar with the Shallow Parser

Each parser has pros and cons. The advantage of Minipar is that it is a broad-coverage parser with relatively high accuracy, and the advantage of the shallow parser is that it is adaptable. For this reason, we intend to use Minipar as our primary parser and the shallow parser a backup. Table 1 shows only a small percentage of sentences parsed incorrectly by both parsers (about 7%). If we always choose the correct tree between the two outputs, we will have a parser with much higher accuracy. Therefore, combining the advantages of the two parsers will achieve better performance in both coverage and accuracy. Now the question is how to decide if a tree is correct or not.

5.1 Detecting Parsing Errors

In an ideal situation, each parser should provide a confidence level for a tree that is comparable to each other. We would choose the tree with higher confidence. However, this is not possible in our case because weightings of the Minipar trees are not publicly available, and the shallow parser is a rule-based system without confidence information.

Instead, we use a few simple heuristics to decide if a tree is right or wrong, based on an analysis of the trees generated for our test sentences. For example, given a sentence, the Minipar tree is incorrect if it has more than one subtree connected by a top-level node whose syntactic category is U (unknown). A shallow parser tree is wrong if there are unparsed words at the end of the sentence after the main verb (except for interjections). We have three heuristics identifying a wrong Minipar tree and two identifying a wrong shallow parser tree. If a tree passes these heuristics, we must label the tree as a good parse. This may not be true, but we will compensate for this simplification later. The module implementing these heuristics is called the *error detector*.

We tested the three heuristics for Minipar trees on a combination of 84 requestive, interrogative and declarative sentences. The results are given in the upper part of Table 2. The table shows that 45 correct Minipar trees (judged by humans) are identified as correct by the error detector and 18 wrong trees are identified as wrong, so the accuracy is 75%. Tagging errors and some attachment errors cannot be detected.

MP/ED	Correct (76.2%)	Wrong (23.8%)
Correct (56%)	45 (53.6%)	2 (2.4%)
Wrong (44%)	19 (22.6%)	18 (21.4%)
SP/ED	Correct (73%)	Wrong (26%)
Correct (59%)	58 (58%)	1 (1%)
Wrong (40%)	15 (15%)	25 (25%)

Table 2: The performance of the parse tree error detector

We tested the two heuristics for shallow parser trees on 100 sentences from our corpus and the result is given in the lower part of Table 2. The accuracy is about 83%. We did not use the same set of sentences to test the two sets of heuristics because the coverage of the two parsers is different.

5.2 Choosing the Better Parse Trees

We run the two parsers in parallel to generate two parse trees for an input sentence, but we cannot depend only on the error detector to decide which tree to choose because it is not accurate enough. Table 2 shows that the error

detector mistakenly judges some wrong trees as correct, but not the other way round. In other words, when the detector says a tree is wrong, we have high confidence that it is indeed wrong, but when it says a tree is correct, there is some chance that the tree is actually wrong. This motivates us to distinguish three cases:

1. When only one of the two parse trees is detected as wrong, we choose the correct tree, because no matter what the correct tree actually is, the other tree is definitely wrong so we cannot choose it.
2. When both trees are detected as wrong, we choose the Minipar tree because it handles more syntactic structures.
3. When both trees are detected as correct, we need more analysis because either might be wrong.

We have mentioned in the previous sections the problems with both parsers. By comparing their pros and cons, we come up with heuristics for determining which tree is better for the third case above.

The decision flow for selecting the better parse is given in Figure 1. Since the shallow parser cannot handle negative and passive sentences as well as sentences with the main verb *be*, we choose the Minipar trees for such sentences. The shallow parser outperforms Minipar on tagging and some PP attachment because it checks the WordNet concepts. So, when we detect differences concerning part-of-speech tags and PP attachment in the parse trees, we choose the shallow parser tree as the output. In addition, we prefer the parse with bigger NP chunks.

We tested these heuristics on 200 sentences and the result is shown in Table 3. The first row specifies whether a Minipar tree or a shallow parser tree is chosen as the final output. The first column gives whether the final tree is correct or incorrect according to human judgment. 88% of the time, Minipar trees are chosen and they are 82.5% accurate. The overall contribution of Minipar to the accuracy is 73.5%. The improvement from just using Minipar is about 7%, from about 75.5% to 82.5%. This is a significant improvement.

The main computational expense of running two parsers in parallel is time. Since our shallow parser has not been optimized, the extended parser is about 2.5 times slower than Minipar alone. We hope that with some optimization, the speed of the system will increase considerably. Even in the current time frame, it takes less than 0.6 second to parse a 15 word sentence.

Final tree	*MP tree* (88%)	*SP tree* (11%)
Correct (82.5%)	73.5%	9%
Wrong (16.5%)	14.5%	2%

Table 3: Results for the extended parser

6 Conclusions and Future Work

In this paper we described a parser that extends a broad-coverage parser, Minipar, with a domain adaptable shallow parser in order to achieve generality and higher accuracy at the same time. This parser is an important component of a general NLP Toolkit, which helps programmers quickly develop an NLP front end that handles natural language input from their end users. We tested the parser on 200 sentences from our corpus and the result shows significant improvement over using Minipar alone.

Future work includes improving the efficiency and accuracy of the shallow parser. Also, we will test the parser on a different domain to see how much work is required to switch to a new domain.

References

Alam H. (2000) *Spoken Language Generic User Interface (SLGUI)*. Technical Report AFRL-IF-RS-TR-2000-58, Air Force Research Laboratory, Rome.

Brill E. (1992) *A Simple Rule-based Part of Speech Tagger*. In Proceedings of the 3rd Conference on Applied Natural Language Processing.

Charniak E. (2000) *A Maximum-Entropy-Inspired Parser*. In Proceedings of the 1st Meeting of NAACL. Washington.

Chomsky N. (1995) *Minimalist Program*. MIT Press.

Collins M. (1999) *Head-Driven Statistical Models for Natural Language Parsing*. PhD Dissertation, University of Pennsylvania.

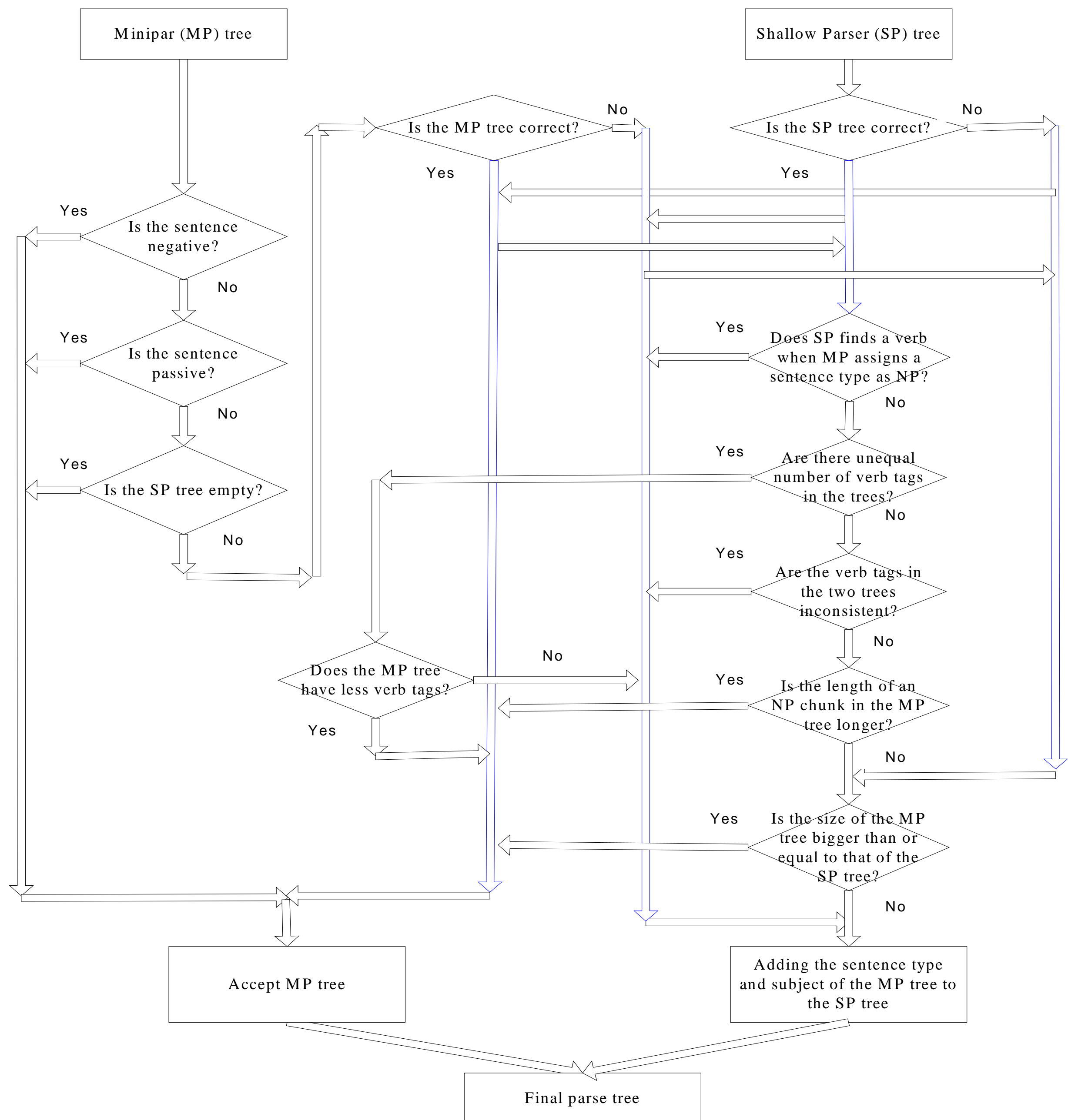

Figure 1: Decision flow for parse tree selection

Levin B. (1993) *English Verb Classes and Alternations: A Preliminary Investigation*. University of Chicago Press, Chicago.

Lin D. (1998a) *Dependency-based Evaluation of Minipar*. In Workshop on the Evaluation of Parsing Systems, Spain.

Lin D. (1998b) *Extracting Collocations from Text Corpora*. In Workshop on Computational Terminology, Montreal, Canada, pp. 57-63.

Lin D. (2001) *Latat: Language and Text Analysis Tools*. In Proceedings of Human Language Technology Conference, CA, USA.

Marcus M., Santorini B. and Marcinkiewicz M. (1993) *Building a Large Annotated Corpus of English: The Penn Treebank*, Computational Linguistics, vol. 19, no. 2, pp. 313-330.

Sampson G. (1995) *English for the Computer*. Oxford University Press.