SUPPLE: A Practical Parser for Natural Language Engineering Applications

Robert Gaizauskas, Mark Hepple, Horacio Saggion,
Mark A. Greenwood and Kevin Humphreys*
Department of Computer Science, University of Sheffield
{robertg,hepple,saggion,m.greenwood}@dcs.shef.ac.uk

Abstract

We describe SUPPLE, a freely-available, open source natural language parsing system, which is implemented in Prolog, and which has been designed for practical use in language engineering applications. SUPPLE can be run as a stand-alone application, but is also available as a component within the GATE General Architecture for Text Engineering. The description covers the SUPPLE parsing approach and the grammar formalism used. SUPPLE is distributed with a particular example grammar, which has been developed over a number of years through use in numerous language engineering projects, and the key characteristics of this grammar are also described.

1 Introduction

In this paper we describe SUPPLE[1] — the Sheffield University Prolog Parser for Language Engineering — a general purpose parser that produces syntactic and semantic representations from input sentences in English and which is well-suited for a range of natural language processing applications. SUPPLE is freely available both as a language processing resource within the General Architecture for Text Engineering (GATE)[1] and as a standalone Prolog program requiring various preprocessing steps to be carried out in advance.

The parser package, as distributed, consists of two separate components. The first is the parser proper — a general purpose bottom-up chart parser for feature-based context free phrase structure grammars (CF-PSGs) written in Prolog that has a number of characteristics that make it well-suited for practical use in language engineering applications. The second is a particular example

*At Microsoft Corporation since 2000 (Speech and Natural Language Group). Email: kevinhum@microsoft.com.

[1]In previous published materials and in the current GATE release the parser is referred to as buChart. This is name is now deprecated.

grammar for use with the parser, which has been developed over a number of years through use in numerous language engineering projects and which now covers a considerable range of English syntax. While neither bottom-up chart parsing for feature-based CF-PSGs nor wide coverage grammars of English are new, both the parsing mechanism and the grammars have features that make the SUPPLE approach distinctive and well-suited to language engineering applications.
Key features of the parser include:

1. The parser allows multiword units identified by earlier processing components — e.g. named entity recognisers (NERs), gazetteers or terminology processors — to be treated as units which are non-decomposable for syntactic processing. This is important in practical settings as the identification of such items is an essential part of analyzing real text in many domains.
2. The parser allows a layered parsing process, with a number of separate grammars being applied in series, one on top of the other, with a "best parse" selection process between stages so that only a subset of the constituents constructed in each stage is passed forward to the next stage. While this may make the parsing process incomplete with respect to the total set of analyses licensed by the grammar rules, it makes the parsing process much more efficient and makes modular grammar development more straightforward.
3. Facilities are provided to simplify handling feature-based grammars. The grammar representation uses flat, i.e. non-embedded, feature representations which are combined used Prolog term unification for efficiency. Features are predefined and source grammars compiled into a full form representation, allowing grammar writers to ignore feature ordering in rules and to include only relevant features in any given rule. The grammar representation formalism also permits disjunctive and optional constituents in the right hand side of rules.
4. The chart parsing algorithm is simple, but very efficient, and in informal testing achieved a roughly ten-fold increase in speed over a related Prolog implementation of a standard bottom-up active chart parsing algorithm.
5. The parser does not fail if full sentential parses cannot be found, but instead outputs partial analyses as syntactic and semantic fragments for user-selectable syntactic categories. This makes the parser robust in applications which deal with large volumes of real text.

Key features of the grammar include:

1. The morpho-syntactic and semantic information required for individual lexical items is minimal — inflectional root and word class only, where the word class inventory is basically that used in the Penn Treebank [4].
2. Given the preceding, a conservative philosophy is adopted regarding identification of verbal arguments and attachment of nominal and verbal post-modifiers, such as prepositional phrases

and relative clauses. Rather than producing all possible analyses or using probabilities to generate the most likely analysis, the preference is not to offer a single analysis that spans the input sentence unless it can be relied on to be correct. This means that in many cases only partial analyses are produced, but the philosophy is that it is more useful to produce partial analyses that are correct than full analyses which may well be wrong or highly disjunctive. Output from the parser can be passed to further processing components which may bring additional information to bear in resolving attachments.

3. Also in line with the foregoing, an analysis of verb phrases is adopted in which a core verb cluster consisting of verbal head plus auxiliaries and adverbials is identified before any attempt to attach any post-verbal arguments. This stands in contrast to the grammatical approach adopted in frameworks such as that underlying the Penn TreeBank in which complements are attached to the verbal head at a lower level in the tree than auxiliaries and adverbials. This decision is again motivated by practical concerns: it is relatively easy to recognise verbal clusters, much harder to correctly attach complements.
4. A semantic analysis, or simplified quasi-logical form (SQLF), is produced for each phrasal constituent. In this representation all tensed verbs are interpreted as referring to unique events and all noun phrases are interpreted as referring to unique objects. Where relations between syntactic constituents have been identified from the syntactic constraints expressed in the grammar, relations between associated events and objects will be asserted.

While linguistically richer grammatical theories could be implemented in the grammar formalism underlying SUPPLE, the emphasis in our work has been on building robust wide-coverage tools — hence the requirement for only minimal lexical morphosyntactic and semantic information. As a consequence the combination of parser and grammars developed to date results in a tool that, although capable of returning full sentence analyses, more commonly returns results that include chunks of analysis with some, but not all, attachment relations determined.

2 Grammars in SUPPLE: basic formalism and extensions

2.1 Basic Grammar Formalism

The basic grammar formalism used within SUPPLE is a straightforward Prolog implementation of a feature-based CFG, whose rules have a character that will be familiar to anyone who has worked with Prolog Definite Clause Grammars. Categories are represented as compound terms in which the main syntactic category appears as the functor, and the remaining features appear in the associated argument positions, i.e. they take the form: `Category(Feat1:Val1,...,FeatN:ValN)`.

For example, we might have `np(person:first,num:sing,case:_)` as the (simplified) representation of an NP category. Here, each argument position contains a pair consisting of the feature name (some atom) and the associated value, which might be underspecified. The sharing of feature values between different categories in a rule can be achieved using Prolog variables, e.g. to enforce agreement. Although features names are included to aid reabability for grammar debugging, the fundamental encoding of feature identity is by argument position. Features representations are matched and combined using Prolog term unification. As in the above example, our use of this formalism has employed only *flat* feature representations, i.e. where feature representations do not embed. Rules provided to the system are represented in a Prolog fact style, e.g. for a rule $A \longrightarrow A_1, \ldots, A_n$, this would be `rule(`$A$`,[`$A_1$`,...,`$A_n$`])`.

2.2 Grammar Compilation

Grammars that are loaded into SUPPLE undergo a compilation process, which principally serves to provide additional functionality to simplify the work of the grammar writer. Firstly, compilation allows grammar rules to be written whose righthand side indicates optional or disjunctive constituents. Compilation enumerates the full set of category sequences that such rules allow, and creates a different compiled rule for each alternative, avoiding the need for the grammar writer to have to explicitly enumerate these alternatives. Compilation also provides another facility for the grammar writer, which is a matter of expressive power rather than convenience, which is to allow the grammar writer to specify that a given rule can only be used at the beginning or end of the sentence, i.e. requiring that the constituent constructed must appear in this position.

In addition, compilation provides facilities to help the grammar writer in working with feature-based representations. The basic feature representation approach described above, with its order-dependent encoding of feature identity, provides the grammar writer with ample opportunity to introduce errors, e.g. transposing the argument position of two adjacent features, omitting a feature, or mis-spelling its name. To avoid this problem,the set of features that may appear with each syntactic category can be defined as part of the grammar. The grammar writer can then specify category representations which mention only the features for which a constraint is specified, with the features given in any order. The compilation process then expands such category representations to include all argument slots, with the provided feature values being mapped to the correct positions, with all other feature values being left unspecified, and at the same time checking that only licensed features are used (catching any mis-spellings). A secondary use of compilation is to reorder the components of rules so as to be conveniently available to the chart process that searches for possible edge combinations, as will be briefly described in the next section.

3 Layered Chart Parsing

SUPPLE provides a layered parsing architecture, where several separate grammars may be applied in sequence, with the option available for the results of each layer to be filtered before being passed on as input to the next. Before describing this layered approach in more detail, we will firstly sketch the basic parsing algorithm employed within the layers.

3.1 The Basic Chart Algorithm

The parsing algorithm used within SUPPLE is a simple bottom-up chart approach. In contrast to familiar *active* chart algorithms, results are stored only for complete constituent analyses, i.e. all edges are inactive. The input to the parsing process is a sequence of edges, which might be for lexical items or multi-word units identified in preprocessing (e.g. by NERs), or might be constituents constructed by an earlier layer of parsing. Whenever an edge E is added to the chart, a check is made for grammar rules for which this edge might be the rightmost daughter constituent. For each such rule R, a scanning process is initiated, which exhaustively searches leftwards through the chart from the position of edge E, looking for edges corresponding to the other daughters of rule R. Whenever a contiguous sequence of edges for the daughters is found, a new edge is introduced spanning the width of the daughter edges and labelled with the mother category of the rule R. Such new edge additions are done immediately, i.e. a depth-first strategy is followed. Part of grammar compilation is geared towards facilitating this process, i.e. the compiled version of a rule $A \longrightarrow A_1, \ldots, A_n$ is stored in the form `compiled_rule(`A_n`,[`A_{n-1}`,...,`A_1`],`A`)`, allowing it to be accessed by its rightmost daughter category, and presenting the remaining daughters in the correct order for a leftwards traversal.

The completeness of this algorithm crucially requires that when an edge E is added to the chart, any edges for constituents that can be built to the left of E must *already* be present in the chart, so that they are available to the scanning process seeking edge combinations for E. Given the depth-first strategy for edge addition just mentioned, this ordering requirement for completeness can be met by sorting the initial input edges and adding them in order of ascending right-vertex value. However, this ordering method is only sufficient for parsing where all edges have non-zero widths, i.e. it is possible to contrive artificial examples for grammars involving empty categories, which might have zero-width inactive edges, for which the approach might not be complete, although these cases seem unlike anything that would arise in an actual linguistic analysis.

The implementation of the above algorithm exploits the characteristics of Prolog in a number of ways. Firstly, the edges that make up the chart are stored as facts in the Prolog database, and Prolog's capacity for matching into the database is exploited as the means for identifying edges that

might participate in edge combinations, and for identifying when a possible new edge is redundant. The exhaustive search for possible edge combinations, performed whenever a new edge is added to the chart, is achieved by exploiting Prolog's depth-first proof search mechanism, using a failure-driven method. Also, the buffering of edges awaiting addition is in effect handled by the Prolog control mechanism, i.e. there is no explicit *agenda* data-structure.

Many of these Prolog-related characteristics of SUPPLE were inherited from an earlier Prolog parser used at Sheffield, which was closely based on the Prolog-based bottom-up chart approach described in Gazdar and Mellish [3]. The latter is an active chart method, using familiar chart rules, i.e. the fundamental rule, and bottom-up rule-invocation. It uses active edges to represent partial analyses, i.e. where not all daughter edges needed to apply a grammar rule have yet been found, and this is its crucial difference to SUPPLE. The charts produced in practical parsing of real language can be very large. By not using active edges, the SUPPLE approach substantially reduces the size of its charts, reducing the costs of creating, storing and accessing edge representations, and trading this, in effect, against the cost of some redundancy in search effort that can be avoided using active edges. In informal experiments over a limited number of examples, we found that switching to the SUPPLE algorithm could reduce parsing time by as much as a factor of ten, although we cannot claim to have done a serious benchmark comparison.

3.2 Layered Parsing

SUPPLE allows parsing to be performed as a layered process, with a series of different grammars being specified for use at the different layers. The initial input to parsing will first be parsed using the grammar of the first layer, and then, when the chart algorithm has run to completion, results from that layer will serve as input to parsing at the next layer, and so on, until the last layer is reached. In our use of this layered approach, a best-parse selection process is applied after parsing at each layer to determine the results passed forward to the next. For each layer, a set of best-parse categories is defined alongside the grammar. Best-parse selection identifies a set of non-overlapping edges bearing only these categories, that achieve maximal coverage of the overall span of the chart, with a preference for edge sets involving the fewest edges. As might be expected, the use of such a selection process is associated with a loss of completeness in terms of what the grammars might yield without any filtering, but in a language engineering context, completeness is often a consideration that is secondary to efficiency and effectiveness.

There are two key motivations for this use of selection between layers of parsing. Firstly, the work done by some sets of grammar rules is naturally seen as separate from that done by other parts of the grammar. For example, we might want an initial grammar layer to do some work in building or extending NE expressions, or building terms of a special class such as protein names.

In the layered approach, such terms can be passed forward as non-decomposable units, with the subcomponent edges from which they have been built being left behind. A subsequent layer might build core NP chunks, which again are treated as non-decomposable units by the next level.

The second key motivation for the use of best parse selection is efficiency. Parsing real sentences with a wide-coverage grammar can produce unmanageably large charts, causing parsing to be slow, or even aborted. Furthermore, multiple alternative analyses will often be produced, and if only one of these can be carried forward for deeper processing in subsequent modules, we might as well filter less successful analyses at earlier stages so that fewer overall analyses are produced. By applying best-parse selection after each parsing layer, we can greatly reduce the number of edges generated overall through the parsing process, making it in general faster, and feasible for application to sentences that might otherwise exceed memory limits.

4 Robust, Wide Coverage Grammars for LE Applications

The preceding sections have presented the SUPPLE grammar formalism and parsing algorithm. In this section we discuss the example grammars we have developed in this framework for use in a range of language engineering applications dealing with text genres as diverse as newswire stories, biological science journal papers, and clinic letters from physicians.

4.1 Syntax

4.1.1 External Named Entity/Terminology Processing

To begin parsing a sentence SUPPLE requires a chart containing initial edges for single or multi-word lexical items with lexical category information. This information can be derived from a lexicon or from a part-of-speech tagger. However, in many cases general purpose English lexicons or POS taggers are inadequate for processing naturally occurring text due to the large numbers of propers names and/or technical terms.

Most applications that have used SUPPLE have presumed prior processes of NER and/or term recognition. Typically, this prior processing involves two stages: a gazetteer/terminology lexicon lookup step, and a pattern matching step using some form of grammar. The first stage allows for exact matching against known names or terms; the second stage implements some sort of generative capacity, where previously unseen names or terms and their types may be recognised from context or other clues. For example, orthography and known company indicators such as *plc* allow us to recognise *Global Widgets plc* as a company even without any prior information about *Global Widgets*.

If such a named entity or terminology recognition process exists the entities recognised by it are passed into SUPPLE as non-decomposable units with a syntactic category of basic noun phrase and a semantic type of whatever category the external process has assigned. This approach both exports the treatment of extensive domain specific lexical compounds to the appropriate specialised components and also speeds up the general parser which does not have to bother with the analysis of the internals of named entities or terminology.

4.1.2 General Phrasal Grammars

The general phrasal grammar in SUPPLE has been manually developed, being created partially with reference to Quirk *et al.* [7]. It is made up of eight subgrammars stored in separate files:

1. *Basic noun phrases* Noun phrases consisting of a single head noun, possibly preceded by a determiner or quantifier and/or more simple premodifiers. No recursive structure, co-ordination or post-modifiers.
2. *Prepositional phrases* Prepositions followed by basic noun phrases or noun phrases generally. Includes rules to build multiword prepositions out of expressions such as *close to* or *next to*.
3. *Noun phrases* Noun phrases which contain one or more basic noun phrases in more complex syntax arrangements that can still be unambiguously identified. For example, certain cases of prepositional phrase attachment (following a sentence intial basic noun phrase or involving the preposition *of*), certain cases of apposition, certain cases of co-ordination.
4. *Core verb clusters* Clusters consisting of verbal heads plus any preceding auxiliaries/modals or preceding/trailing adverbs. These rules add tense and aspect information to the semantics.
5. *Verb phrases* Finite or non-finite verb phrases containing a core verb cluster plus complements or attached prepositional phrases, where rules to do this can be written without introducing incorrect analyses.
6. *Relative clauses* Simple relative clauses consisting of wh-pronouns followed by verb phrases. Includes noun phrase rules for attaching relative clauses as post-modifiers in selected situations, such as following a sentence-initial basic noun phrase.
7. *Sentences* Sentential constructions including basic active and passive sentences, sentences with subordinate clauses, and sentences with sentential complements.
8. *Questions* Separate from the sentence grammars is a question grammar which has been developed for the TREC/QA evaluations. Deals with different types of *who, what, which, how, where*, and *when* questions.

The upper part of Figure 1 shows example rules from the sentence and verb phrase grammars.

A fragment of the grammar used in the analysis

```
%% S -> NP FVP
rule(s(sem:Event^Agent^[NPSem,VPSem,[lsubj,Event,Agent]]),
  [np(person:P,number:N,sem:Agent^NPSem),
   fvp(voice:active,person:P,number:N,sem:Event^VPSem)]).
%% FVP -> VP(tensed)
rule(fvp(person:P,number:N,tense:present,voice:V,m_root:R,
    sem:E^[VPSem]),
  [vp(m_root:R,voice:V,person:P,number:N,tense:present,sem:E^VPSem)]).
%% VP -> VPCORE NP
rule(vp(person:P,number:N,tense:T,voice:active,m_root:R,
    sem:E^X^[VPSem,[lobj,E,X],NPSem]),
  [vpcore(voice:active,m_root:R,person:P,number:N,tense:T,sem:E^VPSem),
   np(sem:X^NPSem)]).
```

Output semantic representation

```
[eat(e1), voice(e1,active), aspect(e1,simple), time(e1,present),
 cat(e2), det(e2,the), number(e2,sing), lsubj(e1,e2),
 mouse(e3), det(e3,the), number(e3,plural), lobj(e1,e3)]
```

Figure 1: Analysing the *The cat eats the mice*

As noted in Section 1, the grammar relies little on lexical semantics or syntactic constraints, so we take minimal risks in PP attachment, complementation, and co-ordination, providing rules to create such structures only when it is safe to do so. If the parser is unable to create a full analysis of a sentence, then fragments are produced. Attachments can be solved later by a more-informed process, using (possibly domain-specific) lexical or discourse information.

4.2 Semantics

The semantic representation of a sentence is constructed *compositionally* from the semantics of the sentence's constituents when the syntactic analysis takes place. Each noun phrase and each verb phrase in the sentence leads to the introduction of a unique identifier, e_k, which is used as a representation of the "entity" or "event" referred to by the noun/verb phrase. The semantics of a sentence is a conjunction of unary and binary predicates represented as a list of Prolog terms. We refer to this representation as a simplified quasi-logical form, or SQLF. Unary predicates in SQLF assert the existence and type of entities (e.g. `cat(e1)`) or events (e.g. `eat(2)`), while binary terms assert relations between or attributes of entities and events (hence the first argument of a binary predicate is always identifier). The set of unary predicates is derived from the citation form produced by a lemmatisation (or morphological) analysis and the argument of a unary predicate is an entity identifier. Some binary predicates encode grammatical information such as the logical subject (`lsubj`) or logical object (`lobj`) of a given event. Other binary predicates are used to

encode attributes of entities (e.g. adjectives) or events (e.g. time, modality).

Figure 1 illustrates aspects of the analysis of the sentence *The cat eats the mice*. The upper portion of the figure shows some of the grammar rules used in the analysis. Note how the `sem` feature is used to pass semantic information up the tree, combining elements from the semantic representations of constituents, possibly through the introduction of relational predicates such as `lobj` and `lsubj` which explcitly capture verbal arguments. The operator `^` is used to make event/entity identifiers available for unification steps in combining representations (a notation adapted from its use by Pereira and Shieber [6] in simulating lambda abstraction).

The two noun phrases of the sentence – *The cat* and *the mice* – are mapped into terms `cat(e2)` and `mouse(e3)` respectively which come from the morphological analysis of the head nouns. The verb phrase leads to the introduction of the predicate `eat(e1)`, created from the morphological analysis of the verb *eats*. The example illustrates the predicates used to represent the time, aspect, and voice of the verb, and the 1 quantification and number of nominals (e.g. `det`).

5 Evaluation

The combination of SUPPLE parser plus grammars has not been formally evaluated. This is principally due to the lack of an appropriate gold standard. The *de facto* community standard for parsing evaluation is the Penn TreeBank (PTB) [4]. Unfortunately the PTB and SUPPLE approaches to grammar are sufficiently different as to make it very difficult to compare outputs. We devoted some effort to attempting to devise a mapping (see [2]) but this effort did not yield quantitative results. Further work could be done in this direction, perhaps considering another resource such as Suzanne [8] which, while not appropriate for evaluating phrase structure, could perhaps be used to evaluate, e.g., logical subject and object relations.

Aside from evaluating SUPPLE against a gold standard annotated corpus, the other potential approach to parser evaluation is to use test suites [5]. This approach has the disadvantage of not providing an indication of how a parser will fare on unseen "real" text. However, it has the advantages of being well-suited to grammar development and being practically feasible for small development teams. We have also pursued this approach and have developed test suites for a number of grammatical phenomena (base noun phrases, core verb clusters and sentences). These test suites consist of sets of examples of each phenomenon plus expected output represented in the PTB bracketted parse notation (thus we evaluate only syntactic structure). To analyse the results we have developed two tools: one a program called *treediff*, which highlights differences in sets of paired trees, and another called it treeval, which is a Perl re-implementation of the Sekine and Collins *evalb* program that produces labelled and unlabelled crossing brackets measures. Together

these programs support regression testing during grammar development.

6 Using SUPPLE

At present SUPPLE may be used either as a standalone Prolog program or within GATE, the General Architecture for Text Engineering [1]. In both cases, text to be parsed is preprocessed to produce an initial chart represented as a sequence of Prolog `chart` terms, one per sentence. Each `chart` term contains a list of `edge` terms, one per single or multi-token lexical item, containing the feature-value information for that item that the grammar requires. For the distributed sample grammar, each lexical item has information of basic word class, morphological root, inflectional affix and surface form. Other information is word class specific, e.g. person and number features for nouns, and person, number, tense and verb form features for verbs. Items identified by NER or term recognition form edges with word class `ne_np`, with their semantic class (organization, location, etc.) marked as a feature.

If SUPPLE is used as a standalone module the user must provide preprocessing up to the point of producing the initial chart terms. Within GATE a sequence of modules which performs the preprocessing steps is available – a tokeniser, sentence splitter, morphological analyser, part-of-speech tagger (to assign word class) and various named entity and terminology recognisers for different domains. Furthermore the parser has been integrated into this sequence. That is, a wrapper is provided that assembles the relevant information produced by the preprocessing stages, which is stored as document annotations in the GATE document manager, and builds from it the input representation (i.e. initial chart) required by the parser. One feature of the wrapper is a configuration file which allows the user to specify declaratively the mapping between various GATE annotations produced in the preprocessing stages and the feature-value information used in the parser chart edges. This means that if a user wants either to replace the distributed grammar with their own, or wants to use different preprocessing modules (e.g. a POS tagger using a different tagset), a change to the mapping in this configuration file is all that is required.

The GATE wrapper also translates the output of the parser – both the syntactic and semantic analysis – into GATE annotations and stores them back into the GATE document manager. Then, the output may be passed on to other modules integrated into GATE, as part of a pipeline forming a larger application. Also, viewing tools available in GATE may be used to inspect the output. These include a standard annotation viewer which allows the syntactic and semantic representations to be viewed as text strings and a parse tree viewer which displays the phrase structure forest underlying syntactic analysis. SUPPLE can be compiled and run using SICStus Prolog, SWI Prolog, and Prolog Cafe, and GATE wrappers are provided for all three.

7 Conclusion

The SUPPLE parser has served as a component in a large set of research projects in Natural Language Processing. It is currently in use in a Question Answering system which participated in recent TREC/QA evaluations, and also in the CubReporter Project. We hope that its availability as a GATE component will facilitate its broader use by NLP researchers, and by others who others who wish to build more general applications that exploit NL technology.

Acknowledgements

The authors would like to acknowledge the support of the UK EPSRC under grants R91465 and K25267. They would also like to acknowledge the contributions of Chris Huyck and Sam Scott to the parser code and grammars.

References

[1] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

[2] R. Gaizauskas, M. Hepple, and C. Huyck. A scheme for comparative evaluation of diverse parsing systems. In *Proceedings of the 1st International Conference on Language Resources and Evaluation (LREC'98)*, pages 143–149, Granada, 1998.

[3] G. Gazdar and C. Mellish. *Natural Language Processing in Prolog*. Addison-Wesley, Wokingham, 1989.

[4] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[5] S. Oepen and D. P. Flickinger. Towards systematic grammar profiling. test suite technology 10 years after. *Computer Speech and Language*, 12(4):411–435, 1998.

[6] F.C.N Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis*. Number 10 in CLSI Lecture Notes. Stanford University, Stanford, CA, 1987.

[7] S. Quirk, R.and Greenbaum. *A University Grammar of English*. Longman, Harlow, Essex, 1973.

[8] G. Sampson. *English for the Computer: The SUSANNE Corpus and Analytic Scheme*. Clarendon Press, Oxford, 1995.