Software Support for Practical Grammar Development

Bran BOGURAEV, John CARROLL

Computer Laboratory, University of Cambridge
Pembroke Street, Cambridge CB2 3QG, England

Ted BRISCOE, Claire GROVER[1]

Department of Linguistics, University of Lancaster
Bailrigg, Lancaster LA1 4YT, England

Abstract

Even though progress in theoretical linguistics does not necessarily rely on the construction of working programs, a large proportion of current research in syntactic theory is facilitated by suitable computational tools. However, when natural language processing applications seek to draw on the results from new developments in theories of grammar, not only the nature of the tools has to change, but they face the challenge of reconciling the seemingly contradictory requirements of notational perspicuity and efficiency of performance. In this paper, we present a comparison and an evaluation of a number of software systems for grammar development, and argue that they are inadequate as practical tools for building wide-coverage grammars. We discuss a number of factors characteristic of this task, demonstrate how they influence the design of a suitable software environment, and describe the implementation of a system which has supported efficient development of a large computational grammar of English.[2]

1. Tools for Grammar Development

A number of research projects within the broad area of natural language processing (NLP) and theoretical linguistics make use of special purpose programs, which are beginning to be known under the general term of "grammar development environments" (GDEs). Particularly well known examples are reported in Kaplan (1983) (see also Kiparsky, 1985), Shieber (1984), Evans (1985), Phillips and Thompson (1985), Jensen *et al.* (1986) and Karttunen (1986). In all instances the software packages cited above fall in the class of computational tools used in theoretical (rather than applied) projects. Thus Kaplan's Grammar-writer's Workbench is an implementation of a particular linguistic theory (Lexical Functional Grammar; Kaplan and Bresnan, 1982); similarly, Evans' ProGram incorporated an early version of Generalized Phrase Structure Grammar (GPSG, Gazdar and Pullum, 1982), whilst PATR-II is a "virtual linguistic machine", developed by Shieber as a tool for experimenting with a variety of syntactic theories.

These systems differ in their goals. Particular implementations of a theory may be used for observing how theory-internal devices interact with each other, or to maintain internal consistency as the grammar is being developed. On the other hand, formalisms for encoding linguistic information in a uniform way underpin efforts to compare and evaluate alternative linguistic theories (Shieber, 1987). Neither type of system is adequate to the task of grammar development on a large scale or for incorporating such a grammar into a practical NLP system, due to factors such as efficiency of encoding (largely neglected in such systems) or verbosity and redundancy of the formal notation. Within the frameworks of their accomodating projects, these are in no way inadequacies of the computational tools; still, the applicability of the tools remains limited outside the strictly theoretical concern.

On the other hand, a number of syntactic formalisms have been used to develop wide-coverage grammars for use in practical NLP systems. The best known of these is the Augmented Transition Network formalism due to Woods (1970). More recent examples are the DIAGRAM grammar (Robinson, 1982) of SRI's TEAM natural language interface (Grosz *et al.*, 1987) and the PEG grammar developed at Yorktown Heights (Jensen *et al.*, 1986). Both are capable of impressive coverage and this is, to some extent, due to the more flexible formalisms employed. A common feature of these formalisms is that they all fall prey to what Kaplan (1987) refers to as "the procedural seduction" of computational linguistics: whatever the basis for the notation is, it incorporates a handle for explicit intervention into the interpretation of the grammar at hand.

Sometimes the nature of the task for which the grammar is being developed justifies a formal notation incorporating 'hooks' for explicit procedures. Thus a number of matchine translation (MT) projects, especially ones employing a transfer strategy, make use of formal systems for grammar specification, which, in addition to mapping surface strings into corresponding language structures, identify operations to be associated with nodes and / or subtrees (Vauquois & Boitet, 1985; Nagao *et al.*, 1985).

In general, the effects of the temptation to allow, for example, the EVALuation of arbitrary LISP expressions on the arcs of the ATN or the addition of "procedural programming facilities" to the rule-based skeleton of IBM's PLNLP have been discussed at length in the recent literature addressing the issues of declarative formalisms from a theoretical perspective (see Shieber, 1986a, and references therein). However, from the point of view of developing a realistic grammar with substantial coverage, the opening of the procedural 'back door', while perhaps useful for 'patching' the inadequacies in the linguistic theory during the exercise, can turn the whole process of grammar development and maintenance into an organisational nightmare, as side effects accumulate and ripple effects propagate.

A separate problem with allowing procedural attachment into the grammar formalism stems from the inevitable commitment to a particular version of a particular theory. Even when a deliberate effort is made to develop a flexible and general framework capable of accomodating a range of 'underlying' linguistic operations, such a framework is bound eventually to become inadequate, especially as modern theories of grammar (strive to) become more declarative and tend to make reference to larger bodies of knowledge. A case in point is the ARIANE system (Vauquois & Boitet, 1985): even though it was designed as a completely integrated programmaing environment, with the aim of enabling implementation of, and experimentation with, different linguistic theories, in reality the system has been unable to cope with radically new grammatical frameworks and computational strategies for text analysis.

The question then arises of the optimal way of developing a practical grammar. This paper will report on our experience in building such a grammar, with a particular emphasis on how a number of constraining factors have influenced the design and implementation of the software tools for supporting the linguist's work.

2. Design Considerations

For the last two years we have been engaged in a project aimed at substantial grammar development, as part of a larger effort to produce an integrated system for wide-coverage morphological and syntactic analysis of English. The overall objectives of the combined effort are described in a number of papers (see Russell *et al.*, 1986; Phillips and Thompson, 1987, and Briscoe *et al.*, 1987). We aimed to achieve comprehensive coverage of English in two years, using only one linguist and one programmer full-time; the complete natural language toolkit was to be made available to the research community outside the immediate environment where the grammar was being developed. Consequently, the software support for the linguist had to exhibit a number of characteristics to encourage high productivity. Particularly

[1] Currently with the IBM (UK) Science Centre.

[2] The work described here was supported by research grant GR/D/87321 from the UK Science and Engineering Research Council.

critical among these are *efficiency of implementation, perspicuity of representation, ease of use* and *robustness of performance.*

Current theories of syntax have much to offer to practical systems; such theories, however, are under constant development. For very pragmatic reasons, a project like ours ought to exploit a developed theory. For equally pragmatic reasons, it ought to be able to take advantage both of developments within the particular theory and of the evolving treatment of various linguistic phenomena. The question of the relationship between the theoretical formalism and the formalism adopted to implement a practical grammar based on that theory then becomes of central importance. For instance, it would be inappropriate to adopt a direct implementation of, say GPSG, since the rate of change of the theory itself is likely to make such an implementation obsolete (or at least incapable of incorporating subsequent linguistic analyses) quite rapidly — the brief lifespan of Evans' ProGram is a case in point. Only when theory and grammar are being developed in very close collaboration, or even within the same group — as in, for example, the Hewlett-Packard NLP project, whose cornerstone is the linguistic framework of Head-Driven Phrase Structure Grammar (Proudian and Pollard, 1985; Pollard and Sag, 1987) — could such an approach work.

However, in an effort like ours, it is of critical importance to strike the right balance between being faithful to the spirit of a theory and being uncommitted with respect to a particular version of it, as well as remaining flexible within the overall framework of 'close' or related theories. Attempts to be too flexible, however, are likely to lead to situations of which the PATR-II system is an example: the ability to model a wide range of theoretical devices and analytical frameworks is penalised by its unsuitability "for any major attempt at building natural-language grammars" (Shieber, 1984:364).

3. Our Approach

For a variety of reasons, intellectual and pragmatic, we chose to carry out the grammar development within the framework of GPSG (see Boguraev, 1988, for more details). Briscoe *et al.* (1987) discuss further some of the major issues concerning the dynamic interaction between the various rule types and constraints in GPSG and their impact on the implementability of the theory presented in Gazdar *et al.* (1985). From the practical perspective of computational grammar development, there are two important conclusions. In order to achieve implementability, the interpretation of the GPSG formalism requires a number of restrictions. In order to provide flexibility and expressive power, the formalism itself needs a number of extensions. In this light, the design of software support for grammar development becomes similar to the task of designing a special purpose, high level computer language, followed by an implementation of an interactive programming environment for it.

3.1 The Formalism

The grammar specification formalism, presented in detail in Carroll *et al.* (1988), is in fact a meta-grammatical formalism which avoids the direct implementation of one particular syntactic theory. While remaining close to the notation of GPSG, this formalism is nonetheless capable of specifying a range of syntactic theories and grammars. The specific choices during the design of such language have been heavily influenced by the desire to be moderately committed to a theoretical framework without being unnecessarily constrained within that framework. Finding the right balance places our system half way between the extreme positions exemplified by ProGram and the Grammar-writer's Workbench, on the one hand, and PATR-II, on the other.

The meta-grammatical formalism is designed to support a particular model of grammar development, suggested by, for example, Kay (1985). We maintain clear separation between a *meta-grammar*, which is "the seat of linguistic universals" (Kay, 1985:276), and an equivalent (in the sense that it describes exactly the same language) *object grammar*, coupled directly to the parser. The process of compiling the former into the latter constitutes the core of our GDE. A number of MT projects, also seeking substantial coverage, make a seemingly similar distinction between a source and object grammar—see, for instance, ARIANE's static and dynamic (or procedural) grammars (Vauquois & Boitet, 1985). However, there are differences, firsly in interpretation—the dynamic grammar largely incorporates whatever execution strategy is employed for transfer—and secondly in emphasis—a dynamic grammar is (necessarily) derived manually from a static one. Such efforts, then, do not have the notion of meta-grammar compilation, and consequently require less functionality from their support environments. We amplify this point below.

The separation between source and object grammars is the key to two of the considerations discussed in the previous section. By stopping short of embodying a particular theory, the formalism of the meta-grammar provides the linguist with an expressively flexible and powerful device for grammar writing. By assuming a parser, whose underlying operation is based on a restrictive version of unification, the object grammar allows an efficient implementation. More specifically, the object grammar is made up of phrase structure rules with feature complexes as categories; parsing with it is based on fixed-arity, term unification.

The meta-grammatical formalism is flexible and powerful. For example, it incorporates rule types for explicitly specifying feature propagation patterns, rather than 'hard-wiring' feature propagation into the interpretion of the rules (as in GPSG), and provides a variety of alternative rule formats, for example, PS or ID/LP rules, (non)-linear, (non)-lexical metarules, and so forth. The meta-grammar can be designed to be perspicuous, flexible and expressively powerful with little regard for issues of computational complexity because this complexity 'disappears' during compilation into the object grammar, leaving a well-defined, invariant and computationally tractable object grammar to be deployed at parse time. The process of compilation is based on ordered application of the various types of meta-grammatical rule to a set of 'base' PS or ID rules.

3.2 The Environment

The questions of optimal software environment for supporting grammar development, particularly in a rule-based formalism like ours, are very similar to the questions of interactive support for program development. A number of special-purpose tools have to be brought together in a tightly integrated 'shell' and organised around the core linguistic 'engine', which performs the reduction (compilation) of meta- into object grammar. These tools must suport

(1) rapid, incremental grammar development,

(2) interactive grammar debugging, and

(3) version maintenance and control.

The grammar development environment incorporates a number of modules, organised round the compilation process. In particular, the core functionality is provided by a *morphological analyser*, a *parser* for the object grammar, and a *generator*. The user interface consists of a command line interpreter, a number of special purpose viewing modules for meta-grammatical constructs, and a component for displaying parse trees on non-graphics terminals. The system is designed to be completely portable and machine-independent, which influenced the deliberate choice not to use any advanced graphics facilities. (These can incorporated if desired — indeed the system has been ported to both the Apple Macintosh and Xerox 1186 workstation).

3.2.1 Detecting Overgeneration

The need for a parser for grammar development is uncontroversial; it assists the linguist in finding gaps in grammatical coverage, checking the correctness of the syntactic description and weeding out spurious analyses. Our parser provides facilities for viewing syntactic descriptions in a variety of ways and batch parsing a growing corpus of examples to check the consistency of the developing grammar. Less obvious is the utility of a generator. Karttunen & Kay (1985:295f) discuss the use of such a component to generally explore the predictions made by a grammar concerning particular constructions.

However, their approach would not highlight the rules involved in overgeneration, particularly as the grammar grows in size. Our generator allows the linguist to guide generation either implicitly, by specifying rule-sets of interest, or explicitly, by directly manipulating (partial) syntax trees. For example, if the focus of interest is relative clauses, then she can request the GDE to ignore inappropriate rules (for example, those relating to coordination) and ask for automatic generation of examples with a specified maximum length whose root node is that appropriate to dominate a relative clause. Alternatively, she can build a syntax tree interactively by selecting the rule to apply from a menu of rule names generated automatically on specification of the next node to expand. Combining the two approaches allows automatic generation, for example, of specific types of relative clause; generation after building the partial syntax tree:

```
              Rel
             .   .
           .       .
         .           .
  NP[+wh]     S[SLASH NP]
```

would produce object relative clauses such as:

```
who every cat liked
who kim likes e
```

Automatic generation is a more natural technique for aiding discovery of overgeneration than parsing, because with the latter it is necessary for the linguist to guess where overgeneration may occur.

3.2.2 Efficient Grammar Compilation

The major potential bottleneck in grammar development is compilation, since changes to the grammar can only be fully evaluated by parsing or generating relevant examples. Complete grammar compilation is increasingly time consuming as the grammar grows; however, it does not have to be performed that often, given the ability to perform *incremental grammar compilation*. The term "incremental" here is taken to mean both as little as possible and as rarely as possible. By analogy with high-level languages for rapid prototyping, where disruptions of the program development cycle are avoided at all costs (consider, for instance, asynchronous garbage collection in Lisp), the intrusion of the grammar compiler into the linguist's work is kept to a minimum. Firstly, grammar compilation takes place 'on demand', so that the user need never worry about having to explicitly invoke it. Secondly, even though rules in large grammars tend to interact quite closely, it is rarely necessary to recompile the whole source every time an individual rule is changed. The GDE software caches compiled data to minimise the effort required during recompilation, and, by maintaining a model of the dynamic dependencies between a cluster of interconnected rules, it is able to ensure that the minimum amount of cached data is discarded when the grammar is changed. A consequence of this design is that individual components and rules at source level can be declared, and redefined, in any order. For example, the rule

```
S --> NP, VP.
```

may be defined before it is even decided which features make up Ss, NPs and VPs. The user may postpone this decision until she actually wants to use the rule for parsing a sentence. This experimental style of development parallels even further that promoted by highly interactive systems, since it allows easy experimentation with small fragments of the grammar, without requiring, for instance, compilation of the complete source or loading of all declarations.

Incremental compilation is made possible by designing the grammar compiler as a modular unit, comprising separate components for the interpretation of each of the statement types (for example alias declarations, feature propagation rules, or feature default statements) in the source (meta-grammatical) language. This has made it possible to combine these components into an integral package for efficient grammar compilation, as well as to incorporate them into individual commands, directly available to the linguist.

3.2.3 Effective Grammar Debugging

There are two further important consequences of our grammar compilation design. The first is the ability to *monitor the effects of grammar expansion*, by selectively filtering subsets of source grammar rules through specific compilation procedures. So, for example, the effects of a particular metarule can be assessed by applying it to a specified subset of 'base' rules. The second is the crucial capacity of *source level debugging*. In a development model which distinguishes between meta- and object grammars, efficient work is only possible if faulty grammar rules can be traced back to their original source in the meta-grammar. In our system, the output of a single command is usually sufficient to pinpoint an error in the source. Nodes in parse trees are labelled with the name of the grammar rule licensing the local tree rooted at that node. Unlike some other systems, such as ProGram, the name of an object grammar rule always uniquely encodes the complete derivation path of the rule. Thus, for example, the rule name VP/TAKES_NP(PASSIVE/+) uniquely identifies the rule derived from the application of the PASSIVE metarule to the rule introducing VPs taking a single NP complement which requires a PP 'agent' phrase (distinguished from the version without the PP by /+). Thus faults in object grammar rules can easily be traced back to their meta-grammatical source.

The use of unique rule names enhances the ability to view all or parts of the meta-grammar, as well as the results of partial compilation, along a number of dimensions, by means of patterns, with wild cards ranging over rule types and the names of rules. To facilitate this type of *grammar browsing*, arbitrary view requests can be constructed by using patterns compositionally; thus in a particular grammar of ours, the pattern

```
VP/PHRASAL*(*) & =NULL
```

refers to the collection of VP rules introducing phrasal verbs which have had metarules applied to them resulting in the introduction of the feature NULL. View requests may be further modified by indicating the level of detail required, i.e. whether the rules should be shown in their original source form, or partially or fully compiled.

Viewing parse trees particularly facilitates source level debugging. Displaying a tree from the perspective of rule names associated with the nodes, for example that resulting from parsing the phrase 'men and women':

```
      N/COORD1
        .  .
      .      .
    .          .
CONJ/NA  CONJ/NB
   .         . .
   .       .    .
   .     .        .
  men   and   women
```

can reveal whether right or wrong rules get activated. Fully displaying the category structures on tree nodes (Figure 1) gives an indication whether feature propagation regimes have been specified correctly. Viewing the gross structure of the tree, in this case

```
((men) (and women))
```

suggests whether the parse is correct or not; furthermore, in the case of multiple parses, nodes with common analyses can be factored out, thus helping localise the source of the error.

Errors are only dealt with at source level; editing facilities incorporate knowledge about the syntax of all constructs in the meta-grammatical formalism. The process of editing is integrated with *extensive bookkeeping*, which frees the grammar writer from the task of explicitly maintaining version backups and checks for consistency of the object grammar with respect to a particular meta-grammar.

The command interpreter is sensitive to work context and is capable, at any stage, to prompt for input appropriate to the current state in the grammar development process. For example, if the linguist has parsed a sentence which resulted in three analyses, she can display the category associated with any node of any of the analyses by typing

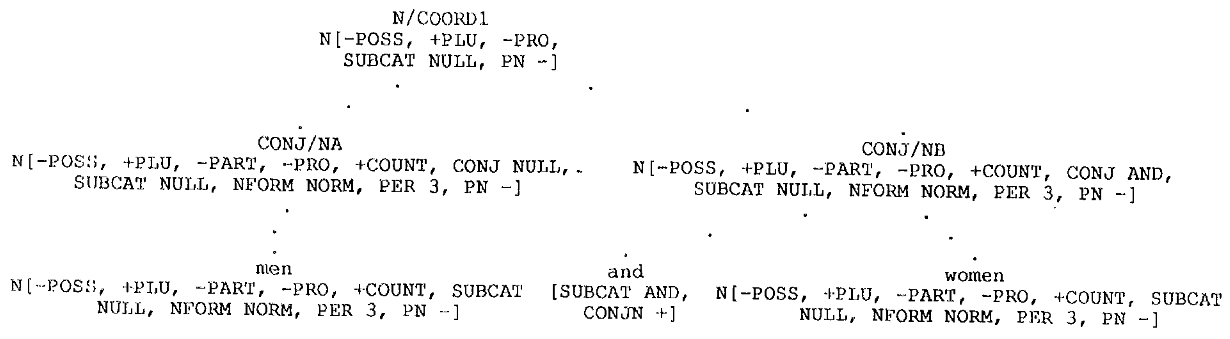
N/COORD1
N[-POSS, +PLU, -PRO, SUBCAT NULL, PN -]
CONJ/NA
N[-POSS, +PLU, -PART, -PRO, +COUNT, CONJ NULL, SUBCAT NULL, NFORM NORM, PER 3, PN -]
CONJ/NB
N[-POSS, +PLU, -PART, -PRO, +COUNT, CONJ AND, SUBCAT NULL, NFORM NORM, PER 3, PN -]
men
N[-POSS, +PLU, -PART, -PRO, +COUNT, SUBCAT NULL, NFORM NORM, PER 3, PN -]
and
[SUBCAT AND, CONJN +]
women
N[-POSS, +PLU, -PART, -PRO, +COUNT, SUBCAT NULL, NFORM NORM, PER 3, PN -]

Figure 1. Fully Detailed Parse Tree for 'men and women'.

a single command requiring two arguments. Alternatively, by just typing carriage return after the command name, she can request the GDE command interpreter to prompt for values for these parameters by displaying menus of values only applicable to the current work context, for example

```
> view
Rules/Full/CAtegory ...? category
Parse tree number (1 to 3)? 1
Appropriate tree nodes are:
   1. N/COORD1 2. CONJ/NA  3. men
   4. CONJ/NB  5. and      6. women
Which one (give its number)? 1
[N +, V -, BAR 0, SUBCAT NULL, PRD @544,
   NFORM @545, PER @546, PLU +, COUNT @547,
   CASE @548, PN -, PRO -, PART @549, POSS -]
```

In this fashion, potentially highly-ambiguous commands, such as *view* are localised to the current context. One of the unexpected consequences of this design is that it makes the system relatively accessible to inexperienced users and has made feasible the use of the system for educational purposes.

4. Conclusion

The design of a software system for grammar development clearly depends on the linguistic choices for, and pragmatic requirements of, the NLP task. It is not surprising that a number of MT efforts, motivated by the need for substantial coverage, have implemented their own GDEs. Perhaps the most comprehensive of these is the METAL-SHOP research environment of the METAL MT system (White, 1987), which includes facilities for selective viewing of parse trees, tracing of the grammar rules as they are invoked by the parser, and editing the grammar at source. The system makes, and conforms to, a clear-cut distinction between descriptive grammar rules and separate mechanisms for their interpretation. However, since the formal model used is that of augmented phrase structure grammar which does not undergo any compilation into object grammar, the functionality of the METAL-SHOP GDE, while adequate in the practical context it is used in, remains below that of the system we describe.

Even though we have worked within a particular theoretical framework, there are generalisations to be made concerning practical grammar development within the framework of any of the current syntactic theories. In particular, it is important to realise that software support for such a task does not imply, and should not be reduced to, the provision of a set of computational tools for e.g. grammar editing, inspecting the output from a parser, or comfortably interacting with the system. An effort of this scale crucially requires critical evaluation of the underlying linguistic theory, so that the right combination of pragmatically motivated and linguistically correct modifications and revisions is found and implemented. We are not alone in our findings; our approach to making computational sense of GPSG is similar to the (unimplemented) proposals of Shieber (1986b) and Ristad (1987).

The system described above is fully implemented and running on a number of hardware configurations. A wide-coverage grammar involving two woman/years of effort has been developed. A set of programs in Common Lisp, together with a user manual (Carroll *et al.*, 1988) and description of our grammar (Grover *et al.*, 1988) are available through the Artificial Intelligence Applications Institute in Edinburgh.

References

Boguraev, B. (1988) 'A natural language toolkit: reconciling theory with practice' in Rohrer C. & U. Reyle (ed.), *Natural Language Parsing and Linguistic Theories*, Reidel, Dordrecht, pp. 95–130.

Briscoe, E., C. Grover, B. Boguraev & J. Carroll (1987) 'A formalism and environment for the development of a large grammar of English', *Proceedings of 10th International Conference on Artificial Intelligence*, Milan, Italy, pp. 703–708.

Carroll, J., B. Boguraev, C. Grover & E. Briscoe (1988) *The Grammar Development Environment: User Manual*, Technical Report no. 127, Computer Laboratory, University of Cambridge.

Evans, R. (1985) 'ProGram — a development tool for GPSG grammars', *Linguistics, vol. 23(2)*, pp. 213–243.

Gazdar, G., E. Klein, G. Pullum & I. Sag (1985) *Generalized Phrase Structure Grammar*, Oxford: Blackwell and Cambridge: Harvard University Press.

Grover, C., E. Briscoe, B. Boguraev & J. Carroll (1988) *The Alvey Natural Language Tools Project Grammar: A Wide-Coverage Computational Grammar of English*, Lancaster Working Papers in Linguistics, no. 47.

Grosz, B., D. Appelt, M. Douglas & F. Pereira (1987) 'TEAM: An experiment in the design of transportable natural language interfaces', *Artificial Intelligence, vol. 32(2)*, pp. 173–244.

Jensen, K., G. Heidorn, S. Richardson & N. Haas (1986) *PLNLP, PEG and CRITIQUE: Three contributions to computing in the humanities*, Research Report RC 11841, Computer Science Department, IBM TJ Watson Research Center, Yorktown Heights, NY.

Kaplan, R. & J. Bresnan (1982) 'Lexical-functional grammar: a formal system for grammatical representation' in J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, MA, pp. 173–281.

Kaplan, R. (1987) 'Three seductions of computational psycholinguistics' in P. Whitelock et al. (ed.), *Linguistic Theory and Computer Applications*, Academic Press, New York, pp. 149–188.

Karttunen, L. (1986) 'D-PATR: A development environment for unification-based grammars', *Proceedings of 11th International Congress on Computational Linguistics*, Bonn, Germany, pp. 74–80.

Karttunen, L. & M. Kay (1985) 'Parsing in a free word order language' in Dowty, D., L. Karttunen & A. Zwicky (ed.), *Natural Language Parsing*, Cambridge University Press, Cambridge, pp. 279–306.

Kay, M. (1985) 'Parsing in functional unification grammar' in Dowty, D., L. Karttunen & A. Zwicky (ed.), *Natural Language Parsing*, Cambridge University Press, Cambridge, pp. 251–278.

Kiparsky, C. (1985) *LFG manual*, Manuscript, XEROX Palo Alto Research Center, Palo Alto, CA.

Nagao, M., Tsujii, J. & Nakamura, J. (1985) 'The Japanese Government Project for Machine Translation', *Computational Linguistics, vol. 11(2)*, pp. 91–110.

Phillips, J. & H. Thompson (1985) 'GPSGP — A parser for generalised phrase structure grammars', *Linguistics, vol. 23(2)*, pp. 245–261.

Phillips, J. & H. Thompson (1987) 'A parser and an appropriate computational representation for GPSG' in Klein, E. & N. Haddock (ed.), *Cognitive Science Working Papers*, Centre for Cognitive Science, University of Edinburgh.

Pollard, C. & I. Sag (1987) *Head-driven Phrase Structure Grammar*, CSLI Lecture Notes Number 12, CSLI, Stanford, CA.

Proudian, D. & C. Pollard (1985) 'Parsing head-driven phrase structure grammar', *Proceedings of 23rd Annual Meeting of the Association for Computational Linguistics*, Chicago, IL, pp. 167–171.

Ristad, E. (1987) 'Revised generalized phrase structure grammar', *Proceedings of 25th Annual Meeting of the Association for Computational Linguistics*, Stanford, CA, pp. 243–250.

Robinson, J. (1982) 'DIAGRAM: A grammar for dialogues', *Communications of the ACM, vol. 25(1)*, pp. 27–47.

Russell, G., S. Pulman, G. Ritchie & A. Black (1986) 'A dictionary and morphological analyser for English', *Proceedings of 11th International Congress on Computational Linguistics*, Bonn, Germany, pp. 277–279.

Shieber, S. (1984) 'The design of a computer language for linguistic information', *Proceedings of 10th International Congress on Computational Linguistics*, Stanford, California, pp. 362–366.

Shieber, S. (1986a) *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Notes Number 4, CSLI, Stanford, CA, and University of Chicago Press.

Shieber, S. (1986b) 'A simple reconstruction of GPSG', *Proceedings of 11th International Conference on Computational Linguistics*, Bonn, Germany, pp. 211–215.

Shieber, S. (1987) 'Separating linguistic analyses from linguistic theories' in P. Whitelock et al. (ed.), *Linguistic Theory and Computer Applications*, Academic Press, New York, pp. 1–36.

Vauquois, B. & Boitet, C. (1985) 'Automated Translation at Grenoble University', *Computational Linguistics, vol. 11(1)*, pp. 28–36.

White, J. (1987) 'The research environment in the METAL project' in Nirenburg, S. (ed.), *Machine translation: Theoretical and methodological issues*, Cambridge University Press, Cambridge, UK, pp. 225–246.

Woods, W. (1970) 'Transition network grammars for natural language analysis', *Communications of the ACM, vol. 13(8)*, pp. 591–606.