January 1992

Specifying Filler-Gap Dependency Parsers in a Linear-Logic Programming Language

Joshua S. Hodas
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/306
For more information, please contact repository@pobox.upenn.edu.
Specifying Filler-Gap Dependency Parsers in a Linear-Logic Programming Language

Abstract
An aspect of the Generalized Phrase Structure Grammar formalism proposed by Gazdar, et al. is the introduction of the notion of "slashed categories" to handle the parsing of structures, such as relative clauses, which involve unbounded dependencies. This has been implemented in Definite Clause Grammars through the technique of gap threading, in which a difference list of extracted noun phrases (gaps) is maintained. However, this technique is cumbersome, and can result in subtle soundness problems in the implemented grammars. Miller and Pareschi have proposed a method implementing gap threading at the logical level in intuitionistic logic. Unfortunately that implementation itself suffered from serious problems, which the authors recognized. This paper builds on work first presented with Miller in which we developed a filler-gap dependency parser in Girard's linear logic. This implementation suffers from non of the pitfalls of either the tradition implementational, or the intuitionistic one. It serves as further demonstration of the usefulness of sub-structural logic in natural language applications.

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/306
Specifying Filler-Gap Dependency Parsers in a Linear-Logic Programming Language

Joshua S. Hodas
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6839 USA
hodas@saul.cis.upenn.edu

Abstract

An aspect of the Generalized Phrase Structure Grammar formalism proposed by Gazdar, et al. is the introduction of the notion of "slashed categories" to handle the parsing of structures, such as relative clauses, which involve unbounded dependencies. This has been implemented in Definite Clause Grammars through the technique of gap threading, in which a difference list of extracted noun phrases (gaps) is maintained. However, this technique is cumbersome, and can result in subtle soundness problems in the implemented grammars. Miller and Pareschi have proposed a method of implementing gap threading at the logical level in intuitionistic logic. Unfortunately that implementation itself suffered from serious problems, which the authors recognized. This paper builds on work first presented with Miller in which we developed a filler-gap dependency parser in Girard's linear logic. This implementation suffers from none of the pitfalls of either the traditional implementation, or the intuitionistic one. It serves as further demonstration of the usefulness of sub-structural logic in natural language applications.

1 Introduction

It is now standard in linguistics and natural language processing to view a relative clause as being formed by a relative pronoun followed by a sentence that is missing a noun phrase. For example, the sentence:

John wrote the book [that Jane read].

can be thought of as having the following parse tree, where gap marks the spot where the missing noun phrase would be, if the clause were a sentence:

---

1This paper will appear in the proceedings of the 1992 Joint International Conference and Symposium on Logic Programming. A gap missing from the second example on page 11, has been inserted.
A common way to implement this idea in logic programming is the technique of *gap threading* in which a difference list of gaps is passed around as a parameter in a Definite Clause Grammar (DCG). The state of this list changes as gaps are introduced (by encountering relative pronouns, for example) and discharged (by completing a parse that uses the gap). In turn the state of the list controls whether the parse is allowed to use a gap in place of an NP at a given point in the parse.

Unfortunately, this technique has several drawbacks. It requires tedious modification of the entire grammar, even the parts that are not involved in parsing structures that need the gap list. Further, the difference list representation induces subtle bugs in the grammar that admit certain ungrammatical sentences which the underlying grammar being implemented rejects.

Pareschi and Miller proposed a method of handling unbounded filler-gap dependencies at the logic level, rather than in the term language. Their technique made use of the enhanced goal language of XProlog, which is based on hereditary Harrop formulas. The basic idea was to temporarily augment the grammar with a new rule for gap noun phrases only during the parse of a relative clause. Unfortunately, because the management of proof context in intuitionistic logic is too coarse, the grammars that result are unsound (though in a different way than the difference-list grammars), accepting many non-grammatical sentences.

In this paper I will begin by describing the Generalized Phrase Structure Grammar formalism which first proposed the basic ideas underlying all of these systems. I will then present the traditional gap threading technique as well as the system proposed by Pareschi and Miller and explain their shortcomings. Finally, I will present a solution to the problem inspired by Pareschi and Miller's work but implemented in Girard's linear logic. This solution, first briefly presented in joint papers with Miller [5, 7], addresses
all of the failings of the previous solutions, while maintaining the naturality of Miller and Pareschi's system.

2 Generalized Phrase Structure Grammar

The Generalized Phrase Structure Grammar (GPSG) formalism developed by Gerald Gazdar \[1, 2, 3\] demonstrated that it is possible to parse grammatical structures involving unbounded dependencies, such as relative clauses and wh questions, using a phrase structure grammar. Previously it had been thought that such constructs were too complex for phrase structure grammars, which are context-free, but rather required the strength of transformational grammar.

The basic ideas in GPSG are quite simple. It posits, for instance, that the body of a relative clause is a sentence that is missing a noun phrase somewhere. So, if sentences belong to the category S, and noun phrases to the category NP, then the rule for (one particular form of) relative clause would be:

\[
\text{REL} \rightarrow \text{that S/NP}^2
\]

where S/NP is the derived category of sentences missing a noun phrase. This requires, in turn, that rules be given for generating/parsing the derived category. These new rules are generated from the original grammar in a relatively straightforward manner. So, if the original grammar were:

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow PN \\
NP & \rightarrow DET N \\
VP & \rightarrow TV NP
\end{align*}
\]

then the new grammar, which allows relative clauses in noun phrases, would consist of that grammar augmented with:

\[
\begin{align*}
NP & \rightarrow DET N \ REL \\
REL & \rightarrow \text{that S/NP} \\
S/NP & \rightarrow NP \ VP/NP \\
VP/NP & \rightarrow TV \ NP/NP \\
NP/NP & \rightarrow \epsilon
\end{align*}
\]

In general, for each rule in the original grammar which defines a category that could dominate an NP (i.e., could occur above an NP in a parse tree) there will be a new version of that rule for each category on the right of the

\[2\] Actually, in the years since GPSG was first proposed it has changed significantly [12]. So, while the name has remained, the formalism no longer uses this sort of phrase structure rule, but instead uses node admissibility rules. In this paper I will use the original phrase structure rule style, as it is easier to understand the connection to DCG's in that form.
rule that could dominate an NP. Note, however, that we have not included
the derived rule:

\[ S/\text{NP} \rightarrow \text{NP}/\text{NP} \text{ VP} \]

in order to block extraction from the subject noun phrase. This is an over-
simplification of a standard restriction, which is intended to guard against
the acceptance of such sentences as:

* John wrote the book [that the story in gap is long].

3 Gap Threading in Prolog

There are many ways to approach implementing GPSG style grammars in
Prolog. Obviously, the grammar could be implemented directly as a DCG
with rules defining the base categories as well as each of the derived cate-
gories. This is not an attractive option, however, since, depending on the
number of derived categories, the resulting grammar can be substantially
(potentially quadratically) larger than the core grammar on which it is based.
Gazdar points out, however, that since the rules for the derived categories
are formed so uniformly from the original grammar, it is possible to use
the original grammar on its own, together with some switches controlling
whether the parser selects rules as is or in derived form [1, page 161].

So, for instance, the grammar above can be implemented with the fol-
lowing DCG:

\[
\begin{align*}
\text{s} & \rightarrow \text{s(nogap)}. \\
\text{s(Gap)} & \rightarrow \text{np(nogap) vp(Gap)}. \\
\text{np(gap)} & \rightarrow [I]. \\
\text{np(nogap)} & \rightarrow \text{pn}. \\
\text{np(nogap)} & \rightarrow \text{det n}. \\
\text{np(nogap)} & \rightarrow \text{det n rel}. \\
\text{vp(Gap)} & \rightarrow \text{tv np(Gap)}. \\
\text{rel} & \rightarrow [\text{that}] \text{s(gap)}. 
\end{align*}
\]

Each rule where the head is parameterized by \text{nogap} corresponds to a rule
in the core grammar only. In contrast, those parameterized by \text{Gap} act as
core rules when the parameter is instantiated to \text{nogap}, but as derived rules
when it is instantiated to \text{gap}. It is easy to see that this DCG implements
the grammar faithfully.

This system of switches is, however, too limited for grammars intended to
handle multiple extractions from nested structures. Therefore gap-threading
parsers (as this sort of system is called) are typically implemented with a dif-
ference list of gaps in place of the simple toggle. In such an implementation,
the DCG above becomes:
Although the difference list in this grammar will never grow beyond a single element, in more complex grammars the list structure is necessary. This technique of implementing GPSG parsers has been developed extensively by Pereira and Shieber [11] and others.

There are many problems with gap-threading parsers of this sort. First, they are difficult to construct. Insuring that the gap list is properly maintained can be quite subtle. Portions of the grammar that seem unconnected with the problem at hand require significant adjustment to insure that they do not interfere with the transmission of the gap information from the gap's introduction to its discharge, since with unbounded dependencies the two may be separated by almost any structure.

More importantly, due to the lack of “occurs check” in the unification algorithm used in languages like Prolog, serious soundness problems can occur with the difference list representation [8]. For instance, it is possible for the difference list to “go negative” in advance of the relative pronoun, only to balance out due to the presence of an extra noun phrase in the relative clause. Similar problems arise in complex grammars that allow multiple extractions from nested structures.

Thus, it is difficult to design a gap threading grammar intended to accept sentences like:

Which violins are these sonatas difficult to play gap on gap.

or:

I told Mary [that John wondered [who Jane saw gap]].

without it also accepting:

* I told gap [that John wondered [who Jane saw Sally]].

4 Gap Threading in Intuitionistic Logic

In 1990 Pareschi and Miller proposed using the expanded goal structure of λProlog to enhance the power of DCG’s [10]. That paper focused both on parsing and the construction of semantics. Here we are concerned only with the former, so I will summarize only that aspect of the work. λProlog is based on the hereditary Harrop formula fragment of intuitionistic logic.
Its enriched formula language provides a number of control structures not available in Prolog, and hence DCG's.

In particular, intuitionistic logic has the following rule for implication introduction:

$$\frac{\Gamma, D \rightarrow G}{\Gamma \rightarrow D \Rightarrow G \Rightarrow R}$$

If we take the view, as λProlog does, that logic programming is the bottom up search for cut-free proofs in intuitionistic logic, then this rule can be seen as giving a notion of scoping of clauses not available in the logic of Horn clauses. The clauses in $D$ are available only during the proof of $G$ [9].

Pareschi and Miller used this rule to provide a clean implementation of filler-gap dependency parsers. The basic idea is relatively simple: rather than use a complex system of parameters to control when the np $\rightarrow$ [] clause can be used — which is what all the mechanics of gap threading is really about — use the control provided by the above rule to scope the gapped-NP clause only over the derivation of the S that forms the body of the relative clause.

Using their implementation, the sample grammar would become:

\[\begin{align*}
\text{s} & : \text{l1 l2 :- np l1 lA, vp lA l2.} \\
\text{np} & : \text{l1 l2 :- pn l1 l2.} \\
\text{np} & : \text{l1 l2 :- det l1 lA, n lA l2.} \\
\text{np} & : \text{l1 l2 :- det l1 lA, n lA LB, rel LB l2.} \\
\text{vp} & : \text{l1 l2 :- tv l1 lA, np lA l2.} \\
\text{rel (that::l1) l2 :- (np Z Z) => s l1 l2.}
\end{align*}\]

Here the syntax of λProlog, which uses a curried notation (i.e. no parentheses) and does not include DCG’s, is used directly. As such, the difference list of words being parsed, which is implicit in DCG’s, is explicit here. The key feature of this implementation is that the bulk of the grammar is unchanged from the core grammar. The only mention of a gapped noun phrase is in the final rule. When this rule is invoked the unit clause np Z Z is added to the grammar and an attempt is made to parse for an S. Since the input and output lists of the assumed rule are the same, the rule represents a noun phrase with no phonological content: a gap. This rule may be used to complete any rule looking for an NP during the parse of the subordinate S.

The quantifier rules of λProlog are used to insure that an introduced gap is used only once. In particular, while λProlog uses the standard Prolog quantifier assumption that variables (identifiers beginning with a capital letter) are universally quantified at the outside of the clause in which they occur, it makes no such assumption about variables in clauses loaded using implication. Therefore the Z in the last clause above is quantified at the same level as the other variables in that clause. When the clause is used, the unit clause that is temporarily loaded into the grammar contains Z as an uninstantiated logic variable. Once the rule is used to fill in for a missing noun phrase, Z is instantiated to that location in the parse. It cannot
be instantiated for some other location, unless the parse fails back to some point before this. If we wanted to be able to use the gap to fill in for more than one noun phrase, then the last rule in the grammar would be written with \( Z \) explicitly universally quantified (for historical reasons, XProlog uses the operators \( \pi \) and \( \sigma \) for universal and existential quantification respectively):

\[
\text{rel (that::L1) L2 :- (pi Z \ (np Z Z)) } \Rightarrow s \ L1 \ L2.
\]

In order to handle restrictions on where a gap is allowable (such as the restriction on extraction from a subject noun phrase already discussed), Miller and Pareschi propose modifying the scheme to load gap locator rules rather than the gaps themselves. While this works, it is roughly equivalent to simply defining the grammar to include rules for derived categories up front; and as described earlier, that is quite cumbersome.

A serious problem with Pareschi and Miller's work, which the authors recognized, is that there is no logical method to require that an introduced gap be used. That is, if the goal \( s \ L1 \ L2 \) can succeed, then so can \( (np Z Z) \Rightarrow s \ L1 \ L2 \). This leads to the erroneous acceptance of:

* John wrote the book [that Jane read a magazine].

since

Jane read a magazine.

is a valid sentence, regardless of the presence of the assumed gap.

Technically, the problem is that in intuitionistic logic the freely available rule of weakening:

\[
\frac{\Gamma \rightarrow B}{\Gamma, A \rightarrow B} \ W
\]

allows unused assumptions to be simply discarded. This problem is familiar to knowledge representation and artificial intelligence researchers, and has led to the interest in relevance logic. In that system weakening is not allowed, so any assumptions must be relevant to the goal in order for a proof to exist.

5 Gap Threading in Linear Logic

In recognizing the limitations of their system, Pareschi and Miller suggested that a solution might be found in Girard's linear logic [4], which places strict limitations on the use of weakening and contraction. That proposal is the idea which underlies this work.

While the rules of intuitionistic linear logic are similar to those of ordinary intuitionistic logic, there is a significant difference: only formulas specifically designated (by being marked with the operator \( ! \)) can be weakened or contracted. Further, because of the restrictions on the structural rules there are two forms of conjunction, disjunction, and truth, which differ
based on the way they treat the proof context. To understand the need for
two conjunctions, for instance, consider two ways of axiomatizing conjunc-
tion introduction in intuitionistic logic:

\[
\frac{\Gamma \rightarrow A}{\Gamma \rightarrow A \land B} \quad \frac{\Delta \rightarrow B}{\Gamma, \Delta \rightarrow A \land B} \quad \frac{\Gamma \rightarrow A \rightarrow B}{\Gamma, \Gamma \rightarrow A \land B} \quad \frac{\Delta \rightarrow B \rightarrow \Gamma}{\Gamma \rightarrow A \land B} \quad \frac{\Gamma \rightarrow \Delta}{\Gamma \rightarrow A \land B} \\
\]

In the presence of the contraction rule, these two formulations are equivalent,
since the former rule can be replaced with the proof:

\[
\frac{\Gamma \rightarrow A \rightarrow B}{\Gamma \rightarrow A \land B} \quad \frac{\Gamma \rightarrow A \rightarrow B}{\Gamma \rightarrow A \land B} \quad \frac{\Gamma \rightarrow A \rightarrow B}{\Gamma \rightarrow A \land B} \quad \frac{\Gamma \rightarrow A \rightarrow B}{\Gamma \rightarrow A \land B} \\
\]

In this work we are concerned with only a fragment of intuitionistic linear
logic, given by the rules in figure 1. The proof theory of this fragment, and a
logic programming language based on it, have been discussed extensively in
two joint papers with Dale Miller [5, 7] and is the main topic of this author’s
dissertation [6]. A crucial point is that there is a straightforward bottom-
up, goal-directed proof procedure (conceptually similar to the one used for
Prolog) that is sound and complete for this fragment of linear logic\(^3\).

The solution, then, is simple. The rules of the grammar are represented
by ‘!ed formulas in the proof context (the left hand side of the sequent arrow)
since they are intended to be available for use as many times as needed, or
not at all. In contrast, the temporary rules for gapped noun phrases are
loaded without the ‘!. Since a gap is stored as a linear resource, it must be
used, and cannot simply be discarded.

The syntax of Lolli\(^4\), the language described in [5, 7], is intended to
be familiar to logic programmers. And, while the elements of the concrete
syntax represent different operators than they do in Prolog, they are designed
to have the same operational behavior. That is, if a pure-Prolog program
is entered into Lolli, its behavior will be the same as it was in Prolog. The
linear nature of the language is not evident if the program does not make
use of it. This operational equivalence is explained and proved in [7]. To
accomplish it, the following assumptions are made:

- ‘:-’ represents reverse \( \rightarrow \), ‘,’ represents \( \otimes \), ‘&’ (which does not occur in
  Prolog) represents &\&, and ‘{‘’ around a formula represent ‘. Finally, as in \( \lambda \)Prolog,’pi’ and ‘sigma’ are used for universal and existential
  quantification, respectively.

- The quantifier assumptions are the same as in Prolog (or more accu-
  rately \( \lambda \)Prolog). To wit, variables not explicitly quantified within a

\(^3\)This statement should be qualified. First, there are restrictions on the ways the logical
operators can be combined. Second, it is the non-deterministic or breadth-first algorithm
which is complete. The deterministic depth-first implementation actually used suffers from
the same failings as standard Prolog interpreters.

\(^4\)The language takes its name from the linear logic implication operator \( \rightarrow \) which, for
obvious reasons, is generally referred to as “lollipop”.
\[
\begin{align*}
    \frac{B \rightarrow B}{\Delta, B} & \quad \text{identity} \\
    \frac{\Delta \rightarrow \top}{\Delta} & \quad \text{unit} \\
    \frac{\Delta \rightarrow B, B & \rightarrow C}{\Delta, B_1 \& B_2 \rightarrow C} & \quad \&L (i = 1, 2) \\
    \frac{\Delta \rightarrow B}{\Delta} & \quad \&R \\
    \frac{\Delta, B \rightarrow C}{\Delta, B_1 \& B_2 \rightarrow C} & \quad \otimes L \\
    \frac{\Delta \rightarrow B, B_1 \& B_2 \rightarrow C}{\Delta, B_1 \otimes B_2 \rightarrow C} & \quad \otimes R \\
    \frac{\Delta \rightarrow C}{\Delta, !B \rightarrow C} & \quad \!W \\
    \frac{\Delta, !B \rightarrow C}{\Delta, !B \rightarrow C} & \quad \!C \\
    \frac{\Delta, !B \rightarrow C}{\Delta, !B \rightarrow C} & \quad \!D \\
    \frac{\Delta, !B \rightarrow C}{\Delta} & \quad \!R \\
    \frac{\Delta, B^{|x|} \rightarrow C}{\Delta, \forall x. B \rightarrow C} & \quad \forall L \\
    \frac{\Delta \rightarrow B^{|y|}}{\Delta} & \quad \forall R
\end{align*}
\]

provided that \( y \) is not free in the lower sequent.

Figure 1: A proof system for a fragment of linear logic

clause are assumed to be universally quantified at the clause’s boundaries.

- Clauses are assumed to be !’ed at the clause’s boundary. For proof theoretic reasons, the implicit !’s are placed outside the implicit quantifiers.

Given these assumptions, the Lolli version of this grammar is changed little from its \( \lambda \)Prolog counterpart. Only the implication in the last clause need be changed:

\[
\begin{align*}
    s & \rightarrow L_1 L_2 \neg \rightarrow \text{np } L_1 L_2, \text{vp } L_1 L_2 \\
    \text{np } L_1 L_2 & \rightarrow \text{pn } L_1 L_2 \\
    \text{np } L_1 L_2 & \rightarrow \text{det } L_1 L_2, \text{ n } L_1 L_2 \\
    \text{np } L_1 L_2 & \rightarrow \text{det } L_1 L_2, \text{ n } L_1 L_2, \text{ rel } L_1 L_2 \\
    \text{vp } L_1 L_2 & \rightarrow \text{tv } L_1 L_2, \text{ np } L_1 L_2 \\
    \text{rel } (\text{that::L1}) & \rightarrow (\text{np } Z Z) \rightarrow \neg s L_1 L_2.
\end{align*}
\]

It is important to note that due to the linear constraint, it makes no difference if the last rule is given as:

\[
\text{rel } (\text{that::L1}) L_2 \rightarrow (\text{pi } Z \backslash (\text{np } Z Z)) \rightarrow \neg s L_1 L_2.
\]

The loaded rule, representing a gapped noun phrase, can still be used only once. Each time a \( \otimes \) conjunction goal is encountered during the parse of the subordinate \( S \), the gap is carried up into the proof of only one side of the conjunction.
In addition to yielding a solution to the problems that Pareschi and Miller encountered, the linear logic setting affords simple treatments of other parsing issues. One particularly attractive feature of this system is its ability to specify the management of gaps across coordinate structures, such as conjuncts. GPSG proposes that any category can be expanded by the conjunction of two or more structures of the same category. So, for instance:

\[ S \rightarrow S \text{ and } S. \]

If the language level conjunction is represented in the grammar by the second form of logical conjunction, \&, then coordination constraints are handled automatically. That is, if the clause:

\[ s \text{ L1 L2} :\sim s \text{ L1 (and::LA)} \& s \text{ LA L2}. \]

is added to the grammar, then the system will accept the sentences

John wrote the book and Jane read the magazine.

and

John wrote the book [that Jane read gap and Jill discarded gap].

but will reject:

* John wrote the book [that Jane read gap and Jill discarded the magazine].

Because \& duplicates linear resources into both branches of the proof, both of the subordinate clauses must consume the same gaps. This scheme does not seem to provide any particular insight into the parsing of so-called “parasitic gaps”, though it is not clear that they are beyond the capabilities of the system.

In a similar manner, it is possible to use the ! operator to specify restrictions on extraction in the grammar. For example, in order to block extraction from subject noun phrases, the first rule of the grammar is rewritten as:

\[ s \text{ L1 L2} :\sim \{np \text{ L1 LA}\}, \text{ vp LA L2}. \]

Recall from Figure 1 that the proof rule for ! in a goal is:

\[
\frac{\frac{! \Gamma \rightarrow A !}{\Gamma \rightarrow ! A ! R}}{}
\]

where \( ! \Gamma = \{! D \mid D \in \Gamma \} \). In essence this states that a !'ed goal can only be proved in the presence of exclusively !'ed assumptions. Thus, if \( ! \Gamma \) is the grammar above, with the first rule modified as stated, then attempting to parse the relative clause:

[who Jane saw gap]
leads to a proof of the form:

\[
\begin{align*}
\Gamma & \rightarrow \text{np((jane::saw::nil), (saw::nil))} \\
\Gamma & \rightarrow \text{np((jane::saw::nil), (saw::nil))} \\
\Gamma & \rightarrow \text{tv((saw::nil), nil)} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow np(nil, nil)} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow np(nil, nil)} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow np((jane::saw::nil), (saw::nil))} \\
\Gamma & \rightarrow \text{np((jane::saw::nil), (saw::nil))} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow tv((saw::nil), nil)} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow np(nil, nil)} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow np(nil, nil)} \\
\Gamma & \rightarrow \text{np(Z, Z) \rightarrow np(nil, nil)} \\
\Gamma & \rightarrow \text{rel((who::jane::saw::nil), nil)}
\end{align*}
\]

The proof is somewhat abridged, in that applications of the various left hand ! rules (dereliction, weakening, and contraction), as well as implication on the left, have been hidden. In contrast, attempting to parse the clause:

* [that the story in gap is long]

will fail, because the gap np formula will be unavailable for use in the branch of the proof attempting to parse the NP "*the story in", since the l'ed np goal forces the gap np in the context to the other side of the tree.

Unfortunately, this technique is at once a bit too coarse-grained and too fine-grained. For instance, it blocks the acceptance of

[who gap saw Jane]

which should be allowed – the restriction on extraction from the subject noun should not block gapping of the entire subject, only its substructures. This problem can be circumvented by having multiple rules for relative clauses: one, as we have already shown, which introduces a gap and attempts to parse for an S, and one which simply attempts to parse for a VP.

A subtler problem is that the use of ! blocks all extractions from the subject, not just the extraction of noun phrases. This will become an issue when other types of gaps are introduced. Thus, while this technique can be used to implement this and other similar “island constraints”, there are some complications.

While the examples in the paper thus far have dealt only with relative clauses, GPSG proposes solutions to many other sorts of unbounded dependencies. For instance, given a category Q of non-wh questions, the category can be expanded to cover some wh questions with GPSG rules of the form:

\[
Q \rightarrow \text{wh-PP Q/PP}
\]

So that from questions like:

Did Jane read the book under the table?

one gets:

Where did Jane read the book?

It should be apparent that such extensions are easy to add in this setting. Figure 2 shows a larger grammar than those presented up till now that parses several forms of sentences and questions. (Only the grammar itself is shown, the pre-terminals and lexicon are removed for the sake of brevity.) Figure 3 shows a sample interaction with the parser, with several examples like those from the paper properly parsed or rejected.
parse Str Tree :- explode_words Str Lst, (s Lst nil Tree ; q Lst nil Tree).

s P1 P2 (s NP VP) :- {np P1 PA NP}, vp PA P2 VP.
s P1 P2 (and (s NP1 VP1) (s NP2 VP2)) :-
{np P1 PA1 NP1}, vp PA1 (and::PO) VP1) & {np P0 PA2 NP2}, vp PA2 P2 VP2).

q P1 P2 (q VFA NP VP) :- vfa P1 PA VFA, np PA PB NP, vp PB P2 VP.
q P1 P2 (q NW Q) :- NW P1 PA NW, {(pi P\ np P P (np gap)) -o q PA P2 Q}.
q P1 P2 (q PPW Q) :- ppwh P1 PA PPW, {(pi P\ pp P P (pp gap)) -o q PA P2 Q}.
q P1 P2 (q NW VP) :- NW P1 PA NW, vp PA P2 VP.

npwh P1 P2 (NW NWH) :- nwh P1 P2 NWH.
nwh which::P1 P2 (npvh which N) :- n PA P2 N.
ppvh P1 P2 (ppwh PWH) :- pwh P1 P2 PWH.

sb (that::Pi) P2 (sbar that S) :- s P1 P2 S.
qb P1 P2 (qbar NW VP) :- npwh P1 PA NW, vp PA P2 VP.
qb P1 P2 (qbar NW S) :- npwh P1 PA NW, {(pi P\ np P P (np gap)) -o s PA P2 S}.

np P1 P2 (np PNposs) :- npposs P1 P2 PNposs.
np P1 P2 (np Det Nposs OptPP OptRel) :-
det P1 PA Det, nposs PA PB Nposs, optpp PB PC OptPP, optrel PC P2 OptRel.

npposs P1 P2 (npposs PN) :- pn P1 P2 PN.
npposs P1 P2 (npposs PN s Nposs) :- pn P1 (s::PA) PN, nposs PA P2 Nposs.

nposs P1 P2 (nposs OptAP N) :- n PA P2 N.
nposs P1 P2 (nposs OptAP N s Nposs) :- n PA (s::PB) N, nposs PB P2 Nposs.

vp P1 P2 (vp DV NP PP) :- dv P1 PA DV, np PA PB NP, pp PB P2 PP.
vP P1 P2 (vp TV NP) :- tv P1 PA TV, np PA P2 NP.
vP P1 P2 (vp IV OptPP) :- iv P1 PA IV, optpp PA P2 OptPP.
vP P1 P2 (vp Stv Sb) :- stv P1 PA Stv, sb PA P2 Sb.
vP P1 P2 (vp TV NP Sb) :- tv P1 PA TV, np PA PB NP, sb PB P2 Sb.
vP P1 P2 (vp Qv Qb) :- qv P1 PA Qv, qb PA P2 Qb.
vP P1 P2 (vp Vfa VP) :- vfa P1 PA Vfa, vp PA P2 VP.

optpp P1 P2 (optpp epsilon). optpp P1 P2 (optpp PP) :- pp P1 P2 PP.
pp P1 P2 (pp P NP) :- p P1 PA P, np PA P2 NP.

optrel P1 P2 (optrel epsilon). optrel P1 P2 (optrel Rel) :- rel P1 P2 Rel.
rel (that::Pi) P2 (rel that VP) :- {vp P1 P2 VP}.
rel (who::Pi) P2 (rel who VP) :- {vp P1 P2 VP}.
rel (that::Pi) P2 (rel that S) :- {(pi P\ np P P (np gap)) -o s P1 P2 S}.
rel (whom::Pi) P2 (rel whom S) :- {(pi P\ np P P (np gap)) -o s P1 P2 S}.
rel P1 P2 (rel whom S) :-
p P1 (whom::PA) P, {(pi P\ pp P P (pp gap)) -o s PA P2 S}.
rel P1 P2 (rel which S) :-
p P1 (which::PA) P, {(pi P\ pp P P (pp gap)) -o s PA P2 S}.

Figure 2: An expanded filler-gap dependency parser
%lolli
Starting Lolli version 0.6, July 10, 1992
(built with Standard ML of New Jersey,
Version 75, November 11, 1991)...

?- nl2 --o top.

?- parse 'the program that john wrote halted' T.

?T <- s (np (det the) (nposs (n program)) (optpp epsilon)
 (optrel (rel that (s (np (pnposs (pn john)))
 (vp (tv wrote) (np gap))))))
(vp (iv halted) (optpp epsilon)).

?- parse 'i told mary that john wondered who jane saw' T.

?T <- s (np (pnposs (pn i)))
 (vp (tv told) (np (pnposs (pn mary)))
 (sbar that (s (np (pnposs (pn john)))
 (vp (qv wondered)
 (qbar (npwh (nwh who))
 (s (np (pnposs (pn jane)))
 (vp (tv saw) (np gap))))))).

?- parse 'i told that john wondered who jane saw sally' T.

no

?- parse 'which computer did john write the program on' T.

?T <- q (npwh which (optap epsilon) (n computer))
 (q (vfa did) (np (pnposs (pn john)))
 (vp (dv write) (np (det the) (nposs (n program)))
 (optpp epsilon) (optrel epsilon))
 (pp (p on) (np gap)))).

?- bye.
Closing proLLog.

Figure 3: A sample interaction with the expanded gap-threading parser
6 Conclusion

In this paper I have shown that the use of a linear logic programming language yields an extremely attractive and understandable implementation of many of the features of Generalized Phrase Structure Grammar. These include proper management of gaps in unbounded dependencies, as well as the handling of a simple form of island constraint and some forms of coordinate structures.

In addition, the implementations that result from using these techniques are particularly perspicuous, in that, in contrast to traditional techniques such as gap threading, they require very few changes to the core grammar. Finally, the system is immune to the soundness problems that occur in many difference-list-based gap threading systems.

7 Availability of the Lolli System

An implementation of Lolli, written in Standard ML of New Jersey, can be retrieved by anonymous ftp from ftp.cis.upenn.edu, in the directory pub/lolli. The system comes with several example programs including the full version of the parser given in Figure 2. The directory also includes DVI versions of most of the papers pertaining to Lolli. If you do retrieve the Lolli system, please send mail to hodas@saul.cis.upenn.edu so that you can be informed of updates to the system.

8 Acknowledgements

The author is grateful to Mark Steedman, Dale Miller, Fernando Pereira, Pat Lincoln, and Elizabeth Hodas for their helpful conversations about this work, and to Remo Pareschi for conversations about the work that led up to it. He is also thankful to Mark Johnson for his ideas relative to the problems with traditional gap threading systems. Finally he is most grateful to the paper's referees, who helped correct certain mis-statements about GPSG, and provided improved example sentences.

The author has been funded by ONR N00014-88-K-0633, NSF CCR-91-02753, and DARPA N00014-85-K-0018 through the University of Pennsylvania.

References


Generalized Phrase Structure Grammar. Harvard University Press, 


[5] Joshua Hodas and Dale Miller. Logic programming in a fragment of
intuitionistic linear logic: Extended abstract. In G. Kahn, editor, Sixth
Annual Symposium on Logic in Computer Science, pages 32 – 42, Am-

Theory, Design, and Implementation. PhD thesis, University of Penn-
sylvania, Department of Computer and Information Science, expected
1993.

of intuitionistic linear logic. Journal of Information and Computation,


International Logic Programming Conference, pages 810–827, Seattle,

[10] Remo Pareschi and Dale Miller. Extending definite clause grammars
with scoping constructs. In David H. D. Warren and Peter Szeredi,
eeditors, 1990 International Conference in Logic Programming, pages

[11] Fernando C. N. Pereira and Stuart M. Shieber. Prolog and Natural-

[12] Peter Sells. Lectures on Contemporary Syntactic Theories. Number 3 in
CSLI Lecture Notes. Center for the Study of Language and Information,