Chapter 6

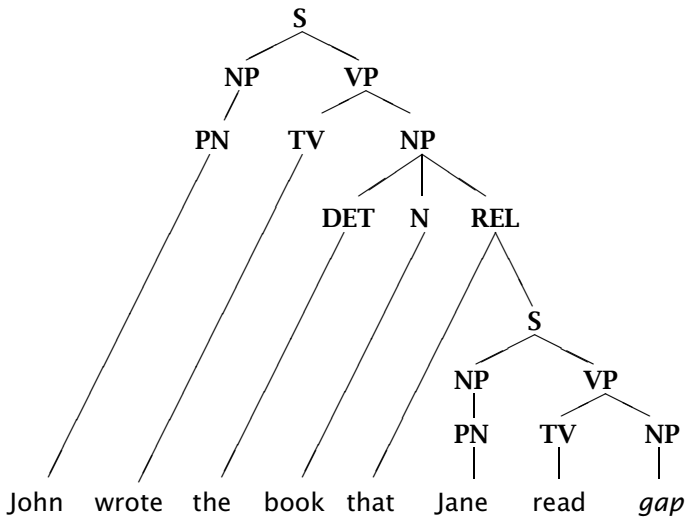
$\mathcal{L}$ as an Extension of Phrase Structure Grammar

6.1 Unbounded Dependencies in Natural Language Processing

It is now standard in the fields of computational 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 *gap*].

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:



This sort of construct is called an *unbounded dependency* (in this case an *unbounded filler-gap dependency*), because two features (in this case the filler, that, and the *gap*) which stand in a dependency relationship can be separated in a sentence by arbitrary grammatical structures. And while in this example the extraction site is at the rightmost end of the subordinate sentence, it can in fact occur almost anywhere. So, for instance, the following sentence is also acceptable:

John wrote the book that [Jane gave *gap* to Jill].

A common technique for implementing parsers that can handle such structures 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 [Pereira and Shieber, 1987].

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 can induce subtle bugs in the system that cause it to accept certain ungrammatical sentences which the underlying grammar rejects.

Pareschi 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 λProlog. 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 [Pareschi, 1989; Pareschi and Miller, 1990].

In this chapter 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 Lolli. This solution, addresses all of the failings of the previous solutions, while maintaining the naturality of Miller and Pareschi's system. I will assume that the reader is familiar with the basics of phrase structure grammars and definite clause grammars (their Prolog analogues).

The ideas in this chapter were first briefly presented in the joint papers with Miller which introduced our system of linear-logic programming [Hodas and Miller, 1991; Hodas and Miller,

1994].[1] The idea was developed in greater depth by this author in a paper for the 1992 Joint International Conference and Symposium on Logic Programming [Hodas, 1992]. This chapter is largely a revision of that paper.

6.2 Generalized Phrase Structure Grammar

The Generalized Phrase Structure Grammar (GPSG) formalism developed by Gerald Gazdar [Gazdar, 1981; Gazdar, 1982; Gazdar *et al.*, 1985] 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:

REL⟶ REL-PRON S/NP

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:

S ⟶ NP VP
NP ⟶ PN
NP ⟶ DET N
VP ⟶ TV NP
VP ⟶ DTV NP NP

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

NP ⟶ DET N REL
REL ⟶ REL-PRON S/NP

S/NP ⟶ NP VP/NP
VP/NP ⟶ TV NP/NP

[1] In his dissertation Pareschi proposed the idea of using linear logic to solve the problems in his system. He did not, however, predict all the additional benefits that would arise from this sort of solution.

VP/NP $\longrightarrow$ **DTV NP/NP NP**
VP/NP $\longrightarrow$ **DTV NP NP/NP**
NP/NP $\longrightarrow \epsilon$

6.2.1 Island Constraints

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 rule that could dominate an **NP**. Note, however, that we have not included the derived rule:

S/NP $\longrightarrow$ **NP/NP VP**

This was done in order to block extraction from within the subject noun phrase. If this rule were included in the grammar then the grammar would incorrectly admit such sentences as:

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

This sort of restriction, in which certain constructions are disallowed in certain positions is called an *island constraint*. We will have more to say about this particular island constraint, which is sometimes called the *generalized left-branch condition*, below.

6.3 GPSG 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 categories. 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 [Gazdar, 1981, page 161].

To understand how this idea can be implemented in DCG grammars, consider for example the two rules:

S $\longrightarrow$ **NP VP**
S/NP $\longrightarrow$ **NP VP/NP**

If we think in terms of a switch that can have two values, gap and nogap, then these two rules can be mapped to the single DCG rule:

```
s(Gap) --> np, vp(Gap).
```

When the variable Gap is instantiated to nogap this rule corresponds to the first GPSG rule. When it is instantiated to gap it corresponds to the second GPSG rule. So, for instance, the grammar above can be implemented with the following DCG:

```
s          --> s(nogap).
s(Gap)     --> np(nogap), vp(Gap).
np(gap)    --> [].
np(nogap)  --> pn.
np(nogap)  --> det, n.
np(nogap)  --> det, n, rel.
vp(Gap)    --> tv, np(Gap).
vp(Gap)    --> dtv, np(Gap), np(nogap).
vp(Gap)    --> dtv, np(nogap), np(Gap).
rel        --> rel-pron, s(gap).
```

Each rule where the head is parameterized by nogap corresponds to a rule in the core grammar only. Rules with heads parameterized by gap are slashed rules only. Finally, those parameterized by Gap act as core rules when the parameter is instantiated to nogap, but as slashed rules when it is instantiated to gap. It is easy to see that this DCG implements the grammar faithfully.

This implementation has problems though. First, as pointed out by Pereira and Shieber, because the two DCG rules for verb phrases with ditransitive verbs are identical when Gap is instantiated to nogap, simple sentences with ditransitive verbs will have two identical parses, depending on which rule is used in the parse [Pereira and Shieber, 1987]. In addition, the system of switches is too limited for grammars intended to handle multiple extractions from nested structures. Therefore gap-threading parsers (as these sorts of systems are called) are typically implemented with a difference list of gaps in place of the simple toggle. In such an implementation, the DCG above becomes:

```
s(S)           --> s([]-[]).
s(F0-F)        --> np([]-[]), vp(F0-F).
np([gap|F]-F)  --> [].
np(F-F)        --> pn.
np(F-F)        --> det, np([]-[]).
np(F-F)        --> det, np([]-[]), rel.
vp(F0-F)       --> tv, np(F0-F).
vp(F0-F)       --> dtv, np(F0-F1), np(F1-F).
rel            --> rel-pron, s([gap|F]-F).
```

Although the difference list in this grammar will never grow beyond a single element, in more complex grammars it can. Notice, though, that there is now just a single rule for ditransitive verb phrases. The difference list of gaps is divided between the two object noun phrases; if

there is just one gap in the list it will go to one of the two noun phrases. This technique for implementing GPSG parsers has been developed extensively by Pereira and Shieber [Pereira and Shieber, 1987] and others.

Unfortunately, while using a difference list to manage gap distribution solves these problems, it causes others. The most serious is that, 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 [Johnson, 1992]. 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 are designed to admit multiple extractions from nested structures (a construction not handled by GPSG).

Thus, it is difficult to design a gap threading grammar intended to admit 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 admitting:

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

A problem common to both implementations is that they are difficult to construct. Insuring that the gap distribution is correct can be quite subtle. Precisely because the filler-gap dependency is unbounded, 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 the two may be separated by almost any structure.

6.4 GPSG in Intuitionistic Logic

Notice that the first DCG implementation of the grammar above includes the rule:

```
np(gap) --> [].
```

which, if we ignore the switch argument, corresponds to the grammar rule:

$\mathbf{NP} \longrightarrow \epsilon$

In some sense the purpose of the switches, the difference lists, and even the slashes in the original grammar, is to control when this rule can be used. These machinations are necessary because DCG's and PSG's, like Horn clauses, are flat.

In his Dissertation, and later in a paper with Miller, Pareschi proposed using the expanded formula language of λProlog to enhance the expressiveness of DCG's [Pareschi and Miller, 1990].[2] In particular, Pareschi proposed using the $\supset_r$ rule to restrict the scope of the **NP** $\longrightarrow \epsilon$ rule to the derivation of the **S** that forms the body of the relative clause.

If we imagine augmenting the standard DCG syntax with the '=>' operator from λProlog then using this idea we end up with the following implementation of the grammar:

```
s   --> np, vp.
np  --> pn.
np  --> det, n.
np  --> det, n, rel.
vp  --> tv, np.
vp  --> dtv, np, np.
rel --> rel-pron, (np --> []) => s.
```

The key feature of this implementation is that the bulk of the grammar is unchanged from the core grammar. The only gap-related machinery is in the final rule. When this rule is invoked the rule `np --> []` is added to the grammar and an attempt is made to parse for an **S**. This rule may be used to complete any rule looking for an **NP** during the parse of the subordinate **S**.

Actually, λProlog does not directly support DCG syntax, and Pareschi and Miller described their techniques in terms of the underlying Prolog (and λProlog) clauses. Such clauses contain a difference list of words as an argument to every predicate, which is used to distribute the words in the string being parsed in much the same way that the gap-list above distributes gaps. This representation was used to advantage by Pareschi and Miller, as will be described below. This presentation will mostly be in terms of the augmented DCG syntax. For more on the relationship between DCG syntax and the underlying clauses see Pereira and Shieber [Pereira and Shieber, 1987].

6.4.1 Island Constraints

In order to handle restrictions on where a gap is allowable (such as the restriction on extraction from a subject noun phrase already discussed), Pareschi proposed modifying the general scheme to load so-called "gap locator" rules rather than the gap noun phrase rules themselves. These gap locator rules are just partially interpreted versions of the rules that can use the gap. So for instance, rather than assuming a gap noun phrase rule, the rule for relative

[2] Pareschi's dissertation, and the paper with Miller focused both on parsing and the construction of semantics. In this chapter we will be concerned only with the former, so I will summarize only that aspect of the work. However, if we consider a version of Lolli with λ-terms and unification in the style of λProlog then the treatment of semantics could be mapped directly to this presentation.

clauses might become:

```
rel --> rel-pron, (vp --> tv) => (vp --> dtv np) => s.
```

While this works, it is roughly equivalent to defining the grammar to include rules for derived categories up front; and can become quite cumbersome if there are many categories immediately dominated by **S** which can dominate an **NP**.

6.4.2 Problems Due to Contraction and Weakening

A serious problem with this simple representation, which Pareschi recognized, is that, as mentioned above, the `np --> []` rule can be used to complete *any* rule looking for an **NP** during the parse of the subordinate **S**. In particular, it could be used to fill both the direct and indirect object positions in a ditransitive verb phrase. So the grammar above incorrectly admits:

* John wrote the book that [Jane gave *gap gap*].

Pareschi's solution to this problem was to take advantage of the fact that he was working with λProlog clauses rather than in the DCG syntax by directly manipulating quantifiers to block this situation. The solution relies on the fact that the intended translation of the augmented DCG syntax:

```
rel --> rel-pron, (np --> []) => s.
```

is that the assumed rule is to have the same quantifier treatment as any other rule. So the translation into λProlog clauses would be:

```
rel S0 S1 :- rel-pron S0 S2, (pi T0 \ (np T0 T0)) => s S2 S1.
```

Pareschi noted that if the explicit quantification of `T0` is dropped, then when the gap noun phrase rule is added to the program in an attempt to parse a relative clause, `T0` will be an uninstantiated logic variable. The first time that rule is used to fill a noun phrase in the subordinate sentence structure the variable will become instantiated to that string position. After that the rule cannot be used to complete a different noun phrase, unless the parse fails back to some point before this.[3]

A similar problem with this system is that there is no logical method to require that an introduced gap be used. That is, the grammar will admit:

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

[3] If this is unclear, look at the discussion of quantification in hypothetical queries in the registrar's data base example in Chapter 3.

since

Jane read a magazine.

is a valid sentence, and that is what the parser is looking for once it has seen the relative pronoun and assumed the scoped noun phrase rule.

In this case Pareschi's solution is reliant on the fact that his system builds semantic terms in conjunction with parse trees, and that this erroneous acceptance would lead to a vacuous abstraction —i.e. a term of the form $\lambda x.t$ where x is not free in t— in the semantic term constructed for the relative clause. It is possible to check for the presence of such a term using the λProlog unification mechanism in conjunction with negation-as-failure. Pareschi, therefore, discards any parse that constructs a semantic term with a vacuous abstraction.

6.5 GPSG in Lolli

While Pareschi's solutions to the problems in the last section work, they are unsatisfying in several ways. Most seriously, it seems unfortunate that dual problems of multiplicity and relevance in the treatment of the assumed noun phrase rule should have such disparate solutions. The solutions feel a bit too close to "hacking" the grammar.

In recognizing the limitations of his system, Pareschi suggested that a solution might be found in Girard's linear logic, which at the time had only recently been introduced [Girard, 1987*a*].

The solution, should be readily apparent. The rules of the grammar are loaded into the unbounded portion of the proof context, 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 into the linear context. Since a gap is stored as a bounded resource, it must be used exactly once in the subordinate sentence.

Thus if we further augment DCG syntax with the Lolli operator `-o`, then the Lolli version of the grammar becomes:

```
s   --> np, vp.
np  --> pn.
np  --> det, n.
np  --> det, n, rel.
vp  --> tv, np.
vp  --> dtv, np, np.
rel --> rel-pron, (np --> []) -o s.
```

In order to insure that the gap rules are managed correctly, we must further assume, by analogy to the embedding of λProlog into Lolli given in Chapter 4, that the comma and DCG arrow now stand for '⊗' and '⟜' respectively. Each time a '⊗' 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.

6.5.1 Coordination Constraints

In addition to yielding solutions 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 $\longrightarrow$ **S** and **S**.

It is required, however, that any gaps discharged in one conjunct also be discharged in the other. This is known as a *coordination constraint*. If the language level conjunction is represented in the grammar by the second form of logical conjunction, '&', then coordination constraints are handled automatically. Since the '&' goal duplicates the linear context into both branches of the proof, both of the subordinate parses must consume the same gaps. Thus, if the clause:

```
s  --> (s, [and]) & s.
```

is added to the grammar,[4] then the system will admit 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].

6.5.2 Island Constraints

In a similar manner, it is possible to use the ! operator in goals 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 can be rewritten as:

[4] In reality, since performing a parse in this setting corresponds to top-down parsing, we cannot add this or any similar rule to the proof as it would cause the parser to loop. Thus we must actually add a rule of the form:

```
s  --> (np, vp, [and]) & (np, vp).
```

but this is a technical issue only.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\varnothing \longrightarrow pn((jane::saw::\mathsf{nil}),(saw::\mathsf{nil}))}}{\Gamma;\varnothing \longrightarrow np((jane::saw::\mathsf{nil}),(saw::\mathsf{nil}))}\,BC}{\Gamma;\varnothing \longrightarrow !np((jane::saw::\mathsf{nil}),(saw::\mathsf{nil}))}\,!_R \quad \dfrac{\dfrac{\dfrac{}{\Gamma;\varnothing \longrightarrow tv((saw::\mathsf{nil}),\mathsf{nil})} \quad \dfrac{}{\Gamma;\forall t_0.np(t_0,t_0) \longrightarrow np(\mathsf{nil},\mathsf{nil})}\,BC}{\Gamma;\forall t_0.np(t_0,t_0) \longrightarrow tv((saw::\mathsf{nil}),\mathsf{nil}) \otimes np(\mathsf{nil},\mathsf{nil})}\,\otimes_R}{\Gamma;\forall t_0.np(t_0,t_0) \longrightarrow vp((saw::\mathsf{nil}),\mathsf{nil})}\,BC}{\Gamma;\forall t_0.np(t_0,t_0) \longrightarrow !np((jane::saw::\mathsf{nil}),(saw::\mathsf{nil})) \otimes vp((saw::\mathsf{nil}),\mathsf{nil})}\,\otimes_R}{\Gamma;\forall t_0.np(t_0,t_0) \longrightarrow s((jane::saw::\mathsf{nil}),\mathsf{nil})}\,BC}{\Gamma;\varnothing \longrightarrow (\forall t_0.np(t_0,t_0)) \multimap s((jane::saw::\mathsf{nil}),\mathsf{nil})}\,{\multimap}_R}{\Gamma;\varnothing \longrightarrow rel((that::jane::saw::\mathsf{nil}),\mathsf{nil})}\,BC*$$

where the boxed premise is $t_0 = \mathsf{nil}$.

Figure 6.1: The Parse-Proof of "that jane saw"

```
s --> {np}, vp.
```
[5]

Recall from Figure 4.2 that the proof rule for ! in a goal is:

$$\dfrac{\Gamma;\varnothing \longrightarrow A}{\Gamma;\varnothing \longrightarrow !A}\,!_R$$

Thus, if Γ is the grammar above, with the first rule modified as stated, together with lexical definitions for a sufficient set of words, then attempting to parse the relative clause:

that [Jane saw *gap*]

leads to a proof of the form shown in Figure 6.1. 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 *gap*", since the !'ed *np* goal forces the gap *np* in the linear context to the other side of the tree.

Unfortunately, this technique is a bit too coarse-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 extraction 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**. The other choice would be to avoid the use of '!' entirely by using gap locator rules as in Pareschi's system; but this seems unattractive for reasons already cited.

[5] This use of '`{}`' conflicts with their existing use for embedding Prolog code in a DCG clause. Since the square brackets, '`[]`', which are used for lists in Prolog and DCG clauses are not used for that purpose in Lolli, they are used for embedding Lolli code in Lolli DCG clauses.

6.5.3 Pied Piping

An interesting construction related to relative clauses is the so-called "pied-piping" construction, in which the relative pronoun is not the filler, but rather is properly embedded as the rightmost constituent within the filler. An example of this, taken from Pareschi, is the relative clause:

the sister of whom [Paul married *gap*]

In this case, the filler can be seen as being formed by concatenating the relative pronoun onto an **NP/NP**, with the added restriction that the *gap* occur only as the rightmost element of the **NP/NP**. It is interesting that this sort of restricted extraction is exactly the sort that can be modeled well in Categorial Grammar, which cannot model arbitrary extraction well. In contrast, in our system, which handles arbitrary extractions easily, there is no straightforward way to restrict the location of the gap to one particular position.

If we are willing to leave the representation of grammar rules as propositional formulas, and move to the representation with explicit string lists and quantifiers (as Pareschi did) then we can also adopt Pareschi's solution to this problem. Consider the clauses:

```
rel  S0 S1 :- relf S0 S2, (forall T0 \ (np T0 T0)) -o s S2 S1.
relf S0 S1 :- (np S2 S2) -o np S0 S2, rel-pron S2 S1.
```

The first clause just replaces the clause for relative pronouns with one in which the filler is in the new category **RELF**. The second clause says that a **RELF** is either just a relative pronoun (if the assumed gap is used immediately to fill in for the noun phrase, in which case $s_2 = s_0$), or it is an **NP/NP**followed by a relative pronoun. Notice, though, that the rule assumed for the gap does not have its string position variables universally quantified. Rather they are pre-bound to the same position as the end of the **NP/NP**. Thus it can only be used in that position.

6.5.4 Parasitic Gaps

Historically, one of the most difficult filler-gap constructions to handle is the so-called "parasitic gaps" in which a single filler can be used to fill two gaps, but, unlike coordinate structures, the second gap is optional. So, for instance, both of the following standard examples are acceptable noun phrases:

The book that [I discarded *gap* [without reading *gap*]]

The book that [I discarded *gap* [without reading the index]]

Pareschi proposes parsing such constructions by introducing two gaps but relaxing the relevance constraint on the second. Unfortunately in this system there is no way to relax the relevance constraint, though an extended system in which this is possible will be discussed in

Chapter 8. Another possible solution is to construct a special coordination constraint between the two verb phrases as in the following two clauses:

```
vp  --> (vp, [without]) &  vp.
vp  --> (vp, [without]) , {vp}.
```

The first rule is for the case in which the parasitic gap occurs, and the second for when it does not. In the second rule it is necessary to ensure that the gap is used in the first verb phrase and not the second, since otherwise the grammar would admit:

* The book that [I discarded the magazine [without reading *gap*]]

6.5.5 Other Types of Filler-Gap Dependencies

While the examples in this chapter have thus far 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 ⟶ wh-NP Q/NP

So that from questions like:

Did Jane read the book?

one gets:

What [did Jane read *gap*]?

It should be apparent that such extensions are easy to add in this setting. Figure 6.2 shows a larger grammar than those presented up till now that parses several forms of sentences and questions. This grammar is designed to return the parse tree it constructs as an argument of the non-terminal parsed. Only the grammar itself is shown, the pre-terminals and lexicon are removed for the sake of brevity. Figure 6.3 shows a sample interaction with the parser, with several examples like those from the paper properly parsed or rejected. Note that a translator (written in Lolli) from the augmented DCG syntax to Lolli syntax is included with the current Lolli distribution.

6.5.6 The Problem of Crossing Dependencies

Using $\mathcal{L}$ as the foundation of this sort of system induces what may be considered either a bug or a feature, depending on one's point of view. In constructing a grammar designed to admit constructions with multiple extractions from nested structures, it is not possible to block the generation of parse trees in which the filler-gap pairs are allowed to cross. Thus, when asked to parse the question:

```
GRAMMAR big_grammar.

parse Str Tree :- explode_words Str Lst,
                  (sent Tree Lst nil ; quest Tree Lst nil).

sent (sent NP VP)  --> {np NP}, vp VP.
sent (and (sent NP1 VP1) (sent NP2 VP2)) -->
        {np NP1}, vp VP1, (and::nil)   &   {np NP2}, vp VP2.

quest (quest VFA NP AP) --> vfa VFA, np NP, ap AP.
quest (quest VFA NP VP) --> vfa VFA, np NP, vp VP.
quest (quest NPWH Q)    --> npwh NPWH, np (np gap) -o quest Q.
quest (quest APW  Q)    --> apwh APW,  ap (ap gap) -o quest Q.
quest (quest PPW  Q)    --> ppwh PPW,  pp (pp gap) -o quest Q.
quest (quest NPWH VP)   --> npwh NPWH, vp VP.

npwh (npwh NWH) --> nwh NWH.
npwh (npwh which OptAP N) --> (which::nil), optap OptAP, n N.
apwh (apwh AWH) --> awh AWH.
ppwh (ppwh PWH) --> pwh PWH.

sbar (sbar that S) --> (that::nil), sent S.
qbar (qbar NPWH VP) --> npwh NPWH, vp VP.
qbar (qbar NPWH S)  --> npwh NPWH, np (np gap) -o sent S.

np (np PNposs) --> pnposs PNposs.
np (np Det Nposs OptPP OptRel) --> det Det, nposs Nposs, optpp OptPP, optrel OptRel.
pnposs (pnposs PN) --> pn PN.
pnposs (pnposs PN s Nposs) --> pn PN, (s::nil), nposs Nposs.
nposs (nposs OptAP N) --> optap OptAP, n N.
nposs (nposs OptAP N s Nposs) --> optap OptAP, n N, (s::nil), nposs Nposs.

vp (vp DV NP PP)   --> dv DV,   np NP, pp PP.
vp (vp TV NP)      --> tv TV,   np NP.
vp (vp IV OptPP)   --> iv IV,   optpp OptPP.
vp (vp Stv Sbar)   --> stv Stv, sbar Sbar.
vp (vp TV NP Sbar) --> tv TV,   np NP, sbar Sbar.
vp (vp Qv Qbar)    --> qv Qv,   qbar Qbar.
vp (vp Vfa VP)     --> vfa Vfa, vp VP.
vp (vp Vfa AP)     --> vfa Vfa, ap AP.

optpp (optpp epsilon) --> nil.
optpp (optpp PP)      --> pp PP.
pp (pp P NP) --> p P, np NP.

optap (optap epsilon) --> nil.
optap (optap AP)      --> ap AP.
ap (ap A) --> a A.

optrel (optrel epsilon) --> nil.
optrel (optrel Rel)     --> rel Rel.
rel (rel that VP)   --> (that::nil), {vp VP}.
rel (rel who VP)    --> (who::nil),  {vp VP}.
rel (rel that S)    --> (that::nil), {np (np gap) -o sent S}.
rel (rel whom S)    --> (whom::nil), {np (np gap) -o sent S}.
rel (rel P whom S)  --> p P, (whom::nil), {pp (pp gap) -o sent S}.
rel (rel P which S) --> p P, (which::nil), {pp (pp gap) -o sent S}.
```

Figure 6.2: A Large Lolli Definite Clause Grammar.

```
?- big_grammar --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)))) .
```

Figure 6.3: A Sample Interaction With The Large Lolli Definite Clause Grammar.

Which book [did you wonder who [john told *gap* that [you wrote *gap*]]]?

a version of the large grammar modified to mark gap introduction/discharge site pairs with unique names produces the following two parse trees:

```
quest (npwh which (optap epsilon) (n book) Gap_5492)
      (quest (vfa did)
             (np (pnposs (pn you)))
             (vp (qv wonder)
                 (qbar (npwh (nwh who) Gap_30922)
                       (sent (np (pnposs (pn john)))
                          (vp (tv told) (np Gap_30922)
                              (sbar that (sent (np (pnposs (pn you)))
                                               (vp (tv wrote) (np Gap_5492))
      )))))))

quest (npwh which (optap epsilon) (n book) Gap_5492)
      (quest (vfa did)
             (np (pnposs (pn you)))
             (vp (qv wonder)
                 (qbar (npwh (nwh who) Gap_30922)
                       (sent (np (pnposs (pn john)))
                             (vp (tv told) (np Gap_5492)
                                 (sbar that (sent (np (pnposs (pn you)))
                                               (vp (tv wrote) (np Gap_30922))
      )))))))).
```

In the first parse tree, which is correct, the introduction/discharge pairs are nested. In the second pair they overlap, which is not allowed in English. Crossing dependencies are, however, allowed in a limited form in some languages. Until now there have been few formalisms that could admit such constructions. Nevertheless, it is necessary to reject this ambiguity in English.

The problem is caused by the fact that $\mathcal{L}$ is a commutative logic. Thus, once the gap rules are loaded into the linear context they can be rearranged in any order. This is the same problem that occurred in the multiset rewriting example in Chapter 5. To solve this problem at the logical level will presumably require examining non-commutative variants of this logic. In the meantime, a simple extra-logical solution is available which relies on the ordering of the Lolli search procedure: modify the gap introductions to be of the form:

```
rel --> (npgap --> []) -o s.
```

We then add explicit gap termination rules of the form:

```
np -->  once npgap.
```

This way when a gap is used, only the most recently introduced gap of the appropriate type can be used. The parse cannot re-succeed using an earlier gap. It is interesting to note that this reintroduces gap termination rules, which are used in GPSG but were unnecessary in this setting.

6.5.7 The Question of Complexity

When faced with a new grammar formalism, one of the first questions one wants answered is what the computational complexity of the accepted languages is. This is important both because it offers a lower bound on the efficiency of parsers that can be built, and because it provides an immediate feel for how likely an explanation of natural language the system might be. Interest is currently particularly focused on systems that are slightly more than context free, such as Steedman's Combinatory Categorial Grammars [Steedman, 1998] and Joshi's Tree Adjoining Grammars [Joshi, 1986; Joshi, 1983].

It is a legitimate question to ask then, what is the computational complexity of the formalism underlying these parsers. At present we are prepared to provide only a partial answer. It is well known that fragments of propositional linear logic are far more complex than equivalent fragments of intuitionistic logic. Lincoln, et al. have shown that full intuitionistic linear logic is undecidable [Lincoln *et al.*, 1990]. More recently, Kanovich has claimed that if G is a formula built out of a single propositional letter, '$\multimap$', and '&', and Γ and Δ are multisets of such formulas, then the system in which sequents are of the form:

$$!\Gamma, \Delta \longrightarrow G$$

is also undecidable [Kanovich, 1993]. This is significant, since it is clearly a sublanguage of $\mathcal{L}$. This is not necessarily bad news, though. While propositional $\mathcal{L}$ is apparently undecidable, the grammars we have constructed do not begin to utilize the full formula language. In particular, our grammar rules have had only a single level of nested implications. It seems likely that if we impose this as a formal restriction we may obtain a significantly less expressive language.