such that $\delta(S_1, Dummy) = S_2$ holds. [27] By modifying the LR machine to use this unbounded lookahead mechanism, three to six-fold improvements in PS recovery times were observed.

- *Subcategorization.*

  Dummy non-terminals and extra pre-conditions also allow phrase structure construction to be 'driven' by selectional properties of heads. As discussed before, in general, lexical heads select semantic roles such as patient, agent and proposition. Normally, these roles may be syntactically realized as NPs or CPs. Hence, the above schema may be expanded into 'ground' rules by consulting the lexical entries for each head category. For example, consider the following (incomplete) list of production rules that expand $\overline{\mathrm{V}}$:

$\overline{\mathrm{V}} \rightarrow \mathrm{V}$	e.g. *John* [$_{\bar{V}}$ *slept*]
$\overline{\mathrm{V}} \rightarrow \mathrm{V}$, NP	e.g. *John* [$_{\bar{V}}$ *saw* [$_{NP}$ *Mary*]]
$\overline{\mathrm{V}} \rightarrow \mathrm{V}$, CP	e.g. *John* [$_{\bar{V}}$ *believes* [$_{CP}$ *Mary left*]]
$\overline{\mathrm{V}} \rightarrow \mathrm{V}$, NP, CP	e.g. *John* [$_{\bar{V}}$ *asked* [$_{NP}$ *Mary*] [$_{CP}$ *to leave*]]

  Of course, it would be grossly inefficient to allow the LR machine to operate directly on these rules without subcategorization restrictions. For example, consider the sentence *John slept.* Assuming the LR table to contain the list of ground rules shown above, once *slept* has been shifted, the machine would go through each possibility, inventing empty constituents, e.g. [$_{NP}$-*e*], if necessary in an attempt to complete each RHS. Roughly speaking, by immediately consulting the lexical features of the verb as soon as it has been shifted (onto the structure stack), we can eliminate the unnecessary search to complete the last three productions. For instance, we might impose conditions of the following form:

  $\overline{\mathrm{V}} \rightarrow$ V, `takesNoObjects(`tos`(SS))`
  $\overline{\mathrm{V}} \rightarrow$ V, `takesNPObject(`tos`(SS))`, NP
  $\overline{\mathrm{V}} \rightarrow$ V, `takesCPObject(`tos`(SS))`, CP
  $\overline{\mathrm{V}} \rightarrow$ V, `takesNPCPObjects(`tos`(SS))`, NP, CP

[27] This is a standard trick used in compiler construction. Note that the dummy non-terminal must be made invisible to the reduce clause for the CP production. Normally, the reduce action for a production with RHS of length three would involve popping the top three elements of both the control and structure stacks. The elements popped off the control stack are discarded and those popped off the structure stack will become immediate constituents of the new phrase. However, the dummy production builds no structure. Hence, we must modify the default behaviour of the reduce action to pop only the top two elements off the structure stack. (Note however, we must still pop three elements off the control stack.) The need to accommodate such dummy non-terminals is one reason for separating the control and structure stacks so that they need not operate in lockstep.

> Here, we run into the same problem with semantic actions as in the unbounded lookahead case. The solution will also be the same: just invent new dummy non-terminals and empty productions containing just the semantic action for each case.

In the actual implementation, the separation between such productions is taken one step further. In the current system, the parser computes equivalence classes for each set of conflicts. Each class represents the dummy productions that can be (ambiguously) satisfied by some lexical element. For example, currently the verb *buy* is analyzed as selecting a 'theme', i.e. something to be bought, and (optionally) a 'location', i.e. the place of purchase, as complements. Hence, the two productions `vNP -> v(V) if takesNPObject(V)` and `vNPNP -> v(V) if takesNPNPObjects(V)` will overlap, and thus belong to the same class. However, if there is no verb in the lexicon that can either take no complements and a single NP as a complement (or none and a double NP), then `vZero -> v(V) if takesNoObject(V)` will not belong in the same class as the previous two dummy productions. Hence, if the current verb in the input can be reduced using the production for `vZero`, there will be no need to test if it takes one or two NPs as objects. The current system exploits this wherever possible to effectively reduce the branching factor in states with multiple reduce-reduce conflicts of this nature. (The effect on the conflict distribution of action table entries will be analyzed in section 4.6.)

Note that the introduction of such dummy non-terminals may cause only a linear increase in the number of productions of the expanded grammar. Hence, such modifications should not unduly impact grammar size. More importantly, the modifications that we have discussed should be transparent to the grammar writer. In the current implementation, the grammar writer merely specifies phrase structure at the abstract level of the rule schema; the details of the underlying machinery such as the LR stacks and action clauses being invisible (and inaccessible). The generation of the LR table and various efficiency modifications that we have described are all handled automatically when the rule schema is compiled. In the following section, we will briefly describe this compilation process.

4.5.3 Compilation

In order to maintain transparency, it is necessary to automate the process of translating the rule schemas that encode phrase structure representations down to the level of the target LR machine. This is carried out in two stages as shown in figure 4.9.

In the first stage, schema rules that contain variables ranging over $\overline{\mathrm{X}}$ categories will be expanded, or 'grounded', by instantiating over each category. Furthermore, the variety of complement structure for each lexical head category will be determined by consulting the lexicon. The expanded rules for $\overline{\mathrm{V}}$ shown in the previous

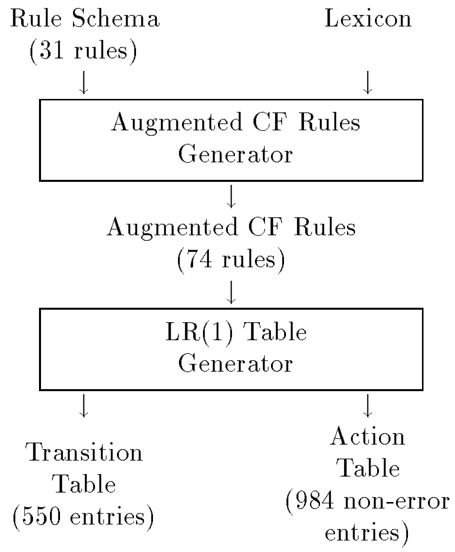


Figure 4.9 The two stage LR-table generator

section are an example of this process. The various optimizations that have been described are also carried out at this stage. The resulting output will therefore consist of ground $\overline{\mathrm{X}}$ rules augmented with extra conditions. This intermediate representation is then fed to the 'back-end' stage. In the current implementation, this stage contains a fairly standard procedure that constructs the PROLOG clauses that encode LR(1) tables. (For details of the method, see Algorithm 6.2 in Aho & Ullman [7].) This modular two stage approach facilitates the substitution of a different back-end, say, a more sophisticated LR(1) table generator, e.g. Pager [42], that generates a more compact machine; or, indeed, any of the other CFG parsing techniques mentioned earlier.

To provide a rough measure of the machine size for the phrase structure grammar of S-structure, the augmented CFG consists of about 74 productions derived from a schema of 31 rules. The resulting characteristic finite state automaton (CFSM) consists of 123 states with 550 transitions between the various states. The action table consists of a total of 984 individual (non-error) entries, counting conflicting actions as separate entries. This compares very favourably with typical (artificial) programming languages. For example, Aho & Ullman state that a typical programming language has about 100 productions resulting in a LR(1) CFSM of several thousand states. Even with a less powerful shift-reduce method such as the LALR(1) algorithm, the CFSM may still contain several hundred states. Also, the corresponding action table may easily have 20000 entries. Direct comparisons with natural language grammars are obviously difficult for reasons of coverage. However, Tomita [56] mentions two natural language CFGs that contain about 220 and 400 productions. He reports that these grammars have LR(0) CFSMs of the order of 400 and 600 states, respectively. Although, the corresponding numbers for a canonical LR(1) CFSM are not available, it is likely that such machines would contain many thousands of states. Bearing in mind the caveat just mentioned, these numbers suggests that canonical LR(1) parsing is not an available option for systems that encode most of the grammatical knowledge using CF rules only. The grammars cited by Tomita would seem to fall in this category. LR(1) parsing is still feasible for theories in the principles-and-parameters framework such as the one described in this thesis because the search for explanatory adequacy has led to the elimination of much of the work traditionally borne by the phrase structure component. (See section 3.3.2. in Chomsky [12] for discussion on this point.) Outside the principles-and-parameters framework, Graham *et al.* [27] discusses the use of a CF grammar for English of about 160 productions that was supplied by Pratt who conducted an experiment to "see how simple the grammar could be made at the expense of complicating other portions of his system."

Although canonical LR(1) parsing is certainly feasible, we have not yet addressed whether its major feature, the (full) one-symbol lookahead mechanism, actually contributes to making the parser more deterministic or not. In the next section, we will address the suitability of the LR(1)-based approach for the S-structure grammar.

4.6 Results and Conclusions

To summarize, we have described a multi-stage approach to recovering phrase structure. This process is driven off the first stage; that of recovering structure at the level of S-structure. By using a method based on the LR(1) algorithm, we have described how such structures may be efficiently recovered. From the level of S-structure, both LFs, the designated level of output of the system, and D-structure may be recovered. Constituents of the latter level, being a theory-internal level of phrase structure, are only computed on a demand basis. In this section, we will evaluate how this approach measures up to the design goals outlined in chapter 1.

- *Abstraction and transparency.*

  Although the current implementation makes considerable use of compilation techniques to automate the process towards producing an efficient phrase structure recovery procedure; still, a significant portion of the gap between the definitions found in linguistic theory and a working parser has to be bridged by the grammar writer. For example, the input to the compilation process is a phrase structure grammar for S-structure. This grammar is derived manually, as opposed to being automatically generated, from a definition of $\overline{\mathrm{X}}$-theory (which applies at D-structure) plus a specification of how much of Move-α to use. Similarly, the procedure that recovers D-structure constituents from S-structure is not automatically generated. Much more work must be done, in particular, involving the addition of automated theorem proving techniques to deduce the effects of Move-α on phrase structure at D-structure, in order to close the abstraction gap.[28]

The various extensions to the basic LR(1) model such as multiple actions, the addition of the environment stack, and arbitrary conditions on LR actions drastically increase the power of the resulting machine. In fact, the last extension allows the modified machine to have access to general computing power. However, note that each extension has been motivated either by the need to handle problematic features of the grammar, e.g. traces and empty categories in general, or by the goal of computational efficiency. Note also that the modifications for purely efficiency reasons, such as the unbounded lookahead mechanism or equivalence class computation, do not affect the language generated by the grammar, and, hence, are transparent to the grammar writer. Moreover, the additional capabilities over the CF model are essentially hidden; that is, there is no direct access to the features of the target machine. The compilation process here serves the dual purpose of insulating the grammar writer from the power of the underlying implementation. Of course, the insulation will not be complete unless the grammar represented by the rule schema can be debugged at a higher level of abstraction than LR actions. Currently, such 'source-level' debugging facilities remain to be added to the system.[29]

A potentially more serious problem is that correctness could be compromised by the limitation placed on recursion by the trace licensing mechanism. For example, as discussed earlier, repeated adjunction of adverbs to a VP is disallowed to ensure termination. Hence, if there are well-formed structures that involve cases of repeated adjunction of adverbs, then the current system will fail to recover them. Although the bound on the number of such adjunctions can be adjusted by the grammar writer, and that no such structures have been found to be necessary for the sentences that the parser has been tested on in

[28] Recently, there has been some work in this direction (Stabler [52]).

[29] Note that the process of generating an LR table can also provide useful tools for debugging phrase structure rules. For example, one by-product of computing lookahead symbols is the set of all non-terminals that derive the empty string. This information can be provided to the grammar writer as a tool for detecting parsing termination problems.

Lasnik & Uriagereka, a more sophisticated control mechanism to detect cases of vacuous adjunction would be a more satisfactory solution. That is, ideally the system should be able to detect and suppress all cases where further adjunction would not affect well-formedness (or ill-formedness) of the structure being built.[30]

- *Computational Efficiency.*

  As mentioned at the beginning of the chapter, one of the motivating factors for optimizing phrase structure recovery was that, in general, there will be many possible structures, within the space of possibilities specified by a rule schema, for a given sentence. By simply taking advantage of one of the extant efficient algorithms for parsing with CF rules, the ratio of the amount of work spent on phrase structure recovery as opposed to application of the principles described in chapter 3 changed drastically. (See note 1 for specific details.) Previously, grammar rules had simply been interpreted in an straightforward, on-line fashion. Although the original LR(1) model has been 'tweaked' considerably, there are still areas that could be improved. Perhaps the most significant among these is that there is no special mechanism for structure sharing between multiple parses — cf. Tomita's packed forest representation. For example, the VP built for [*CP* [*NP John*] [*IP* [*NP-e*] [*VP saw Mary*]]] will be discarded when the parser backtracks past the VP (in order to re-position *John*) to build the second parse [*CP* [*IP* [*NP John*] [*VP saw Mary*]]]. Preserving shared constituents in the backtracking framework would require the implementation of some kind of memoization scheme.[31]

We now conclude the chapter by considering how well the LR(1)-based algorithm 'matches' the phrase structure grammar for S-structure.

[30] See Stabler [52] (section 13.6) for discussion on this point.

[31] In the backtracking framework, the amount of structure sharing between parses depends on how far the parser has to backtrack to find the next valid choice point. When the parser backtracks to an earlier choice point, it also (automatically) restores the state of all the stacks, effectively discarding any structure built on the structure stack since that choice point. If some common structure had been built before the choice point, then it is still shared; otherwise, it must be rebuilt from scratch. To escape this restriction, the lifetimes of structures to be saved must be extended by storing those structures in an external database. Depending on the PROLOG implementation, such a memoization technique can be very expensive. In informal experiments, memoization turned out not to be a cost-effective optimization. The problem was that, for performance reasons, structures are usually allocated room on a special stack. Such structures must be copied into a less transient area of memory if they are to be preserved under backtracking. Unfortunately, this copying process turned out to be expensive enough to more than compensate for the shortcut of not having to re-build any shared structure. (Note that the costs involved may very well be different given some other implementation of PROLOG.)

Of course, the obvious alternative is to abandon the backtracking approach and construct all possible parses in parallel. This is essentially what happens in Tomita's extended LR parser.

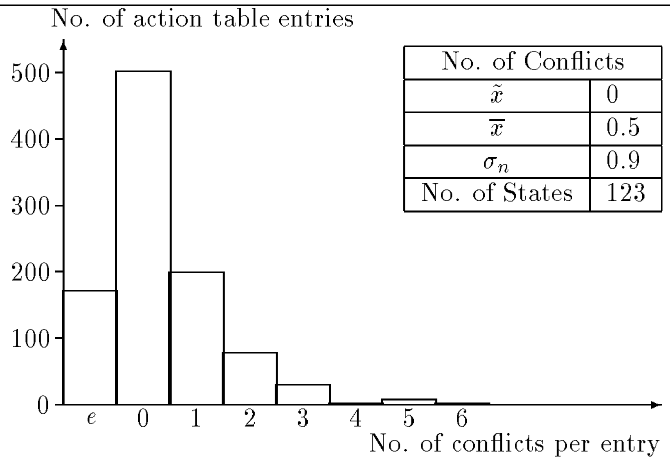
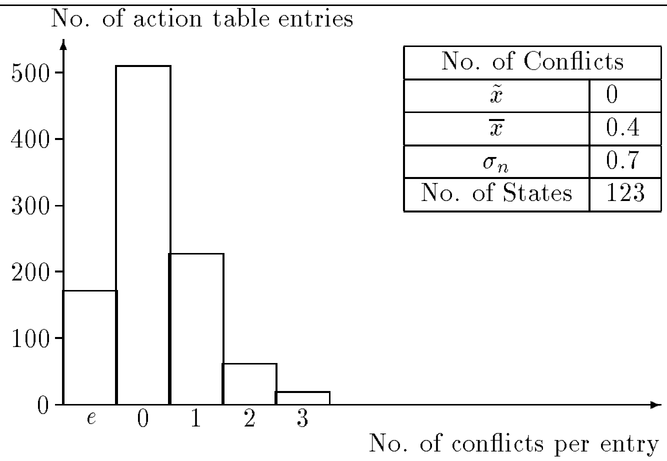


Figure 4.10 English S-structure grammar conflict statistics.

(Here, 'e' represents an entry with no possible actions, i.e. the error entry.)

Tomita has suggested that LR-based algorithms can be much faster than Earley's algorithm in practice because 'natural language grammars' are considerably closer to LR than other general context-free grammars. The essential claim is that the extended-LR parser is able to handle non-LR grammars with little loss of LR efficiency, provided that the grammar is 'close' to LR. In the following discussion, we address the question: How close to LR(1) is the phrase structure grammar for S-structure? We also consider a related question: Does (one symbol) lookahead help? That is, does lookahead reduce the degree of non-determinism represented by multiple actions? What about the effect of the equivalence class shortcut procedure?

We can get an idea of the 'LR(1)-ness' of a grammar by considering the distribution of the number of conflicts per entry in the action table. The basic intuition is this: if a grammar is 'close' to LR(1), then its LR(1) action table should have relatively few entries with one or more conflicts. On the other hand, if a grammar is relatively non-LR(1)-like, then a large proportion of its table entries should have multiple conflicts. Consider the conflict set distribution for the phrase structure grammar of S-structure for English shown in figure 4.10. Note that the number of conflicts shown does not take into account the reduction in non-determinism due to the unbounded lookahead mechanism. With a more sophisticated LR(1) table generator such as Pager's (as mentioned earlier), certain states may be merged, thereby reducing the frequency of some of the conflict entries. With these caveats in mind, the median and average number of conflicts per state is zero and 0.5, respectively.



No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.4
σ_n	0.7
No. of States	123

Figure 4.11 Revised English S-structure grammar conflict statistics.

For interpretation purposes, an average of one would roughly correspond to binary branching in the computation search space every time the machine was forced to pick an action.[32]

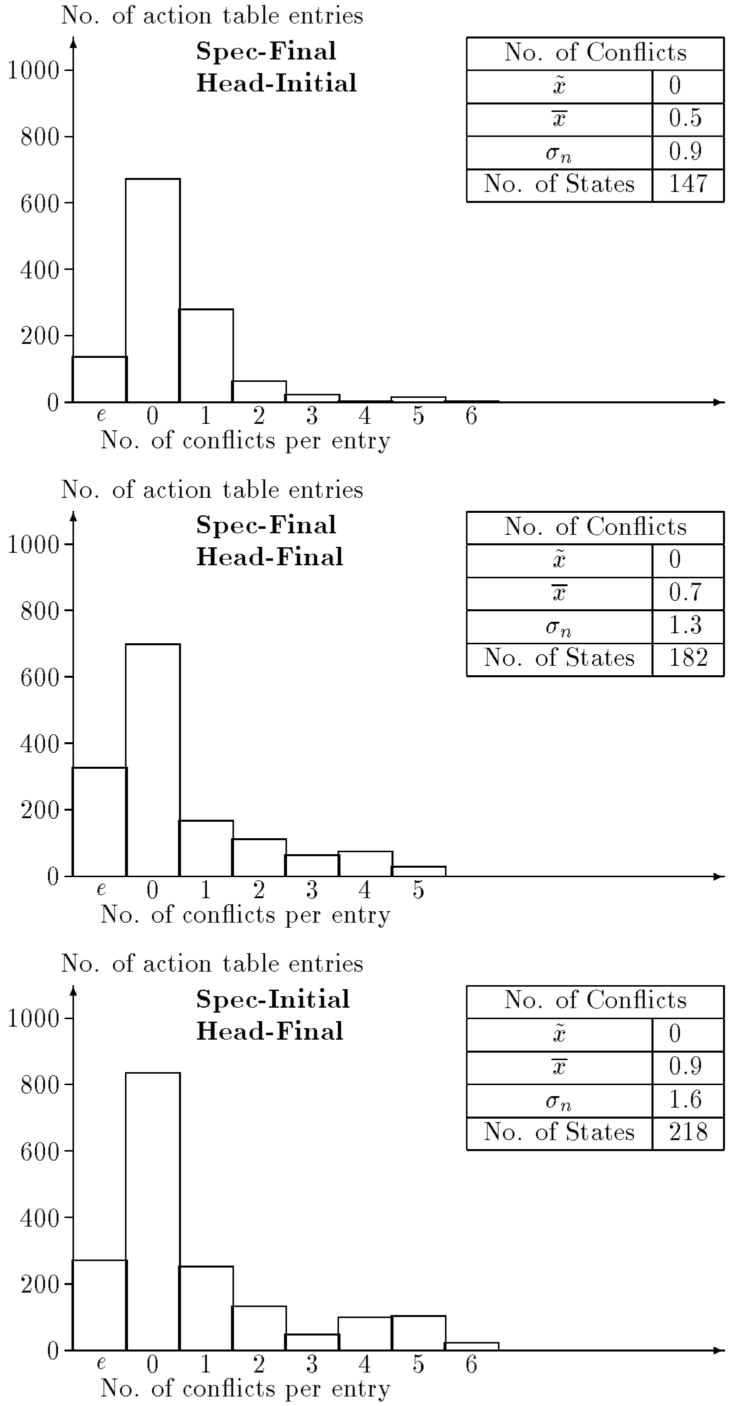
If the equivalence class shortcut described in the previous section is taken into account, there is a significant shift in the overall conflict distribution as shown in figure 4.11.[33] However, the conflict distribution essentially retains the same profile.

As for the question whether the one symbol lookahead mechanism is profitably utilized in the grammar of S-structure: out of the 123 states generated, 83 (or approximately 67%) benefitted in some degree; that is, there exists at least one table entry that had strictly fewer conflicts when lookahead is added. There were 252 entries (out of a total of 984) that had the number of conflicts reduced by (exactly) one. The degree of conflict was reduced by two in 26 other entries. Hence, we can conclude that lookahead is certainly a useful feature.

The rule schema used to encode the S-structure grammar also allows different values of the specifier-head and head-complement parameters to be specified. The conflict statistics shown so far are only valid when the specifier precedes its head, i.e. specifier-initial, and the head precedes its complements, i.e. head-initial, as in English. The question is: Will the distribution of conflicts change significantly for different values of the $\overline{\mathrm{X}}$-parameters? As figure 4.12 shows, there seems to be rela-

[32] Of course, strictly speaking, it is a vast oversimplification to correlate the average number of conflicts (which is only a *static* measure) with the branching factor at parse time. For example, not all entries are equally likely to be visited at parse time — which, presumably, should be a function of sentence type.

[33] In this case, we have taken the number of conflicts to be the maximum number of different actions that can be taken before shortcutting occurs.



Spec-Final Head-Initial

No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.5
σ_n	0.9
No. of States	147

Spec-Final Head-Final

No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.7
σ_n	1.3
No. of States	182

Spec-Initial Head-Final

No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.9
σ_n	1.6
No. of States	218

Figure 4.12 Conflicts statistics for various parameter settings.

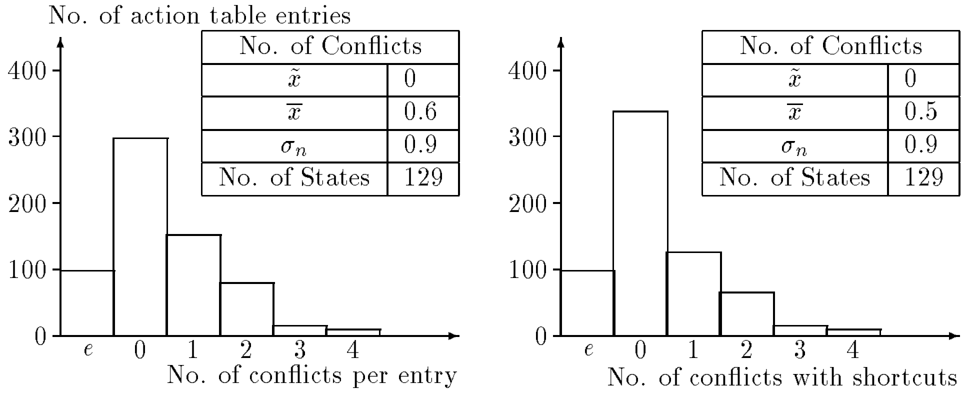


Figure 4.13 Japanese S-structure grammar conflict statistics.

tively little difference in the distribution of conflicts for all four parameter settings. Hence we can conclude that the results obtained for the S-structure grammar seems to be robust, at least with respect to parameter setting. However, informally speaking, there seems to be some correlation between the average number of conflicts per table entry and the relative position of the head with respect to its specifier and complements. The lowest and highest averages were obtained when the head preceded and followed both its specifier and complements, respectively. In the other two cases, i.e. when the head separates the specifier from the complements, the averages fell inbetween the two extremes. This result is perhaps not surprising for a left-to-right parsing algorithm since, typically, a head will carry lexical information that helps to prune the space of possible complements and specifiers.

The grammar for Japanese differs in coverage with respect to scrambling and syntactic *wh*-movement from that of English, and therefore, strictly speaking, should not be directly compared. However, for completeness, the corresponding conflict distributions (both with and without the conflict shortcut procedure) are shown in figure 4.13. We will just comment that the reduction in the number of conflicts due to the equivalence class shortcut is less marked than for the English grammar. This can be traced to the fact that heads precedes their complements in English, and hence subcategorization information of individual heads, e.g. the θ-grid, can be profitably used to prune the search space for parsing following complements in a left-to-right parser. (By contrast, in Japanese, heads will follow their complements, and hence, complements will generally be parsed before heads.) However, despite these differences, note that the basic distribution of the conflicts is similar to that for English.

Finally, let us compare the canonical LR(1)-based machines built for English with the corresponding LALR(1)-based machines that, as mentioned earlier, trade early error detection for reduced table size. It is a simple matter to derive the corresponding LALR(1) CFSM given a canonical LR(1) CFSM. (See Algorithm 6.4 in Aho & Ullman [7].) LALR(1) CFSMs were built for each of the four parameter settings. The pertinent results are summarized in the following table:

	Spec-Initial Head-Initial	*Spec-Final Head-Initial*	*Spec-Final Head-Final*	*Spec-Initial Head-Final*
No. of states LR(1)/LALR(1):	123/79	147/76	182/86	218/92
Percentage of error entries LR(1)/LALR(1):	17%/6%	11%/6%	22%/15%	16%/5%

In each case, the LALR(1)-based machine contains approximately half the number of states of the corresponding canonical LR(1)-based machine. This difference is not significant since, in both cases, the CFSM can be considered to be relatively small. Also, the distribution of conflicts were found to be very similar (modulo the general reduction in the size of the LALR(1)-based tables), except in the ratio of error entries as a percentage of the total number of table entries. As the table shows, in all cases, a larger proportion of error entries were consistently found in the LR(1)-based machines. This difference is due to the fact that for some LR(1) error entries, the corresponding entry in the LALR(1) table will contain an (erroneous) reduce action that will be detected at a later state. In general, this result is expected because an LALR(1) machine may perform more reduce actions (but not more shifts) than its LR(1) counterpart.

Chapter 5

Principle Ordering

In this chapter we will examine if, and how, a parser can exploit the freedom to re-order principles to avoid doing unnecessary work. In particular, we will investigate the following questions:

1. What effect, if any, does principle ordering have on the amount of work needed to parse a given sentence?
2. If the effect of principle ordering is significant, then are some orderings much better than others?
3. If so, is it possible to predict (and explain) which ones these are?

By characterizing principles in terms of the purely computational notions of *filters* and *generators*, we will show how principle ordering can be exploited to minimize the amount of work performed in the course of parsing. Basically, some principles such as Move-α (a principle relating 'gaps' and 'fillers') and free indexation (a principle relating referential items) are "generators" in the sense that they may provide more structures as output than they receive as inputs. For example, in the case of Move-α, there are likely to be many possible assignments of movement chains to a given structure. Other principles, like the θ-criterion which places restrictions on the assignment of thematic relations, or the Case Filter which requires certain noun phrases to be marked with abstract Case, act as filters and weed-out ill-formed structures.

The effects of principle ordering can be investigated in the current system by specifying (separately from principle definitions of the kind discussed in the previous chapter) the order in which principles are to be processed. Note that no matter which ordering is chosen, the system makes use of the same set of principles, and thus will make the same grammaticality judgements and return the same LF structures as output. The only visible difference will be that different orderings will result in different parsing times.

Roadmap

We will begin by reviewing the underlying processing model that will be used for principle ordering. Then, we will investigate how significant a factor re-ordering principles can be for parsing in that model. We will also explain under what conditions certain orderings are preferable to others. Using the observation that the efficiency of a particular order depends on local conditions, i.e. at the level of individual structures, we will go on to describe a dynamic ordering mechanism that attempts to re-schedule the order of processing based on structural cues in conjunction with simple heuristics using the filters-and-generators model. Finally, we will describe a small revision to the computational notion of a filter that will allow the use of partial dependency-directed control mechanism.

5.1 The Model of Processing

We will adopt a very simple model of processing for exploring the effects of principle ordering. In this model, parser operations that correspond (roughly speaking) to linguistic principles will be the atomic elements that are subject to re-ordering. For example, consider the following:

$$\mathrm{SF} \rightarrow \boxed{PS} \rightarrow \underbrace{\boxed{P_2} \rightarrow \boxed{P_1} \cdots \rightarrow \boxed{P_{n-1}} \rightarrow \boxed{P_n}}_{\pi(P_1, P_2, \ldots, P_n)} \rightarrow \mathrm{LF}$$

Here, each box represents a single principle that is connected to both the previous and next principle to be processed in a "production line" sequence. Given an input Surface Form (SF), the initial PS parser operation will produce the corresponding phrase structures that will be passed from one parser operation to the next in a "stream-like" fashion. Because the current system employs a backtracking depth-first search strategy, the set of permissible structures at each point will be produced one at a time. Note that the structures passed between each operation will be phrase structures for the *complete* input. (The case where individual operations will act on partial structures that represent portions of the input SF will be dealt with in the next chapter.) The principle ordering problem thus is to find the permutation

$\pi(P_1, P_2, \ldots, P_n)$ that minimizes the cost of producing all possible output Logical Forms (LFs) corresponding to a given input SF. In this and the following chapter, we will adopt the number of parser operations performed during parsing as a simple, machine-independent measure of the cost of parsing.[1]

5.2 The Principle Ordering Problem

Let us begin by examining the question of whether there exists a significant performance difference between various orderings? Equivalently, is the "principle ordering problem" really a problem at all? An informal experiment was conducted using the system to provide some indication of the magnitude of the problem. As mentioned earlier, the current system consists of about two dozen principles. Clearly, it would be impractical to examine all the possible permutations.[2] However, it turns out that order-of-magnitude variations in parsing times can be achieved merely by permuting the order of a few principles. For instance, consider the graph shown in figure 5.1.

All of the parser configurations shown here have been derived from a single "default" ordering (`Default`). That is, the order of principles will be the same except that a few selected principles will have been fronted as far forward as possible. For example, `Case` represents a parser with the same ordering as `Default` except that the operations of Case theory have been 'favoured' over all other modules. In the current system, this means that the operations *inherent Case assignment*, *structural Case assignment*, *Case filter* and *Case condition on traces* will be processed as early as possible.

Each configuration was tested on a small set of fifteen examples (including both well-formed and ill-formed sentences) from Lasnik & Uriagereka. This selection represents a cross-section of different linguistic phenomena including parasitic gaps, *that*-trace (ECP) effects, strong crossover, superiority, subjacency, exceptional Case marking (ECM), passivization, and Binding effects. These examples are shown in figure 5.2.

---

[1]For simplicity, we assume that all parser operations will have comparable cost. The actual ratios will, of course, depend on the precise definition of each principle and the efficiency of the compilation process. However, this measure correlates quite well with actual parsing times. One reason why the approximation seems to hold is that one of the major tasks involved in applying a principle is the work required in simply "walking" through each sub-phrase for a given structure.

[2]Actually, only a fraction of the possible $n!$ permutations are *valid* orderings due to logical dependencies between various parser operations — see the dependency graph (figure 1.6) and accompanying discussion in section 1.5.2. The relevant notion of validity here is that correctness with respect to grammaticality judgements should be preserved. Estimates of the number of valid permutations have been made for an earlier version of the system which contained one dozen principles. Out of the almost 500 million possible permutations, only about half a million are actually valid. Even so, this is far too many to test exhaustively.

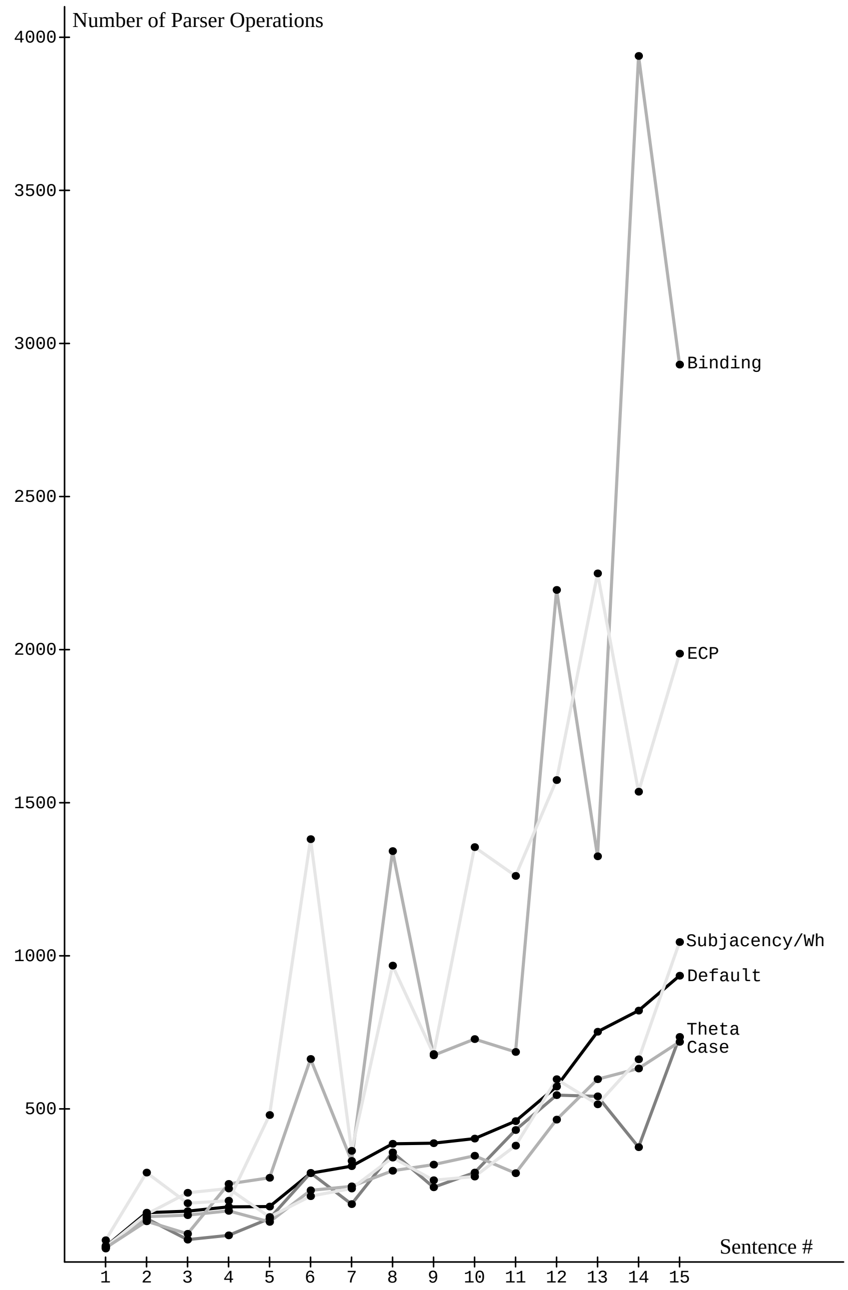


Figure 5.1 The effect of principle ordering. [3]

(1)	*What will who read (4:21c)	
(2)	Who will read what (4:21b)	*Superiority*
(3)	*John's mother likes himself (2:8)	*Binding condition A violation*
(4)	John's mother likes him (2:16b)	*Pronoun Binding ambiguity*
(5)	*Which book did you file the report without reading (3:18)	*cf. (11)*
(6)	*Who does Mary wonder why John hit (4:57)	*Subjacency*
(7)	*Who do you think that saw Bill (4:8a)	that-*trace effect*
(8)	John was believed to be intelligent (1:49c)	*Exceptional passivization*
(9)	Who do you think saw Bill (4:8b)	*cf. (7)*
(10)	Who that John knows does he like (1:15a)	Wh-*movement/pronoun ambiguity*
(11)	Which report did you file without reading (3:17)	*Parasitic gap*
(12)	*It was believed John to be intelligent (1:49b)	*cf. (8)*
(13)	Who does he think Mary likes (2:45a)	*Strong crossover*
(14)	I want to visit you (2:66a)	*Control of PRO*
(15)	*I am eager John to be here (1:22)	*Case violation*

Figure 5.2 Test examples from Lasnik & Uriagereka.

Each point on the graph represents the total number of parser operations invoked for a given sentence. (The examples are not given in any particular order.) As the graph clearly illustrates, the order in which principles are processed can make a large difference in the amount of work required to obtain the same grammatical judgements. Other experiments on the system have shown that this order of magnitude in variation is possible even for four word sentences such as *I want to win*.

5.2.1 Explaining the Variation in Principle Ordering

The variation for various principle orderings that we observed in the previous section can be explained by assuming that overgeneration is the main problem, or bottleneck, for systems such as the current one. That is, in the course of parsing a single sentence, the parser will hypothesize many different structures. These structures are produced by operations that function as generators in the sense discussed earlier. Most of these structures, the ill-formed ones in particular, will be accounted for by one or more linguistic filters. Computationally speaking, filters are operations that produce N or fewer structures given N structures as input.[4] A sentence

[3]Note that the discrete data points for each parser configuration have been connected for presentation purposes only.

[4]However, one complicating factor is that some parser operations may behave either as filters or generators depending on the degree of instantiation of features in the phrase structures that are presented as input to such operations. For instance, the *theta criterion* operation can function in either mode depending on processing order. For example, (following Chomsky [9]) the last element

will be deemed acceptable if there exists one or more structures that satisfy every applicable filter. Note that even when parsing grammatical sentences, overgeneration will produce ill-formed structures that need to be ruled out. Given that our goal is to minimize the amount of work performed during the parsing process, we would expect a parse using an ordering that requires the parser to perform extra work compared with another ordering to be slower.

Overgeneration implies that we should order the linguistic filters to eliminate ill-formed structures as quickly as possible. For these structures, applying any parser operation other than one that rules it out may be considered as doing extra, or unnecessary, work (modulo any logical dependencies between principles).[5] However, in the case of a single well-formed structure, principle ordering cannot improve parser performance. By definition, a well-formed structure is one that passes all relevant parser operations. Unlike the case of an ill-formed structure, applying one operation cannot possibly preclude having to apply another.

5.2.2 Optimal Orderings

Since some orderings perform better than others, a natural question to ask is: Does there exist a "globally" optimal ordering? The existence of such an ordering would have important implications for the design of the control structure of any principle-based parser. For example, the current system also implements a novel "dynamic" control structure in the sense that it tries to determine an ordering-efficient strategy for every structure generated. If such a globally optimal ordering could be found, then we can do away with the run-time overhead and parser machinery associated with determining individual orderings. That is, we can build an ordering-efficient parser simply by "hardwiring" the optimal ordering into its control structure. Unfortunately, no such ordering can exist.

The impossibility of the globally optimal ordering follows directly from the "eliminate unnecessary work" ethic. Computationally speaking, an optimal ordering is one that rules out ill-formed structures at the earliest possible opportunity. A *globally* optimal ordering would be one that always ruled out every possible ill-formed structure without doing any unnecessary work. Now, consider the following three

---

of a chain must be assigned a θ-role if and only if it is headed by an argument. Consider the case when the head of the chain is an empty NP without any features (all empty NPs are initially inserted with null features.) In the current implementation, a simple generate-and-test strategy has been adopted; that is, *theta criterion* will function as a generator. If the empty head has to be an argument, it will try each possibility: *PRO*, *pro* and variable (but not an anaphor). On the other hand, if the empty head has been previously determined as one of these, then it will function as a filter.

[5] In the current system for example, the Case Filter operation which requires that all overt noun phrases have abstract Case assigned, is dependent on both the inherent and structural Case assignment operations. That is, in any valid ordering the filter must be preceded by both operations.

structures (taken from Lasnik & Uriagereka):

(1) a. *John$_1$ is crucial $[_{CP}[_{IP}$ t_1 to see this]]
b. *$[_{NP}$John$_1$'s mother $][_{VP}$ likes himself$_1]$
c. *John$_1$ seems that he$_1$ likes t_1

Example (1a) violates the Empty Category Principle (ECP). Hence the optimal ordering must invoke the *ECP* operation before any other operation that it is not dependent on. On the other hand, example (1b) violates a Binding theory principle, namely Condition A. Hence, the optimal ordering must also invoke *Condition A* as early as possible. In particular, given that the two operations are (locally) independent, the optimal ordering must order *Condition A* before the *ECP* and vice-versa. Similarly, example (1c) demands that the *Case Condition on Traces* operation must precede the other two operations. To conclude, because optimality is a local property rather than a global one, there can be no single optimal ordering.

5.3 Dynamic Ordering

The principle ordering problem can be viewed as a limited instance of the well-known conjunct ordering problem (Smith & Genesereth [50]). Given a set of conjuncts, we are interested in finding all solutions that satisfy all the conjuncts simultaneously. The parsing problem is then to find well-formed structures (i.e. solutions) that satisfy all the parser operations simultaneously. Moreover, we are particularly interested in minimizing the cost of finding these structures by re-ordering the set of parser operations (i.e. conjuncts).

This section outlines the heuristics used by the parser to automatically determine the minimum cost ordering for a given structure. The parser contains a dynamic ordering mechanism that attempts to compute a separate minimum cost ordering for each phrase structure generated during the parsing process.[6] The mechanism can be subdivided into two distinct phases as shown in figure 5.3. First, we will describe how the dynamic ordering mechanism decides which principle is the most likely candidate for eliminating a given structure. Then, we will explain how it makes use of this information to re-order parser operation sequences to minimize the total work performed by the parser.

[6] In their paper, Smith & Genesereth drew a distinction between "static" and "dynamic" ordering strategies. In static strategies, the conjuncts are first ordered, and then solved in the order presented. By contrast, in dynamic strategies the chosen ordering may be revised between solving individual conjuncts. We have employed a dynamic strategy in the current implementation.

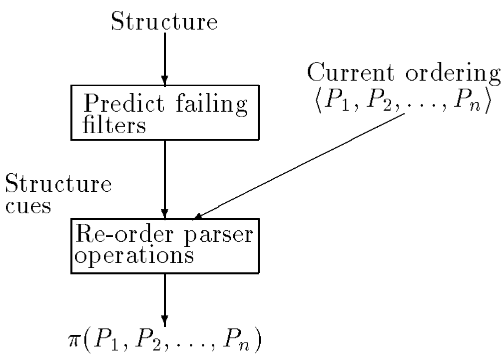


Figure 5.3 The dynamic ordering mechanism.

Predicting Failing Filters

Given any structure, the dynamic ordering mechanism attempts to satisfy the "eliminate unnececessary work" ethic by predicting a "failing" filter for that structure. More precisely, it will try to predict the principle that a given structure violates on the basis of simple structure cues. Since the ordering mechanism cannot know whether a structure is well-formed or not, it assumes *a priori* that all structures are ill-formed and attempts to predict a failing filter for every structure. In order to minimize the amount of work involved, the types of cues that the dynamic ordering mechanism can test for are deliberately limited. Only inexpensive tests such as whether a category contains certain features (e.g. [±anaphoric], [±infinitival], or whether it is a trace or a non-argument) are used. Any cues that may require significant computation, such as searching for an antecedent, are considered to be too expensive. Each structure cue is then associated with a list of possible failing filters. Some instances of this mapping is shown in figure 5.4. The system then picks one of the possible failing filters indicated by the structure cues.[7]

Note that the correspondence between each cue and the set of candidate filters may be systematically derived using simple causality from the definitions of the relevant principles. For example, Principle A of the Binding theory deals with the conditions under which antecedents for anaphoric items, such as *each other* and *himself*, must appear. Hence, Principle A can only be a candidate failing

[7] Obviously, there are many ways to implement such a selection procedure. Currently, the system uses a voting scheme based on the frequency of cues. The (unproven) underlying assumption is that the probability of a filter being a failing filter increases with the number of occurrences of its associated cues in a given structure. For example, the more traces there are in a structure, the more likely it is that one of them will violate some filter applicable to traces, such as the Empty Category Principle (ECP).

Structure Cue	*Possible Failing Filters*
`ecm(_)` *Exceptional Case Marker*	Case filter, Case condition on traces, $\overline{\mathrm{S}}$-deletion
`inf(_)` *Infinitival clause*	Case filter, Case condition on traces,
`wh(+)`	*Wh*-in-syntax
`trace`	ECP, Case condition on traces, Subjacency
`noCaseMark` (*e.g. passive or intransitive*)	Case filter, Theta criterion D-structure Theta condition
`nonarg`	Theta criterion, D-structure Theta condition
`a(+)` *anaphor*	Binding theory Principle A
`p(+)` *pronoun*	Binding theory Principle B
`op(+)` *operator*	License Operator/Variables
`adj(nonhead)` *non-head adjunction*	License adjuncts

Figure 5.4 Mapping structure cues to failing filters.

filter for structures that contain an item with the [+anaphoric] feature. Other correspondences may be somewhat less direct: for example, the Case Filter merely states that all overt noun phrase must have abstract Case. Now, the conditions under which a noun phrase may receive abstract Case are defined by two separate operations, namely, *inherent Case assignment* and *structural Case assignment*. It turns out that an instance where *structural Case assignment* will not assign Case is when passive morphology occurs with a verb that normally assigns Case. Hence, the presence of a passive verb in a given structure may cause an overt noun phrase to fail to receive Case, which in turn may cause the Case Filter to fail.[8]

[8] It is possible to automate the process of finding structure cues simply by inspecting the closure of the definitions of each filter and all dependent operations. One idea is to collect the negation of all conditions involving category features. For example, if an operation contains the condition "`\+ Item has_feature noCaseMark`", then we can take the presence of an intransitive item as a possible reason for failure of that operation. However, this approach has the potential problem

The failing filter mechanism can been seen as an approximation to the *Cheapest-First* heuristic in conjunct ordering problems. It turns out that if the cheapest conjunct at any given point will reduce the search space rather than expand it, then it can be shown that the optimal ordering must contain that conjunct at that point. Obviously, a failing filter is a "cheapest" operation in the sense that it immediately eliminates one structure from the set of possible structures under consideration.

There are two other notable features of the failure filter mechanism:

- First, the failing filter approach does not take into account the behaviour of generators. For example, the operations corresponding to $\overline{\mathrm{X}}$ phrase structure rules, Move-α, free indexation and LF movement are the main generators in the current system. (Similarly, the operations that we have previously referred to as "filters" may be characterized as parser operations that, when given N structures as input, pass N and possibly fewer than N structures.) Due to logical dependencies, it may be necessary in some situations to invoke a generator operation before a failure filter can be applied. For example, Principle A of the Binding theory is logically dependent on the generator *free indexation* to generate the possible antecedents for the anaphors in a structure.[9] Consider the possible binders for the anaphor *himself* in *John thought that Bill saw himself* as shown below:

  (2) a. *John$_i$ thought that Bill$_j$ saw himself$_i$
  b. John$_i$ thought that Bill$_j$ saw himself$_j$
  c. *John$_i$ thought that Bill$_j$ saw himself$_k$

  Only in example (2b), is the antecedent close enough to satisfy the locality restrictions imposed by Principle A. Note that Principle A had to be applied a total of three times in the above example in order to show that there is only one possible antecedent for *himself*. This situation arises because of the general tendency of generators to overgenerate. This characteristic can greatly magnify the extra work that the parser performs when the dynamic ordering mechanism picks the wrong failing filter. Consider the ill-formed structure **John$_i$ seems that he likes* t$_i$ (a violation of the principle that traces of noun

of generating too many cues. Although, it may be relatively inexpensive to test each individual cue, a large number of cues will significantly increase the overhead of the ordering mechanism. Furthermore, it turns out that not all cues are equally useful in predicting failure filters. One solution may be to use "weights" to rank the predictive utility of each cue with respect to each filter. Then an adaptive algorithm could be used to "learn" the weighting values, in a manner reminiscent of Samuels [49]. The failure filter prediction process could then automatically eliminate testing for relatively unimportant cues. This approach will be the subject of future work.

[9] See figure 1.6 on page 30.

phrase cannot receive Case.) If however, Principle B of the Binding theory is predicted to be the failure filter (on the basis of the structure cue *he*), then Principle B will be applied repeatedly to the indexings generated by *free indexation*. On the other hand, if the *Case condition on traces* operation was correctly predicted to be the failing filter, then *free indexation* need not be applied at all. The current system takes a conservative approach by also adopting the heuristic of delaying generators as far as possible to avoid unnecessary "fan-out". All potential filters are prioritized by the number of generators that they are dependent on.

- The mechanism is dynamic because it will re-evaluate (and perhaps revise) its choice of failing filter after a given phrase structure becomes more fully instantiated. For example, after chains are formed by *trace theory*, it re-computes the failure filter because some empty categories may be instantiated as traces. Similarly, after *functional determination* has applied, all empty noun phrases in A-positions will have their [$\pm$anaphoric,$\pm$pronominal] features instantiated.

In informal experiments, the implemented mechanism performs quite well in many of the test cases drawn from the reference text. Figure 5.5 compares the dynamic ordering mechanism (represented by the dashed line) to the fixed orderings shown previously in figure 5.1.[10] All results shown have been normalized with respect to the number of parser operations performed by `Dynamic`. (The results for `Binding` and `ECP` have been omitted from the graph for presentation purposes.) As the ratios indicate, `Dynamic` performs better than all the fixed orderings except in a few cases. This result is highlighted in the table of average ranks shown in figure 5.6. We should add that no "tweaking" (e.g. by adjusting the weights assigned to each structure cue) has been done for the comparison. Figure 5.7 should provide some idea of the range of accuracy of the prediction mechanism. Although the dynamic ordering mechanism performs well in comparison to the fixed orderings (even in some cases where the "hit" rate is low), it is by no means foolproof. There are many cases where the prediction mechanism will trigger an unprofitable re-ordering of the default order of operations. (We will return to discuss how this might be partially remedied in the next section.)

A few comments about further improvements in performance are in order here:

1. Given the poor hit rates for some of the examples (illustrated in figure 5.7), it should be possible to further improve the overall performance of dynamic

[10] Like the other parsers, `Dynamic` is also based on `Default`. As a last resort, in cases where the prediction mechanism cannot decide on an appropriate filter, `Dynamic` will adopt the underlying order used by `Default`.

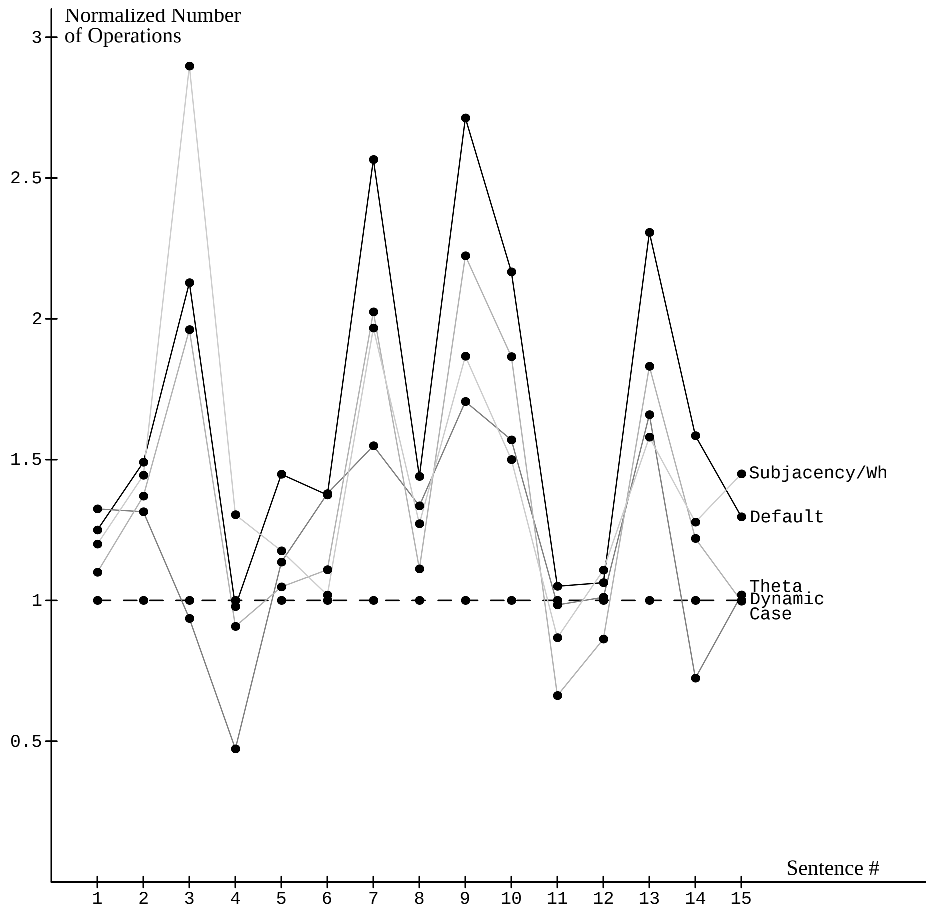


Figure 5.5 Comparison of dynamic ordering with fixed orders using normalized counts.

Parser Configuration	Average Rank
`Dynamic`	1.7/7
`Case`	2.7/7
`Theta`	2.9/7
`Subjacency/`*Wh*`-in-syntax`	3.8/7
`Default`	4.7/7
`Binding`	5.7/7
`ECP`	6.5/7

Figure 5.6 Comparison of dynamic ordering with fixed orders using ranks.

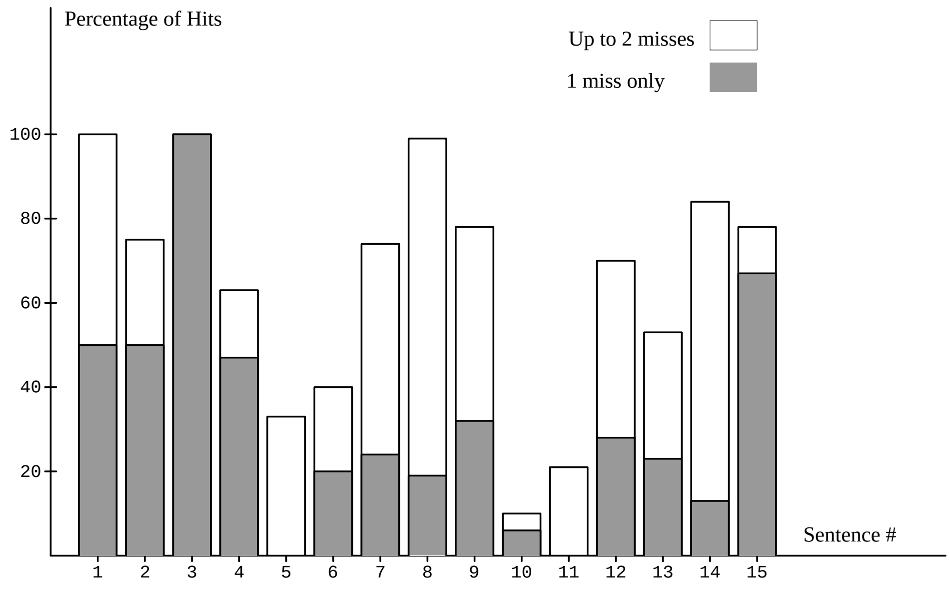


Figure 5.7 Percentage of hits achieved by dynamic ordering.

ordering, by tweaking the relative weights of structural cues, or possibly by "training" the mechanism over a suitable sample of sentences.

2. A more sophisticated prediction scheme, perhaps one based on more complex cues, could also increase the accuracy of the ordering mechanism. However, it is probably not cost-effective to do so. In general, there seems to be no *simple* way to determine whether a given structure will violate a certain principle.[11] That is, as far as one can tell, it is difficult to produce a cheap, but effective approximation to a filter (relative to the cost of the real operation). For example, in Binding theory, it is difficult to determine if an anaphor and its antecedent satisfies the complex locality restrictions imposed by Principle A without actually doing some search for a binder. Simplifying the locality restrictions is one way of reducing the cost of approximation, but the very absence of search is the main reason why the overhead of the present mechanism is not overwhelming.[12]

Logical Dependencies and Re-ordering

The second phase of the dynamic ordering mechanism is to re-schedule the parser operations so that the failing filter is performed as early as possible. Simply "fronting" the predicted failing filter to the front of the operations queue may violate logical dependencies between various parser operations. For example, suppose the Case filter was chosen to be the failing filter. To create the conditions under which the Case filter can apply, both Case assignment operations, namely, *inherent Case assignment* and *structural Case assignment*, must be applied first. Hence, fronting the Case Filter will also be accompanied by the subsequent fronting of both assignment operations — unless, of course, they have already been applied to the structure in question. Finally, in the case where the predicted failing filter fails to rule some structure, the re-ordering mechanism will then front the second choice and (if necessary) a third, and so on.

---

[11] If such a scheme can be found, then it can effectively replace the definition of the principle itself.

[12] The machine-independent results given above do not include the work spent on the dynamic ordering mechanism. Informal profiling suggests that the overhead of the mechanism varies from about 10% to 30%. This is already significant enough to outweigh the advantage of doing dynamic ordering on some cases. (Of course, the actual percentage would depend on the particular PROLOG system used.) For example, under symbolics PROLOG, the actual time taken to parse sentence (1) in figure 5.2 is virtually identical to the second best fixed ordering (`Theta`).

5.4 Linguistic Filters and Determinism

In this section we describe how the filters and generators model may be exploited to further improve the performance of the system for some operations. More precisely, we make use of an additional (computational) property of linguistic filters to improve the backtracking behaviour of the parser. The behaviour of this optimization will turn out to complement that of the ordering selection procedure quite nicely. That is, the optimization is most effective in exactly those cases where the selection procedure is least effective.

We hypothesize that linguistic filters, such as the Case filter, Binding conditions, ECP, and so on, may be characterized as follows:

> **Hypothesis**: Linguistic filters are side-effect free conditions on configurations

In terms of parser operations, this means that filters should never cause structure to be built or attempt to fill in feature slots.[13] Moreover, computationally speaking, the parser operations corresponding to linguistic filters should be deterministic. That is, any given structure should either fail a filter or just pass. A filter operation should never need to succeed more than once, simply because it is side-effect free.[14] By contrast, operations that we have characterized as generators, such as *trace theory* and *free indexation*, are not deterministic in this sense.

Given the above hypothesis, we can cut down on the amount of work done by the parser by modifying its behaviour for filter operations. Currently, the parser employs a backtracking model of computation. If a particular parser operation fails, then the default behaviour is to attempt to re-satisfy the operation that was called immediately before the failing operation. In this situation, the parser will only attempt to re-satisfy the preceding operation if it happens to be a generator. When

[13] So far, we have not encountered any linguistic filters that require either structure building or feature assignment. Operations such as θ-role and Case assignment are not considered filters in the sense of the definition given in the previous section. In the current implementation, these operations will never fail. However, definitions that involve some element of 'modality' are potentially problematic. For example, Chomsky's definition of an *accessible SUBJECT*, a definition relevant to the principles of Binding theory, contains the following phrase "*... assignment to α of the index of β would not violate the (i-within-i) filter* $*[_{\gamma_i} \ldots \delta_i \ldots]$". A transparent implementation of such a definition would seem to require some manipulation of indices. However, Lasnik (p.58) points out that there exists an empirically indistinguishable version of accessible SUBJECT without the element of modality present in Chomsky's version.

[14] It turns out that there are situations where a filter operation (although side-effect free) could succeed more than once. For example, the linguistic filter known as the "Empty Category Principle" (ECP) implies that all traces must be "properly governed". A trace may satisfy proper government by being either "lexically governed" or "antecedent governed". Now consider the structure $[_{CP}$ *What*$_1$ *did you* $[_{VP}$ *read* $t_1]]$. The trace t_1 is both lexically governed (by the verb *read*) and antecedent governed (by its antecedent *what*). In the current implementation, the ECP operation can succeed twice for cases such as t_1 above.

the preceding operation is a filter, then the parser will skip the filter and, instead, attempt to resatisfy the next most recent operation and so on.[15] For example, consider the following calling sequence:

$$\boxed{G_1} \cdots \longrightarrow \boxed{G_2} \longrightarrow \boxed{F_1} \longrightarrow \boxed{F_2} \longrightarrow \boxed{F_3}$$

Suppose that a structure generated by generator G_2 passes filters F_1 and F_2, but fails on filter F_3. None of the three filters could have been the cause of the failure by the side-effect free hypothesis. Hence, we can skip trying to resatisfy any of them and backtrack straight to G_2.

Note that this optimization is just a limited form of *dependency-directed backtracking.* Failures are traced directly to the last generator invoked, thereby skipping over any intervening filters as possible causes of failure. However, the backtracking behaviour is limited in the sense that the most recent generator may not be the cause of a failure. Consider the above example again. The failure of F_3 need not have been caused by G_2. Instead, it could have been caused by structure-building in another generator further back in the calling sequence, say G_1. But the parser will still try out all the other possibilities in G_2 first.

Consider a situation in which the principle selection procedure performs poorly. That is, for a particular ill-formed structure, the selection procedure will fail to immediately identify a filter that will rule out the structure. The advantages of the modified mechanism over the default backtrack scheme will be more pronounced in such situations — especially if the parser has to try several filters before finding a "failing" filter. By contrast, the behaviour of the modified mechanism will resemble that of the strict chronological scheme in situations where the selection procedure performs relatively well (i.e. when a true failing filter is fronted). In such cases, the advantages, if significant, will be small. (In an informal comparison between the two schemes using about eighty sentences from the reference text, only about half the test cases exhibited a noticeable decrease in parsing time.)

5.5 Conclusions

We have seen that principle ordering is an important consideration in the design of efficient principle-based parsers. The order-of-magnitude (or greater) differences

[15] This behaviour can be most easily simulated using a 'throw' exception mechanism or the 'cut' predicate in PROLOG. We can route all calls to filter operations through such devices.

in the amount of work required for parsing suggests that it is worthwhile to try to optimize parser control. We have seen how these differences can be explained by characterizing parser operations as being filters or generators. The fact that no globally optimal ordering exists has motivated the dynamic ordering mechanism described in the last section. An important question that remains is whether the same performance results hold for different languages. Preliminary experiments using the Japanese *wh*-questions (from Lasnik & Saito [34]) have been inconclusive.[16] Obviously, more tests with a wider range of examples will be necessary to reveal any significant differences. Also, it remains to be seen whether dynamic ordering will be profitable if the principles are revised or extended. However, the experience with early prototypes of the current system suggests that as the number of principles increases the effect of principle ordering also becomes more pronounced.

[16] Due to language parameterization, there are differences in the number of choice points of some parser operations. For example, functional determination is deterministic for English. However, the possibility of assigning *pro* in Japanese (*pro* not being available in English) turns functional determination into a generator. Due to the small number of test cases, it is not clear whether this particular difference will be significant enough to cause different orderings to be preferred for Japanese. (The situation is also complicated by the fact that the phrase structure grammars for S-structures are not strictly comparable.)

Chapter 6

Principle Interleaving

In the previous chapter, we explored a simple model of parsing that consists of two distinct phases: the first being the phase in which phrase structures are built, and the second being the phase in which ill-formed structures are "weeded out" by applying each well-formedness condition, or principle, in turn. Note that in such a scheme, there is a strict separation between phrase structure construction and principle application. That is, phrase structure for a complete sentence will be built before it is tested for well-formedness. An alternative strategy may be based on the observation that we need not wait until phrase structure construction is complete before applying certain principles. In many cases, principles exhibit *locality of reference* in the sense that may apply only to certain sub-structures within a given phrase structure. (We discuss such an example immediately below.) We can take advantage of this by co-routining, or *interleaving*, principle application with phrase structure construction as individual phrases are built. The motivating principle here is that if some sub-structure is found to be ill-formed, then any complete structure built from that sub-structure will also be ill-formed. For example, the sentence *John saw he* is ill-formed — cf. the well-formed sentence *John saw him*. In fact, the verb phrase corresponding to *saw he* is ill-formed no matter what elements precede or follow it. For example, *Mary believes John saw he*, *John saw he win*, and *I thought that Mary believed John saw he* are all ill-formed sentences. The problem here is that Case assignment, a principle of Case theory, states that verbs such as *see* assign accusative Case to noun phrase elements in object position such as *he*. Therefore, the Case conflict can be discovered by applying the principle of Case assignment as soon as the object of *see* has been identified. Hence, it would seem to make sense computationally speaking (as suggested by Crocker [17]), to favour the interleaved over the non-interleaved approach of the previous chapter. Such a strategy of principle interleaving may provide an answer to the problem of controlling the combinatorial explosion due to overgeneration. In this chapter, we examine three basic questions for this alternative model:

- Is it worthwhile, in general, to interleave or not?
- Are there some principles that can be more profitably interleaved than others? If so, which ones are these?
- What is the cumulative effect of interleaving many principles?

To investigate these issues, a simple model of off-line interleaving was used. An interleaving mechanism merges definitions of linguistic principles with the LR(1)-based machine used to build phrase structure (described earlier in chapter 4). The modified LR machine will then apply principles on an incremental basis to phrase structure constituents in accordance with the order in which the LR algorithm builds structure. The interleaver makes use of a type inference mechanism to decide how best to merge principles with LR actions. The notion of the type of a principle will be introduced as an example of a computational property that can be used to help make design decisions. In particular, we will argue that the type cardinality of a principle is an important factor in determining which principles lend themselves to interleaving. We will also argue that the utility of interleaving is parasitic on the efficiency of the algorithm used to build phrase structure.

Roadmap

The remainder of the chapter is divided into four major sections:

- Section 6.1 develops the case for an off-line model of interleaving and discusses the problem of maintaining a transparent representation.
- Next, section 6.2 introduces the relevant notion of the "type of a principle" and presents a case study of how type construction proceeds on definitions of parser operations that implement conditions of θ-role and inherent Case assignment.
- Next, section 6.3 describes the principle interleaving mechanism that makes use of the type inference procedure to merge designated principles into the LR tables used by the structure building procedure.
- Finally, section 6.4 discusses how the type inference procedure performs in the current system. Here, we will also present and explain the differences in efficiency resulting from interleaving various combinations of principles.

6.1 A Model of Off-Line Interleaving

We will begin by outlining why the obvious on-line polling model is an impractical one. We will then proceed to sketch the requirements for an alternative off-line model based on type inference.

6.1.1 The Naïve Polling Model

Perhaps the obvious (and general) method for interleaved execution is to simply try to apply each principle every time some structure is built. We will call this the *naïve polling model*. Unfortunately, this is an impractical model because of the relatively high overhead of 'polling' constituents. The reason why polling is expensive comes from the general observation that many linguistic principles are fairly restrictive about the range of configurations that they apply to. In other words, many principles have the following structure: "for all constituents X such that X satisfies some property P, then some property Q *must* hold or else some addition (or modification) R *must* be made to constituent structure." For example, consider Case assignment. Roughly speaking, the principle states that: "for all configurations X, X being a configuration of Case assignment, assign Case to an appropriate noun phrase in X." However, the basic problem here is that it may take a considerable amount of computation to identify and retrieve the relevant components of a Case configuration. For example, we will need to find an element to assign Case, we will also need to find a corresponding element to receive Case, and finally, we will need to check that these two elements bear the appropriate structural relation to each other. But since these requirements are fairly restrictive, most constituents will fail at one of three points mentioned. However, this repeated evaluation of the pre-conditions will result in a high overhead above the amount of computation required to process those configurations that turn out to be Case configurations. (A more concrete example of a principle along these lines may be found in the definition of θ-role assignment to be discussed in section 6.2.)

Note however that, unless we exhaustively test each structure, we run the risk of either failing to rule out an ill-formed structure or causing a well-formed structure to be ruled out (by failing to assign Case.) In other words, non-exhaustive testing would constitute an *unsound* implementation of the (relativized) universal quantifier. Of course, if certain classes of structures that never satisfy the pre-conditions of a given principle can be identified and eliminated from consideration (on an *off-line* basis), this can lead to a substantial reduction in the number of (unsuccessful) tests necessary at *parse-time*. In the next section, we will outline an improved model that will only need to selectively poll partially-constructed phrase structures.

6.1.2 The Revised Model

Basically, the idea will be to cut the overhead by using information about the range of possible phrases that the pre-conditions of a principle may apply to. For instance, if we can determine that a certain pre-condition only applies to, say, noun, verb and adjectival phrases; then we can safely eliminate polling, say, of all adverbial phrases. Of course, the problem is how to extract this categorial information simply by inspecting the logic definition of a principle. Roughly speaking, a *type* will be a set that represents the range of category labels associated with a configuration. A type inference mechanism that operates over logic definitions will used to compute type values. Before proceeding further, let us briefly review some of the properties that such a mechanism must have to satisfy the design goals (discussed previously in chapter 2).

In the current system, an attempt has been made to abstract principle definitions away from control issues such as the question of non-interleaved versus interleaved processing. The basic idea here being that it should be possible for the grammar writer to specify principles in a manner that is both neutral and transparent with respect to such issues. In the current system, the fact that types are computed when principles are to be interleaved is completely transparent to the grammar writer. (However, the grammar writer is free to explore different control options independently of the representation.) To achieve this, a type inference mechanism must have the following properties:

1. The type inference mechanism must be *automatic* in the sense that it should compute types solely from a principle definition without having to solicit extra information from, or be volunteered by, the user, e.g. in the form of type specifications as program annotations. This also implies that the mechanism must be capable of *degrading gracefully* in cases where there is insufficient information to determine a particular type. In the current system, no annotations are allowed.

2. The *off-line* interleaving process must preserve *soundness*. For example, as discussed earlier, type computation must never cause the parser to (erroneously) fail to impose a well-formedness condition on an ill-formed structure.

3. Finally, of course, the inference mechanism must always halt. As will be discussed in a subsequent section, this is a potential problem with the type relations to be defined, given the "multiple-pass" nature of the implementation.

Although, type inference must be sound, it need not be *complete* (in a sense to be made clear below) in order to retain transparency. Suppose T is the smallest possible sound type for a configuration X, i.e. for every category c in T, there exists a constituent of category c that satisfies the conditions of X. Now, suppose a type T' computed for X by an inference algorithm is *strictly larger* than T in the sense that $T \subset T'$. Then, we say that the algorithm is incomplete (but still sound) since it has failed to determine the (minimal) type of X. To make the point clearer, we can say that the original naïve polling algorithm is equivalent to the revised system with a type inference algorithm that always returns the set of all possible category labels of the (linguistic) theory; that is, the algorithm is effectively failing to compute any meaningful type. (We will denote the set of all possible category labels, i.e. the *universal type*, by U.) The algorithm that we will describe in this paper will automatically substitute U for any part of the principle definition that a type inference fails to apply. As the section on inference rules will describe, this is usually not "fatal". In fact, this default substitution allows the inference algorithm to degrade gracefully by computing a strictly larger type but remaining sound.

6.2 Case Study: Types and θ-Role Assignment

In this section we will illustrate the use of type inference on the conditions of θ-role assignment — one of the parser operations that the θ-criterion is logically dependent on. First, we will show how θ-role assignment may be formalized in a logic program using the notation defined in chapter 3. Next, we will introduce the notion of a type of a principle, and define type inference rules that operate over primitive linguistic relations and logic programs in general . We will then apply these rules to the case of θ-role assignment, and further extend the example to include inherent Case assignment, to illustrate how the type computation is actually carried out. Finally, we will outline some of the limitations and tradeoffs in the implemented procedure.

6.2.1 The Conditions of θ-Role Assignment

θ-theory is concerned with the proper distribution of *thematic* roles. For example, consider the sentence:

(1) the police arrested John

Here, *the police* and *John* are considered to bear the thematic roles of 'agent' and 'patient', or 'theme', of the predicate *arrest*, respectively. In syntax, these relations are established by having a lexical head such as the verb *arrest* (which has an 'agent' and a 'theme' θ-role), assign its θ-roles to, i.e. *θ-mark*, the subject and object positions, respectively. Now, the fundamental principle of θ-theory is the θ-criterion, a two-part condition which ensures that θ-roles are properly discharged to *arguments*; roughly speaking, syntactic elements that have a referential function. Here is a typical definition of the θ-criterion (from Lasnik & Uriagereka [35]):

(2) θ-Criterion:
a. Every argument must be assigned a θ-role.
b. Every θ-role must be assigned to an argument.

For example, the θ-criterion rules out the following ill-formed examples:

(3) a. *the police arrested John Bill
b. *the police arrested
c. *there arrested John
d. *there arrested

In (3a), (2a) is violated since *Bill* is an argument without a θ-role. Here, the theme θ-role is assigned to *John*, and the agent role to *the police*. In (3b), no argument is available to bear the theme θ-role; and in (3c), the agent role is assigned to a non-argument, the pleonastic element *there*. In both cases, (2b) is violated. Finally, (3d) violates both sides of the θ-criterion.

The (Extended) Projection Principle requires that the θ-criterion must hold at all syntactic levels, i.e. at D-structure, S-structure and LF. For example, one consequence is that movement from a $\bar{\theta}$-position (a position to which no θ-role is assigned) to a θ-position is prohibited, as in *John$_i$ mentioned that t$_i$ rains — cf. *John mentioned that it rains* where no movement of a noun phrase has taken place.

To apply the θ-criterion, we have to first assign θ-roles. In general, *internal* θ-roles will be assigned to complement positions, e.g. the 'theme' θ-role of *arrest*, and *external* θ-roles will be assigned to subject positions, e.g. the agent θ-role of *arrest*. Following Chomsky [12], the possible cases of θ-marking are:

1. *Lexical heads θ-mark their complements.*

   We have already seen the case when the head is a verb. Nouns also θ-mark their complements. For example, the nominal form *destruction* assigns the same (internal) theme θ-role to *the city* in the configuration [[$\overline{\text{N}}$ *destruction*] [*NP of the city*]] that *destroy* assigns in [[$\overline{\text{V}}$ *destroyed*] [*NP the city*]]. Similarly, the adjective *proud* θ-marks the pronoun *he* in [$\overline{\text{A}}$ *proud* [*NP of him*]], and the preposition *to* assigns a destination θ-role in [$\overline{\text{P}}$ *to* [*NP the city*]]. Note that some heads such as the verbs *persuade* and *ask*, as in [$\overline{\text{V}}$ *persuaded* [*NP John*] [*CP that he should leave*]] and [$\overline{\text{V}}$ *asked* [*NP John*] [*CP to leave*]], may assign two or more internal roles. Others such as *sleep* take no objects and therefore have no internal roles to assign.

2. *Verbs that possess an* external *θ-role will θ-mark their subject.*

   For example, the verb *hit* will assign an 'agent' θ-role to *John* in [*IP* [*NP John*] [*VP hit Bill*]]. Note that in some cases, a verb may combine with an internal predicate to θ-mark the subject. For example, in *John is crazy*, the argument *John* is indirectly θ-marked by the adjective *crazy*.

Let X and Y be constituents, and let F_X be the feature set associated with X.
Let F be an instantiated feature.

Primitive Operations
`cat(`X`,`c`)` holds if c is the category label of X.
X `complement_of` Y holds if X occupies a complement position in phrase Y.
X `has_constituent` Y holds if Y is an immediate sub-constituent of X.
X `has_constituent_head` Y holds if Y is an immediate sub-constituent of and heads X.
X `has_constituents` L holds if L represents all the immediate sub-constituents of X.
X `has_feature` F holds if $F \in F_X$.
X `specifier_of` Y holds if X occupies the specifier position in phrase Y.

Figure 6.1 Primitive operations.

3. *Nouns that possess an external θ-role (optionally) θ-mark the subject position of the noun phrase that they head.*

   Consider the two noun phrases shown in (4) (taken from Chomsky [12]):

   (4) a. Bill's fear of John
       b. the fear of John

   In (4a), *Bill* receives same θ-role 'experiencer' that it receives in *Bill fears John*, but in (4b) the corresponding θ-role is not assigned.

(In the following discussion, we will make use of the linguistically-motivated primitive operations shown in figure 6.1.)

The configurations of θ-marking are summarized by the predicate `thetaConfig/3` shown in figure 6.2. `CF` is a θ-configuration with two components: (1) a list of the θ-roles to be assigned, and (2) a list of the constituents to receive the θ-roles. Note that the thematic representation of a head, the *θ-grid*, is encoded using a lexical feature `grid(L1,L2)`, where the lists `L1` and `L2` hold the external and internal θ-roles of the head, respectively. For example, *arrest* would have the feature `grid([agent],[theme])`.

The translation from each θ-configuration case described above to the formal definition in terms of the phrase structure primitives provided is relatively straightforward. Let us briefly review the first clause of `thetaConfig/3`. This clause is designed to extract the two components of a θ-configuration if `CF` is a configuration where a lexical head would θ-mark its complements. In general, `CF` need only be the

```
thRoleAssign in_all_configurations CF where
     thetaConfig(CF,Roles,Els) then assignRoles(Roles,Els).

thetaConfig(CF,[Role|Rs],Complements) :-% Case 1: [Xi X Complements]
     CF has_feature grid(_,[Role|Rs]),
     CF has_constituents L,
     CF has_constituent_head Head,
     pick(Head,L,Complements).          % List op: pick out Head from L
                                        % leaving Complements.
thetaConfig(IP,[Role],[NPSpec]) :-      % Case 2: [IP Spec [I1 I VP]]
     cat(IP,ip),
     VP complement_of IP,
     extRoleToAssign(VP,Role),
     NPSpec specifier_of IP.

thetaConfig(NP,Roles,Args) :-           % Case 3: [NP Spec [N1 N ..]]
     cat(NP,np),
     NP has_constituent NPSpec,
     cat(NPSpec,np),
     NP has_feature grid([Role],_),
     optionalThRole(Role,NPSpec,Roles,Args).

optionalThRole(Role,Arg,[Role],[Arg]).  % Commit to assigning the role.
optionalThRole(Role,Arg,[],[]).         % Omit assigning the role.
```

Figure 6.2 The definition of θ-role assignment.

smallest constituent containing both components.[1] This occurs when the head and its complements are both immediate constituents of `CF`. The identification of this condition is performed by the last three conjuncts of the clause. The first conjunct is there to make sure that the head has at least one θ-role to assign. For example, this would rule out [*VP* [*V sleeps*]] as a potential case of head–complement θ-marking. (Note that here we make use of the inheritance of features to examine the θ-grid associated with the head.) The definition of the other cases proceeds in

[1] It is not strictly necessary to pick the smallest configuration; in fact, the definition could be written to accept any arbitrary element containing both components. But, the reason for preferring the smallest lies in the bottom-up nature of the algorithm that builds structure. Recall that, in the interleaved model, we would like to be able to apply principles as early as possible. By defining the θ-configuration to be the smallest possible constituent, we can apply θ-role assignment as soon as the relevant components have been found.

```
% NB: assignRoles/2 will be defined in terms of assignRole/2
assignRole(Role,Argument) :- Argument has_feature theta(Role).
```

Figure 6.3 Assigning a θ-role to an argument.

a similar fashion. Finally, the operation of θ-role-assignment itself is formalized by universally quantifying over the configurations of phrase structure. The definition shown is meant to be read as follows:

> "In all configurations named by `CF` such that `CF` is a θ-configuration with components `Roles` and `Els`; then assign the θ-roles `Roles` to elements `Els`."

The actual assignment of θ-roles to constituents is carried out by `assignRoles`. Constituents will contain θ-slots that will be represented as a complex feature `theta(`S`)`, where S will hold the actual θ-role. Hence, assignment simply boils down to unifying a θ-slot with a given θ-role, as shown in figure 6.3. (Here, to keep the definition short, we only illustrate the case for a single θ-role and constituent.[2])

The issue of the high overhead involved in repeated evaluation of the preconditions of a principle can be seen clearly in this definition. The cost of determining the θ-configurations is by far the most expensive part of applying the principle. As we have seen, the conditions under which θ-marking takes place are quite restricted. Hence, a large majority of constituents that are constructed should fail to satisfy the predicate `thetaConfig`. But, given that there are three clauses to test and that several conjuncts of the clauses may succeed in the course of testing, each failure can be quite expensive. By comparison, the amount of additional work required to actually assign the θ-roles, once the appropriate components of the configuration have been determined, is small.

[2] The version of `assignRoles/2` discussed here is a little more complicated for two reasons: (1) θ-grid roles are assumed to be unordered. (More recent versions of the system adopt a revised theory in which some canonical order of complements exists.) For example, verbs such as *give* have two internal θ-roles that may permute freely in syntax, as in *gave John a book* and *gave a book to John*. (2) Also, θ-roles may have several syntactic realizations. For example, following Chomsky [12], *persuade* select a goal and a proposition as internal θ-roles. A proposition may be realized as either as a noun phrase or a clause, as in *persuaded* [*NP John*] [*CP that he should leave*] or *persuaded* [*NP John*] [*NP of the importance of leaving*]; but a goal may only be realized as a noun phrase. Note however, that although θ-roles are unordered, principles may apply to exclude ungrammatical permutations. For example, **persuaded* [*CP that he should leave*] [*NP John*] is ruled out by the Case adjacency requirement which is present in English. That is, a noun phrase must be adjacent to its Case assigner in order to receive structural Case.

6.2.2 Type Definitions and Inference Rules

In this section, we will formally define the notion of a type together with type equations for linguistic primitives and type inference rules over logic programs. In the following section, we will illustrate the process of type computation on the definition of θ-role assignment introduced in the previous section.

We begin by defining the possible category labels:

Definition 1 (Category labels)
Let L be the set of $\overline{\mathrm{X}}$-theory head category labels, i.e. $\{N, V, A, P, I, C\}$. Let L' be the set of non-$\overline{\mathrm{X}}$ category labels, e.g. $\{Adv, Det\}$. Now, let proj denote the $\overline{\mathrm{X}}$ "projects to" relation. That is, let c proj $\overline{c}$ and $\overline{c}$ proj $\overline{\overline{c}}$ hold, for $c \in L$. Then, let proj* be the reflexive transitive closure of proj. Finally, let V denote the (finite) set of all possible category labels, i.e. $\{c' \mid c \text{ proj}^* c', \text{for } c \in L\} \cup L'$.

We can now define the notion of a type in terms of category labels:

Definition 2 (Type)
A *type* is simply a member of 2^V. In particular, let U, the *universal type*, be the largest member of this set, i.e. V.

In a logic program, types will be associated with logical variables that represent constituents. We will use the expression $X:T$ to represent a constituent variable X with associated type T. We will now define some of the type equations associated with linguistically-motivated primitives such as those in figure 6.1.

Definition 3 (Category labelling)
Given a constituent X and a category label c:
`cat(`X`:`T`,`c`)` holds $\Rightarrow$
    $T = \{c\}$ if c is a category label, or
    $T = \text{typeValue}(c)$ if c is uninstantiated.

For example, consider [Defn. 3] which defines the type of its first argument as follows:[3]

1. When c is instantiated to some label, say `NP`, as in "`cat(`X`,NP)`," we can immediately determine that X can only range over constituents with that label — for our example, we say that the type of X is $\{$`NP`$\}$.

2. On the other hand, c may be left uninstantiated: for example, we might write "`cat(`X`,`C`), head(`C`)`" to state that X must be a head. Here, the type of X

[3]Note that these type definition rules are transparent to the user. However, if the grammar writer wishes to incorporate a new primitive relation involving constituents, then a corresponding type definition should be written — although one is not strictly necessary, that is, the inference algorithm will still function, albeit, less effectively.

cannot be determined solely from "cat(X,C)." In general, if 'typeValue(C)' is returned, the inference algorithm will interpret it to mean that should make a second pass through the predicate in which cat(X,C) occurs to find other literals that involve C and compute all possible values of C where feasible.[4] In this case, **head** is a simple primitive of $\overline{\text{X}}$-theory that holds only for categories N, V, A, P, I, and C. Hence, the type of X will become $\{\mathtt{N}, \mathtt{V}, \mathtt{A}, \mathtt{P}, \mathtt{I}, \mathtt{C}\}$.

The lexicon can also play a part in type determination. Consider [Defn. 4] which applies to the feature membership primitive:

Definition 4 (Feature membership)
Given a constituent X and an instantiated feature F. Let E and F_E be a lexical entry and its feature set:
$X:T$ **has_feature** F holds $\Rightarrow$
$T = \{c \mid \exists\ E$ of category c' st. $F \in F_E, c'$ proj$^* c\}$
when F is a lexical feature, else
$T =$ typeValue(F) if F has a given type, otherwise
$T = U$.

For example, if only verbs and adjectives may be exceptional case markers (ECM), and "**X has_feature ecm**" holds, then we can infer (assuming feature inheritance) that **X** must have type (or be a sub-subset of) $\{\mathtt{V}, \overline{\mathtt{V}}, \mathtt{VP}, \mathtt{A}, \overline{\mathtt{A}}, \mathtt{AP}\}$. This is captured by the first type equation. The second equation applies to non-lexical features that are assigned independently of the lexicon, but have type definitions specified. Finally, the last equation introduces the universal type U. This applies as a 'catch-all' for all features not handled by the earlier type equations. (In fact, unless otherwise stated, type U will be the default type assigned whenever no type equation applies.)

Next, to conclude our examples of primitive relations, [Defn. 5] illustrates the notion of a type relation:

Definition 5 (Head relation)
Given two constituents X and Y:
$Y:T_Y$ **head_of** $X:T_X$ holds $\Rightarrow$
$T_X = \{c \mid c'$ proj$^* c$, for each $c' \in T_Y\}$,
$T_Y = \{c \mid$ head(c), c proj$^* c'$ for each $c' \in T_X\}$

head_of holds if the first constituent is the head of the second. In this case, two type equations, one for each argument, are defined in terms of the type of the other.

[4] The conditions under which typeValue(C) causes partial evaluation of a literal are too detailed to cover here. In this case, the predicate head is defined by the clauses "head(a) head(n) head(v) ...head(c)." The algorithm detects that head is defined using ground unit clauses, from which it can collect values for C without going into an infinite loop.

We now move on to examples of type composition rules for logic programs. In the following definitions, a *simple formula* F may be an atom. If F_1 and F_2 are simple formulas than so are: (`\+`F_1), (F_1`,`F_2), (F_1`;`F_2). For example, [Defn. 6] defines how types may be composed for conjunction '`,`':

Definition 6 (Conjunction) Given simple formulas F_1 and F_2:

$$\frac{X : T_1(F_1), \qquad X : T_2(F_2)}{X : T_1 \cap T_2(F_1\texttt{,}F_2)}$$

This inference rule has the following interpretation: "if X has type T_1 in F_1, and X also has type T_2 in F_2, then X has type T_1 intersect T_2 in the conjunction of F_1 and F_2." Although, strictly speaking, Horn clauses do not have disjunctions in the body of the clause, (see Lloyd [36]); it is both simple and convenient to allow the corresponding rule for disjuncts:

Definition 7 (Disjunction) Given formulas F_1 and F_2:

$$\frac{X : T_1(F_1), \qquad X : T_2(F_2)}{X : T_1 \cup T_2(F_1\texttt{;}F_2)}$$

The type rule for predicate definition is quite similar to that for disjunction, except, of course, for the variable substitutions:

Definition 8 (Predicate Definition)
Let predicate P be defined by n clauses of the form $A_i\texttt{:-}F_i, \quad 1 \leq i \leq n$, where A_i and F_i are atoms and simple formulas, respectively. Let A be an atom and let σ_i be a most general unifying substitution for A_i wrt. A. Then:

$$\frac{\forall i \text{ st. } \exists \sigma_i \text{ st. } A = A_i\sigma_i \quad X : T_i(A_i\texttt{:-}F_i)\sigma_i}{X : \bigcup_i T_i(A)}$$

Finally, we conclude the series of type definitions with a brief discussion of negation `\+` (as failure):

Definition 9 (Negation)

$$\frac{X : T(F)}{X : U(\backslash{+}F)}$$

Note that X has type U after negation, irrespective of its original type. This is a conservative rule which essentially provides no type information. Note however, that exceptions are defined for certain goals. For example,the primitive `has_feature` [Defn. 4] is an example where type information is not deleted across negation:

Definition 10 (Negation and feature membership)
Given a constituent X and an instantiated feature F. Let E and F_E be a lexical entry and its feature set:
\+ $X:T$ **has_feature** F holds $\Rightarrow$

$T = \{c|\ \exists E$ of category c' st. $F \notin F_E,\ c'$ proj$^* c\}$ if F is a lexical feature, else
$T = U - T'$ where $T' =$ typeValue(F) if F has given type, otherwise
$T = U$.

(Note: a complete list of the type rules used are given in appendix B.)

6.2.3 Type Computation

In this section, we will present an example of type computation using θ-role assignment, as defined in section 6.2.1. Let us consider again the clauses for `thetaConfig` shown previously in figure 6.2.

1. The first clause encodes the case of head–complement θ-marking. For this to occur, the head must have a θ-grid with one or more internal θ-roles. This is encoded by the first conjunct "`CF has_feature grid(_,[Role|Rs])`." The type inference algorithm will consult the type rule [Defn. 4]. According to this rule the algorithm must consult the lexicon to find all categories c' that have one (or more) lexical entries with a θ-grid containing one or more internal arguments. In the current lexicon, such entries will only exist for categories `N`, `V`, `A` and `P`. (For examples, see section 6.2.1.) Then, the type of `CF` is determined as the set of labels c such that c is zero or more (levels of) projections from c'. Hence, `CF` has type constrained by $\{\mathtt{N},\overline{\mathtt{N}},\mathtt{NP},\mathtt{V},\overline{\mathtt{V}},\mathtt{VP},\mathtt{A},\overline{\mathtt{A}},\mathtt{AP},\mathtt{P},\overline{\mathtt{P}},\mathtt{PP}\}$.

   Not all the primitives have associated type equations. For example, there is no equation for the second conjunct "`CF has_constituents L`." The list operation used in the fourth conjunct `pick` is a generic predicate that has no special linguistic significance. Hence, `CF` is given type U in both cases.

   However, the third conjunct, namely "`CF has_constituent_head Head`," specifies that `CF` immediate dominates its head; in other words, `CF` must be a single-bar category. Hence, `CF` has type upper-bounded by $\{\overline{\mathtt{N}}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}, \overline{\mathtt{I}}, \overline{\mathtt{C}}\}$.

   Using the type rule for conjuncts, we intersect each type instance to obtain $\{\overline{\mathtt{N}}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}\}$.

2. The second and third clauses encode two cases of external θ-role assignment: a noun assigning its external θ-role to its NP specifier, and the subject of IP being assigned an external θ-role, either directly or indirectly, via the VP predicate. In both cases, by applying the type rule for category labelling [Defn. 3],

```
inCaseAssign in_all_configurations CF
   where inCaseConfig(CF,Case,Items)
        then assignInherentCase(Case,Items).

inCaseConfig(CF,Case,Items) :-
      thetaConfig(CF,_,Items),
      Head head_of CF,
      cat(Head,C),
      inherentCaseAssigner(C,Case).

inherentCaseAssigner(n,gen).
inherentCaseAssigner(a,gen).
inherentCaseAssigner(p,obq).
```

Figure 6.4 A definition of inherent Case assignment.

> `CF` will be immediately restricted to a singleton set, {`NP`} in the former and {`IP`} in the latter case. Note that the algorithm will still try to explore the other conjuncts present in both clauses.[5]

Finally, using the type rule for predicate definition [Defn. 8] together with the precondition in **`assignThRoles`**, we can infer that the type of **`thConfig`** is the union of the types computed for each of the three clauses: that is, {$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, `IP`}. Note that this type is obviously much smaller than the universal type. Although not all structures that have its category labelled as one of these type components, e.g. [$\overline{V}$ [*V sleep*]] is not a θ-configuration as mentioned before, we have safely and automatically eliminated much of the unnecessary overhead incurred by the naïve polling model.

Let us extend this example one step further to illustrate the capability of the inference procedure to trace a number of different type variables. In general, evaluating type relations may require the algorithm to make multiple-passes through principle definitions. We have already seen one example of this, namely "**`cat(`**X**`,`**C**`),`** **`head(`**C**`)`**", in which the computation of the type of X will require the computation of the possible values for the type value C. In the current implementation, this is done by making a separate pass for each such unknown. A more sophisticated example can be found in the fragment shown in figure 6.4. This example is taken from

[5] Incidentally, if the empty type for a definition is ever returned, then that definition can never succeed. Although, we have argued for the transparency of the type inference procedure, this is perhaps one case in which debugging information can and should be conveyed to the grammar writer.

the definition of inherent Case assignment, an assignment operation that operates on a proper subset of θ-configurations. The additional restriction here being that the head must be an inherent Case assigner.

By using the type rule for the `head_of` primitive [Defn. 5], the algorithm can infer that the type of `CF` is related to the type of `Head`. Therefore, by switching to computing the type of `Head`, the algorithm may be able impose further constraints on the type of `CF`. By using rule [Defn. 3] on the atom "`cat(Head,C)`," we can infer that the type of `Head` is given by the set of possible values for `C`. Switching once more to compute the values of `C`, we can follow the definition of `inherentCaseAssigner` to obtain $\texttt{C} = \{\texttt{N}, \texttt{A}, \texttt{P}\}$. By plugging these values back into the type equation for `head_of`, `CF` is constrained to be $\{\texttt{N},\overline{\texttt{N}},\texttt{NP},\texttt{A},\overline{\texttt{A}},\texttt{AP},\texttt{P},\overline{\texttt{P}},\texttt{PP}\}$. Finally, we know that `CF` has type $\{\overline{\texttt{N}}, \texttt{NP}, \overline{\texttt{V}}, \overline{\texttt{A}}, \overline{\texttt{P}}, \texttt{IP}\}$ in `thetaConfig`, so by type intersection, we have that inherent Case assignment has type $\{\overline{\texttt{N}}, \texttt{NP}, \overline{\texttt{A}},\overline{\texttt{P}}\}$.

Although, type relations allow the algorithm to infer the type of one variable in terms of another, as illustrated by in the preceding example; unfortunately, this also allows *circularity* in a chain of type inferences to occur. Perhaps, the simplest example occurs in the definition of `head_of` where the type of the first argument is defined in terms of the second, and vice-versa. Currently, the adopted solution is to keep track of the names of the variables being traced, and to terminate the search as soon as a previously encountered variable has been detected. Then, type computation will be resumed with the universal type substituted for all outstanding type variables. This system can be extended to cope with looping caused by recursive predicates by treating a recursive call as if the system switched to computing a new type variable. For example, this will prevent the algorithm from looping on `p(...,X,...) :- p(...,X,...)`, where X is the constituent variable being traced. This rudimentary loop detection scheme seems to be sufficient to guarantee termination, but at the cost of generality.[6]

This completes our description of the basic type inference procedure. We will return to discuss further extensions in a later section. The procedure has been successfully tested on all of the (relevant) principles found in the current system. (The pertinent results are summarized in section 6.4.) Next, we will turn our attention to the description of the mechanism (built on top of type inference) that transparently compiles principle definitions into the appropriate forms for interleaved execution.

6.3 The Interleaver

[6] It is easy to construct an example to show the limitations of the loop detection scheme. For example, consider the following (artificial) definition of a principle p: `p(0,X) :- cat(X,np).` `p(1,X) :- p(0,X).` The loop detection algorithm will prematurely abort the type computation process in the case of the goal p(1,X).

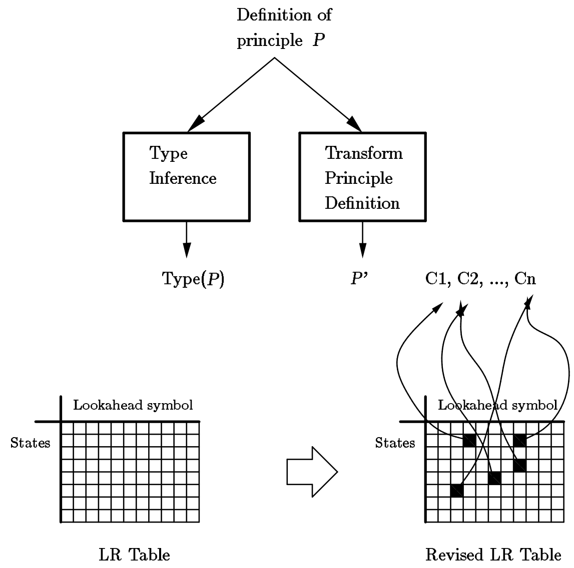


Figure 6.5 The interleaving process.

The job of merging principle definitions with the LR tables used by phrase structure recovery procedure is carried out by the off-line interleaver. (See chapter 4 for an overview of the LR parsing model.) Given the definitions of one or more principles, it uses the type inference mechanism to deduce the appropriate points in the LR tables where the parser operations corresponding to the (interleaved) principles should be called. Figure 6.5 illustrates this process schematically. For example, as we saw in the last section, θ-role assignment has type $\{\overline{\texttt{N}}, \texttt{NP}, \overline{\texttt{V}}, \overline{\texttt{A}}, \overline{\texttt{P}}, \texttt{IP}\}$. This means that θ-role assignment need only be consulted when the phrase structure recovery procedure completes a constituent with a category label belonging to the type. In this section, we will describe how principle definitions are transformed into a form that may be directly called by LR table actions. We will also discuss the problem of respecting logical dependencies in the case where more than one principle is interleaved.

6.3.1 Compiling Principle Definitions

Adding external calls to LR action entries is a standard technique used to implement side effects or "semantic actions" in compiler construction. In our case, this is implemented by having all reduce actions call a "hook" predicate, and passing along the structure it has just built. There is a separate hook predicate for each non-terminal symbol. For example, all reduce actions for rules matching $\texttt{NP} \rightarrow \alpha$ will call a predicate reduceNP/1 (passing along an NP constituent as input). In general, if a principle P has type T, then for each category label c belonging to T, there will be a goal P_c(X) in the body of the clause reduce$c$(X) :- β, where $P_c/1$ will be a specialized version of the definition for P (with respect to c). In this model, the task of the off-line interleaver is to recast principle definitions (written initially using one of the two forms, in_all_configurations or compositional_cases_on, as described in chapter 3), into a series of predicates specialized for each category given by the type of the principle. Here, we will describe the procedure in detail for the in_all_configurations form only.

Compiling the in_all_configurations form

Principles defined using in_all_configurations have the following general form:

```
<Name> in_all_configurations <CF>
        where <Pre-Cond-1> then <Post-Cond-1>
        else <Pre-Cond-2> then <Post-Cond-1>
        ...
        else <Pre-Cond-n> then <Post-Cond-n>
```

The type of the principle will be the type of the logical variable <CF>. This will be the union of the type of <CF> in each pre-condition. For each category c, we can determine if c belongs to the type of each pre-condition and produce a version of the general form specialized with respect to c. For example, if c belonged to the type of <CF> for <Pre-Cond-1> and <Pre-Cond-n> only, then we can construct the following single clause predicate:

```
<Namec>(<CF>) :-
       <Pre-Cond-1>
       -> <Post-Cond-1>
       ;  <Pre-Cond-n>
       -> <Post-Cond-n>
       ;  true
```

(Note: we have used standard PROLOG notation for "if–then–else" constructions. Also, assume that the goal true always succeeds.)

Here, type inference also eliminates unnecessary run-time testing (with certain failure) for <Pre-Cond-2> through <Pre-Cond-(n-1)>. In general, there may be n different specialized predicates generated, where n is the cardinality of the type of the principle. This compilation procedure is summarized in figure 6.6. There are two aspects of the procedure worth noting:

1. A preliminary expansion stage (Step 1) is used to further eliminate unnecessary testing in some cases. For example, consider again the definition of θ-role

Given a principle definition of the form:

```
:- <Name> in_all_configurations <Variable>
        where <Pre-conditions-1> then <Conditions-1>
        else <Pre-conditions-2> then <Conditions-2>...
```

with `<Variable>` having type T. Then:

1. **Preliminary Unfold Step**: If the definitions of any pre-condition has multiple clauses, then expand that pre-condition by adding an extra condition for every clause.
2. For each category label c in T do the following:
3. Let sequence S_c be the empty sequence $\langle\rangle$.
4. **Collect Compatible Conditions**: For each condition "P `then` C" in the definition:
   (a) Let T_P be the type of `<Variable>` wrt. P.
   (b) If $c \in T_P$ then c is compatible with "P `then` C". If so, append $\langle(P, C)\rangle$ to S_c.
5. **Construct Predicate**: Combine the conditions in S_c into a nested form F according to the following rules:

$$\langle\rangle \longmapsto \texttt{true}.$$
$$\langle(P_1, C_1), S_1\rangle \longmapsto (P_1 \texttt{ -> } C_1 \texttt{ ; } F_1)$$
$$\text{if sub-sequence } S_1 \longmapsto \text{form } F_1.$$

   If F has been constructed before, do nothing. Otherwise, generate a new predicate symbol p and emit p(`<Variable>`) `:-` F.

Figure 6.6 Compiling the `in_all_configurations` form.

assignment from figure 6.2. Note that **thetaConfig/3** is defined using three clauses. As we saw in section 6.2.3, the type of CF ($\{\overline{\mathtt{N}}, \mathtt{NP}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}, \mathtt{IP}\}$) was determined to be the union of $\{\overline{\mathtt{N}}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}\}$, $\{\mathtt{NP}\}$ and $\{\mathtt{IP}\}$ — the corresponding types for each clause. By unfolding the definition of **thetaConfig/3** first, we can apply the same optimization trick. Because the clauses are independent (i.e. the types do not overlap), the result will be the following three specialized predicates:

```
thRoleAssign1(CF) :-
        thetaConfig1(CF,Roles,Els)
        -> assignRoles(Roles,Els)
        ;  true.
thRoleAssign2(CF) :-
        thetaConfig2(CF,Roles,Els)
        -> assignRoles(Roles,Els)
        ;  true.
thRoleAssign3(CF) :-
        thetaConfig3(CF,Roles,Els)
        -> assignRoles(Roles,Els)
        ;  true.
```

where each thetaConfig$i$/3 is a predicate with just one clause; namely, the ith clause of the original thetaConfig/3 predicate.

2. Specialized predicates are shared between type members that generate the same sequence of pre-conditions (Step 5).

Compiling the compositional_cases_on form

The corresponding compilation procedure for compositionally defined principles has the same structure as the in_all_configurations form. In this situation, the conditions to be considered are simply all the base cases plus the single compositional case. (Because of the similarity to the previous form, we omit the detailed description of the procedure.) However, one important difference is that compositional definitions compute intermediate values for each phrase. For example, condition A of the Binding Theory (as described in chapter 3) maintains a set of unbound anaphors for each (sub-) phrase. Since the value for an aggregate phrase is computed as a combination of the values of its sub-constituents, the value computed for each constituent must be preserved for subsequent use. Hence, the expanded forms will contain extra goals for memoization.[7]

[7] The internally maintained values for each constituent are stored as features within the constituent itself.

6.3.2 Respecting Logical Dependencies

So far we have assumed a simple model of processing where interleaved principles are applied to phrases as soon as they are recovered. This "eager" model breaks down in the case where an interleaved principle depends on information that may not be available at the time it is first called. For example, consider the inherent Case assignment operation. Now, the system recovers phrase structure at the level of S-structure, but inherent Case is assigned at D-structure. A D-structure constituent recovery operation (discussed previously in section 4.3.2) is used to determine the appropriate constituent that occupies any given position at D-structure. This operation depends, of course, on the prior assignment of movement chains (by the Trace theory operation). Hence, the inherent Case assignment operation must wait on chain assignment.

As another example, consider the Case filter operation. Inherent and structural Case assignment have type {$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{A}}$,$\overline{\texttt{P}}$} and {`NP`, `VP`, `AP`, `PP`, `IP`, `CP`}, respectively. But, the Case Filter (which is logically dependent on both operations) has type {`NP`}. The interleaver cannot directly link the Case Filter to reduce `NP` $\rightarrow \alpha$ actions because Case may not have been assigned before the filter is invoked. For example, [$\bar{V}$ [V saw] [NP Mary]] is a configuration of structural Case assignment, but the Case filter must be delayed for *Mary* until the $\overline{\text{V}}$ constituent has been identified (and Case assigned).

The interleaver automatically reformulates principle definitions with such dependencies to use a simple constraint mechanism.[8] Constraints are associated with individual constituents — generally, those with category labels from the type of the principle. An interleaved principle may cause a constraint (i.e. an unevaluated goal) to be "posted" on any completed constituent. Constraints are posted with respect to a arbitrary signal. When another principle posts a signal S to a constituent with constraint C on signal S, it will trigger the execution of C. Of course, constraints and signal may also be posted in reverse order; that is, if constraint C on S is posted on a constituent that has had signal S raised, C will not be delayed, but immediately executed.

Now, returning to the previous two examples:

- The inherent Case assignment operation will post a constraint (i.e. delay the execution of) `assignInherentCase(`$Case$`,`X`)` for signal `chain` on the S-structure NP X at the position where inherent Case is to be assigned (at D-structure). When a chain involving X is completed (by the Trace theory operation), the signal `chain` will go out to all elements of the chain. Because computing the chain for X allows the D-structure constituent recovery proce-

[8] However, as mentioned in the previous chapter, the dependency relations are not yet automatically generated.

dure to determine the corresponding constituent at D-structure, the delayed goal `assignInherentCase(`*Case*`,`X`)` may now be executed. (Also, inherent Case assignment will, in turn, raise the signal `case` (for "Case assigned") on successful completion of the delayed goal.)

- In the second example, because the Case filter has type {`NP`}, the constraint `assignedCase(`X`)` on signal `case` will be attached to each NP X for which `lexicalNP(`X`)` holds. When X is assigned (either inherent or structural) Case, the delayed goal which completes the Case filter operation for X will be triggered.

To summarize, a given constituent may have a number of posted constraints on various signals.[9] A simple signal applied to a constituent may cause the evaluation of a single constraint or trigger a "cascade" of evaluations. Any untriggered constraints left over will be automatically triggered after the complete structure has been built. For example, *John* (which must satisfy `assignedCase(`*John*`)`) cannot be assigned Case by any element in *I am eager John to win.* Operations (e.g. structural Case assignment) which have no logical dependencies (other than the phrase structure recovery procedure) do not make use of the constraint mechanism, but are interleaved directly.[10] Finally, to complete our description, we will briefly mention two noteworthy features of this mechanism:

1. *Selective polling is maintained.*

   In contrast to schemes that satisfy constraints by propagating constraints posted at a particular node up to dominating nodes (and attempt to merge compatible constraints from sub-nodes whenever possible at each interior node), the present scheme only attempts to test a posted constraint when the necessary information (e.g. chains or case) has been computed.

2. *Constraint mechanism used only when required.*

   Since no general meta-interpretation is needed to implement this scheme, unnecessary overhead will be avoided for principles that can be directly interleaved.

[9] Operations that have multiple immediate dependencies are handled using the same scheme. For example, the Trace Case condition operation (which ensures that *wh*-traces, but not NP-traces, get Case) depends directly on both Case assignment (signal `case`) and chain formation (signal `chain`). The operation will post *two* mutually dependent constraints of the form `(signalled(`S_1`,`X`) ->` *applyCondition*`(`X`) ; true)` on signal S_2, where S_1 and S_2 are instantiated to `chain` and `case` (in turn). The `signalled(`S_1`,`X`)` goal ensures that the condition is not applied unless S_1 has also been signalled.

[10] However, directly interleaved operations may signal other operations. For example, the structural Case assignment operations signals `case` on each NP that it assigns Case to.

1. *EC Reduce for $V \rightarrow \lambda$.*

   Builds an empty category with label V.

2. *Adjunction Reduce for $V_i \rightarrow \alpha V_i \beta$.*

   Builds a larger category V_i with identical category label and features to V_i on the right side of the rule.

3. *Normal Reduce for $V^{j+1} \rightarrow \alpha V^j \beta$.*

   Projects V^j up to the next bar-level.

Figure 6.7 The reduce actions of the LR machine.

```
thetaConfig(NP,RoleList,Arguments) :- % [NP Spec [N1 N ..]]
     cat(NP,np),
     NP has_constituent SpecOfNP,
     cat(SpecOfNP,np),
     NP has_feature grid([Role],_),
     optionalThetaRole(Role,SpecOfNP,RoleList,Arguments).
```

Figure 6.8 A case of θ-configuration.

6.3.3 Restrictions: An Extension to Type Inference

We have seen how type information allows the interleaver to *safely* restrict the categories of constituents that need to be polled for a given interleaved principle. As described earlier in section 6.3.1, each element V of a type maps onto the reduce actions in the extended-LR table for rules of the form $V \rightarrow \alpha$. By further distinguishing the various kinds of reduce actions, it is possible to implement a more "finely-grained" partitioning of the underlying LR table, and thus further improve the selectivity of the interleaving mechanism. In this section, we will introduce the notion of a *restriction set* that may contain elements of the form `ec` (i.e. restricted to empty categories (ECs) only), `adjS` (i.e. restricted to adjunction structures), `head` (i.e. must be at the level of a head projection), and $\neg R$ (i.e. must not be R) — where R is one of the primitive restrictions. In general, a principle will have an associated restriction set in the same way that it has a given "type".

In the current system, the extended-LR machine distinguishes three different