```
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.

can eliminate useless polling of all empty categories (irrespective of category label). Consider again the definition of θ-role-assignment — in particular, the second case (repeated in figure 6.8). This clause covers the case of a noun (optionally) assigning its external θ-role to its NP specifier. Since the type of this configuration is {`NP`}, only reduce NP actions need call this configuration. However, note that all phrases forwarded by the EC Reduce action for NP can never satisfy this configuration. For instance, in order for the second goal (`NP has_constituent SpecOfNP`) to hold, the constituent represented by the variable `NP` cannot be empty. In the current system, the restriction to non-empty categories is encoded by associating a set {¬`ec`} with the first argument of the primitive operation `has_constituent`. This restriction suffice to prevent this instance of θ-role-assignment from being called every time the parser hypothesizes an empty NP.

In general, restriction sets will be defined for the arguments of linguistic primitives — just as for regular types. Similarly, propagation rules can be defined for the usual logical connectives. (We will omit the description of the corresponding restriction rules in this chapter. The interested reader is referred to the merged (type and restriction) definitions given in appendix B.) The only interesting difference occurs for conjunction (and disjunction) where restrictions are combined using set union (resp. intersection), rather than using intersection (resp. union) as in the case of types. Because of the similarity between types and restrictions, the same inference procedure is used for both.

6.4 Results and Conclusions

In this section, we will discuss the performance of the type inference mechanism on the principles that apply at S-structure. We will also correlate type cardinality with parsing times. We will explain the correlation by reference to the interaction between the degree of error detection in the phrase structure recovery procedure

Parser operation P	*Type(P)*	$\|Type(P)\|$	*Restrictions*
Theta-role assignment	$\{\overline{\texttt{N}}$, NP, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, IP$\}$	6	$\{\neg$ec$\}$
Theta criterion	$\{$NP, CP$\}$	2	$\{\}$
D-structure theta condition	$\{\overline{\texttt{N}}$, NP, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, IP$\}$	6	$\{\neg$ec$\}$
Inherent Case assignment	$\{\overline{\texttt{N}}$, NP, $\overline{\texttt{A}}$, $\overline{\texttt{P}}\}$	4	$\{\neg$ec$\}$
Structural Case assignment	$\{NP$, VP, PP, AP, IP, CP$\}$	6	$\{\neg$ec$\}$
$\overline{\text{S}}$-*Deletion*	$\{$VP, AP$\}$	2	$\{\neg$AdjS, $\neg$ec$\}$
Case filter	$\{$NP$\}$	1	$\{\neg$ec$\}$
Trace Case condition	$\{$NP$\}$	1	$\{$ec$\}$
Wh-*in-syntax*	$\{$CP$\}$	1	$\{\}$
Coindex subject and Infl	$\{$IP$\}$	1	$\{\neg$ec$\}$
License syntactic adjuncts	$\{$VP, NP$\}$	2	$\{$adjS$\}$
Trace theory	$\{$NP, Adv$\} \cup U$	20	$\{\}$
Subjacency	$\{$NP$\} \cup U$	20	$\{\}$
Free indexation	$\{$NP$\} \cup U$	20	$\{\}$
Functional determination	$\{$NP$\} \cup U$	20	$\{$ec$\}$
Binding condition A	$\{$NP$\} \cup U$	20	$\{\}$
Binding condition B	$\{$NP$\} \cup U$	20	$\{\}$
Binding condition C	$\{$NP$\} \cup U$	20	$\{\}$
Control	$\{$NP$\} \cup U$	20	$\{$ec$\}$
ECP	U	20	$\{$ec$\}$

Figure 6.9 Computed types for parser operations at S-structure.

and principle interleaving. More precisely, we will conclude that the effectiveness of interleaving is dependent on the amount of "garden-pathing" done by the LR(1)-based phrase structure recovery mechanism.

6.4.1 Type Computation Results

Figure 6.9 illustrates the type, the associated type cardinality and restriction set computed by the inference procedure for each principle. The procedure is powerful enough to come up with the minimal type for almost all of the twenty or so parser operations that operate at S-structure in the current system.[12] Compositionally defined principles will incrementally compute values for each constituent of phrase structure, and therefore, will have the universal type U.

[12] Due to an (artificial) limitation of the multi-pass algorithm, the procedure fails to take into account the set of possible structural Case assigners, thereby failing to compute the minimal type for structural Case assignment (NP, PP and AP being extraneous values). Nevertheless, soundness is preserved. Note also that (unlike D-structure operations) no LF operations can be interleaved in the current system.

We conclude that automatic type inference is practical in the context of the current system because the principle definitions make considerable use of the linguistic primitives for which type definitions exist. In general, we can probably rely on the grammar writer to make use of the supplied primitives. Hence, the type computation process usually has access to enough type information so that the universal type may be avoided, despite the conservative handling of negation and recursive definitions. Moreover, since the language used to represent the grammar is a fairly simple one, we have avoided many complexities for type inference present in general programming languages.[13] These two factors conspire to make the type computation problem simple enough to be practical for use in principle-based systems.

6.4.2 Principle Interleaving and Parsing Efficiency

Given the interleaving model described in the previous section, how well does principle interleaving perform in practice? Figure 6.10 illustrates the typical effect of interleaving on parsing performance for a series of six parsers. Each parser (numbered $i + 1$) builds on the preceding parser (numbered i) by adding one or two interleaved operations. (The actual parser specifications are shown separately in figure 6.11.) The fifteen test sentences are the same examples taken from Lasnik & Uriagereka that were used to test different principle orderings in chapter 5 (see figure 5.2). As the graph clearly shows, the general trend is that parsing times will increase dramatically as more principles are interleaved. Observe also that:

- There is a large jump between the second and third configurations (`DirectOnly` and `Chains`).[14] The crucial difference between the two is that the operation responsible for chain formation (Trace theory), one of the major generators in the system, has been interleaved in the third configuration.
- Adding filters such as the Case filter and the Trace Case condition (e.g. configuration `CaseTheory`) does *not* result in a reduction in parsing time.
- Although it is difficult to tell from the scale of the graph, the second configuration (`DirectOnly`) almost always performs better than the original, non-interleaved configuration (`Default`) — on the average about 30% faster.

Explaining the results

Why should there be such a difference between interleaving and non-interleaving? After all, in both cases, the parser will make use of the exact same definitions (i.e.

---

[13] For example, there is no data type polymorphism or coercion. (See Milner [39].)

[14] Note also that the results for some sentences (i.e. (1)–(4)) are largely unaffected by the amount of interleaving. These examples are relatively simple sentences (i.e. containing no embedded clauses) with correspondingly simple phrase structure at S-structure.

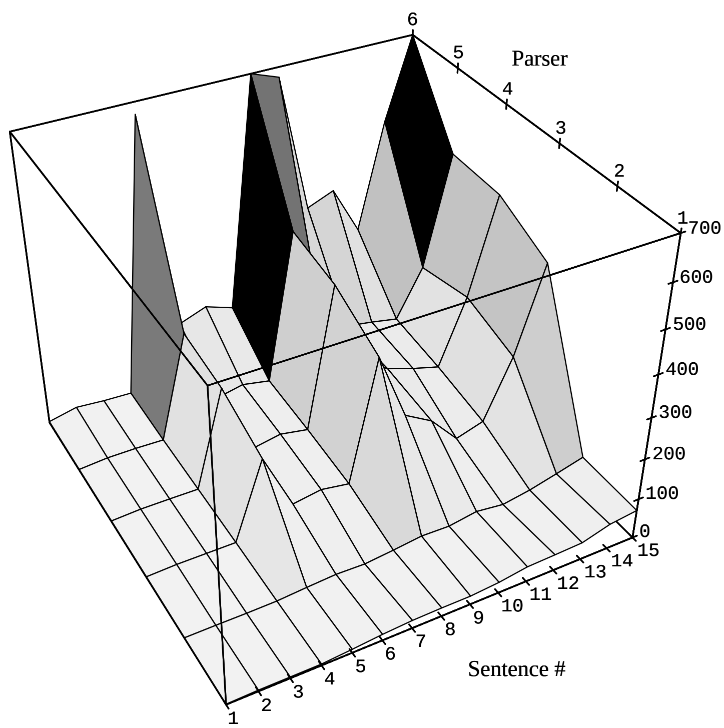

Figure 6.10 The effect of interleaving.

Parser Configuration	*Operations Interleaved*
1. `Default`	None
2. `DirectOnly`	*License syntactic adjuncts* + $\overline{\text{S}}$-*Deletion* + *Coindex subject & Infl*
3. `Chains`	= `DirectOnly` + *Trace theory* + *Inherent Case assignment*
4. `CaseAssignment`	= `Chains` + *Structural Case assignment*
5. `CaseFilter`	= `CaseAssignment` + *Case filter*
6. `CaseTheory`	= `CaseFilter` + *Trace Case condition*

Figure 6.11 Test parser configurations.

execute the same code) for each principle and return the same linguistic judgments. The only possible difference occurs in the underlying mechanism (transparent to the user) used to "hook up" the well-formedness conditions to phrase structure. Let us briefly review both mechanisms in the context of a typical condition "`q(X) if p(X)`":

1. *Non-interleaved processing*: A "tree-walker" is used to explore every node `X` in a structure, and apply `q(X)` to each node satisfying `p(X)`.

2. *Interleaved processing*: No tree-walker is used. Instead, `p(X)` is called directly for each node with category label belonging to the type of `X` (with respect to `p`). For each node `X` satisfying `p`, `q(X)` may be evaluated immediately or delayed pending some signal.

Note that at first glance, it seems that the second mechanism should have less overhead.[15] A number of efforts have been made to keep the overhead of interleaving as low as possible. The use of type inference cuts down on the number of times `p` has to be called (compared with tree-walking). Also, the constraint mechanism is relatively cheap since no polling is used.

The source of the large overhead lies in the *total* number of times `p(X)` (and `q(X)`) are called during parsing. Let us now review how the operation that builds

[15] One caveat: other details (e.g. `call(G)` *vs.* `G`) of the actual PROLOG implementation will also have some effect. But these overheads are relatively minor compared to the cost of `G` itself.

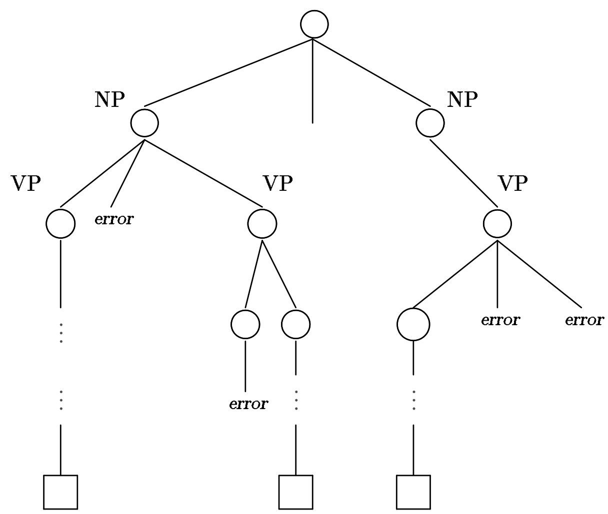

Figure 6.12 Search and error detection.

phrase structure works. Consider the search space diagram shown in figure 6.12. Basically, the parsing mechanism will proceed from LR state to LR state, reducing the current sentential string until it either succeeds in replacing it with a single sentential symbol, *or* it deduces (either with or without the one symbol lookahead) that there can be no derivation from the sentential symbol to the current sentential string. At each step, the parser can shift an input symbol or perform a reduce action, i.e. complete a non-terminal (e.g. an NP or VP). Because the S-structure rules do not form an LR(1) grammar, the parser cannot always determine the appropriate action to take at all times. Hence, it will end up trying several different actions for certain state/lookahead combinations — some of which will not lead to successful derivations. (See chapter 4 for a discussion of LR(1) conflicts for various S-structure phrase structure grammars.) Only at a later stage in the derivation, when the parser has read enough of the input to determine that the current path is incorrect (i.e. the parser has *garden-pathed*), will it automatically retrace its steps to explore alternative paths. (In the current system, the default behaviour is to find all possible parses — so the parser will explore the entire search space.) The crucial difference with respect to interleaving and non-interleaving is that, in the former case, any extra conditions (representing interleaved principles) that are attached to reduce actions will have been tried. If this occurs in a state on a garden path, then extra (wasted) work has been performed in comparison to a non-interleaved configuration. In other words, interleaving poses a large cost penalty for visiting states that do not lead to successful derivations (for the current input). In general, the more principles are interleaved, the larger the associated penalty. Although the addition of off-line type inference keeps the number of actions with attached conditions low, the experiments show that even in this case the penalty overwhelms the advantage of using principle violations to prune the search space, especially as the number of principles interleaved increases. Let us now re-examine the interleaving results:

1. As reported earlier, the second parser (`DirectOnly`) is about 30% faster on average than non-interleaving. In this case, only three operations of low type cardinality have been interleaved. (See figure 6.9 for type information.) Apparently, the penalty here is low enough to make interleaving cost effective.

2. The third parser (`Chains`) adds Trace theory which dramatically slows down the parser for two reasons: (1) it is compositionally defined (and therefore has a large type), thus adding a penalty for every phrase built, and more significantly: (2) it is a major generator, thus multiplying the branching factor in the search space.

3. Interleaving structural Case assignment (`CaseAssignment`), since it is not a filter (except in Case clash situations), results in a slightly slower parser. Next, adding the Case filter does not result in a slow down; in fact, it speeds up the parser slightly in a few cases. This suggests that the cost/penalty ratio for the Case filter is roughly even. Finally, adding a Case condition for *wh*- and NP traces results in another jump in parsing times. Note that this operation has the same type cardinality as the Case filter. The reason for the discrepancy is that the parser predicts many more empty NPs than overt NPs. In general, NPs can appear in a large number of different positions in phrase structure. Hence, a large number of empty NPs will be hypothesized during parsing. However, the need to first shift a noun in the input prevents the parser from doing the same with overt NPs. Note also that the use of restrictions means that the two filters are attached to different reduce actions.[16]

Prospects for principle interleaving

In this chapter, we have discussed and explained why it is generally not cost effective to interleave principles, even in this system where an attempt has been made to minimize overhead costs. However, experience with the current system indicates that it *is* cost effective to selectively interleave those principles that do not depend

[16] Unlike the condition on traces, the Case filter is not attached to *EC Reduce* (see figure 6.7). In fact, since the trace condition is only attached to *EC Reduce*, the two filters share no common attachment.

on, and thus do not require the prior interleaving of, generators. To make interleaving a considerably more attractive model, it seems that a large reduction in the amount of garden-pathing during phrase structure construction is necessary. The fact that the current system has employed a fairly efficient canonical LR(1)-based algorithm extended to include unbounded lookahead to further improve error detection suggests that the prospects for such an improvement is not good. As a final note, we should add that interleaving may be more effective in a parser that minimizes search by only looking for a single parse.

Chapter 7

Conclusions

This chapter discusses whether the current system meets its original goals and outlines possible directions for future work.

7.1 Remaining Problems and Limitations

In the preceding chapters, we have described how the parsing system represents linguistic principles and compiles those descriptions into forms that can be more efficiently processed. We have also described the (sometimes surprising) effects and tradeoffs from varying simple parameters of control. At this point, it is appropriate to review how the system measures up to the original goals and point out limitations and difficulties that were encountered.

- *Linguistic coverage.*

  The current set of principles will correctly account for almost all of the examples from the first four chapters in Lasnik & Uriagereka— plus a small set of additional examples taken from *Knowledge of Language* (Chomsky [12]) and the Japanese *wh*-questions from Lasnik & Saito [34]. This can probably be considered to represent broad coverage with respect to existing principle-based systems. It should be pointed out here that the problem of assembling a mutually compatible set of twenty-five or so principles turned out to be the most expensive portion of the project, both with respect to time and computational resources. The degree of interaction between the various principles and the need to import principles from sources other than Lasnik & Uriagereka were the major contributing reasons. In particular, the almost *chaotic* nature of the system (in the sense of being sensitive to perturbations in the definitions of principles) made it very difficult to maintain a monotonically increasing set of correctly analyzed examples.[1] Hence, substantial additions or revisions to the current theory will require much careful planning. Actually, the small size of the lexicon is probably the most significant barrier to having increased coverage. Currently, the lexicon contains exactly those entries required to handle the test cases and no more.

- *Transparency of representation.*

  One of the major goals for the representation was to try to minimize the cost of formalizing linguistic principles by pitching the level of definitions as closely as possible to the notation used by linguists whilst still being able to parse. Independence from control parameters and keeping as much of the underlying machinery as possible (e.g. the LR machine and type inference mechanism) hidden from the grammar writer were also important goals. Whilst it is not claimed that someone completely unfamiliar with the underlying system could successfully encode and debug principles, it is claimed that the majority of the definitions should be readily accessible to linguists (i.e. the semantic import of the definitions should be immediately obvious without knowing how the definitions are executed). Probably the most obvious defects in this area are the following:

  1. The grammar writer is insufficiently insulated from the default search strategy used by PROLOG. In general, one will have to worry about the order in which goals are processed, particularly when negation (as failure) is involved.
  2. The process of turning principle definitions into a parser has not been fully automated. In particular, the grammar writer will have to manually identify any logical dependencies with respect to the existing base of principles. In general, this is necessary for both the simple principle ordering and interleaving models. Moreover, structural cues for dynamic ordering are not automatically identified either. However, in either case there seems to be no foreseeable reason why the process cannot be made automatic.
  3. Finally, as discussed in chapter 3, problems remain with principles that are not stated in a form that can be directly used for parsing. Free indexation (*"assign indices freely"*) and Trace theory are prime examples of this. Currently, the system has no built-in knowledge of how to go about computing the possible indices or chain assignments in a sensible manner. It is unreasonable to expect the grammar writer to write a com-

[1] Note that this same property can be considered to be a virtue in linguistic theory.

positional definition and to have to prove that the procedure correctly enumerates all and only the proper assignments on a case-by-case basis.

Nevertheless, it should be mentioned that the relative ease of formalizing principles and the independence of the resulting definitions from the underlying machinery played a large role in keeping the cost of debugging the system low.

- *Parsing efficiency.*

  The modified LR machine, the conjunct (i.e. principle) ordering mechanism and the type inference procedure are all examples of well-known methods that together make it possible to obtain results in a reasonable amount of time. Even with these optimizations, the system can still take anything from around a hundredth of a second to about five minutes to search for all possible parses on a single example from Lasnik & Uriagereka.[2] Moreover, it seems that the amount of work required in some cases can be relatively unpredictable (i.e. governed by theory-internal artifacts rather than, say, sentence length or similarity of input). These figures are reasonable for use as a research tool, but not for applications such as text retrieval or understanding. Clearly, much more work is required before the current system can be deemed truly practical.

7.2 Future Work

Future extensions to the system will be in two main directions: (1) towards a tool to aid in testing linguistic theories, and (2) towards a practical system by improving computational efficiency. Also, expanding the coverage and sophistication of the basic system is another obvious place to start. The extension to handle the Japanese *wh*-questions from Lasnik & Saito with only a small modification in existing principles is a recent addition to the system. Further developments along these lines, e.g. to handle phenomena such as Japanese passive constructions at the same time as moving to VP-internal subjects, are also planned. Note that the current system implements just one of many possible theories in the principles-and-parameters framework. Supporting other theories might help to raise the level of abstraction or point out inadequacies in the current representation language. One obvious starting point would be to replace specific sub-theories of grammar. For instance, chapter 5 of Lasnik & Uriagereka discusses alternatives to the implemented Binding theory

[2] Times mentioned are for a Quintus PROLOG/sparcstation platform. The median time taken for configuration `Default` (introduced in chapter 5) with respect to all the examples in the first four chapters of Lasnik & Uriagereka is 4.38 (s).

that seem largely compatible with other existing sub-theories; namely, Aoun's Generalized Binding theory (which also replaces the ECP) and Higginbotham's Linking theory.

A test tool

Ideally, the system should be usable as a tool to help verify or develop linguistic theories, i.e. as the linguist's equivalent of a "hand calculator" that saves the linguist, say, from having to manually verify that some particular structure does not violate some unforeseen principle, or that there is no unlikely structure or derivation sequence that will not be blocked. The availability of a relatively fast and exhaustive (constructive) "proof checker" may also encourage the linguist to explore the ramifications of subtle perturbations in the theory at a relatively low cost. However, the current system requires further development to support debugging at a level that is linguistically-relevant. For example, the current system supports tracing of control and candidate structures down to the level of individual principles, but not within. To find out what is happening within a principle (e.g. if a particular trace is being lexically or antecedent governed in the ECP), one has to drop down into the underlying PROLOG system. Similarly, if there is a problem with the phrase structure recovery procedure (e.g. parser attempts to construct infinite structures), debugging must be performed at the level of LR(1) state sets, rather than $\overline{\text{X}}$-rules. Clearly, it is unreasonable and a violation of transparency to expect the grammar writer to perform anything other than "source level" debugging.

Automatic control configuration for efficiency

The present system will automatically construct a parser from a given control specification. It should be possible to take this one step further. That is, given a set of principles, the user should have the option of allowing the system to choose an appropriate (i.e. efficient) control configuration for that particular set. Automatically tailoring the control structure to fit principles would be another step in raising the level of abstraction of the system. It is a relatively simple matter to construct such a decision procedure. In the current model, we have that type cardinality is a good indicator of whether interleaving is appropriate for a given principle. For principles that have been determined to be unsuitable candidates for interleaving, a determinacy detector can be used to distinguish filters from possible generators, and thus assign an appropriate ordering.[3] Static measures of fan-out could also be

[3] For most classes of logic programs, there only exist sound but incomplete determinacy detectors. That is, such procedures correctly detect many cases of determinacy, but will miss others. It would be interesting to see if determinacy detection (like type inference) will work well in the restricted domain of linguistic principles.

used to prioritize generators.

Towards the generation of efficient principle-based parsers

It is probably safe to say that the current system is much less efficient than an equivalent "hand-compiled" system. The problem lies in the weakness of the compilation procedures used. For example, in chapter 2 we discussed how Correa used information from a variety of different principles to produce more deterministic (and efficient) versions of chain formation and functional determination operations. Except for the possibility of interleaving, the current system (unnecessarily) preserves a strict separation between parser operations. For example, even if free indexation is interleaved with all of the Binding conditions, all indexing possibilities will still be produced (although the ill-formed choices will be eliminated much earlier than in the corresponding non-interleaved case). One method to avoid this might be to use partial evaluation on the conjunction of principle definitions to eliminate inappropriate indexing choice points *off-line*.[4] Another possibility would be to use a constraint propagation approach, provided, of course, that transparency of principle definitions is preserved and overhead is kept low for other principles.

[4] One problem that must be solved is the control of the partial evaluation process (e.g. selection of the goal to evaluate and when to stop useless evaluation). In many cases, such information is communicated to the partial evaluator using program annotations — which would be incompatible with the goal of maintaining transparency.

Bibliography

[1] Abney, S. & J. Cole. "A Government and Binding Parser." *Proceedings of NELS 15.* 1985.

[2] Abney, S. "Licensing and Parsing." *Proceedings of NELS 17. GLSA.* UMass/Amherst. 1987.

[3] M. Abramowitz & I.A. Stegun, *Handbook of Mathematical Functions.* 1965. Dover.

[4] Barton, G.E., Jr. *"Towards a Principle-Based Parser."* MIT A.I. Memo 788. July 1984.

[5] Berge, C., *Principles of Combinatorics.* 1971. Academic Press.

[6] Chomsky, N.A., *Lectures on Government and Binding: The Pisa Lectures.* 1981. Foris Publications.

[7] Aho, A.V. & J.D. Ullman. *"Principles of Compiler Design."* Addison-Wesley. 1977.

[8] Baltin, M.R., and A.S. Kroch eds. *"Alternative Conceptions of Phrase Structure."* The University of Chicago Press. 1989.

[9] Chomsky, N.A. *"Lectures on Government and Binding."* Foris Publications. 1981.

[10] Chomsky, N.A. *"Some Concepts and Consequences of the Theory of Government and Binding."* MIT Press. 1982.

[11] Chomsky, N.A. *"Barriers."* M.I.T. Press. 1986.

[12] Chomsky, N.A. *"Knowledge of Language: Its Nature, Origin, and Use."* Prager. 1986.

[13] Chomsky, N.A. "Linguistics and Adjacent Fields: the State of the Art. A personal view." Israel *(m.s.)* 1988.

[14] Chomsky, N.A. "Some Notes on Economy of Derivation and Representation." *MIT Working Papers in Linguistics.* Vol. 10. 1989.

[15] Correa, N. *"Syntactic Analysis of English with respect to Government-binding Grammar."* Ph.D. thesis. Syracuse University. 1988.

[16] Correa, N. "Empty Categories, Chain Binding, and Parsing." *The MIT Parsing Volume 1987–88.* Center for Cognitive Science. MIT.

[17] Crocker, M.W. *"A Principle-Based System for Syntactic Analysis," (m.s.)* 1989.

[18] Dennis J.B. "Modularity." *Software Engineering.* Springer-Verlag. 1973.

[19] de Marcken, C.G. "Parsing the LOB Corpus." *Proceedings of the 28th Meeting of the ACL.* 1990.

[20] Dorr, B.J. "UNITRAN: A Principle-Based Approach to Machine Translation." S.M. thesis. MIT Department of Electrical Engineering and Computer Science.

[21] Fischer, C. & R.J. LeBlanc, Jr. *"Crafting a Compiler."* Benjamin/Cummings. 1988.

[22] Fodor, J.A. *"The Modularity of Mind."* MIT Press. 1983.

[23] Fong, S. & R.C. Berwick "The Computational Implementation of Principle-Based Parsers," *International Workshop on Parsing Technologies.* Carnegie Mellon University. 1989.

[24] Frank, R. "Licensing and Tree Adjoining Grammar in Government Binding Parsing." *Proceedings of the 28th Meeting of the ACL.* 1990.

[25] Fujisaki, T. *et al.* "A Probabilistic Parsing Method for Sentence Disambiguation." *Internation Workshop on Parsing Technologies.* Carnegie Mellon University. 1989.

[26] Fujita, H. "An Algorithm for Partial Evaluation with Constraints." ICOT TM-367.

[27] Graham S.L., *et. al.* "An Improved Context-Free Recognizer." *ACM Transactions on Programming Languages and Systems.* Vol. 2, No. 3, July 1980.

[28] Hentenryck, P. van. *Constraint Satisfaction in Logic Programming.* MIT Press. 1989.

[29] Hoji, H. "Logical Form Constraints and Configurational Structures in Japanese." Ph.D thesis. Department of Linguistics. University of Washington. 1985.

[30] Hornstein, N. *"Logic as Grammar: An Approach to Meaning in Natural Language."* MIT Press. 1984.

[31] Johnson, M. "Use of the Knowledge of Language," *Journal of Psycholinguistic Research.* 18(1). 1989.

[32] M.B. Kashket, *"A Government-Binding Based Parser for Warlpiri, a Free-Word Order Language."* TR-993, MIT A.I. Laboratory.

[33] Knuth, D.E., *The Art of Computer Programming: Volume 1 / Fundamental Algorithms.* 2nd Edition. 1973. Addison-Wesley.

[34] Lasnik, H. & M. Saito "On the Nature of Proper Government." *Linguistic Inquiry.* Vol. 15, Number 2. 1984.

[35] Lasnik, H. & J. Uriagereka *A Course in GB Syntax: Lectures on Binding and Empty Categories.* 1988. M.I.T. Press.

[36] Lloyd, J.W. *"Foundations of Logic Programming,"* 2nd Ed., Springer Verlag. 1987.

[37] Macías, B. *"Government-Binding Theory and Parsing as Deduction." (m.s.)* 1989.

[38] Makino, S. & M. Tsutsui. "A Dictionary of Basic Japanese Grammar." The Japan Times. 1986.

[39] Milner, R. "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences.* 17, 348–375. 1978.

[40] Naish, L. *Negation and Control in Prolog.* Springer-Verlag. 1985.

[41] Page, E.S., and L.B. Williams. *"An Introduction to Computational Combinatorics."* Cambridge University Press. 1979.

[42] Pager, D. "A Practical General Method for Constructing *LR(k)* Parsers." *Acta Informatica.* 7. 249–268. 1977.

[43] Petrick, S.R. *"A Recognition Procedure for Transformational Grammars."* Ph.D. thesis. Department of Modern Languages. 1965.

[44] Purdom, P.W., Jr. & C.A. Brown, *The Analysis of Algorithms.* 1985. CBS Publishing.

[45] Reingold, E.M., *et al.* *"Combinatorial Algorithms: Theory and Practice."* Prentice-Hall. 1977.

[46] Riemsdijk, H.C. van, and E. Williams. *"Introduction to the Theory of Grammar."* MIT Press. 1986.

[47] Riordan, J. *"An Introduction to Combinatorial Analysis."* John Wiley & Sons. 1958.

[48] Sharp, R.M. "A Model of Grammar Based on Principles of Government and Binding Theory." M.S. thesis. Department of Computer Science. University of British Columbia. 1985.

[49] Samuels, A.L., "Some Studies in Machine Learning Using the Game of Checkers. II — Recent Progress," *IBM Journal.* November 1967.

[50] Smith, D.E., & M.R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence* **26** (1985) 171–215.

[51] Sterling, L. & E. Shapiro, *"The Art of Prolog."* MIT Press. 1986.

[52] Stabler, E.P., Jr. *"The Logical Approach to Syntax: Foundations, Specifications and Implementations of Theories of Government and Binding." (m.s.)* M.I.T. Press. 1989.

[53] Stabler, E.P., Jr. *"Avoid the Pedestrian's Paradox." (m.s.)* 1990.

[54] Takeuchi, A., and H. Fujita. "Competitive Partial Evaluation — Some Remaining Problems of Partial Evaluation." ICOT TR-361.

[55] Tamaki, H., and T. Sato. "Unfold/fold Transformations of Logic Programs." *Proceedings of the 2nd International Logic Programming Conference.* Uppsala University, Sweden. 1984.

[56] Tomita, M. *"Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems."* Kluwer. 1986.

[57] Webelhuth, G. "Syntactic Saturation Phenomena and the Modern Germanic Languages." Ph.D thesis. Dept. of Linguistics. University of Massachusetts. 1989.

[58] Wehrli, E. *"A Government-Binding Parser for French."* Working Paper No. 48. University of Geneva.

Appendix A

The Linguistic Theory

The following definitions constitute a complete description of the linguistic theory implemented in the current system. Most of the theory described is taken directly from Lasnik & Uriagereka [35]. We have attempted to supply references to the appropriate sources in cases where modifications and additions to the basic theory have been made.

A.1 Basic Definitions

In this section, we define basic notions such as arguments, c-command, chains, and government, that will be used in the definitions of the principles in the next section. Assume that the order of constituents is unimportant unless otherwise stated. (The definitions are given in alphabetical order.)

Definition 1 (Anaphor)

NPs are marked with the feature $\pm$A (Anaphoric), implemented as `a(+/-)`. The value of this feature is lexically determined for overt NPs. For empty NPs, this feature is instantiated by functional determination [Defn. A.2.6].

Definition 2 (A-position)

A-positions are potential θ-positions, that is, positions where θ-roles may be assigned. $\overline{\text{A}}$-positions are simply those positions that are not A-positions. The following positions are defined to be A-positions:

a. Complement positions of lexical heads corresponding to internal θ-grid roles.

b. Specifier of IP.

c. Specifier of NP, but only when occupied by a NP (not when occupied by a determiner).

The feature `apos` will be associated with all A-positions, i.e. if a constituent X occupies an A-position, then X `has_feature apos` will hold.

Definition 3 (Argument) [pg101,Chomsky [9]]

Arguments are the elements that bear a θ-role. The status of elements as arguments will be determined by lexical and positional features. Note that some elements considered to be arguments at D-structure may have non-argument status at other levels of representation, e.g. at S-structure and LF. The following overt elements are arguments:

a. Quantifiers in A-positions only.

b. *wh*-NPs in A-positions only (see [pg115,Chomsky [9]]).

c. All other overt NPs unless explicitly marked as non-arguments (lexical feature `nonarg`).

d. All clauses in A-positions.

The following empty elements are also arguments:

e. *pro* [Defn. 15].

f. *PRO* [Defn. 16].

g. Variables [Defn. 21].

h. Empty NPs in A-positions (at D-structure) from which empty operators (*Op* [Defn. 14]) are formed by $\overline{\mathrm{A}}$-movement.

Definition 4 (Binding)

An element A *binds* an element B (equivalently, B is *bound* by A) if:

a. A c-commands B, [Defn. 5], and

b. A and B are coindexed [Defn. 8].

c. A and B are not coindexed for agreement purposes, e.g. subject and Infl ([Defn. 66]).

An element is said to be *free* if it is not bound. An element A *A-binds* an element B if A binds B and A is in an A-position. Similarly, A $\overline{\mathrm{A}}$*-binds* B if A binds B and A is in an $\overline{\mathrm{A}}$-position.

Definition 5 (C-Command)

"First branching node" definition: In $[\alpha\ X \ldots Y\ ]$, where X and Y are immediate constituents of α:

a. X c-commands all constituents *dominated by* Y ([Defn. 10]).

b. If X has form $[\beta\ Z\ ]$, Z being the only immediate constituent of β, then case (a) applies with Z in place of X. Also case (b) applies recursively with Z in place of X.

(Case (b) is actually implemented by having non-branching structures automatically collapsed during structure building.)

Definition 6 (C-Command′) [pg63f.,Lasnik & Uriagereka]

"Accessible SUBJECT" ([Defn. 25]) extension to c-command:

a. X c-commands′ Y if X c-commands Y.

b. I(AGR) c-commands′ NP in $[_{IP}$ NP $[_{\bar{I}}\ [_I\ ] \ldots]]$, where I(AGR) denotes an inflection element with a non-vacuous AGR (agreement) component.

(AGR is implemented as a feature `agr(F)` where `F` encodes person, number and gender features (see section A.3.1). Hence I `has_feature agr(_)` must hold in the second case above.)

Definition 7 (Chains)

Chains are formed by movement [A.2.9]. A chain $\langle E_1, \ldots, E_n \rangle$, $n \geq 1$ is a sequence of elements where E_1 is the *head* of the chain, and E_n, the last element of the chain. Each *link* $\langle E_i, E_{i+1} \rangle$ of a chain represents a single instance of movement of the head E_1 from the position occupied by the trace E_{i+1} to the position occupied by E_i. E_i is known as the *antecedent* of E_{i+1}. All elements of a chain will be assigned the same index.[1] When $n > 1$, E_n is known as a *base trace*. Furthermore, for $n > 2$, elements $E_2, \ldots, E_{n-1}$ are known as *intermediate traces*.
(Chains are represented by having each element of a non-trivial chain contain a syntactic feature `chain(`X`,`T`)`, where X is the antecedent of the element (or `[]` if none exists) and T is one of `head`, `base` and `medial`.)

Definition 8 (Coindexation)

Two elements A and B are said to be coindexed only if they have both been assigned indices, and the values of the indices are identical.

[1] Note, it need not be the case that all elements with same index will be part of the same chain.

(Indices are represented using the syntactic feature `index(I)`, `I` being a logic variable. This feature is assigned by either movement, free indexation ([Defn. 4]) or coindexation of subject and inflection ([Defn. 66]).)

Definition 9 (Comp)

The specifier of CP, if one exists, and the head of CP, C, are collectively referred to as Comp.

Definition 10 (Dominates)

Dominates is the primitive structural relation that captures the notion of hierarchy:

a. Any constituent dominates itself.

b. Let S be an element with immediate constituents $S_1, \ldots, S_n$, $n \geq 1$. S dominates a consituent P if for some S_i, $1 \leq i \leq n$, S_i dominates P.

An element S is said to *properly dominate* an element P if S dominates P, and S and P are distinct elements.

Definition 11 (Government)

Goverment is a structural relation between two elements:

a. *Core government.*

Let X be the head of a maximal projection XP. Then X governs any element P that XP dominates, provided:

1. X is a governor [Defn. 12], and
2. P does not dominate, nor is it dominated by X, and
3. There is no maximal projection YP, distinct from XP, that is dominated by XP yet dominates P.

b. *Exceptional government I.*

C governs the NP specifier of its complement (IP), provided C has lexical feature `ecm`, e.g. *for* in English.

c. *Exceptional government II.*

A lexical head X governs the IP specifier of its clausal complement, provided one exists, if:

1. X has lexical feature `ecm(oblig/opt)`, and
2. the clausal complement is a non-finite clause.

Definition 12 (Governor)

The following categories are considered to be governors:

a. Lexical heads, i.e. N, V, A and P.

b. Inflection I(AGR) containing a non-vacuous AGR element. ([Defn. 6] describes how AGR is represented.)

Definition 13 (NP-trace) [pc,Lasnik]

An NP-trace is the trace of movement of an NP from an A-position to another A-position.

Definition 14 (Operator)

The status of overt elements as operators is determined by lexical features. The following elements are considered to be operators:

a. Quantifier NPs marked with lexical feature `op(+)`, e.g. *everyone* and *someone* in English. Such NPs will form operator-variable constructions at LF.

b. [$+wh$] NPs (marked with lexical feature `wh`), e.g.*who* in English.

c. [$+wh$] adverbs, e.g. *why* and *how* in English.

d. An empty NP in an $\overline{\mathrm{A}}$-position marked as an operator *Op*.

(This is implemented by a feature `ec(op)`. It is assumed that empty operators are only formed through movement.)

Definition 15 (*pro*)

pro is an empty NP with Binding features $[-A, +P]$. *pro* is stipulated to not appear in complement positions. (Also, the *pro*-Drop parameter controls its availability for Japanese but not for English.)

Definition 16 (*PRO*)

PRO is an empty NP with Binding features $[+A, +P]$.

Definition 17 (Pronominal)

NPs are marked with the feature ±`P` (Pronominal), implemented as `p(+/-)`. As with anaphors [Defn. 1], the value of the feature will be determined lexically for overt NPs, and functionally, for empty NPs.

Definition 18 (R-expression)

An R-expression is any NP (overt or non-overt) marked with Binding features $[-A, -P]$.

Definition 19 (Subject)

The following positions are subject positions:

1. NP specifier of IP, and

2. NP specifier of NP.

Definition 20 (θ-grids)

Thematic grids encode the selectional properties of lexical heads. That is, they specify the range of external and internal θ-roles that a lexical head may assign. In the following sections, θ-grids will be denoted by a tuple $(\langle\theta_1 \ldots \theta_m\rangle, \langle\theta'_1 \ldots \theta'_n\rangle)$, $n, m \geq 0$. The first element contains the external θ-roles, and the second, the internal θ-roles.

Definition 21 (Variable)

A variable is an empty NP that is locally $\overline{\text{A}}$-bound by an operator. It has Binding features $[-A, -P]$. Note:

1. A variable may or may not be a trace — for example, a parasitic gap.

2. Not all empty NPs with features $[-A, -P]$ are variables (see [Defn. 47] for details).

Definition 22 (Wh-trace) [pc,Lasnik]

A wh-trace is the trace of the movement of a NP with feature $[+wh]$ from an A-position to an $\overline{\text{A}}$-position. (In particular, this definition will exclude intermediate traces in Comp.)

A.2 The Modules of Grammar

In this section we define the principles that make up the modules of grammar. The principles are organized as shown in figure A.1.

A.2.1 Binding Theory

Binding Conditions impose restrictions on the coreference possibilities for anaphors, pronominals and R-expressions. In the current theory, PRO, defined as having Binding features $[+A, +P]$, is treated specially. In particular, PRO will be allowed to 'escape' both Binding conditions A and B by virtue of not having a *Governing Category* (GC).

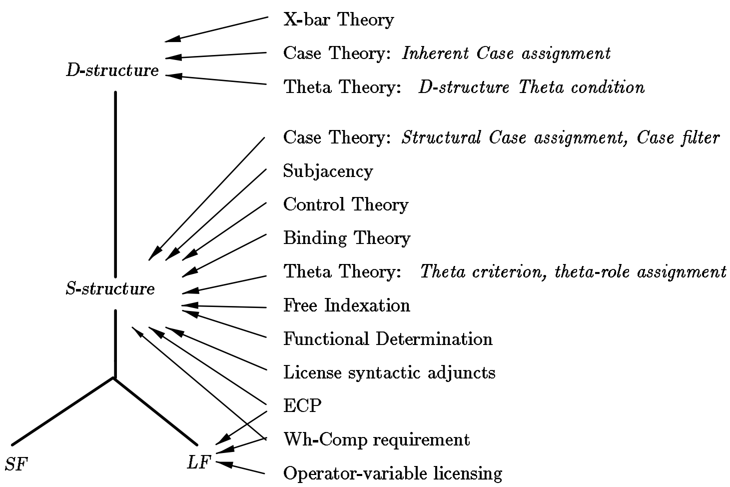

Figure A.1 The modules of grammar.

Definition 23 (Governing Category)

The governing category of an element A, GC(A), is the smallest *GC domain* containing:

a. A itself, and

b. an element G that *governs* A ([Defn. 11]), and

c. an element S such that S is an *accessible SUBJECT* to A ([Defn. 25]).

Definition 24 (GC Domain)

The GC domains are IP and NP.

Definition 25 (Accessible SUBJECT)

A constituent S is an accessible SUBJECT to A if:

a. S is a SUBJECT, and

b. S is accessible to A, that is:

1. S *c-commands'* A ([Defn. 6]), and
2. S is not coindexed with any category properly dominating A.

Definition 26 (SUBJECT)

An element X is a SUBJECT if:

1. X is a *subject* ([Defn. 19]), or

2. X is I(AGR) (see [Defn. 5]).

Definition 27 (Condition A)

All anaphors, i.e. NPs marked with feature `a(+)`, must be A-bound ([Defn. 4]) within its governing category ([Defn. 23]), if one exists.

Definition 28 (Condition B)

All pronominals, i.e. NPs marked with feature `p(+)`, must be A-free in its governing category, if one exists.

Definition 29 (Condition C) [pg98,Chomsky [12]]

Let X be an R-expression ([Defn. 18]). Then:

a. If X is part of a non-trivial chain, i.e. $X = E_i$ for some chain $\langle E_1, \ldots, E_n \rangle$, $1 \leq i \leq n$ and $n > 1$, then X must be *A-free* ([Defn. 4]) in the domain of the head of the chain E_1. Otherwise:

b. X must be A-free.

A.2.2 Control Theory

The following definitions deal with the cases in which *PRO* appears in a declarative clause complement or in a purposive adjunct clause, that is, in cases where the obligatoriness of local binding of *PRO* (wherever possible) holds. The definitions do not work for freer cases (e.g. interrogative clausal complements) where *PRO* is *optionally* controlled by the nearest appropriate binder.

Definition 30 (Control of *PRO*) [pg124,Chomsky [12]]

PRO must be controlled by the nearest *appropriate binder*, if one exists.

Definition 31 (Appropriate Binder) [pg124,Chomsky [12]]

To be an appropriate binder, an NP X must satisfy the following conditions:

a. X must *A-bind* *PRO* ([Defn. 4]).

b. X must not be lexically marked with feature `nonarg`, e.g. excludes the non-argument *it*.

c. X must not in a position associated with the feature `noControl`.

This definition deviates slightly from Chomsky's version. The last condition is designed to handle sentences like *John promised Mary PRO to leave* (cf. *John persuaded Mary PRO to leave*) where the nearest binder, *Mary*, cannot be the controller. Instead, the appropriate binder in this case is the next nearest binder *John*. This is accomplished by lexically marking the θ-role corresponding to *Mary* with the feature `noControl` (see section A.3.1).

A.2.3 Case Theory

The version of Case theory adopted is basically the "morphological" account of the Case Filter [pg9ff.,Lasnik & Uriagereka] — as opposed to an account based on the notion of 'visibility'. The only significant addition is for inherent Case assignment which occurs at D-structure (following Chomsky [12].[2])

Definition 32 (Structural Case assignment)

A head X assigns Case C at S-structure to an NP S under the following conditions:

a. X is a structural Case assigner that assigns Case C, and

b. X *governs* S ([Defn. 11]), and

c. X is adjacent to S if language parameter `CaseAdjacency` holds.

Note:

- Verbs only assign structural Case to direct objects. (See the description of inherent Case assignment below.)
- Elements marked as being optional Case assigners, i.e. having feature `ecm(opt)` (see section A.3.1), should have government operate consistently for structural Case assignment and Binding theory. For example, in *I want PRO to win*, *want* should not govern *PRO* — otherwise *PRO* will be Case-marked and also (incorrectly) have a governing category.

Definition 33 (Structural Case assigners)

The following elements are considered to be structural Case assigners:

a. I(AGR) assigns nominative Case.

b. V assigns accusative Case if:

1. V is not marked `noCasemark`, and

[2] Actually, Chomsky's approach to *of*-insertion is discussed very briefly in the context of illicit NP-movement in Lasnik & Uriagereka ([pg152]).

2. V is not marked as a dative Case assigner (in which case, it assigns dative Case to its direct object).

   For example, some verbs, e.g. in Japanese *nar* ('become') as in *Taro-wa naze kubi*-DAT *natta no* ((37b) in Lasnik & Saito [34]), seem to assign dative Case to their direct objects.

c. C with feature `ecm` assigns oblique Case, e.g. *for* in English.

Definition 34 (Inherent Case assignment) [pg192,KofL,Chomsky,86]

A head X assigns Inherent Case at D-structure to an NP S if:

a. X assigns a θ-role to S, and

b. X is an Inherent Case assigner.

The configurations of inherent Case assignment parallel those of θ-role assignment ([Defn. 51]).

Definition 35 (Inherent Case assigners) [pg193,KofL,Chomsky,86]

The following elements are the inherent Case assigners:

a. Prepositions assign oblique Case.

b. Nouns assign genitive Case.

c. Adjectives assign genitive Case.

d. Verbs assign genitive (or dative) Case for indirect objects only (roughly following [note 130, pg.219,Chomsky,86]).

Definition 36 (Genitive Case Realization) [pg194f,Chomsky [12]]

In English, genitive Case assigned to an NP S is realized according to the following rules:

a. If S is in a complement position, then genitive Case is realized as the semantically null marker *of*, otherwise:

b. If S is in a subject position, then genitive Case is realized as the possessive marker *'s*. Genitive Case may also be realized morphologically for pronouns, e.g. *me* + *'s* is equivalent to *my*.

Note, it is assumed that the markers *of* and *'s* do not form constituents at S-structure, but instead are inserted at Surface Form.

Definition 37 (Case Filter)

At S-structure, every lexical NP needs Case.

Definition 38 (Case Transmission)

The current implementation deviates from Lasnik & Uriagereka in requiring Case transmission to take place for A- to $\overline{\text{A}}$-movement for two reasons:

1. Because of the possibility of scrambing in Japanese, e.g. [*VP nani*-ACC$_i$ [*VP te*-DAT [*NP-t*]$_i$ *ireta*]] as in *Taro-ga nani-o te-ni ireta koto-o sonnani okotteru no* ((39a) in Lasnik & Saito [34]), it is assumed that accusative Case is transmitted from the trace to a VP-adjunct position (an $\overline{\text{A}}$-position according to [Defn. 2]) occupied by the scrambled direct object. Also, the current definition of government excludes the verb from governing VP-adjunct positions.

2. Without Case transmission, the Case filter will have to be modified to exclude operators in $\overline{\text{A}}$-positions, as in *who*$_j$ *did John see* t_j where it is assumed that the trace rather than the operator *who* requires Case.

However, the present definition is somewhat unsatisfactory since Case transmission overlaps the condition that *wh*-traces must have Case ([Defn. 39]).

Definition 39 (Trace Case Condition)

---

a. All Wh-traces must have Case.

b. No NP-trace can have Case.

A.2.4 Empty Category Principle (ECP)

The ECP holds both at S-structure and at LF.

Definition 40 (ECP)

All empty categories except for the empty complementizer C, empty operators and *PRO* must be properly governed. An empty category is properly governed if it is *lexically governed* or *antecedent governed.*

Definition 41 (Lexical Government)

An element S is lexically governed if there exists an element P such that:

a. P governs S [Defn. 11], and

b. P is a lexical governor.

Definition 42 (Lexical Governors)

The following elements are the lexical governors:

a. Lexical heads [Defn. 62].

b. I(AGR) if the *pro*-Drop parameter holds.

Definition 43 (Antecedent Government)

An element S is antecedent governed if there exists a local binder P such that there is no intervening *barrier*.

Definition 44 (Barriers for Antecedent Government)

The following categories are considered to be barriers to antecedent government:

a. All NPs.

b. All clauses (CP) unless:

1. Element S is in specifier of COMP — for antecedent government into COMP, or
2. Clause CP has been marked with feature `noBarrier`.

Here, we deviate slightly from Lasnik's account of the *That-trace* effects (sections 4.1.2–4.1.4 in the reference text). Lasnik assumes that the trace in an example like '*Who[1] do you think [that [t[1] saw Bill]]' is not properly governed because there is no trace (in COMP) that antecedent governs it. The assumption is that there cannot be both 'that' and a trace in COMP. Furthermore, it is assumed that this is not a Subjacency violation despite the absence of an intermediate trace. The apparent conflict between the absence of an intermediate trace for ECP and Subjacency is reconciled by having movement *optionally* leave a trace. However, in this implementation, we stipulate that movement always leaves traces, but the same judgements are achieved by having 'that' (as the head of COMP) mark the trace in the specifier position as being 'invisible' to ECP.

Definition 45 (Invisible traces)

A trace is invisible to ECP (it need not be properly governed, nor can it be a proper governor) if it is marked `invisible`.

A further modification to the above concerns head movement (see [A.2.9] for details.) The ECP also applies to all traces of head movement. Unfortunately, the above definition also rules out all cases of INFL lowering to V (e.g. when AGR is "weak", but the verb is "strong"). That is, the trace of INFL is not antecedent governed when INFL adjoins to V. Hence, the actual definition has an extra clause tailored to permit precisely this case.

A.2.5 Free Indexation

In this theory, elements do not intrinsically possess indices. Instead, indices are assigned through one of three mechanisms. Binding Theory relevant elements may receive indices through movement [A.2.9], or through the coindexation of subject of IP and I [Defn. 66]. Also, at S-structure, free indexation stipulates that:

Definition 46 (Indexation at S-structure)

Indices are freely assigned to all elements in A-positions only.

The A-position restriction prevents operators such as wh-words in COMP from receiving an index at S-structure by any means other than through movement. For example, the preferred analysis for the sentence 'who left' is 'who[i] t[i] left' where 'who' has moved from the subject position occupied by the trace. In the case where the A-position stipulation is eliminated, there is another analysis 'who[i] e[i] left', where 'who' and the empty element are both 'base-generated' in the positions shown, and then (freely) assigned identical indices at S-structure. With free indexing restricted to A-positions, this second analysis becomes 'who e[i] left' which will be ruled by Full Interpretation [A.2.7].

A.2.6 Determination of Empty Categories

Empty noun phrases initially are devoid of the two Binding theory-relevant features, $\pm$`A` and $\pm$`P`. These two features are instantiated according to the following algorithm:

Definition 47 (Functional Determination)

Let A be an empty NP in an A-position. Then:

1. Let B be the nearest binder of A (if one exists). Then:
   - (a) If B is in an A-position, then:
     A has feature `p(-)` if A and B do not have an independent θ-role, i.e. are part of the same chain. Otherwise, A has feature `p(+)`.
   - (b) If B is in an $\overline{\text{A}}$-position, then:
     A is a variable [Defn. 21] (with features `[a(-),p(-)]`) if B is an operator. Otherwise, A has feature `p(-)`.
2. Otherwise A must be unbound. Then A has feature `p(+)`.

Except for the case where A is a variable, the feature `a(+/-)` is freely assigned as follows:

a. A may have feature `a(+)`.

b. A may have feature `a(-)` if the *pro*-Drop parameter is set.

Notes:

1. Functional Determination is deterministic only when the *pro*-Drop parameter is not set (as in English, but not Japanese).

2. Functional Determination does not apply to intermediate traces and empty operators in Comp. These do not receive anaphoric and pronominal features.

A.2.7 Full Interpretation

Full Interpretation requires that every element of the interface to syntax, that is, the two levels of PF and LF, must receive an appropriate interpretation. The following principles constitute a partial implementation of Full Interpretation for licensing of LF representations only.

Definition 48 (License Operator-Variable Constructions)

In this theory, we do not adopt the Bijection Principle. In other words, operators and variables need not be in strict one-to-one correspondence. For example, in the case of parasitic gap constructions such as *which report*[i] *did you file* t[i] *without reading* e[i], the operator *which report* binds both the trace variable, and the parasitic gap e[i].

a. An operator [Defn. 14] must bind at least one variable [Defn. 21], and

b. A variable must be *strongly bound* by an operator, and

c. A variable that is also a trace must be bound by the operator that heads its chain.

Definition 49 (Strong Binding) [pg108ff,KofL,Chomsky,86]

A variable x is strongly bound if:

a. x is bound by an overt operator, otherwise:

b. if x is bound by an empty operator, then x's range must be determined by being also bound by an overt, non-pleonastic element.

Definition 50 (License Syntactic Adjuncts) [pg94f,van Riemsdijk & Williams,86]

Basically, all syntactic adjuncts must be one-place predicates. The predicate status of adjuncts may be either determined structurally or through lexical features. (In the actual implementation, this condition is enforced at S-structure.) The following adjuncts are licensed as one-place predicates at LF:

a. A clause of the form $\lambda x.(\ldots x \ldots)$ where λx is an operator and x a variable.

The operator may be realized at LF as an empty operator *Op* as in the restrictive relative clause construction [*NP* [*NP the man*] [*CP Op I saw*]], or as an overt operator such as *who* in [*NP* [*NP the man*] [*CP who I saw*]]. Furthermore, the operator and the element to which it is adjoined cannot have different category labels — cf. **The man how fast we ran.*

b. An adjective with one (unsaturated) external θ-role in its θ-grid — e.g. *the sad man* versus *the man is sad.*

c. A head with lexical feature `predicate(Type)` where `Type` is one of `location, manner, time`, etc. — e.g. the adverb *sincerely* and the PP headed by preposition *by* have type 'manner' and 'agentive', respectively.

The above types do not cover all the possible cases of adjunction. *Wh*-adverbs that adjoin to VP are excluded from this condition. Furthermore, adjunction resulting from head movement, VP adjunction from scrambling and LF movement are also all excluded.

A.2.8 θ-Theory

Definition 51 (θ-Assignment)

Modified from Chomsky's definition ([pg142,KofL,Chomsky,86]) to accommodate the three-bar level scheme used for verbs. The configurations of θ-role-assignment are:

a. [$\overline{\mathrm{X}}$ X YP]

A head X with θ-grid $(\ldots, \langle\theta_1 \ldots \theta_n\rangle)$ (n=1,2) assigns its internal role θ_1 to its complement (i.e. direct object) YP.

b. [$\overline{\overline{\mathrm{X}}}$ $\overline{\mathrm{X}}$ YP]

A head X with θ-grid $(\ldots, \langle\theta_1 \ldots \theta_2\rangle)$ assigns its internal role θ_2 to its complement (i.e. indirect object) YP.

c. [*NP* NP-SPEC [$\overline{\mathrm{N}}$ ... N ...]]

A head noun N with a θ-grid $(\langle\theta\rangle, \ldots)$ *optionally* assigns its external θ-role θ (provided one exists), to its specifier NP-SPEC.

d. [*IP* IP-SPEC [$\overline{\mathrm{I}}$ I VP]]

A clausal subject *IP-SPEC* may receive the external θ-role of a predicate, either directly from the VP, or by percolation from a lower predicate. For example, in [*IP John* [*VP is* [*AP happy*]]], the external θ-role of the adjective *happy* will be assigned to the subject *John*.

Definition 52 (θ-criterion) [pg42,LGB,Chomsky,81]

The θ-criterion at S-structure is formulated as a property of movement chains. Given a chain $\langle E_1, \ldots, E_n \rangle$, $n \geq 1$. Then:

a. E_n must be assigned a θ-role, if and only if, the chain is an A-chain.

b. E_i, $1 \leq i < n$, must not be assigned a θ-role.

Definition 53 (A-chain)

The definition of an A-chain, that is, a chain headed by an argument, must take into account whether the head is in an A-position or not, because the argumenthood of some elements hinge upon this distinction — see [Defn. 3]. The reference text is unclear on this point. However, it seems that the correct definition of an A-chain must consider the argument status of its head *at D-structure*, not at S-structure — otherwise, examples involving Wh-movement such as *who*[i] *t*[i] *left* will be incorrectly ruled out. Hence:

> A chain $\langle E_1, \ldots, E_n \rangle$, $n \geq 1$, is an A-chain if E_1 is an argument at D-structure.

Definition 54 (D-structure θ-Condition) [pg97f,KofL,Chomsky,86]

By the projection principle, the θ-criterion must hold at every syntactic level of representation. The θ-criterion as formulated in the previous section only applies at S-structure. The fact that the θ-criterion must also hold at D-structure is separately expressed as a uniformity condition with respect to θ-marking:

> At D-structure, if α T-governs X then X is an argument if and only if α θ-marks X.

Note: For example, [*CP someone*$_i$ [*IP* x_i [*VP likes everyone*]]] is a candidate S-structure generated by the parser. Here, the variable x_i has been coindexed with the quantifier *someone* not through movement but through free indexation. (The θ-criterion [Defn. 52] will fail to rule this out.) However, the corresponding D-structure will violate the uniformity condition. The null element that occupies the position of the variable at D-structure is θ-marked by the VP but it is not an argument.

Definition 55 (T-government) [pg98,KofL,Chomsky,86]

There are two configurations where T-government obtains:

a. [X^{i+1} X^i YP], $\quad i = 0, 1$

A head X T-governs its complement(s) YP.

b. A predicate T-governs its subject.

A.2.9 Trace Theory

Movement in Syntax

We assume that empty operators in constructions such as restrictive relative clauses, as in "the man [*Op* [I saw *e*]]," are base-generated in A-positions at D-structure and fronted by Move-α (see [pg84f, KofL, Chomsky, 86]).

Definition 56 (Chain Formation)

Movement chains are formed in accordance with the "history of movement" hypothesis. Furthermore, we assume that traces are obligatory for movement.

Head Movement

A very simple theory of head movement was implemented for two basic reasons: (1) to allow cases of *Do-support* where an auxiliary verb raises to COMP, and (2) to account for the particular ordering of verbal adjuncts. In this theory, languages are parameterized according to whether AGR is *strong* or *weak*. In English, AGR is weak, but for languages like French, AGR is considered to be strong. For simplicity, AGR and TNS are not considered to be distinct entities from INFL. We adopt the following restrictions:

a. Only inflection and verb heads are allowed to move.

b. Movement of an I–V (or V–I) complex is permitted to the head of Comp.

c. Movement is by adjunction, not substitution.

d. V moves in preference to I wherever possible.

The traces left by head movement must obey the ECP. However, the case where Infl must lower to V, e.g. for regular (non-auxiliary) verbs in English, is problematic for the ECP — see [A.2.4] for details.

Conditions on Syntactic Movement

Definition 57 (Subjacency)

Let E_i and E_{i+1} be a pair of elements from a chain $\langle \ldots E_i, E_{i+1}, \ldots \rangle$. Then E_i and E_{i+1} are subjacent if there exists an element S such that:

a. S dominates both E_i and E_{i+1}, and

b. there does not exists two distinct *bounding nodes*, B_1 and B_2, such that B_1 and B_2 both properly dominate E_{i+1} and are properly dominated by S.

For English, IP and NP are considered to be the bounding nodes.

Definition 58 (*Wh*-movement in Syntax) [Lasnik & Saito,84]

Languages are parameterized according to whether they allow *wh*-movement in syntax; that is, between D- and S-structure and D-structure. For example, English is +WhInSyntax, and Japanese as -WhInSyntax. If +WhInSyntax, then:

a. A [+*wh*] Comp must have a [+*wh*] head.

b. A [-*wh*] Comp must not have a [+*wh*] head.

In the current implementation, "head" refers to both the head and the specifier position of Comp (the target position for *wh*-movement).

LF Movement

LF movement is restricted to the processes of Quantifier Raising (QR) and LF *wh*-movement.

Definition 59 (Quantifier Raising)

Quantifiers marked with lexical feature `op(+)` are freely raised to adjoin to the nearest IP.

Note: unlike LF *wh*-movement, QR is a local operation with no successive cyclic movement allowed.

Definition 60 (LF *Wh*-movement)

Elements marked with lexical feature `wh` are freely raised to Comp.

Movement is always accomplished by substitution unless the position in Comp is already occupied, in which case, moved elements are adjoined.

Conditions on LF Movement

The ECP and Operator–variable licensing operations are applied at LF. Note that Subjacency does not apply to LF *wh*-movement. The *Wh*-movement in Syntax filter also applies, but with one modification:

Definition 61 (*Wh*-Comp Requirement) [Lasnik & Saito,84]

Universally:

a. A [+*wh*] Comp must have a [+*wh*] head.

b. A [-*wh*] Comp must not have a [+*wh*] head.

There is an extra semantic parallelism requirement for the [+*wh*] head case to handle examples such as **Mary-wa John-ga nani-o katta ka do ka shiranai*, for which *ka do ka* ('whether or not') — a yes/no question element — and *nani* (meaning 'what') cannot both be present in the same Comp at LF.

A.2.10 $\overline{\text{X}}$-Theory

Definition 62 (Projection Levels)

In this theory we adopt a two-bar-level system for all categories except verbs (which makes use of a three-bar-level system). Heads constitute the zeroth projection level. N, V, A, and P are the *lexical heads*. I (Inflection) and C (Complementizer) are the *non-lexical* heads. *Intermediate projections* (or single-bar categories) are the first level projections of heads. Double-bar categories are *maximal projections* for N, A, P, I, and C, but still an intermediate projection for V. $\overline{\overline{\overline{\text{V}}}}$ is the maximal projection for V.

Definition 63 (Complements)

In general, complements of heads are attached as sisters of heads, and as sisters of intermediate projections *not* immediately dominated by maximal projections. The complement structure for lexical heads is primarily determined by individual θ-grid properties. For verbs, direct and indirect objects are attached to the $\overline{\text{V}}$ and $\overline{\overline{\text{V}}}$ projections, respectively. For non-lexical heads I and C, the possible complements are not determined by lexical properties, but simply stipulated as VP and IP, respectively.

Definition 64 (Specifiers)

Specifiers are attached as sisters of intermediate projections immediately dominated by a maximal projection. At most one specifier is permitted. The specifier position for I and C may be filled by NPs.[3] The specifier position of NP can be filled by an NP or a determiner. The specifier positions for V, P and A are currently left empty.

A.2.11 Miscellaneous Conditions

Definition 65 ($\overline{\text{S}}$-Deletion)

Heads (e.g. *believe* and *likely*) that normally take clausal complements (CPs), but are lexically marked as being $\overline{\text{S}}$-deleters will take IPs as complements instead.

In the current implementation, no deletion actually takes place. Instead, the CP node is marked as being "transparent" for conditions such as the ECP.

Definition 66 (Coindex subject of IP and Inflection)

$[_{IP}\ \text{NP}_i\ \ [_{\bar{I}}\ \ [_I\ ]_i \ldots\ ]]$

There are two reasons for coindexing the subject of IP and inflection:

[3] *Wh*-adverbs can also move to the specifier of Comp as the target site of *wh*-movement.

1. Subject–verb agreement in conjunction with head movement.
2. The determination of accessible SUBJECTS [Defn. 25].

A.3 Features

This section describes the lexical and syntactic features used in the current system.

A.3.1 Lexical Features

This section briefly lists the features used in lexical entries together with various examples of their use. The following table contains the basic lexical features:

Feature	Categories	Example	Explanation
`a(±)`	N	*himself*: `a(+)`	[±anaphoric]
`agr([P,N,G])`	N,det,V	*few*: `agr([3,pl,[]])`	Encodes person, number and gender features. Boolean combinations can be built using `not(_)` and lists (to represent disjunction). `[]` encodes a *don't care*, i.e. irrelevant, feature.
`count(+/-/_)`	det, N	*much*: `count(-)`	Used to associate determiners with particular classes of nouns.
`ecm(oblig/opt)`	V, A	*believe*: `ecm(oblig)`	Obligatory and optional Exceptional Case Markers.
`grid(ERs,IRs)`	N,V,A,P	*kau* ('buy'): `grid([agent], [theme,[location]])`	`ERs` and `IRs` hold the list of external and internal θ-roles, respectively. Optional roles are bracketed.
`left(C,F)`	mrkr	*'s* (possessive marker): `left(n,case(gen))`	A marker is an input element that becomes a feature `F` on the closest item of category `C` to its left.
`morph(R,F)`	V	*eaten*: `morph(eat,ed(2))`	Encodes the root `R` and particular form `F` of an individual entry.
`morphC(Case)`	N	*his*: `morphC(gen)`	Used to indicate morphologically realized Case on noun forms.
`noCasemark`	V	*seem*: `noCasemark`	Does not Case-mark its complement.
`noControl`	V	*promise*: `grid([agent], [patient$noControl, proposition])`	Indicates the unavailability of a θ-grid element as a controller for *PRO*.
`nonarg`	N	*it*: `nonarg`	Indicates the non-argument status of a noun.
`p(±)`	N	*him*: `p(+)`	[±pronominal]
`right(C,F)`	mrkr	*of* (mrkr): `right(n, case(gen))`	Similar to `left(C,F)`. Indicates that a marker becomes feature `F` on the nearest item of category `C` to its right.
`subcat(C$F`$_1$`,F`$_2$`)`	V(aux)	*be* (passive): `subcat( vp$[morph(_,ed(2))], [noCasemark])`	Subcategorization: `C` is the category of the complement which must have features `F`$_1$. `F`$_2$ are the features to be added to the complement.
`wh`	adv, N	*why* (adv): `wh`	[+wh]

More complex features can be used to construct arbitrary conditions from the basic features listed above. The following table lists the feature prototypes that are used to express various idiosyncratic or peripheral properties of lexical items not yet properly handled by the conventional set of principles.

Feature	Categories	Example	Explanation
`selR(R)`	all heads	*that* (C): `selR( not( feature(inf(_))))`	An escape mechanism that allows arbitrary restrictions to be placed on the complement of a head. `R` may be constructed using the forms `feature(F)` (complement must have feature `F`), `addFeature(F)` (add feature F to the complement), `goal(G,X)` (complement (named by `X`) must satisfy a goal `G`, and `not(R)` (negation) in conjunction with any of the previous forms.
`specR(R)`	all heads	*that* (C): `specR( addFeature(invisible))`	The corresponding mechanism for specifiers of heads.
`adjR(Rs)`	all heads	*by* (P): `adjR([goal( extRoleToAssign(XP, agent), XP), feature(passive)])`	Similar to `selR(R)` and `specR(R)`. Allows a sequence of conditions (`Rs`) to be imposed on adjuncts.

APPENDIX B

The Notation for Encoding Principles

This chapter defines the various primitives and definitions that form the "programming language" that the system provides for the grammar writer. Type and restriction set definitions are also supplied where relevant.

B.1 Primitive Operations

In this section, we define the basic operations provided for manipulating constituent structure such as the extraction of complements and heads, category labelling, and feature set maintenance, e.g. membership, addition of new features, and chain link traversal. Many of these are "one-way" or non-reversible in the sense that they have restricted modes of operation.[1] For example, `complement_of` can only be used to extract complements but not to return the corresponding head (or projections of the head). Note that this does not constitute an exhaustive list of primitives; in particular, there are also many more basic operations deal with set and open list manipulation.

Definition 67 (`addFeature`)

Given a constituent X and a feature F, `addFeature(`F`,`X`)` adds F to the feature set of X.

[1] This is a side-effect of the simple term-based representation used for phrase structure. See chapter 3 for the details.

Definition 68 (`adjoin`)

Given two constituents X and Y, and a parameter `Dir` $\in$ {`left`,`right`}, `adjoin(`X`,`Y`,`Z`,Dir)` holds when Z is the constituent formed by adjoining X to Y according to `Dir`. (Can also be used to extract sub-phrases of an adjunction structure.)

Definition 69 (`adjoined`)

Given two constituents X and Y, `adjoined(`X`,`Y`)` holds if Y is an immediate adjunct under X. Similarly, `adjoined(`X`,`Y`,`Z`)` holds if Y has been adjoined to Z forming the adjunction structure X.

Definition 70 (`antecedent`)

Chain link chasing: given a trace E_{i+1}, `antecedent(`E_{i+1}`,`E_i`)` holds if there exists a non-trivial chain $\langle \ldots, E_i, \ldots, E_{i+1}, \ldots \rangle, \quad 1 \leq i < i+1 \leq n, n > 1$, where n is the number of elements in the chain.

Definition 71 (`baseTrace`)

Given a constituent E_n, `baseTrace(`E_n`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_n \rangle, \quad n > 1$. The corresponding type definition is:

> `baseTrace(`$E_n : T : R$`)` holds $\Rightarrow T = U, R = \{\text{ec}\}$, i.e. E_n must be an empty category.

Definition 72 (`cat`)

Category labelling: given a constituent X and a label c, `cat(`X`,`c`)` holds if c is the category label of X. (Note, this may be used for both testing and setting the label of a constituent.) There are two type definitions corresponding to the cases where c is instantiated or not:

> `cat(`$X : T : R$`,`c`)` holds $\Rightarrow$
> $T = \{c\}, R = \emptyset$ if c is a category label, or
> $T = \text{typeValue}(c)$ if c is uninstantiated.

In the latter case, T is determined by direct constraints on c in the definition of the principle in which the goal `cat(`X`,`c`)` occurs. For example, the value of c may be constrained through $\overline{\text{X}}$-theory using a goal such as `head(`c`)`.

Definition 73 (`coindex`)

Given two constituents X and Y, `coindex(`X`,`Y`)` first assigns indices to X or Y if they have not already been indexed. Then, it makes their indices identical with respect to `sameIndex` [Defn. 88].

Definition 74 (`complement_of`)

The following is a recursive definition for extracting complements from a non-head projection. Given a constituent X, Y **`complement_of`** X holds if:

a. Y is an immediate constituent of X st. Y is a complement of X according to $\overline{\mathrm{X}}$-theory if X immediately dominates its head, or

b. Y **`complement_of`** X^i where X^i is an immediate constituent of X st. X is a projection of X^i, if X does not immediately dominate its head.

The following type definition applies to the second argument:

Y **`complement_of`** $X:T:R$ holds $\Rightarrow T = U, R = \{\neg\mathrm{ec}, \neg\mathrm{head}\}$,
i.e. X cannot be an empty category or a head.

Definition 75 (`consFeature`)

Given a constituent X and a feature F, **`consFeature(`**F**`,`**X**`,`**X'**`)`** creates a new constituent X' identical to X — the only difference being the addition of F to the feature set of X'.

Definition 76 (`ec`)

Given a constituent X, **`ec(`**X**`)`** holds if X has no sub-constituents, i.e. X is an empty category. The type definition for X is given by:

`ec(`$X:T:R$`)` holds $\Rightarrow T = U, R = \{\mathrm{ec}\}$.

Definition 77 (`has_constituent`)

Designed to pick out an immediate constituent. X **`has_constituent`** Y holds if X immediately dominates the set of constituents C_X st. $Y \in C_X$.

The type of the first argument is described by the following definition:[2]

$X:T:R$ **`has_constituent`** Y holds $\Rightarrow T = U, R = \{\neg\mathrm{ec}\}$.

Definition 78 (`has_constituents`)

Given a constituent X, X **`has_constituents`** C_X holds if C_X is the set of constituents that X immediately dominates.

The type of the first argument is given by the following definition:[3]

---

[2] Obviously, a stronger (more constrained) definition can certainly be written if we make use of $\overline{\mathrm{X}}$-theory to relate the types of X and Y. However, this was found to be unnecessary for the current set of principle definitions.

[3] Also see note 2.

$X:T:R$ `has_constituents` Y holds $\Rightarrow T = U, R = \{\neg\text{ec}\}$.

Definition 79 (`has_feature`)

Feature membership: given a constituent X (with associated feature set F_X) and an instantiated feature F, X `has_feature` F holds if $F \in F_X$.

Here is the definition for inferring the type of the first argument:

$X:T:R$ `has_feature` F holds $\Rightarrow$
$T = \{c | \forall c' \in V_T$ st. $\exists$ lexical entry E with feature set F_E st. $F \in F_E, c'\text{proj}^* c\}$ provided F is a lexical feature, else
$T = \text{typeValue}(F)$ if (non-lexical) feature F has given type, otherwise
$T = U$.
$R = \emptyset$ in each case.

Negation (`\+`) is handled specially:

`\+` $X:T:R$ `has_feature` F holds $\Rightarrow$
$T = \{c | \forall c' \in V_T$ st. $\exists$ lexical entry E with feature set F_E st. $F \notin F_E, c'\text{proj}^* c\}$ if F is a lexical feature, else
$T = U - T'$ where $T' = \text{typeValue}(F)$ if F has given type, otherwise
$T = U$.
$R = \emptyset$ in each case.

Definition 80 (`head_of`)

The following is a recursive definition for extracting a head. Given a constituent X, Y `head_of` X holds when:

a. Y is X if X is a head or has no sub-constituents, otherwise

b. Y `head_of` X^i where X^i is an immediate constituent of X st. X is a projection of X^i.

A type definition is provided for the second argument in terms of the first. An additional (independent) equation is provided for the first:

$Y:T_Y:R_Y$ `head_of` $X:T_X:R_X$ holds $\Rightarrow$
$T_X = \{c | \forall c' \in T_Y, c'\text{proj}^* c\}, R_X = \emptyset$.
$T_Y = \{c | c$ is a head category label $\}, R_Y = \emptyset$.

Definition 81 (`headOfChain`)

Given a constituent E_1, `headOfChain(`E_1`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_n \rangle, \quad n > 1$.

Definition 82 (`index`)

Indexing: given a constituent X, `index(`X`,`I`)` holds if:

a. I is the index assigned to X if X does not have an index, otherwise

b. I is (or becomes) the index of X if X has already been indexed.

(This may also be used to return the index of a constituent.)

Definition 83 (`inheritFeature`)

Given two constituents X and Y with feature sets F_X and F_Y, respectively, and a feature F, then `inheritFeature(`X`,`F`,`Y`)` updates F_Y to be $F_Y \cup \{F\}$ provided $F \in F_X$. (F_Y is not updated when $F \notin F_X$.)

Definition 84 (`intermediateTrace`)

Given a constituent E_i, `intermediateTrace(`E_i`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_i, \ldots, E_n \rangle, \quad n \geq 3, 1 < i < n$.

Definition 85 (`partOfChain`)

Given a constituent E_i, `partOfChain(`E_i`)` holds if there exists a non-trivial chain $\langle \ldots, E_i, \ldots \rangle, \quad 1 \leq i \leq n$, where n is the length of the chain.

Definition 86 (`partOfSameChain`)

Given two constituents X and Y, `partOfChain(`E_i`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_n \rangle, \quad n > 1$ such that X and Y are members. (Note that two constituents with identical indices need not be part of the same movement chain.)

Definition 87 (`projection_of`)

Given two constituents X and Y with category labels c_X and c_Y, respectively, X `projection_of` Y holds if $c_Y \mathrm{proj}^* c_X$, and X and Y share feature sets.

A type definition in terms of the second argument is provided for the first argument:[4]

$X : T_X : R_X$ `projection_of` $Y : T_Y : R_Y$ holds $\Rightarrow$
$T_X = \{c | \forall c' \in T_Y, c' \mathrm{proj}^* c\}, R_X = \emptyset$.

Definition 88 (`sameIndex`)

Given two indices I and J, `sameIndex(`I`,`J`)` holds if I and J are identical.

[4] Obviously, a symmetrical definition can be written for T_Y in terms of T_X. One is not provided to prevent "race" conditions from occuring during type computation: see chapter 6 for details.

Definition 89 (specifier_of)

Given a maximal projection X, Y **specifier_of** X holds if Y is an immediate constituent of X st. Y is an specifier of X.

The following type definition applies to the second argument:

$$Y \texttt{ specifier\_of } X : T : R \text{ holds} \Rightarrow T = U, R = \{\neg\text{ec}, \neg\text{head}\}.$$

B.2 Principle Definition Forms

In this section, we define the forms used to express conditions over tree structures. For each form, we will first describe its syntax, and then present the compiled PROLOG form Two of these, **in_all_configurations** [Defn. 90] and **compositional_cases_on** [Defn. 92], are particularly noteworthy as they define parser operations. Parser operations form the basic unit for principle ordering and interleaving. Each of them will have one of two possible compiled forms depending on whether the principle will be interleaved or not. Note that no examples of use are provided in this section — there are many examples spread throughout chapters 3 and 6.

Definition 90 (in_all_configurations)

```
:- <Operation-Name> in_all_configurations <Variable>
      where <Pre-conditions> then <Conditions>.

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

(<Operation-Name> names the principle to be defined. <Variable> names the configurations over which the principle operates. Arbitrary combinations of goals may be substituted for all (pre-) conditions.) Roughly speaking, the first form may be read as:

> *"In every configuration (named by* <Variable>*), as defined by* <Pre-conditions>, <Conditions> *must hold."*

By using a variant of the 'if–then–else' construct, the second form allows alternative conditions to be placed on configurations. In either case, the **in_all_configurations** macro expands into two different PROLOG forms. The first form is used when the principle is not interleaved. Expansion will generate a predicate <Operation-Name> with arity 1 that recursively descends a given phrase structure and imposes the specified conditions every time its pre-conditions are satisfied. This "tree-walker" has the following general form:

```
<Operation-Name>(At) :- atomic(At). % case 1: leaf node.
<Operation-Name>([Head|Tail]) :-    % case 2: list of nodes.
        \+ cat([Head|Tail],_),
         !,
        <Operation-Name>(Head),
        <Operation-Name>(Tail).
<Operation-Name>(<Variable>) :-     % case 3: single node
        <Pre-conditions>,           % satisfying
        !,                          % pre-conditions.
        <Conditions>,
        <Variable> has_constituents Cs,
        <Operation-Name>(Cs).       % keep descending...
        etc...
<Operation-Name)(X) :-              % case 4: catch-all for
        X has_constituents Cs,      % single node.
        <Operation-Name>(Cs).
```

(The general form of the third clause is repeated as many times as necessary for alternative conditions in the definition.)

The second form is used when the principle is interleaved. In this case, it is not necessary to use a tree-walker to find the appropriate constituents to impose the conditions. When an appropriate LR reduce action (as defined by the type inference algorithm) is encountered during parsing, the newly-built phrase will be automatically sent to this form for evaluation. Hence, the interleaved predicate will take just one argument and will simply be composed of the "core" if–then–else construct:

```
<Operation-Name>(<Variable>) :-
        <Pre-conditions>
        -> <Conditions>
        ;
        alternative conditions...
        ; true.
```

(Note that the type of the principle being defined is given by the type of `<Variable>`.)

The only exception to this "cascade" of conditions occurs when the interleaver creates specialized versions of the above form for different sub-types of the principle. (See chapter 6 for a detailed description of this process.) In such cases, the interleaved predicate will contain more than one clause, each of which will contain a "reduced chain", i.e. an (order-preserving) subsequence of the various alternatives from the original definition.

Definition 91 (`collect`)

```
:- <Name> collect <Variable-1>
      in_all_configurations <Variable-2>
         where <Conditions>.
```

This form is very similar to `in_all_configurations` [Defn. 90]. The only major difference is that, instead of imposing conditions on constituents that meet certain pre-conditions, this form is designed to collect a list of elements (`<Variable-1>`) from each constituent (`<Variable-2>`) satisfying those (pre-) conditions. For example, with an appropriate instantiation of `<Conditions>`, it can be used to collect, say, all traces or all indices from noun phrases in a given structure.

Its expanded form is virtually identical to the non-interleaved translation for `in_all_configurations`. The basic difference is the presence of an additional argument that returns the elements being collected. Because of the similarity, we omit the description of the expanded form.

Definition 92 (`compositional_cases_on`)

```
:- <Operation-Name> produces <Items-1>
      compositional_cases_on <Variable> where
        <Variable> with_constituents <C-1> and <C-2> st
                     <C-1> produces <Items-2>,
                     <C-2> produces <Items-3>,
                     <Variable> produces <Items-4>
        then <Conditions-1>
      else <Pre-conditions> then <Conditions-2>...

:- <Operation-Name> produces <Items-1>
      compositional_cases_on <Variable> where
        <Variable> with_constituents <C-1> and <C-2> st
                     <C-1> produces <Items-2>,
                     <C-2> produces <Items-3>,
                     <Variable> produces <Items-4>
        then <Conditions-1>
      finally <Conditions-2>
      else <Pre-conditions> then <Conditions-3>...
```

(`<Operation-Name>` names the principle being defined. `<Variable>` names the configurations over which the principle operates. Arbitrary combinations of goals may be used for the conditions `<Conditions-i>` (and any pre-conditions.) Similarly, arbitrary terms may be substituted for `<Items-i>`.)

This basic form is used to encode the compositional definitions introduced in chapter 3. In a compositional definition, values are computed for each sub-phrase in a structure, with the value for an aggregate phrase being computed in terms of the values of its constituents. To ground the recursion, base cases must also defined. This is encoded in a case-by-case basis using either of the forms above. Roughly speaking, the first form may be read as:

> "`<Operation-Name>` *computes* `<Items-1>` *as follows:*
>
> 1. *Constituent* `<Variable>` *consists of two sub-constituents* `<C-1>` *and* `<C-2>`. *Let* `<C-1>` *and* `<C-2>` *have values represented by* `<Items-2>` *and* `<Items-3>`, *respectively. Then the value of the aggregate represented by* `<Variable>`, *namely* `<Items-4>`, *may be computed from* `<Items-2>` *and* `<Items-3>` *using* `<Conditions-1>`, *otherwise:*
> 2. *If the constituent* `<Variable>` *satisfies* `<Pre-conditions>`, *then* `<Conditions-2>` *computes its value, otherwise do nothing."*

Parts 1 and 2 represent the compositional and base cases, respectively. Situations involving more than one base case may be represented by appending extra 'else–then' forms. Although the form of the compositional case only mentions binary branching, the expanded form automatically handles general n-ary branching through iteration. For example, ternary branching will be treated as two instances of binary branching. First, an intermediate value is computed (using `<Conditions-1>`) for the two leftmost branches. Next, the value for the whole branching is computed from the intermediate value and the value for the rightmost branch. The only difference between the first and second form is that the latter allows extra conditions (via the `finally` form) to be invoked once all branches have been collected in this manner.

As for `in_all_configurations`, there will be two possible expansions corresponding to the cases where the operation is interleaved or not. In the non-interleaved case, expansion will generate a tree-walker that recursively descends a given phrase structure and applies the compositional case whenever branching occurs, and the base cases for all other constituents. The values for each sub-phrase will be computed in a bottom-up manner. Two predicates will be defined: one to do the iteration for general n-ary branching, and one to apply the base cases. The general form is as follows:

```
<Operation-Name>(<Variable>|<Items-1>) :- % Compositional
        <Variable> has_constituents Cs,   % case.
        !,
        <Operation-Name-1>(Cs|<Items-1>).
        [<Conditions-2>]                  % Optional
```

```
                                                  % finally form.
<Operation-Name>(<Variable>|<Items-1>) :- % Base case.
        <Pre-conditions>,
        !,
        <Conditions-2>.
        etc...
<Operation-Name>(<Variable>|<Null-Items>).% Catch-all case.
```

(The general form of the second clause is repeated as many times as necessary for alternative conditions in the definition.) The values maintained for each sub-phrase are returned as extra arguments to the predicate. For the last clause which catches any constituents not satisfying the pre-conditions of any of the defined cases, by convention, these values (`<Null-Items>`) are set to the empty list. The occurrence of `<Operation-Name-1>` in the first clause refers to the second predicate which has the following general form:

```
<Operation-Name-1>([]|<Null-Items>).          % Empty list case.
<Operation-Name-1>([Head|Tail]|<Items-4>) :- % Non-empty
        <Operation-Name-1>(Tail|<Items-3>),  % list case.
        <Operation-Name>(Head|<Items-2>),
        <Conditions-1>.
```

This predicate simply iterates over a list of sub-constituents applying `<Conditions-1>` to compute the value of `<Items-4>` from `<Item-2>` and `<Item-3>`. Also, by convention, the values for the empty list are defined to be null.

The second form is used when the operation is to be interleaved. In this case, each time an appropriate reduce action is encountered, the newly constructed phrase will be passed to this form. If the phrase contains branching, then `<Conditions-1>` must be applied (possibly repeatedly) to compute the values for the new phrase from its sub-constituents. On the other hand, if the phrase is simple, then it must be tested against the base cases. In either case, the values computed must be "saved" — so that the next time the operation is called with a larger phrase, it can make use of values already computed for its sub-constituents. In fact, the expansion for interleaving is virtually identical to that for the non-interleaved case, except that every time the values for a constituent are requested, it tests to see whether those values have already been computed — in which case, the saved values are used. Otherwise, it computes the values in the regular manner using `<Conditions-1>`. This is implemented by having an extra (initial) clause for `<Operation-Name>` that performs the memory retrieval function. Because of the similarity of the two forms, we omit the description of the second form.

B.3 Type Composition Rules

In this section, we define rules for propagating types in logic programs. For example, inference rules will be given for the conjunction, disjunction, and negation of goals.

In the following definitions, let G, G_1 and G_2 be goals. Let X be a variable. Let '`,`', '`;`', '`:-`', and '`\+`' be the conjunction, disjunction, conditional, and negation operators, respectively. Let U be the universal type.

Definition 93 (Conjunction)

$$\frac{X : T_1 : R_1(G_1), \qquad X : T_2 : R_2(G_2)}{X : T_1 \cap T_2 : R_1 \cup R_2(G_1\,\texttt{,}\,G_2)}$$

The above rule is meant to be read as: "If X has type T_1 and restriction R_1 in G_1, and if X has type T_2 and restriction R_2 in G_2, then we can infer that X has type T_1 intersect T_2 and restriction R_1 union R_2 in (G_1 `,` G_2)."

Definition 94 (Disjunction)

$$\frac{X : T_1 : R_1(G_1), \qquad X : T_2 : R_2(G_2)}{X : T_1 \cup T_2 : R_1 \cap R_2(G_1\,\texttt{;}\,G_2)}$$

Definition 95 (Negation)

$$\frac{X : T : R(G)}{X : U : R_1(\backslash{+}G)}$$

where R_1 is defined to be $\{\neg r | r \in R\}$. Note that X has universal type after negation — no matter what its original type is. This is a conservative rule which essentially provides no type information. Note however, that exceptions are defined for certain goals. See the definition of the primitive `has_feature` [Defn. 79] for an example where type information is not deleted across negation.

Definition 96 (Predicate Definition)

Let predicate P be defined by n clauses of the form $H_i\texttt{:-}G_i, \quad 1 \leq i \leq n$. Let G be a goal and let σ_i be a unifying substitution for H_i wrt. G. Then:

$$\frac{\forall i \text{ st. } \exists \sigma_i \text{ st. } G = H_i\sigma_i \quad X : T_i : R_i(H_i\texttt{:-}G_i)\sigma_i}{X : \bigcup_i T_i : \bigcap_i R_i(G)}$$

Definition 97 (Single Clause)

Let $H\,\texttt{:-}G_1$ be a clause. Let G be a goal and let σ be a unifying substitution for G_1 wrt. G. Then:

$$\frac{X : T : R(G)}{X : T : R(H\,\texttt{:-}G_1)\sigma}$$