Technical Report 1000

UNITRAN: A Principle-Based Approach to Machine Translation

Bonnie Jean Dorr

MIT Artificial Intelligence Laboratory

Number 20.

This report presents an approach to natural language translation that relies on principle-based descriptions of grammar rather than rule-oriented descriptions. The model that has been constructed is based on abstract principles as developed by Chomsky (1981) and several other researchers working within the "Government and Binding" (GB) framework. The approach taken is "interlingual", *i.e.*, the model is based on universal *principles* that hold across all languages; the distinctions among languages are then handled by settings of *parameters* associated with the universal principles. The design of the UNITRAN (UNIversal TRANslator) system is such that a language may be described by the same set of parameters that specify the language in linguistic theory. Because of the modular nature of the model, the interaction effects of universal principles are easily handled by the system; thus, the programmer does not need to specifically spell out the details of rule applications. Because only a small set of principles covers all languages, the unmanageable grammar size of alternative approaches is no longer a problem.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. T.R. No. 1000 December, 1987

UNITRAN: A PRINCIPLE-BASED APPROACH TO MACHINE TRANSLATION

Bonnie J. Dorr

This report is a revised version of a thesis submitted to the department of Electrical Engineering and Computer Science on May 7, 1987, in partial fulfillment of the requirements for the degree of Master of Science. The research described by this report was done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-80-C-0505 and N00014-85-K-0124, and also in part by NSF Grant DCR-85552543 under a Presidential Young Investigator's Award to Professor Robert C. Berwick. Useful guidance and commentary were provided by Peter Andreae, Ed Barton, Bob Berwick, Dave Braunegg, Michael Brent, Noam Chomsky, Bruce Dawson, Sandiway Fong, Eric Grimson, Ken Hale, Mike Kashket, Boris Katz, and Patrick Winston.

Abstract

Machine translation has been a particularly difficult problem in the area of Natural Language Processing for over two decades. Early approaches to translation failed since interaction effects of complex phenomena in part made translation appear to be unmanageable. Later approaches to the problem have succeeded (although only bilingually), but are based on many language-specific rules of a context-free nature. This report presents an alternative approach to natural language translation that relies on principle-based descriptions of grammar rather than rule-oriented descriptions. The model that has been constructed is based on abstract principles as developed by Chomsky (1981) and several other researchers working within the "Government and Binding" (GB) framework. Thus, the grammar is viewed as a modular system of principles rather than a large set of *ad hoc* language-specific rules.

Typically machine translation systems have used parsing strategies that are based on context-free grammars. To try to capture all of the phenomena allowed in natural languages, context-free rule based systems require an overwhelming number of rules; thus, a translation system either has limited linguistic coverage, or poor performance (due to formidable grammar size). The system I have constructed handles many complex phenomena without relying on a large set of language specific rules.

The approach taken is "interlingual", *i.e.*, the model is based on universal *principles* that hold across all languages; the distinctions among languages are then handled by settings of *parameters* associated with the universal principles. For example, there is a universal principle that requires a particular ordering of constituents with respect to a phrase. The parameter that corresponds to this principle is called "constituent order," which is set to *head-initial* for a language like English, but *head-final* for a language like Japanese. The design of the system is such that a language may be described by the same set of parameters that specify the language in linguistic theory. Because of the modular nature of the model, the interaction effects of universal principles are easily handled by the system; thus, the programmer does not need to specifically spell out the details of rule applications. Because only a small set of principles covers all languages, the unmanageable grammar size of alternative approaches is no longer a problem.

Thesis Supervisor:	Dr. Robert C. Berwick
Title:	Associate Professor of Computer Science and Engineering

Contents

1 Introduction **7**

2 Computational Framework of the Translation Model **16**
- 2.1 Direct and Transfer Approaches: Rule-based Systems . . . 16
- 2.2 Interlingual Approaches: Principle-based Systems . . . 19

3 Linguistic Framework of the Translation Model **22**
- 3.1 $\overline{\text{X}}$-Theory . . . 23
  - 3.1.1 Constituent Order Parameter . . . 24
  - 3.1.2 Infl and Comp . . . 27
- 3.2 θ-Theory . . . 30
  - 3.2.1 θ-Criterion and Projection Principle . . . 30
  - 3.2.2 Canonical Structural Realization and Visibility Condition . . . 31
  - 3.2.3 Clitic Doubling and θ-role Transmission . . . 32
- 3.3 Government Theory . . . 33
- 3.4 Case Theory . . . 35
  - 3.4.1 Case Assignment and Case Filter . . . 35
  - 3.4.2 Clitic Doubling and Choice of Government . . . 36
- 3.5 Trace Theory . . . 37
  - 3.5.1 Null Subject Parameter and ECP . . . 39
  - 3.5.2 V-Preposing and ECP . . . 41
- 3.6 Binding Theory . . . 43
  - 3.6.1 Binding Conditions for Overt Noun Phrases . . . 43
  - 3.6.2 Binding Conditions for Empty Noun Phrases . . . 44
- 3.7 Bounding Theory . . . 46
  - 3.7.1 V-preposing and Choice of Bounding Nodes . . . 46
- 3.8 Principles and Parameters . . . 47

4 Overall Design of UNITRAN **48**

5 Pretranslation Routines **58**
- 5.1 Preprocessing Routines . . . 58
- 5.2 Morphological Analysis . . . 59
  - 5.2.1 Morphological Rule Formalism . . . 59

5.2.2 The Lexicon 65
5.3 Precompilation 72

6 Parsing Component **80**
6.1 Structure-Building Component: Augmented Earley Algorithm 80
6.2 Linguistic Constraints of Parser 85
6.2.1 $\overline{\mathrm{X}}$ Module 87
6.2.2 θ Module 98
6.2.3 Government Module 102
6.2.4 Case Module 107
6.2.5 Trace Module 109
6.2.6 Binding Module 109
6.2.7 Bounding Module 111

7 Final Translation Routines **120**
7.1 Replacement Routines 121
7.1.1 Structural Replacement 121
7.1.2 Thematic Substitution 123
7.1.3 Linguistic Constraints on Replacement Routines 125
7.2 Generation Routines 128
7.2.1 Structural Movement (Move-α) 129
7.2.2 Morphological Synthesis 132
7.2.3 Linguistic Constraints of Generation Routines 134

8 Example Of Translation **136**
8.1 Parsing 136
8.2 Replacement 160
8.3 Generation 163
8.4 Processing Time 169

9 Limitations, Future Work, and Conclusions **170**
9.1 Limitations 170
9.2 Directions for Future Work 174
9.3 Conclusions 179

A Principles and Parameters **181**
A.1 Principles and Parameters of GB Modules 182
A.2 Parameter Values for Spanish and English 184

B Representation of Kimmo Automata **186**
B.1 English Automata 186
B.2 Spanish Automata 189

C Representation of Lexicons **193**
C.1 English Lexicon 193
C.2 Spanish Lexicon 203

D Translation System Parameters **218**

E Lisp Representation of GB Parameters **223**
E.1 $\overline{\text{X}}$-Theory 223
E.2 θ-Theory 225
E.3 Government Theory 225
E.4 Case Theory 225
E.5 Trace Theory 226
E.6 Bounding Theory 226

F Lisp Representation of GB Principles **227**
F.1 $\overline{\text{X}}$-Theory 227
F.1.1 Precompilation Routines 227
F.1.2 Complement Prediction Routines 229
F.1.3 Routines for Associating Complements with Heads 233
F.1.4 Routines for Determining Feature Information and Argument Structure 234
F.1.5 Feature Percolation Routines 236
F.1.6 Structure-Modification Routines 239
F.2 θ-Theory 240
F.2.1 CSR Mapping 240
F.2.2 θ-Role Assignment Routines 240
F.2.3 θ-Role Transmission Routines 242
F.2.4 θ-Criterion Routines 244
F.2.5 θ-Role Matching Routines 245
F.3 Government Theory 246
F.3.1 Routines for Setting up Government Relations 246
F.3.2 Routines for Testing Government Relations 250
F.4 Case Theory 251
F.5 Trace Theory 253
F.6 Binding Theory 254
F.7 Bounding Theory 256
F.7.1 Routines for Trace Linking at PUSH Time 256
F.7.2 Routines for Trace Linking at POP Time 259
F.7.3 Routines for Checking Language-Specific Effects 260
F.7.4 Routines for Performing Move-α at Generation Time 260

G Earley Parser Routines **262**
G.1 Main Earley Parser Loop 262
G.2 Predict Action 265
G.3 Scan Action 267
G.4 Complete Action 269

H Interface Between Structure-Building and GB **271**
H.1 Interface During PUSH Stage 272
H.2 Interface During SCAN Stage 276
H.3 Interface During POP Stage 279
H.4 Interface During Replacement Stage 281
H.5 Interface During Generation Stage 284

I Global Variables for UNITRAN **286**

J Translation Routines of UNITRAN **291**

K Examples **296**
K.1 Free Inversion 296
K.2 Verb-Preposing 298
K.3 Subject-Aux Inversion (SAI) 300
K.4 Thematic Divergence 302
K.5 Raising 303
K.6 Clitic Doubling 305
K.7 Null Subject 306

Chapter 1

Introduction

How can computerized natural language translation be performed without relying on a large set of language-specific rules? A large majority of approaches to translation use parsing strategies that are based on context-free grammars. To try to capture all of the phenomena observed in natural languages, context-free rule-based systems require an overwhelming number of rules; thus, a translation system either has limited the linguistic coverage, or poor performance due to formidable grammar size.[1]

This report concerns an alternative approach to natural language translation. In particular, the computational system described herein, UNITRAN, relies on principle-based descriptions of grammar rather than rule-oriented descriptions.[2] The model that has been constructed is based on abstract principles as developed by Chomsky (1981a, 1981b) and several other researchers working within the "Government and Binding" (GB) framework. Thus, the grammar is viewed as a modular system of principles rather than a large set of *ad hoc* language-specific rules.

What is natural language translation, and what makes it a difficult problem? Natural

[1]As noted in Barton, 1984, in a typical parsing system the description of a language is lengthy, thus increasing the running time of many parsing algorithms. (The Earley algorithm (1970) for context-free language parsing can quadruple its running time when the grammar size is doubled.) The same is true of typical translation systems. For example, Slocum's METAL system (1984a) developed at the Linguistics Research Center at the University of Texas relies on over 1000 language-specific context-free rules per language solely for syntactic processing. In order to ensure computational feasibility, the system requires that linguistic coverage be limited.

[2]The name UNITRAN stands for UNIversal TRANslator, that is, the system serves as the basis for translation across a variety of languages, not just two languages or a family of languages.

language translation is the task of mapping a source (natural) language input into a target (natural) language output. In order to determine this mapping, we must understand the common representation that underlies the source language input and target language output. The "underlying form" that this translation approach uses will be discussed in more detail in section 7.1.1.

The reason that translation is difficult is that it seems to require a massive amount of "knowledge" in order to handle all possible phenomena (and their interaction effects) that might occur in a language. Consider (1):

(1) ¿Qué vio?
'What did {he,she} see?'[3]

Although (1) appears to be simple, it is not simple from the point of view of machine translation since the sentence exhibits interaction among three complex phenomena. The first phenomenon is the absence of a subject in the source language. In many languages (*e.g.*, Spanish, Italian, Greek, and Hebrew), the pronominal subject of a tensed sentence may remain unexpressed; the verbal morphology is rich enough to make the subject pronouns recoverable. Thus, (1) would literally translate as:

(2) * What saw?[4]

In order to rule out (2) a translation system must have the "knowledge" that a null subject is allowed in Spanish, but not in English.

The second phenomenon concerns movement of a sentence constituent. In (1), the verb *vio* (= saw) takes an object, but the object does not appear where it is normally found (*i.e.*, after the verb); instead, the object *qué* precedes the verb. The positioning of *qué* at the beginning of the sentence is conceived of as a type of movement: the object has moved from verb-phrase final to sentence-initial position. This phenomenon, which may occur in either Spanish or English, must be accounted for during the translation process.

[3]The "{.. , ..}" notation will be used to denote optionality. Thus, in (1) the subject of the sentence may either be *he* or *she*.

[4]An asterisk in front of a sentence is used to indicate that it is syntactically ill-formed in some way.

The third phenomenon is called inversion. In Spanish (and other Romance languages), the verb is allowed to invert with its subject. Thus, if the subject of (1) were overt, it would be inverted, as in (3):

(3) ¿Qué vio Juan?
'What did John see?'

Although the subject is phonetically null in (1), it is assumed to have inverted with the verb, just as in (3), where the subject is present.

Early approaches to translation failed since interaction effects of complex phenomena such as those found in (1) made translation appear to be unmanageable. Later approaches to the problem have succeeded in handling many different linguistic phenomena, but these translation systems do not operate cross-linguistically and are generally based on language-specific rules of a context-free nature.[5] The approach I take handles many complex phenomena without sacrificing cross-linguistic application, and without relying on a large set of language-specific rules. It is an "interlingual" approach, that is, it is based on universal *principles* that hold across all languages; the distinctions among languages are then handled by settings of *parameters* to the universal principles. For example, there is a principle that requires a particular ordering of constituents with respect to a phrase. The parameter setting that corresponds to this principle is called "constituent order," which is set to *head-initial* for English, but *head-final* for a language like Japanese. The *head-initial* parameter setting forces the object to follow the verb in English (*e.g.*, "hit the ball"); by contrast, the *head-final* parameter setting forces the object to precede the verb in Japanese (*e.g.*, "the ball hit").

The contribution put forth by this investigation is two-fold: (a) from a linguistic point of view, the investigation allows the principles of GB to be realized and verified; and (b) from a computational perspective, descriptions of natural grammars are simplified, thus easing the programmer's and grammar writer's task. The model not only permits a language to be de-

[5]Slocum's system (1984a) relies on a separate set of context-free language-specific rules for each source and target language. The system is entirely rule-based, and is not easily extendible to other languages. By contrast, Sharp's translation system (1985) approaches a design with applies cross-linguistically (*i.e.*, it does not rely on a large database of language-specific context-free rules and includes some universal principles), but essentially operates only between Spanish and English. No provision is made for the user to freely enter new parameter values for additional languages since most of the principles are implicitly represented (hardwired) in the code.

scribed by the same set of parameters that specify the language in linguistic theory, but it also eases the burden of the programmer by handling interaction effects of universal principles without requiring that the effects be specifically spelled out. For example, trace-antecedent specifications are generally incorporated directly into structure-building rules in a rule-based approach. That is, rules that build sentences require a provision for linking a "moved element" (*e.g.*, *qué* in (1)) with the position from which it is missing (*e.g.*, object position in (1)).[6] By contrast, the principle-based approach abstracts the task of trace-antecedent linking into a module that is allowed to apply across all types of structures regardless of how the structures are built. Thus, there is a single structure-building mechanism that assembles phrasal constituents into phrases, and there is an independent trace-antecedent linking routine that applies uniformly across all types of phrases. Ultimately, the goal is for a small set of principles (grouped into modules) to cover phenomena found in all languages so that unmanageable grammar size is no longer a problem.

Corresponding to the two-fold contribution of the investigation is the two-fold advantage of this principle-based approach: (a) from a scientific point of view the interlingual approach is beneficial because it allows linguistic generalization to be captured (modularized principles eliminate the need for specifying common properties across languages directly in rules), and the approach lends itself to a more plausible theory of learnability (the basic syntactic description for a language is condensed to a set of parameter values, not a large database of rules); and (b) from an engineering point of view the interlingual approach is advantageous because the grammar size is reduced (the multiplicative effects of constraint modules are not spelled out in the form of grammar rules) and the system is easily extendible (a separate description is not required for each language).

The overall design of the system is illustrated in figure 1.1 (using (1) as the translation example). The *structural and lexical processing* includes three stages: parsing, thematic substitution, and generation, each of which will be discussed shortly. During all three stages a structure-generating module operates in a co-routine fashion with a linguistic constraint module. The linguistic constraint module consists of universal principles with user-modifiable parameter set-

[6] For example, the GPSG approach (Gazdar *et al.* (1985)) uses a "slashed-category" mechanism to incorporate the trace-antecedent relation directly into the context-free grammar rules.

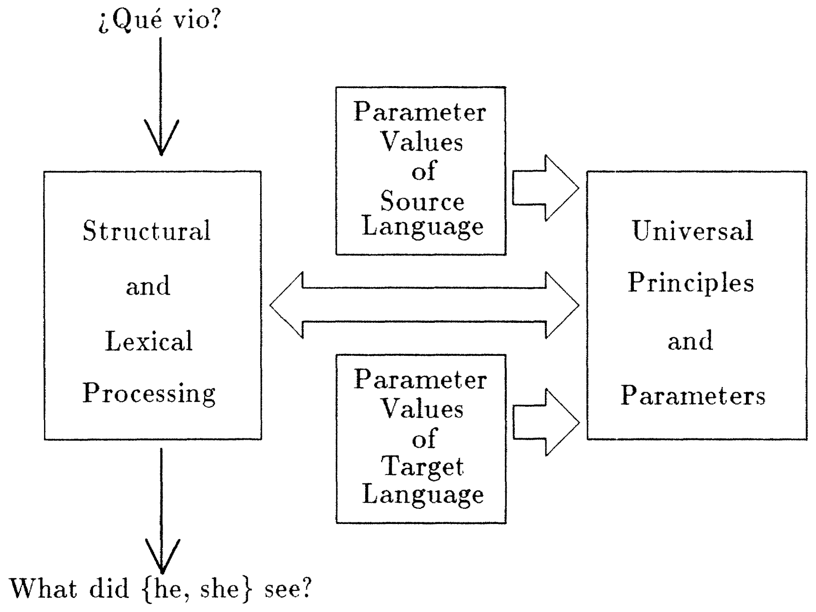

¿Qué vio?
Parameter Values of Source Language
Structural and Lexical Processing
Universal Principles and Parameters
Parameter Values of Target Language
What did {he, she} see?

Figure 1.1: Interaction of Translator with Universal Principles

tings. For example, in the case of the above-mentioned "constituent order" parameter, the user is allowed to specify either *head-initial* or *head-final*, depending on the language being translated. Thus, all universal principles have user-modifiable parameters associated with them. Before the source language processing (parsing) takes place, the parameters are set according to the source language values, but are then *reset* according to the target language values before target language processing (generation) occurs.

The three stages, parsing, thematic substitution, and generation, with the corresponding input-output for each stage (using (1) as the translation example) are illustrated in figure 1.2. The parser takes a morphologically analyzed input and returns a tree structure that encodes structural relations among elements of the source language sentence. Just prior to thematic substitution, a structural replacement routine moves constituents back into their place of origin, thus deriving a representation that underlies the source language. During thematic substitution the parameter settings for the target language are installed, and the target lexical entries that thematically correspond to the source language constituents are substituted. Thus, in the example *vio* is thematically mapped to *saw* when it is discovered that the thematic roles of

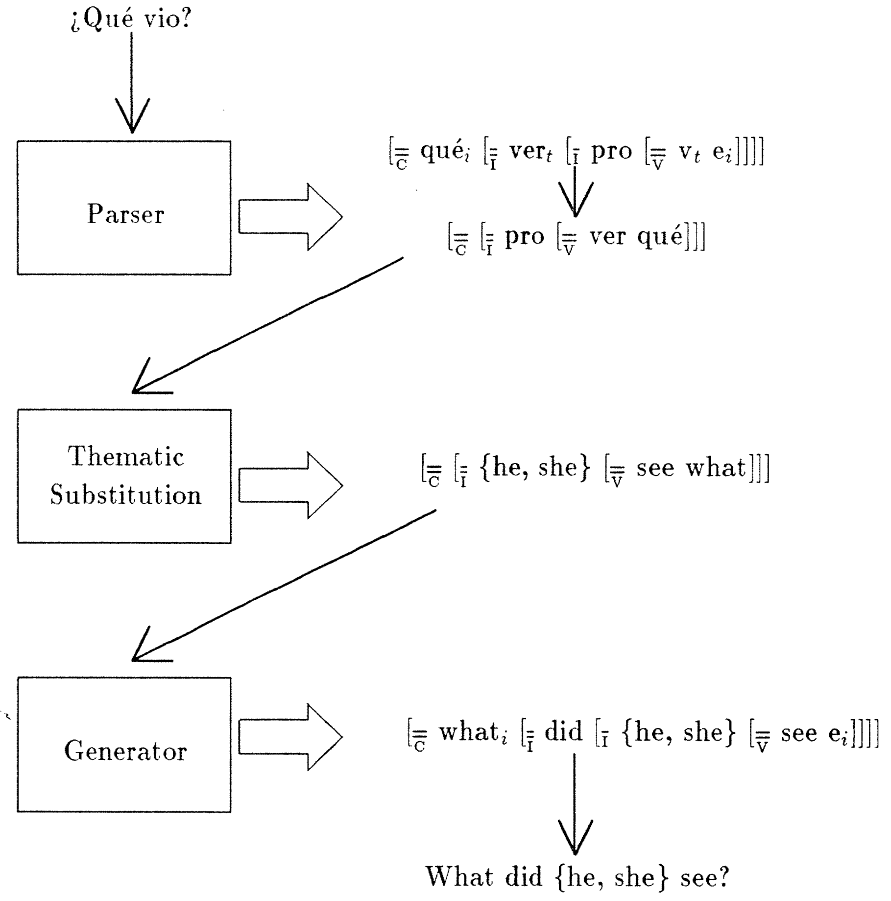

¿Qué vio?
Parser
$[_{\bar{\bar{C}}}$ qué$_i$ $[_{\bar{I}}$ ver$_t$ $[_{\bar{I}}$ pro $[_{\bar{\bar{V}}}$ v$_t$ e$_i$]]]]
$[_{\bar{\bar{C}}}$ $[_{\bar{I}}$ pro $[_{\bar{\bar{V}}}$ ver qué]]]
Thematic Substitution
$[_{\bar{\bar{C}}}$ $[_{\bar{I}}$ {he, she} $[_{\bar{\bar{V}}}$ see what]]]
Generator
$[_{\bar{\bar{C}}}$ what$_i$ $[_{\bar{I}}$ did $[_{\bar{I}}$ {he, she} $[_{\bar{\bar{V}}}$ see e$_i$]]]]
What did {he, she} see?

Figure 1.2: Three stages of Translation

the *vio* and *saw* match. At the generation stage, movement and morphological synthesis take place, thus deriving the target language sentence *What did {he, she} see?*

Although all three stages of translation have been implemented and are discussed in this report, the emphasis of the project is on the parsing component. The generation routines are not as elaborate as the parsing routines, and they do not handle all of the cases that the parser can handle due to problems concerning structural realization (*i.e.*, choosing the *syntactically* correct form to generate from the "underlying form") and lexical selection (*i.e.*, choosing the *semantically* correct form to generate from the "underlying form").[7]

[7]See Dorr (1988) for a lexicon-driven generation approach that addresses the issues of lexical selection and structural realization.

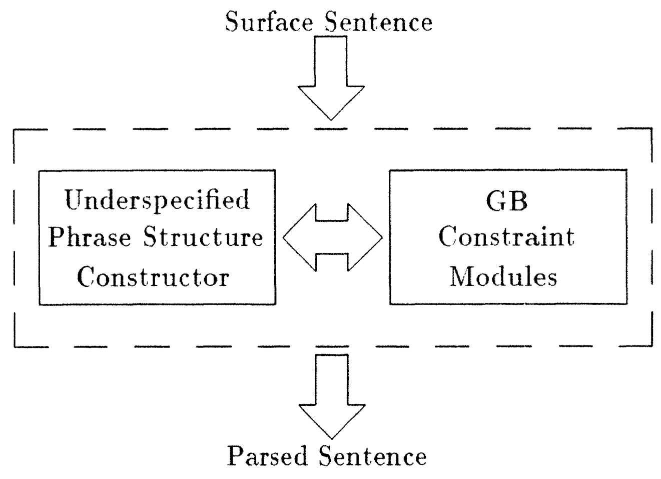


Figure 1.3: Co-Routine Design of Parsing Stage of Translation

Figure 1.3 shows the co-routine design of the parsing stage. During the parse of a sentence, there is a back-and-forth flow between the Phrase Structure Constructor and the GB Constraint Modules. The Phrase Structure component builds underspecified phrase structures (*i.e.*, structures that do not include information about agreement, abstract case, semantic roles, argument structure, *etc.*) based on $\overline{X}$-Theory, while the GB component enforces well-formedness conditions (agreement filters, case filters, etc.) on the structures passed to it, and adds missing information (argument structure, semantic roles, etc.) not available to the structure building component. Note that the model assumes that a syntactic structure will initially be assigned to a sentence, and that this structure may be eliminated or modified according to principles of GB. This design is consistent with several studies that indicate that the human language processor initially assigns a (possibly ambiguous or underspecified) structural analysis to a sentence, leaving lexical and semantic decisions for subsequent processing.[8]

The computational system is built in Lisp and runs on a Symbolics 3600. It is currently bidirectional between Spanish and English, although other languages may easily be added since a universal approach is taken.[9] The primary focus of the investigation falls within the

[8]Frazier 1986 provides recent psycholinguistic evidence that parsing proceeds in this fashion.

[9]The approach is "universal" only to the extent that the linguistic theory is "universal." There are some residual phenomena not covered by the theory that are consequently not handled by the system in a principle-based manner. For example, the language-specific English rules of *it-insertion* and *do-insertion* cannot be

realm of syntax. Thus, there is no global contextual "understanding" (the system translates one sentence at a time). Semantics is incorporated only to the extent of locating possible antecedents of pronouns (*e.g.*, linking *himself* with *he* in the sentence *He dressed himself*), and assigning semantic roles[10] (*e.g.*, designating *he* as "agent-of-action" in *He ate dinner*). to certain elements of the sentence, in particular, arguments of verbs (*e.g.*, in the English sentence "I read the book", the external argument (agent) of *read* is *I*, and the internal argument (goal) is *book*).

This is not to say that semantic issues should be ignored in machine translation; on the contrary, semantics may be the next step in the evolution of the translation system presented here. However, the theory of thematic roles, which falls within the domain of syntax, is a major part of what makes translation work: in order to understand the central action of a sentence, the participants of the action must be identified. In the field of Artificial Intelligence, one of the most important methodological considerations is the selection of a representation that allows natural constraints to be exploited; thus, before delving into semantics to identify the meaning, one must take advantage of syntax, which exposes many of the structural constraints (*e.g.*, relations between nodes in parse trees) required to understand the central action of a sentence. Furthermore, exploiting natural constraints provided by syntax avoids such requirements as small subject domain, narrow linguistic coverage, and enormous lexical entries (as found in exclusively semantic-based systems). Thus, while UNITRAN does not employ semantic processing *per se*, it is the "core" of any modular translation system to which global contextual understanding may subsequently be added, perhaps in the interpretation of the thematic roles identified by the syntactic component of the system.

The organization of this report is as follows: Chapter 2 presents the computational framework for the translation model, including a comparison of this approach with other translation approaches. Chapter 3 provides the linguistic framework for the translation model, including a summary of the principles and parameters that are used in the model. Chapter 4 covers the

accounted for by parameterized principles, but must be individually stipulated as idiosyncratic rules of English. Happily, there appear to be only a few such rules per language since the principle-based approach factors out most of the commonalities across languages. By contrast, in a system like Slocum's, thousands of language-specific rules are employed for each source and target language.

[10]Semantic roles will henceforth be called *thematic* or *theta*-roles (θ-roles) in accordance with GB Theory.

overall design of the system. Chapter 5 contains a description of the pretranslation routines, including preprocessing, precompilation and morphological analysis. Chapter 6 presents the design of the parsing component. Chapter 7 describes the final translation stages (including thematic substitution and generation). Chapter 8 gives an example of execution of the system. Chapter 9 discusses the limitations of the model, directions for future work, and conclusions.

Chapter 2

Computational Framework of the Translation Model

The aim of this chapter is to present the computational framework for UNITRAN, and to put into perspective how the design of the system differs and compares to other approaches. The distinction between rule-based (non-interlingual) and principle-based (interlingual) systems will be presented, and the advantages of the principle-based design over other designs will be discussed.

2.1 Direct and Transfer Approaches: Rule-based Systems

An early approach to translation taken by GAT (the Georgetown Automatic Translation system (1964), as described by Slocum (1984b)) was a *direct* word-for-word for word scheme in which there was a parser and generator for each source-target language pair (see figure 2.1). The primary characteristic of such an approach is that it was designed to translate out of one specific language into another.

Later approaches to translation (*e.g.*, the METAL system by Slocum (1984a)) have taken a *transfer* approach, in which there is only one parser and one generator for each source and target language. In this approach, there are a set of *transfer* components, one for each source-target language pair (see figure 2.2). The transfer phase is actually a third translation

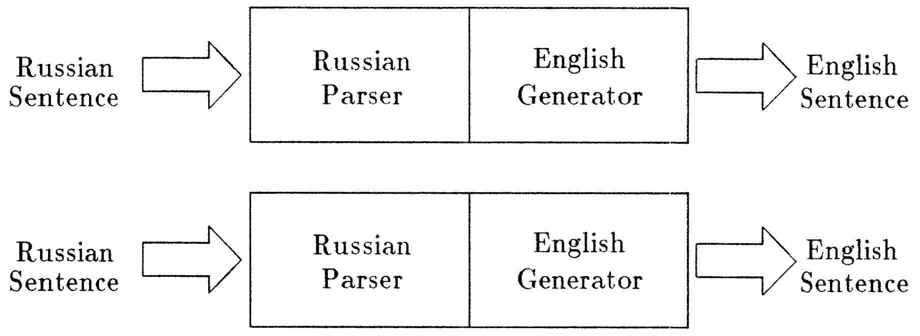


Figure 2.1: Direct Translation Approach *e.g.*, GAT (1964)

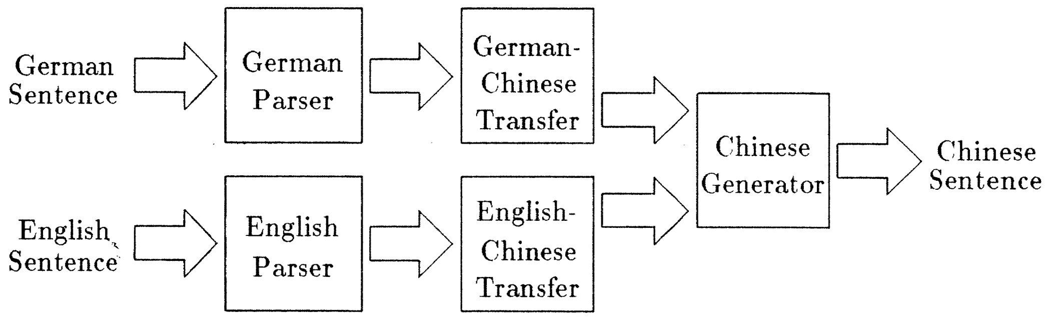


Figure 2.2: Transfer Translation Approach *e.g.*, METAL (1984)

stage in which one language-specific representation is mapped into another. In contrast to the direct approach to translation, the transfer approach has been somewhat more successful, accommodating a variety of linguistic strategies across different languages. The METAL system currently translates from German into Chinese and Spanish, as well as from English into German.

The malady of the transfer approach is that each parsing component is based on language-specific context-free rules.[1] Because the system has no access to universal principles, there is no consistency across the components; thus, each parser has an independent theoretical and engineering basis. Rather than abstracting principles that are common to all languages into

[1]In Slocum's system, the type of grammar formalism is allowed to vary from language to language. For example, the German parser is based on phrase-structure grammar, augmented by procedures for transformations; by contrast, the English parser employs a modified GPSG approach with no transformations. Regardless of the type of grammar formalism, each parser is nevertheless based on thousands of rules of a context-free nature.

```
NN          NST        N-FLEX
 0           1            2
LVL 0       REQ WI     REQ WF

TEST (INT 1 CL 2 CL)

CONSTR (CPX 2 ALO CL)
       (CPY 2 NU CA)
       (CPY 1 WI)
```

Figure 2.3: Context-free Phrase-Structure Rule in METAL

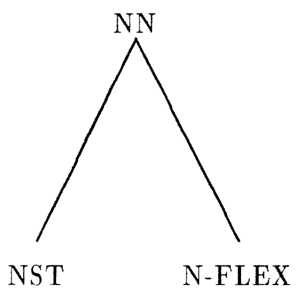


Figure 2.4: Tree generated via Context-Free Phrase-Structure Rule in METAL

separate modules that can be activated upon translation of any language, each parser must independently include all of the information required to translate that language, whether or not the information is universal. For example, agreement information must be encoded into each rule in the METAL system; there is no separate agreement module that can apply to other rules. Furthermore, there is no "rule-sharing" — all rules are language-dependent and cannot apply across several languages.

Figure 2.3 gives an example of a context-free rule in the METAL system. In this example, the "father" node (NN) and "sons" (NST and N-FLEX) are built into a syntax tree as shown in figure 2.4.[2] This tree corresponds to a noun stem (NST) and its nominal ending (N-FLEX).

[2]This example is taken from Slocum (1984a), p. 18.

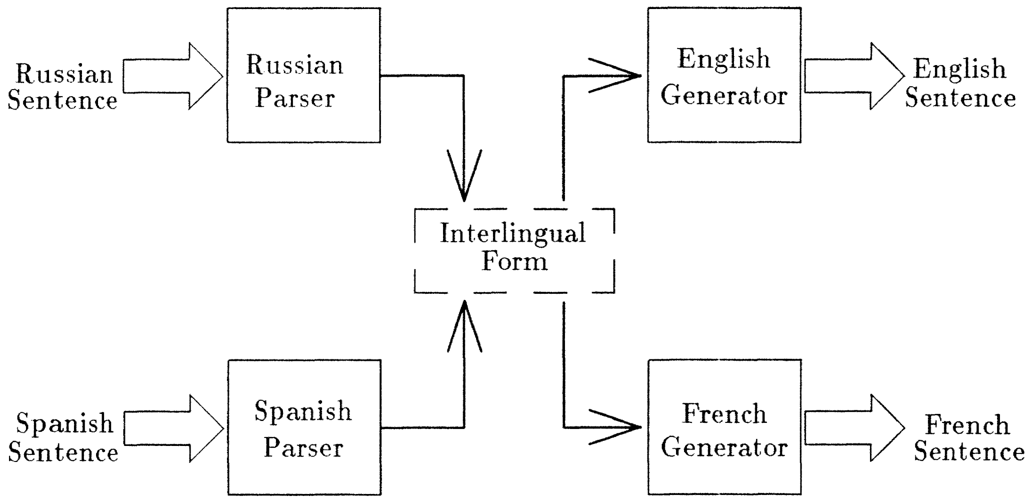


Figure 2.5: Interlingual Translation Design as in CETA (1961) and Sharp (1985)

The syntax tree is built by the CONSTRuctor part of the rule only after the constituent tests (the second and third lines) and the TEST portion of the rule are satisfied. Essentially, what this rule does is associate a noun with the two constituents NST and N-FLEX (using the column tests which require the first element to be word-initial (WI) and the second element to be word-final (WF)), and then test for agreement between the two constituents (the fourth line). Note that the application of such a rule must be restricted by tests on syntax (positioning and agreement in this example) which are encoded directly in the rule, rather than in independent modules which can account for these constraints globally. Furthermore, in order to account for a wide range of phenomena, thousands of such rules are required for each language, thus increasing grammar search time.[3]

2.2 Interlingual Approaches: Principle-based Systems

The translation model described in this report moves away from the language-specific context-free rule approach. It is an *interlingual* approach, (*i.e.*, the source language is mapped into a form that is independent of any language); thus, there are no transfer modules or language-specific context-free rules. The interlingual approach to translation has been taken by CETA

[3]The LRC MT system has approximately 1,000 phrase-structure rules for each source language. Since the GPSG formalism is employed, these 1,000 rules multiply out to an unmanageable grammar size after meta-rules have applied. For more on the complexity of GPSG, see Ristad (1986).

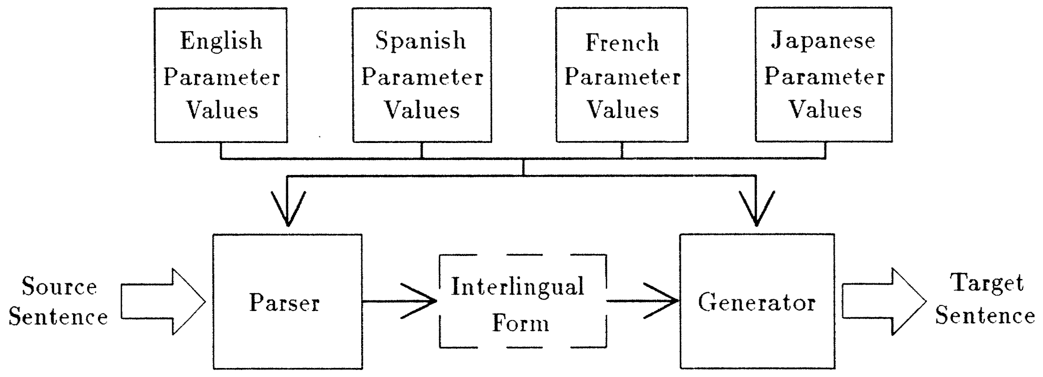


Figure 2.6: The Modified Interlingual Design: Dorr 1987

(Centre d'Etudes pour la Traduction Automatique (1961), as described by Slocum (1984b)) and Sharp (1985). However, the CETA system is not entirely interlingual since there is a transfer component (at the lexical level) that maps from one language-specific lexical representation to another. Sharp's system, although not rule-based, is also not entirely interlingual since it includes some hard-wired principles that are not user-programmable (*i.e.*, not parameterized). The result is that the class of languages that can be translated is limited. The interlingual approach as embodied by CETA and Sharp is illustrated in figure 2.5. Note that there are no transfer components, but that there is a separate parser and generator for each source and target language. The interlingual form is assumed to be a form common to all languages.

One problem with this incarnation of the interlingual approach is that the user must supply a parser for each source language and a generator for each target language. The approach taken for UNITRAN is still interlingual by definition (*i.e.*, the source language is mapped into a form that is independent of any language), but the design is slightly different from that of CETA and Sharp: the same parser and generator are used for all languages. This more closely approximates a true universal approach since the principles that apply across all languages are entirely separate from the language-specific characteristics expressed by parameter settings. Figure 2.6 illustrates the design of the model. The parser and generator are user-programmable: all of the principles associated with the system are associated with parameters that are set by the user. Thus, the user does not need to supply a source language parser or a target language generator since these are already part of the translation system. The only requirement is that the built-in parser and generator be *programmed* (via parameter settings) to process the source and target languages. For example, the user must specify that the English language requires a sentence to contain a subject, but that Spanish does *not* have a subject requirement. This is done by setting the "null subject" parameter to TRUE; by contrast, this parameter must be set to FALSE for English. A dictionary for each language must also be supplied (the dictionary, or *lexicon* is described briefly in section 3.2.2, and in more detail in section 5.2.2) for each language to be translated.

Chapter 3

Linguistic Framework of the Translation Model

Having just seen the computational framework for the translation model, this chapter will now turn to the linguistic basis of the system. The linguistic theory that the system models is the theory of "Government and Binding" (GB). The central idea of the theory is that there is a universal grammar (UG) that represents the linguistic knowledge common to all languages. UG consists of subsystems of principles that are parameterized. The parameter settings interact with the principles, yielding language-specific effects, thus accounting for all the phenomena that are handled by detailed language-specific rules in older translation approaches.

Within the GB framework there are four different levels of grammatical description: *D-structure*, a base form in which semantic participants (or thematic roles) like *agent* or *affected object* are identified; *S-structure*, a surface form in which syntactic movement has taken place; *Logical Form*, a form in which "meaning" (quantification and scope) is specified; and *Phonological Form*, a form in which sound is represented. The relationship between the levels of description is shown in figure 3.1. At the level of S-structure, the D-structure form *John ate what?* would be represented as *What did John eat?*, the corresponding Logical Form would be *For which thing x, John ate x*, and the corresponding Phonological Form would be the literal sound sequences that comprise the surface utterance. Several modules containing subsystems of GB principles are accessible at each of these grammatical levels. The modules are: $\overline{\text{X}}$, θ,

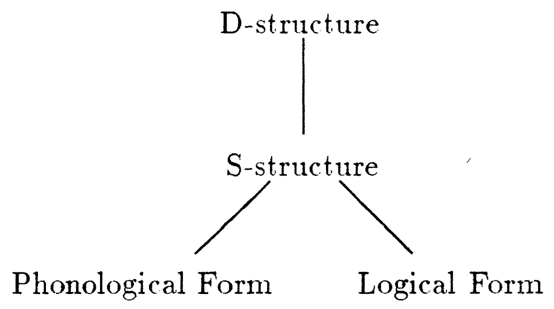


Figure 3.1: Relationship Between Four Levels of Grammatical Description

Government, Case, Trace, Binding, and Bounding. Each of these modules will be discussed in turn in the next seven sections.

3.1 $\overline{\text{X}}$-Theory

There have been many proposals for the phrase-structure representation of sentences. The one adopted here is a modification of $\overline{\text{X}}$-Theory presented by Chomsky (1981b). The central idea is that the dictionary (henceforth *lexicon*) specifies subcategorization frames for lexical items (*e.g.*, the frame for the verb *put* includes two arguments, one that is a noun phrase, and another that is a prepositional phrase, as in *put the car in the garage*), and phrase-structures are projections of a lexical head X (= N, V, P or A)[1] such that the following scheme is obeyed:

(4) $\overline{\overline{\text{X}}} \Rightarrow$ Specifier $\overline{\text{X}}$

$\overline{\text{X}} \Rightarrow$ X Complement

$\overline{\overline{\text{X}}}$ is more familiarly known as XP (= NP, VP, PP or AP). An example of a specifier of a constituent is a determiner of a noun (*e.g.*, *a, the*). In general, specifiers may be optional.[2] A complement consists of one or more arguments as specified in the subcategorization frame of the lexical entry of the head.

[1] I make the assumption that basic categories are specified as values to a parameter of $\overline{\text{X}}$-Theory since it is not clear that the ones used in English and Spanish (*i.e.*, N, V, P and A, and also Comp and Infl (to be discussed shortly)) are universal across all languages. Similarly, pre-terminals such as *determiner* and *adverb* are specified via an $\overline{\text{X}}$ parameter setting.

[2] Choice of specifiers (and their optionality) is specified as a setting of a parameter to $\overline{\text{X}}$.

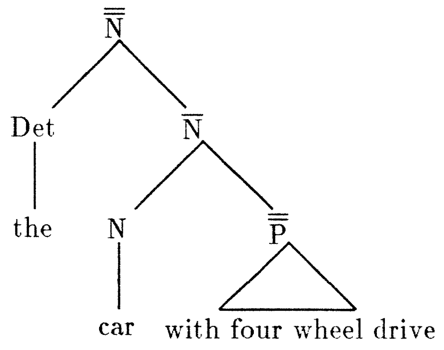


Figure 3.2: Example of Adjunction to a Noun Phrase

The four basic categories of X are characterized in terms of the features [±N] (substantive) and [±V] (predicative):

(5) [+N, +V] = A, [+N, −V] = N, [−N, +V] = V, and [−N, −V] = P

This notation allows generalizations to be made crosscategorially. For example, the two categories that assign objective case (V and P) can be referred to with the single designation [−N].

In addition to Specifiers and Complements, other modifying phrases occur in phrasal projections; these are called adjuncts:

(6) $\overline{X} \Rightarrow$ Adjunct $\overline{X}$
$\overline{X} \Rightarrow \overline{X}$ Adjunct

An example of an adjunct is *with four wheel drive* in the noun phrase *the car with four wheel drive*. Adjuncts are those phrasal elements that are not subcategorized by the head of a phrase. Since the head *car* does not subcategorize for anything, it is assumed that the prepositional phrase *with four wheel drive* has *right-adjoined* to the noun phrase so that the structure in figure 3.2 is derived. Usually adjuncts are maximal projections (*i.e.*, on the $\overline{\overline{X}}$ level).[3]

[3]In certain cases, adjuncts are X^0 nodes. For example, V-preposing left-adjoins a verb (not a verb phrase) to S. Auxiliary verbs and clitics are also allowed to left-adjoin (in this case to V). In general, as pointed out to me by Craig Thiersch (personal communication), minimal (X^0) elements are allowed to left-adjoin but not right-adjoin. By contrast, maximal (X^{max}) elements are allowed to left-adjoin *and* right-adjoin. (For example, N^{max} right-adjoins to V^{max} for the free-subject inversion phenomenon in Spanish.)

3.1.1 Constituent Order Parameter

Note that if we were to ignore adjuncts for the time being, (4) could be characterized by a single ternary branching scheme:

(7) $\overline{\overline{\text{X}}} \Rightarrow$ Specifier X Complement

This is similar to the scheme I will adopt since it lends itself more readily to incorporation of the "constituent order" parameter, which allows the order of specifier, head and complement to be permuted across languages. For example, the parameter must be set to specifier-head-complement in English, complement-specifier-head in Navajo,[4] and head-specifier-complement in Kikuyu[5] (a Bantu language of East Africa). Once the constituent order is set for a particular language, all of the $\overline{\text{X}}$ phrase-structure skeletons can be derived before parsing begins. In order to include adjuncts, (7) must be modified and combined with (6) to derive (8):

(8) $\overline{\overline{\text{X}}} \Rightarrow$ Specifier $\overline{\text{X}}$ Complement
$\overline{\text{X}} \Rightarrow$ Adjunct $\overline{\text{X}}$
$\overline{\text{X}} \Rightarrow \overline{\text{X}}$ Adjunct
$\overline{\text{X}} \Rightarrow$ X

Note that adjuncts are only allowed to occur after Specifiers or before Complements. However, there is evidence that adjuncts can occur before the specifier and after the complement in other languages:

(9) (i) ¿Qué vio el hombre?
'What did the man see?'

(ii) ... entrada del hombre a las ocho ...
'entrance of the man at eight'

In (9)(i) the verb *vio* (= 'saw') is adjoined to the clause in a position *preceding* the specifier *el hombre* (= 'the man'). In (9)(ii) the prepositional phrase *a las ocho* (= 'at eight') is adjoined to the noun phrase in a position *following* the complement *del hombre* (= 'of the man'). Thus,

[4] Hale (1973).
[5] Lightfoot (1982).

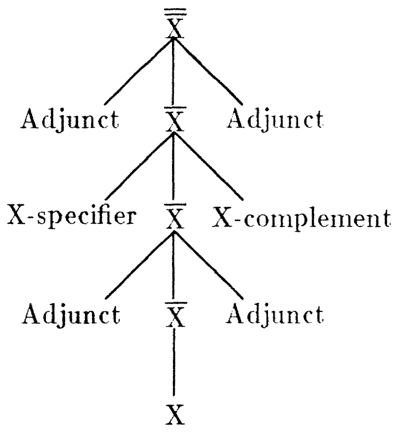


Figure 3.3: Phrase-Structure Skeleton for Specifier-Head-Complement Order

it would seem that the allowable phrase-structure for a specifier-head-complement language should be as in figure 3.3. Here, adjuncts can occur before or after the specifier, and before or after the complement, that is, they can occur "freely" (subject to constraints imposed by other modules of the theory). Thus, the constituent order parameter setting is all that is required in order to determine all of the context-free rules required for the derivation of the phrase-structure skeleton for any given language.

The context-free rules corresponding to the phrase-structure skeleton in figure 3.3 are:

(10) $\bar{\bar{X}} \Rightarrow$ Specifier $\bar{X}$ Complement
$\bar{\bar{X}} \Rightarrow$ Adjunct $\bar{X}$
$\bar{\bar{X}} \Rightarrow \bar{X}$ Adjunct
$\bar{X} \Rightarrow$ Specifier $\bar{X}$ Complement
$\bar{X} \Rightarrow$ Adjunct $\bar{X}$
$\bar{X} \Rightarrow \bar{X}$ Adjunct
$\bar{X} \Rightarrow$ X

However, the context-free rule system I will adopt is a simplified version of (10). According to Chomsky (1986a), adjunction to $\bar{X}$ is not possible; rather, adjunction is made only at the maximal (X^{max}) or zero (X^0) level. This would seem to indicate that the presence of the $\bar{X}$ level is ineffectual since the $\bar{X}$ level in (10) is required only for the introduction of adjuncts. Thus, we can now speak of just two levels, minimal (X^0 = X) and maximal ($\bar{\bar{X}} = X^{max}$). This

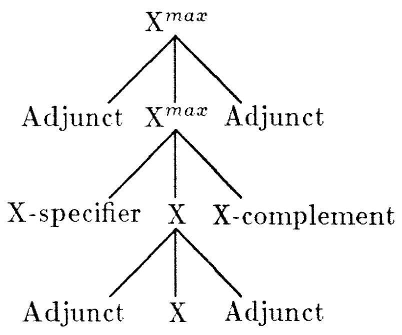


Figure 3.4: Modified Phrase-Structure Skeleton for Specifier-Head-Complement Order

new formulation reduces the 7 rules of (10) to 5 rules:[6]

(11) $X^{max} \Rightarrow$ Specifier X Complement
$X^{max} \Rightarrow$ Adjunct X^{max}
$X^{max} \Rightarrow X^{max}$ Adjunct
$X^{max} \Rightarrow$ Adjunct X
$X^{max} \Rightarrow$ X Adjunct

The phrase structure skeleton corresponding to (11) is in figure 3.4.

Note that the context-free rules in (11) are no longer needed once the phrase-structure skeleton 3.4 has been constructed. Each time a word is encountered in a surface sentence, a phrase-structure projection (where X corresponds to the lexical category of the word) is established. The specifier and adjuncts are optional, and the complement consists of arguments projected by the subcategorization of the head.

So far, the scheme presented here includes specifier-head-complement languages only. However, the order of these three constituents may be permuted for other languages. There are a total of 6 (= 3!) permutations of specifier, head and complement; the phrase-structure skeletons pictured in figure 3.5 correspond to these 6 permutations (3.5 (a) is the same as 3.4). The abbreviations Spec and Comp[7] refer to Specifier and Complement respectively.

[6] In general (for ease of notation), $\overline{\overline{X}}$ or XP will be substituted for X^{max}, and X will be substituted for X^0.

[7] The abbreviation Comp for Complement is not to be confused with the abbreviation Comp for Complementizer. In general, Complement will be abbreviated as Comp and Complementizer will be abbreviated as C.

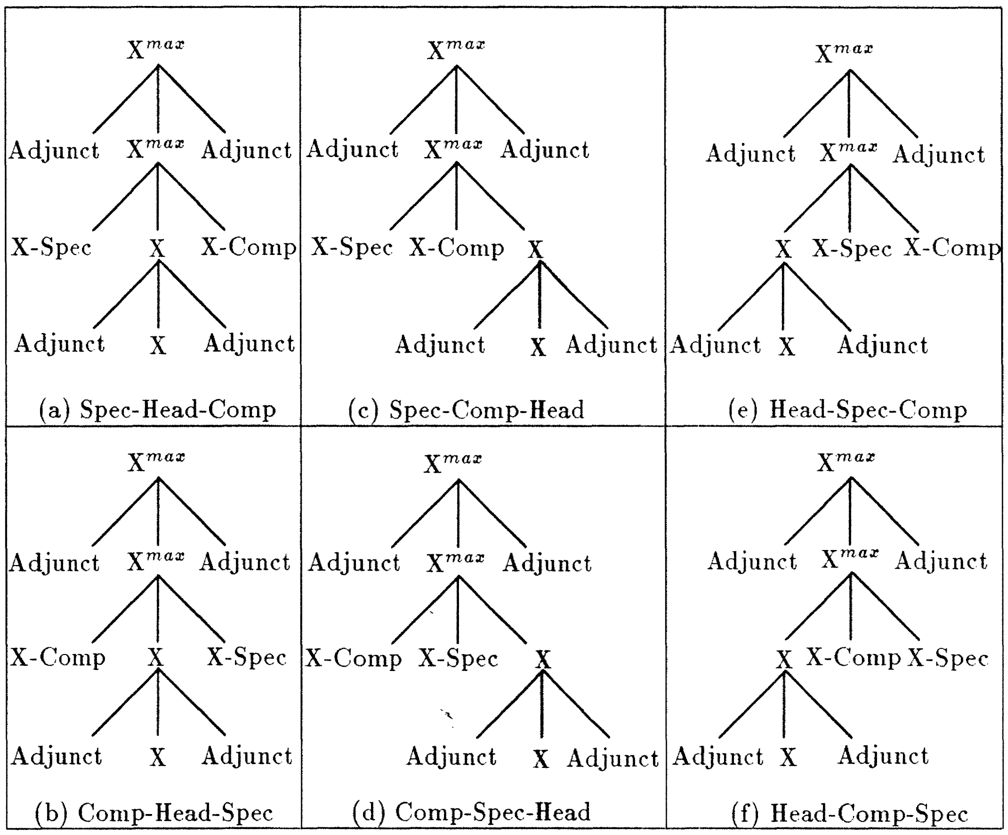


Figure 3.5: Six Permutations of the "Constituent Order" Parameter

3.1.2 Infl and Comp

In addition to the lexical heads N, V, P and A, Chomsky (1981b) includes two other categories: Infl (= I), an inflection node containing tense and agreement information; and Comp (= C), a complementizer (or head) of a clause.[8] The following rules are given for introducing these two categories:

[8]Throughout this report, $\overline{\overline{C}}$ (= C^{max}) and $\overline{S}$ will be used interchangeably, as will $\overline{\overline{I}}$ (= I^{max}) and S. That is, C is the head of $\overline{S}$, and I is the head of S.

(12) $\bar{\bar{\mathrm{I}}} \Rightarrow \bar{\bar{\mathrm{N}}}\ \bar{\mathrm{I}}$

$\bar{\mathrm{I}} \Rightarrow \mathrm{I}\ \bar{\bar{\mathrm{V}}}$

$\bar{\bar{\mathrm{C}}} \Rightarrow$ *Wh-phrase* $\bar{\mathrm{C}}$

$\bar{\mathrm{C}} \Rightarrow \mathrm{C}\ \bar{\bar{\mathrm{I}}}$

Adapting (12) to the ternary branching scheme, we have:

(13) $\mathrm{I}^{max} \Rightarrow$ I-Specifier I I-Complement

$\mathrm{C}^{max} \Rightarrow$ C-Specifier C C-Complement

Here I-Specifier and C-Specifier are allowed to vary from language to language. In the case of English, I-Specifier is N^{max} and C-Specifier is (optionally) *Wh-phrase*. The choice of Complement for I and C is dictated by the lexical entry corresponding to the head. For example, the English modal *would* (of category I) selects V^{max} as a complement, and the English complementizer *that* (of category C) selects I^{max} as a complement. In the case where the head is not overt (*e.g.*, neither C nor I is overt in *I ate pie.*), complement selection can be determined via a parameter setting (*e.g.*, in English I always selects V^{max} and C always selects I^{max}). So (13) follows the more general scheme (7), in which the Specifier of the head is determined by a parameter setting (and may be optional), and the Complement of a head is determined by the subcategorization information associated with the head (or a parameter setting when the head is not overt). Consequently, the only machinery needed to embody $\overline{\mathrm{X}}$-Theory for English is the phrase-structure skeleton 3.3, with X = A, P, V, N, C or I.[9] Superimposed onto this machinery are parameter values and subcategorization information, that determine the exact fit of the $\overline{\mathrm{X}}$ phrase-structure skeleton to each phrase of a sentence.

The lexicon and $\overline{\mathrm{X}}$-Theory come together at the level of D-structure before any transformations apply, thus generating basic phrase-structures upon which other modules operate. The theory of $\overline{\mathrm{X}}$ is crucial in the translation model since it provides the foundation on which the parsing mechanism is built: it allows the characterization of structural variation across languages, and it provides the basis for application of constraints and well-formedness conditions

[9]More generally, the phrase-structure skeletons of figure 3.5 comprise the complete $\overline{\mathrm{X}}$ machinery required to handle all languages.

imposed by principles of other GB modules.

3.2 θ-Theory

In contrast to $\overline{\text{X}}$-Theory, θ-Theory holds at the level of D-structure, S-structure and Logical Form. The central notion associated with θ-Theory is "argument of." The fundamental task of θ-Theory is to account for the relation between verbs and their arguments. θ stands for *thematic*; *θ-roles* are assigned to different arguments of a verb according to the semantic description of the verb. For example, the verb *love* in *John loves Mary* assigns the role of *agent* to *John* (the external argument of the verb) and the role of *patient* to *Mary* (the internal argument of the verb). Both internal and external arguments are specified in the subcategorization frame of the verb in the lexicon, and the assignment of θ-roles is determined from this information.

3.2.1 θ-Criterion and Projection Principle

Noun phrases must be matched up one to one with arguments of a verb. The θ-Criterion (as stated by Chomsky, 1981b) ensures that this bijection holds:

(14) *θ-Criterion*:

Each argument bears one and only one θ-role, and each θ-role is assigned to one and only one argument.

The Projection Principle then ensures that this Criterion holds at D-structure, S-structure, and Logical Form:

(15) *Projection Principle*:

The θ-Criterion holds at D-structure, S-Structure, and Logical Form.

Taken together, the Projection Principle, θ-Criterion, lexicon, and $\overline{\text{X}}$ templates comprise all that is required to generate phrase structure without the need for specific context-free rules. The phrase structure skeletons provide underspecified phrase structure, and lexical items determine (through subcategorization information consisting of internal and external arguments along with their corresponding θ-markings) the elements required to fill out missing constituents of the phrase.

3.2.2 Canonical Structural Realization and Visibility Condition

Within θ-theory the mapping between θ-roles and their structural realization must be parameterized. According to Chomsky (1986b), the lexicon presents, for each lexical item, the "selectional properties" of heads of constructions (N, V, A, P). The two types of selectional properties are *s-selection* ("semantic selection," *i.e.*, the roles, such as *agent* and *goal*, that are assigned to internal and external arguments) and *c-selection* ("categorial selection," *i.e.*, the categories, such as *N* and *C*, of the internal and external arguments). Since c-selection is redundant (*e.g.*, s-selection of a *patient* automatically implies c-selection of an NP, *etc.*), the lexicon may be restricted to s-selection. However, in order to determine the mapping between a semantic category S and a syntactic category C, it is necessary to provide a function, call it "canonical structural realization" (CSR) (in accordance with Chomsky (1986b)), that characterizes the mapping. This function is parameterized so that the CSR of a particular semantic category is allowed to vary across languages. For example, CSR(patient) = NP in English, but CSR(patient) = PP in Spanish:

(16) (i) see: external = agent, internal = patient
ver: external = agent, internal = patient

(ii) John saw $[_{\text{NP}}$ Mary$]$
Juan vio $[_{\text{PP}}$ a María$]$

Although the lexical entries for "see" and "ver" are identical, the complements are not structurally identical; however, this structural distinction need not be specified in the lexicon. Thus, not only is c-selection (*i.e.*, context-free rules) eliminated from syntactic phrase structure, but it is eliminated from the lexicon as well. The lexicon need only specify the s-selection, and the parameterized CSR may then apply to determine the structural realization. Note that in (16) the θ-role *patient* is assigned to $[_{\text{NP}}$ María$]$, not $[_{\text{PP}}$ a María$]$; in general, the structural entity that is c-selected is either the NP to which θ-role is assigned, or the phrase containing the NP to which θ-role is assigned. The only requirement is that the NP be case-marked before θ-role be assigned. This condition is called the Visibility Condition:

(17) *Visibility Condition*:

An NP is visible for θ-marking only if it is assigned case.

As we will see in section 3.4.1, objective case is assigned to an object of a preposition. Since *María* is the object of the preposition *a*, objective case is assigned, at which point θ-role assignment triggers and *María* receives *patient* θ-role.

3.2.3 Clitic Doubling and θ-role Transmission

In addition to parameterization of the CSR function, another parametric variation of θ-theory is within the principle of θ-role transmission. In Spanish, the phenomenon of clitic doubling is relevant to this parametric variation. A clitic is a pronominal constituent that is associated with a verbal object. In contrast to movement theories[10] and clitic placement/deletion theories,[11] Jaeggli (1981) proposes that clitic pronouns are characterized as verbal objects and are base generated *in clitic position*[12] by the rule:

(18) $\overline{\overline{\text{V}}} \Rightarrow \text{clitic+V}\ \overline{\overline{\text{N}}}$

For example, the clitic *le* in the following sentence is base-generated to the left of the verb *regalé*:

(19) Le regalé un libro.
'I gave {him,her} a book.'

The phenomenon of clitic doubling is defined in terms of the pair <*clitic, lexical NP*> where the clitic must agree in number, person, and gender with the lexical NP. Examples of clitic doubling are the following:

(20) (i) Le entregué la carta a él. (pronominal indirect object doubling <*le, él*>)
'I delivered the letter to him.'

(ii) Lo vimos a él. (pronominal direct object doubling <*lo, él*>)
'We saw him.'

[10] See Kayne (1975) and Quicoli (1976).
[11] See Rivas (1977) and Strozer (1976).
[12] Clitic adjunction is assumed to be part of the $\overline{\text{X}}$ module. However, the presence or absence of clitics for a particular language is determined by a parameter setting associated with θ-theory.

(iii) Lo vimos a Guille. (nonpronominal direct object doubling <*lo, Guille*>)[13]

'We saw him.'

In (20), the clitic actually stands for an NP that does not yet have a θ-role. Thus, in order to satisfy the θ-Criterion, a parameter of variation is required for θ-role transmission. Jaeggli proposes that clitics always bear a particular θ-role and that there is a θ-role transmission rule that will supply θ-roles to object NPs that are doubled:

(21) $[\text{CL } +case_i +\theta_j]$ ... $[\text{NP } +case_i] \Rightarrow [\text{CL } +case_i +\theta_j]$ ... $[\text{NP } +case_i +\theta_j]$

Note that in order for this transmission rule to trigger, the clitic and NP must have the same case. (A description of Case Theory is given below.) (21) allows a doubled NP object to receive θ-role. If a clitic is not present, a θ-role is assigned in the usual fashion, (*i.e.*, from the verb that subcategorizes for the NP). Thus, for languages that allow clitics, clitic doubling must be available as a parameter of variation to the θ-role transmission principle of θ-Theory. This is important in a model of translation since languages that allow clitics could not be analyzed without such a parameter of variation.

3.3 Government Theory

Government is a central notion to several of the modules including θ, Case, Trace and Binding; thus, in terms of figure 3.1, government applies at all levels. Within the context of translation, government is important because it is a key determinant of sentence structure possibilities: as a parameterized principle, government accounts for the possibility of null subjects, of clitics, and of V-preposing (these phenomena will be discussed shortly). A familiar example of government in English is that a verb governs its object. Government is defined as follows (adapted from van Riemsdijk and Williams (1986)):

(22) α (= V, N, A, P or AGR)[14],[15] *governs* β if:

[13]Doubling a nonpronominal direct object is acceptable only in certain dialects, for example, in the River Plate Spanish dialect of Argentina, Paraguay, Uruguay and Chile.

[14]AGR is the part of the Infl node that contains agreement features. It is necessarily accompanied by the $+tns$ feature.

[15]In the model presented here, the choice of governors is determined from a parameter setting associated with Government Theory since the categories in (22) (V, N, A, P and AGR) may not be universal across all languages.

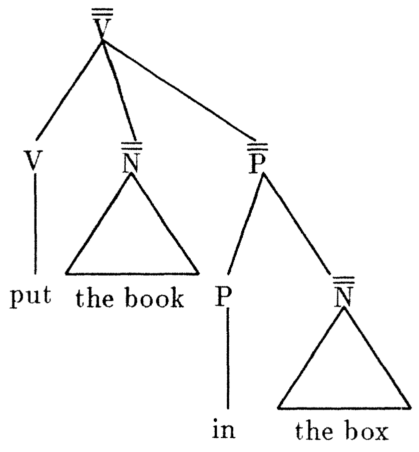

$\overline{\overline{V}}$
V
$\overline{\overline{N}}$
$\overline{\overline{P}}$
put
the book
P
$\overline{\overline{N}}$
in
the box

Figure 3.6: Example of Government Relation

(i) α *c-commands* β, and

(ii) $\overline{\overline{\alpha}}$ is the smallest maximal projection containing β, and

(iii) β is contained in $\overline{\overline{\alpha}}$

The notion of c-command (Chomsky 1981b) is defined as follows:

(23) α *c-commands* β if:

(i) α does not contain β, and

(ii) all maximal projections that contain α also contain β

Figure 3.6 shows an example of the government relation: the verb *put* governs *the book*, but not *the box* since the node $\overline{\overline{P}}$ (not $\overline{\overline{V}}$) is the smallest maximal projection containing *the box*.

Several Spanish-English differences reveal "type of government" as a possible parameter of variation. These differences show up in Case Theory with respect to case assignment requirements, and in Trace Theory with respect to the Empty Category Principle (ECP). The next two sections explore these differences, the parameters of variation that are required, and the relevance of the parameters to the translation model.

3.4 Case Theory

Abstract case (e.g., nominative, objective, *etc.*) is assigned to a noun phrase at S-structure according the structural positioning of the noun phrase with respect to other elements. The notion of government is relevant to case assignment since an element assigns case only if it is a governing case-assigner. For example, in English, a preposition assigns objective case to the object it governs, as in *with her* as opposed to **with she* (here case shows up overtly). A well-defined theory of Case is necessary for translation because it provides an explanation for many of the distinctions between languages, including the existence of clitics in some languages, but not in others.

3.4.1 Case Assignment and Case Filter

Case is assigned as follows:

(24) (A) Objective case is assigned to the object governed by transitive P.

(B) Objective case is assigned to the object governed by transitive V.

(C) Nominative case is assigned to the subject governed by [I +*tns*].[16]

The Case Filter is a principle that requires all lexical NPs to have case:

(25) *Case Filter*:

$*$ NP, where NP has no case

For example, since V and I (with +*tns* features) are case assigners, the following sentence does not violate the Case Filter (because all of the NPs have been assigned case):

(26) $[_S$ I $[_I$ [+PRS 1S]] $[_{\overline{\overline{V}}}$ believe him]] ('I believe him')

Here, *believe* assigns objective case to *him* and +*tns* (realized as the +PRS feature) assigns nominative case to the pronoun *I*, so the Case Filter is satisfied. On the other hand, the Case Filter is violated in:

(27) $[_S$ I $[_I$ [+PRS 1S]] $[_{\overline{\overline{V}}}$ believe $[_{\overline{\overline{I}}}$ him $[_{\overline{\overline{V}}}$ to go]]]] ('I believe him to go')

[16] Since AGR is necessarily accompanied by the +*tns* feature, [I +*tns*] is used interchangeably with AGR.

In (27) nominative case is assigned to *I* as in (26), but *him* cannot receive objective case from *believe* because $\bar{\bar{I}}$ blocks *believe* from governing *him*. Furthermore, *to go* is not tensed, so *him* cannot receive nominative case. The Case Filter provides a principled account of the ill-formedness of sentences like (27) and also of several other types of sentences including those containing certain forms of clitic doubling (as we will see in the next section).

3.4.2 Clitic Doubling and Choice of Government

Clitic doubling figures in case assignment parametric variation. Since case assignment is dependent on the notion of government, one might ask (given rule (18)) whether a clitic is governed. If the clitic is not governed, it cannot be assigned case; consequently sentences (19) and (20) will be ruled out by the Case Filter (25). In order to avoid this wrong prediction, Jaeggli proposes that case assignment be parameterized so that it is dependent on *s-government* rather than on the less refined notion of *government* (henceforth referred to as *c-government*) as defined in (22).

In contrast to *c-government*, which is defined in terms of c-command and maximal projections, *s-government* is defined with respect to a strict subcategorization feature. *S-government* is a unique pairing of c-governed elements of a verb to the subcategorization features of that verb. In Spanish, objective case is assigned to an NP that is *s-governed* by the verb, whereas in English, objective case is assigned to an NP that is *c-governed* by the verb. The following example illustrates the necessity of this parameterization:

(28)∗ Lo vimos Guille.

In (28), the accusative clitic *lo* absorbs s-government and *Guille* remains ungoverned. Thus, *Guille* does not receive case and the sentence is ruled out by the Case Filter. On the other hand, (20)(iii) (the *correct* version of (28)(i)) *is* acceptable. This is because the quasi-preposition *a* assigns objective case (with a +ACC feature marking) to *Guille* and the sentence is not ruled out. Jaeggli proposes that the *a* is introduced immediately after the base by the following rule:

(29) $\phi \Rightarrow a$ / __$[NP,$ +accusative$]$[17]

[17]This is similar to the *of*-insertion rule proposed by Chomsky (1981b) which allows for the correspondence between *destroy the city* and *destruction of the city*. In the former, case is assigned to *the city* by the verb *destroy*, whereas in the latter, case is assigned to *the city* by the preposition *of*.

Note, however, that in sentence (20)(i), there is no particle *a* before the accusative object *la carta*, but case is still assigned successfully. The reason is that the dative clitic *le* receives the +DAT feature from the verb matrix, while the direct object *la carta* receives the +ACC feature from the verb matrix. Since there is no duplication of feature assignment, both NPs are s-governed by the verb, and thus, both are assigned objective case.

It is necessary to parameterize case assignment in (24) so that there is an explanation for the existence or non-existence of clitics across languages. In languages like Spanish, s-government determines objective case assignment by a verb, whereas in English c-government is the determinant of objective case assignment by a verb. The parameter of variation, then, is a setting that determines whether case assignment is dependent on s-government or c-government. Thus, the Spanish and English parameter values for (24) (A), (B) and (C) are set as follows:

(30)

Language	*Government for (A)*	*Government for (B)*	*Government for (C)*
Spanish	C-government	S-government	C-government
English	C-government	C-government	C-government

The parameterization of the Case module is necessary in the translation model for characterization of the distinction between clitic and non-clitic languages. Because of this parameterization, language-specific rules are not required for this characterization.

3.5 Trace Theory

A trace is an empty sentence position that is either base-generated or left behind when a constituent has moved. Principles within this theory apply at the level of S-structure (*i.e.*, *after* movement has taken place and traces are left behind).[18] The principle of empty categories (ECP)[19] requires that traces be *properly governed*, where proper government is defined as follows (taken from van Riemsdijk and Williams, (1986)):

(31) *Proper government*:

α properly governs β if and only if

[18] There is evidence that principles of Trace Theory, namely ECP, apply at the level of Logical Form. (See van Riemsdijk and Williams (1986) for a discussion of this possibility.) However, the application of ECP at Logical Form is not relevant to the discussion presented here since the translation model includes only those principles that apply at D-structure and S-structure (*i.e.*, within the realm of syntax, not semantics).

[19] See Kayne (1981) and Jaeggli (1980).

(i) α is a *governing node*, and

(ii) α governs β (in the sense of (22))

A *governing node* or *governor* is a minimal lexical category (*i.e.*, V, N, A, P) or NP_i, where $\beta = NP_i$. However, we shall see shortly that the choice of *governing node* is indirectly subject to parametric variation due to a property that distinguishes null-subject languages from non-null-subject languages. A well-defined theory of traces is important for translation because it provides an explanation for the distinctions between languages that allow null subjects and V-preposing (like Spanish) and other languages.

In general, there are four types of empty categories, each of which can be characterized in terms of the features $\pm$*pron* (pronominal) and $\pm$*ana* (anaphoric or referential):

(32)

NP-trace	[–pron,+ana]
Wh-trace (or variable)	[–pron,–ana]
pro[20]	[+pron,–ana]
PRO	[+pron,+ana]

An extended version of the above-mentioned ECP includes all of these empty categories (not just NP-trace and *Wh*-trace). The extended version is as follows:

(33) *Extended ECP*:[21]

An empty category is trace if and only if it is properly governed and PRO if and only if it is ungoverned.

Thus, all empty categories except PRO must be properly governed.[22] This formulation is not restricted to the empty NPs in (32); it may apply to other empty categories as well. For example, in Spanish there are several types of verbal traces. (The choice of traces for a language is specified by a parameter value to the Trace module.) As it stands, (33) requires that verbal traces be properly governed. However, as noted in van Riemsdijk and Williams (1986), this

[20]The empty category *pro* corresponds to a a lexical pronoun (*i.e.*, I, you, he, she, it, or they). Unlike its empty anaphoric counterpart (PRO), it must be properly governed.

[21]Taken from Chomsky (1981).

[22]This conflicts with Sharp's claim that *pro* is not subject to the ECP, but rather depends on Condition (B) of the Binding Theory (see section 3.6.2). The Null Subject Parameter analysis by van Riemsdijk and Williams (1986) shows that *pro* is indeed properly governed. Section 3.5.1 gives the details of this analysis.

generalization may not be warranted (in fact, verbal traces provide a counterexample to (33)). I will assume then that (33) holds only for empty categories with the [−V] feature (see (5)). For example, PP-traces (in languages which allow them) must also be properly governed. The term *ECP* will henceforth refer to the Extended version of the ECP.

3.5.1 Null Subject Parameter and ECP

According to the analysis of the null subject parameter introduced by van Riemsdijk and Williams (1986), proper government must be parameterized so that the choice of proper governors is allowed to vary from language to language. We shall look at the analysis given by van Riemsdijk and Williams and discuss their conclusions.

In Spanish, as in Italian, Greek, and Hebrew, morphology is rich enough to make the subject pronouns redundant and recoverable. Thus, we can have the sentence:

(34) Hablé con ella.
     '(I) spoke with her.'

Since the inflection on the verb is first person singular, the subject pronoun *yo* (=I) is optional.

The formulation of the null subject or *pro-drop* parameter by van Riemsdijk and Williams is motivated by the observation that subjects are missing in a variety of constructions, not just in cases like (34).[23] These constructions do not appear in many other languages (*e.g.*, English, *etc.*); thus, there must be a parameter which will account for the distinction between pro-drop and non-pro-drop languages. The assumption is that some factor other than a coindexed *Wh*-phrase or its trace must properly govern the null entity in each of these constructions. The *pro-drop parameter*, then, is a minimal binary difference that does or does not allow this factor to properly govern subject position.

The proposed factor for proper government of a subject is the $[_{\mathrm{I}}\ [+tns]\ \mathrm{AGR}]$ node. The idea is that the features of AGR must agree with the subject NP and are realized on the verb. The agreement relation is expressed by coindexing AGR and the subject:

(35) COMP $\mathrm{NP}_i\ [_{\mathrm{I}}\ [+tns]\ \mathrm{AGR}_i]\ \overline{\overline{\mathrm{V}}}$

[23] The constructions that allow missing subjects include free inversion, V-preposing, embedded clauses, and *Wh*-islands.

The assumption is that AGR$_i$ c-commands NP$_i$ and hence can govern it. However, in non-pro-drop languages, such as English, AGR counts as governor for case marking and binding, but does not count as proper governor for ECP. The reason AGR is allowed to be a proper governor in pro-drop languages is that it has features for gender, number and person (*i.e.*, AGR is said to be *rich* in pro-drop languages). The obvious conclusion is that AGR$_i$ is allowed to be a proper governor when it has nominal features (*i.e.*, [+N,–V]).

Thus, the pro-drop parameter is relevant to the classification of governors. The following two definitions from van Riemsdijk and Williams (1986) make this classification more clear:

(36) *Governors*:

(i) V, A, N or P

(ii) $[_{\text{I}}$ [+*tns*] AGR]

(iii) N$_i$ or NP$_i$, where the governed element is NP$_i$

(37) *Proper governors for ECP:* (36)(i) and (36)(iii)

The parameter of variation that distinguishes Spanish from English, then, is the ability of AGR to take on a noun-like status and thus become a proper governor by (36)(iii).

The underlying structure of (34) is the following:

(38) $[_{\text{S}}$ NP$_i$ $[_{\text{I}}$ [+past] $[_{\text{AGR}}$ +1s$]_i$] $[_{\overline{\overline{\text{V}}}}$ hablar con ella]][24]

The status of the empty subject, then, must be $[_{\text{NP}}$ e$]_i$.[25] (Note that it cannot be PRO since it is governed.) This empty category (sometimes called "little pro" or *pro*) may either be base generated or positioned by movement. Since NP$_i$ (= *pro*) is properly governed by AGR$_i$, ECP is satisfied. In English, AGR could not be coindexed with an NP; thus, pro-drop is not possible. This distinction between Spanish and English is relevant in the context of the translation model in that there must be a parameter that is set according to whether or not

[24] The feature 1s stands for first person singular.

[25] Binding conditions rule out the possibility that the empty subject is a trace. In particular, the subject is neither A-bound (ruling out the NP-trace possibility) nor $\overline{\text{A}}$-bound (ruling out the *Wh*-trace possibility). Section 3.6.1 describes A-binding and section 3.6.2 describes $\overline{\text{A}}$-binding.

AGR is *rich* in the languages undergoing translation. Otherwise there would be no way to account for the phenomenon of pro-drop.

Note that no additional principles have been stipulated: both proper government and the choice of governing nodes are still the same across all languages; all that is required is the value of a single parameter (pro-drop) in order to account for distinctions in government properties that allow or disallow null subjects. The parameter setting approach is more desirable than a rule-based approach since it accounts for several types of null subject constructions without requiring several independently motivated rules.

3.5.2 V-Preposing and ECP

Proper government can be further parameterized as evidenced by an analysis of V-preposing by Torrego (1984). In Spanish, V-preposing is an obligatory rule that occurs only in clauses in which *Wh*-movement takes place. The assumption is that V-preposing moves the verbal element to the left of the subject, adjoining it to S leaving behind a trace of the verb t_v:[26]

(39) (i) ¿Con quién vendrá *Juan* hoy?
'With whom will John come today?'

(ii) $*$¿Con quién *Juan* vendrá hoy?

The structure of (39)(i) is as follows:

(40) $[_{\bar{S}}\ [_{\bar{\bar{P}}}$ Con quién$]_i\ [_S$ [+FUT 3S] venir $[_S$ Juan $[_{\bar{\bar{V}}}\ t_v\ e_i$ hoy$]]]]$

Here the trace of the prepositional argument of the embedded verb is not properly governed since t_v is not a governing node; however, Torrego introduces a variation in the definition of "proper government" so that a trace is allowed to be properly governed when it is part of a *chain*, all of whose elements are governed.[27] The revised version of proper government, then, is the following:[28]

[26]As described by Torrego (1984), V-preposing moves a V projection out of $\bar{\bar{V}}$, adjoining it to the right of COMP under a new S node.

[27]A *chain* is a record of movement. For example, in the sentence *What$_i$ did John see e_i?*, the chain ($What_i$, e_i) characterizes the movement of $What_i$ from the position occupied by the trace e_i.

[28]This version is a modification of the revision presented by Torrego (1984). Part (ii) in Torrego's version is *hard-wired*, *i.e.*, it is not parameterized to allow for variation between pro-drop and non-pro-drop languages. In

(41) *Proper government*:

α *properly governs* β if and only if

(i) α is a governing node and α governs β (in the sense of (22)), or

(ii) β satisfies *chain conditions*, if applicable.

The chain conditions are defined as follows:

(42) *Chain Conditions for* β*:*

β must be in a chain $= (a_1 \ldots a_n)$ such that $n > 1$ and for each i, a_i is governed.

With this new definition of proper government, (39) (i) no longer violates ECP. In Spanish, chain conditions are applicable (as determined by a parameter setting), so (42) applies. The trace e_i is governed by t_v and is part of a chain containing the moved element $[_{\bar{P}}$ con quién$]_i$ which is also governed. Thus, according to the new formulation of proper government, the trace is properly governed, and ECP is not violated.

In English (42) does not apply. Thus, ECP is satisfied only if (41)(i) is satisfied. (This is equivalent to the effect of applying the original definition (31).) In fact, chain conditions are not required for English since V-preposing is not available in non-Romance languages. In English, Subject-Aux Inversion (SAI), not V-preposing, is triggered by *Wh*-movement. (The assumption is that there is a movement parameter that dictates that *Wh*-movement triggers V-preposing in Spanish, and SAI in English.) SAI generates an auxiliary to the left of the subject without leaving a trace behind. Thus, the English equivalent to (40) is:

(43) $[_{\bar{S}}$ $[_{\bar{P}}$ With whom$]_i$ $[_S$ [+FUT 3S] will $[_S$ John $[_{\bar{V}}$ come e_i today$]]]]$

Because the main verb does not move and may still properly govern the empty category to its right, the chain analysis is not required in order to account for proper government of the object trace. In the context of the translation model, this finding is important because it implies that, in order to account for this distinction between V-preposing and SAI, there must be a parameter setting that dictates whether or not proper government is defined in terms of chains.

the version proposed by Torrego, the chain conditions are tested regardless of whether the language is a pro-drop language (even though non-pro-drop languages do not require the chain conditions). The effect of both versions is the same, but the version presented here allows a more efficient implementation since the chain conditions are only tested for a *handful* of languages (namely, the pro-drop languages), not for *all* languages.

As in the null subject analysis, no language-specific stipulations have been made to account for proper government variations associated with V-preposing; rather, the distinction in government properties across languages is determined by a single ECP parameter setting (chain conditions), thus distinguishing between languages that use V-preposing, SAI, or some other type of verbal movement out of a clause.

3.6 Binding Theory

Binding Theory is concerned with the coreference relations among noun phrases. According to van Riemsdijk and Williams (1986), Binding Theory applies at the level of Logical Form. However, there is evidence that Binding Theory may also apply at the level of S-structure; thus, I will adopt the position that Binding principles are available both at S-structure and at Logical Form.

3.6.1 Binding Conditions for Overt Noun Phrases

There are three types of noun phrases: anaphors (*e.g.*, *himself*, *each other etc.*), pronominals (*e.g.*, *he*, *them*, *her etc.*) and R-expressions (referential expressions, including *Mary*, *table*, *etc.*). The three Binding conditions (from Chomsky (1981b)) corresponding to these noun phrases are:

(44) *Binding Conditions*:

(A) An anaphor must be *bound* in its *governing category*.

(B) A pronoun must be *free* in its *governing category*.

(C) A lexical NP must be *free*.

The definitions of *bind* and *governing category* (not to be confused with *governing node* of Case and Trace Theory) are as follows:

(45) α *binds* β if:

α and β have the same index and α c-commands β.

(If there is no such α for a given β, then β is said to be *free*.)

(46) *Governing category*:

γ is the governing category for δ if and only if γ is the minimal category containing δ, a governor of δ, and a *subject* accessible to δ.

Formally, a *subject* of a clause is AGR$_i$ if there is one, otherwise the NP$_i$ immediately dominated by S or NP.[29]

Some examples will clarify the above definitions:

(47) (i) $[_{S_1}$ John$_i$ felt that $[_{S_2}$ $[_{\overline{\overline{N}}}$ his friends$]$ liked him$_{i/j}]]$

(ii) $*[_{S_1}$ John$_i$ felt that $[_{S_2}$ $[_{\overline{\overline{N}}}$ his friends$]_i$ liked him$]]$

(iii) $*[_{S_1}$ John$_i$ felt that $[_{S_2}$ $[_{\overline{\overline{N}}}$ his friends$]$ liked himself$_{i/j}]]$

(iv) $[_{S_1}$ John felt that $[_{S_2}$ $[_{\overline{\overline{N}}}$ his friends$]_i$ liked themselves$_i]]$

(v) $*[_{S_1}$ John felt that $[_{S_2}$ $[_{\overline{\overline{N}}}$ his friends$]_i$ liked them$_i]]$

In each of these sentences, the governing category of the inner clausal object is S$_2$ since the accessible subject is *his friends*. In (47)(i) both the *him$_i$* and *him$_j$* interpretations are allowed since the pronoun is free inside S$_2$. On the other hand, (47)(ii) violates Binding Condition (C) since $[_{\overline{\overline{N}}}$ his friends$]_i$ and $[_{\overline{\overline{N}}}$ John$]_i$ are not free. (47)(iii) violates Binding Condition (A) in both the *himself$_i$* and *himself$_j$* interpretations since the anaphor is not bound within S$_2$. By contrast, (47)(iv) is well-formed since the anaphor *themselves$_i$* is bound within S$_2$. Finally, (47)(v) contains a violation of Binding Condition (C) since *them$_i$* is bound within S$_2$.

3.6.2 Binding Conditions for Empty Noun Phrases

Recall that empty categories are characterized in terms of the features $\pm$*pron* and $\pm$*ana* (see (32)). From this classification it can be determined that NP-trace is subject to condition (44)(A) (since it is anaphoric), *Wh*-trace is subject to condition (44)(C) (since it is neither pronominal nor anaphoric), *pro* is subject to condition (44)(B) (since it is pronominal) and

[29] Actually, according to Wexler and Manzini (1986) the definition of *governing category* is parameterized. In languages other than English, the existence of a subject may not be sufficient for identification of a governing category. The proposed modification of the definition of governing category of an element is: *the minimal category which contains the element and (a) has a subject, or (b) has an Infl, or (c) has a tns, or (d) has an indicative tns, or (e) has a root tns.* The parameterization of governing category is beyond the scope of this report. (See Wexler and Manzini (1986) for more details.) In the model presented here, the governing category is assumed to fall within classification (a) (*i.e.*, it has a subject as defined in (46)).

PRO is not subject to any of the binding conditions (since it is both pronominal and anaphoric, and therefore must not have a governing category).[30] Thus, the NP-trace left by movement of an NP to an argument position (henceforth referred to as A-position) must be bound within its governing category; the *Wh*-trace left by movement to a non-argument position (henceforth referred to as $\overline{\text{A}}$-position) must be free; *pro* must be free in its governing category; and PRO must be ungoverned.

As it stands, Binding Condition (44)(C) does not allow for coreference of a *Wh*-trace (henceforth referred to as a *variable*) with a moved *Wh*-phrase (henceforth referred to as an *operator*). For example, in the sentence $What_i$ *did John eat* e_i*?*, the binding of e_i to $What_i$ is ruled out. In order to admit the operator-trace relationship, we must distinguish between two kinds of binding: A-binding, which refers to referential-dependence (*i.e.*, binding from an A-position); and $\overline{\text{A}}$-binding, which refers to the operator-variable relationship (*i.e.*, binding from an $\overline{\text{A}}$-position). What is meant by Binding Condition C then is that an R-expression must not be bound by an element in an A-position. Chomsky (1986b) proposes the following modification to (44)(C):

(48) An r-expression must be A-free (in the domain of its operator).[31]

With this modification, the binding conditions cover all the cases of A-binding. All that is left is a principle that enforces the requirement that a variable be bound to an operator in $\overline{\text{A}}$-position:

(49) A variable must be $\overline{\text{A}}$-bound.

Binding Theory is relevant to the translation model in two ways: (a) at the level of Logical Form, the referential-dependencies must be determined in order to derive the full "meaning" of a sentence; and (b) at the level of S-structure, antecedent-trace relations must be determined (and checked against A-binding and $\overline{\text{A}}$-binding constraints) in order to derive the D-structure form of a sentence. This report focuses on the syntactic level (S-structure and D-structure), not the semantic level (Logical Form); thus, the primary emphasis with respect to Binding Theory will be on antecedent-trace relationships.

[30] From this is derived the *PRO-theorem* which states that PRO must be ungoverned. This condition is included in the Extended ECP (33).

[31] The domain of an operator is the minimal phrase containing it.

3.7 Bounding Theory

The single rule Move-α, α being an arbitrary category, relates D-structure to S-structure. The restriction placed on this rule is the condition of *Subjacency*[32] which prohibits movement beyond more than one bounding node. The choice of bounding node is allowed to vary across languages. For example, NP and S are bounding nodes for English. Thus, we are unable to say:

(50)* Who did you wonder whether went to school?[33]

The reason (50) is ruled out is that the operator *who* has moved (from the subject position preceding the verb *went*) beyond two S nodes (into the Comp position of the matrix clause).

3.7.1 V-preposing and Choice of Bounding Nodes

Torrego suggests that V-preposing proves relevant to determining the choice of bounding node for movement in Spanish. In contrast to English, NP and $\overline{\text{S}}$ are bounding nodes in Spanish. Torrego's analysis of inversion rules out the possibility that S is a bounding node in Spanish (as it is in English).[34] Consider (51):

(51) ¿Qué libro dice *María* que *Ana* le ha regalado?
'What book does Mary say that Ann has bought her?'

Because no inversion has taken place in the innermost embedded clause, it must be the case that the operator *Qué libro* has skipped a cycle during *Wh*-movement (recall that inversion is obligatory in clauses in which *Wh*-movement takes place); that is, the operator has moved over two S nodes. However, it turns out that Spanish allows no more than one $\overline{\text{S}}$ cycle to be skipped. Thus, Torrego, backed by Rizzi (1978), suggests that $\overline{\text{S}}$, but not S, is a bounding node in Spanish.

[32] Chomsky (1977).

[33] If *who* is spoken emphatically, this sentence can almost be understood as an echo question corresponding to the statement *I wondered whether John went to school.*

[34] According to Rizzi (1978) S is not a bounding node in Italian. However, in order to show this, Rizzi makes an appeal to the fact that Italian allows *Wh*-Movement out of a clause introduced by a *Wh*-phrase. The reason this result cannot be achieved via an appeal to V-Preposing is that V-Preposing occurs only in matrix clauses in Italian.

Returning now to (50), if the above argument is true, then the Spanish sentence corresponding to the English version should be well-formed. It turns out that this is the case; that is, the Spanish sentence corresponding to (50) *is* well-formed:

(52) ¿A quién contempla usted que fue a la escuela?

Bounding Theory is relevant to the translation model in that it limits the possible outcomes of conversion from D-structure to S-structure according to the requirements of the language undergoing translation. The strategy of constraining the application of a single general movement rule is an improvement over a context-free rule-based approach because language-specific grammars are not required, and the constraints restrict overgeneration that might otherwise occur during context-free parsing.

3.8 Principles and Parameters

Appendix A.1 contains a table summarizing the subsystems of principles and parameters (grouped according to subtheory) and the corresponding level of application. The principles that apply at S-structure (SS) and D-structure (DS) are within the realm of syntax; the principles that apply at Logical Form (LF) are within the realm of semantics. Thus, only those that apply at S-structure and D-structure are included in the translation model. Appendix A.2 shows an example (for Spanish and English) of parameter settings for the principles of GB.

Chapter 4

Overall Design of UNITRAN

As mentioned in chapter 1, UNITRAN has three basic stages of translation: parsing, thematic substitution, and generation. All three translation stages operate in a co-routine fashion in which the flow of control is passed back and forth between a structure-building module and a linguistic constraint module. At each of the three stages of translation, processing tasks are divided between the two modules as shown in table 4.1. This chapter provides a brief description of the three stages, and presents an example of how the co-routine design works during the parsing stage. The chapters that follow give a more detailed description of the translation system components.

First, during the parsing stage a preprocessed and morphologically analyzed input sentence is mapped into a tree structure that conforms to the requirements of $\overline{\text{X}}$-theory. The structure-

Translation Stage	*Structure-Building Tasks*	*Linguistic Constraint Tasks*
Parser	Phrase Structure Construction: Predict, Scan, Complete	Phrase Structure Constraints: Argument Structure, Binding, Case, and Semantic Roles
Substitution	Lexical Replacement	Lexical Constraints: Argument Structure and Thematic Divergence Tests
Generator	Structural Movement and Morphological Synthesis	Structural and Morphological Constraints

Table 4.1: Translation Tasks of Structure-Building and Linguistic Constraint Modules

building component, an implementation of the Earley algorithm (1970), applies predicting, scanning, and completing actions,[1] while the linguistic constraint component enforces well-formedness conditions (based on GB principles) on the structures passed to it. The phrase-structures that are built by the structure-building component are underspecified, (*i.e.*, they do not include information about agreement, abstract case, semantic roles, argument structure, and so forth). The structures are based on a set of templates derived during a precompilation phase according to certain source language parameters (discussed in section 5.3).

All phrase structures built during parsing are held in a push-down stack; the top of the stack is the subtree currently being processed. All possible parses are held in parallel push-down stacks. The linguistic constraint component eliminates or modifies the underspecified phrase-structures according to principles of GB (*e.g.*, agreement filters, case filters, argument requirements, semantic role conditions, etc.). This design is consistent with several studies that indicate that the human language processor initially assigns a (possibly ambiguous or underspecified) structural analysis to a sentence, leaving lexical and semantic decisions for subsequent processing.[2] Because the linguistic constraints are available during parsing, the structures built by the structure-building module need not be elaborate; consequently the grammar size need not, and should not, be as large as is found in many other parsing systems.[3] Thus, the system avoids some computational costs due to grammar search time.

Just prior to the second stage, thematic substitution, the source language sentence is in an *underlying form*, *i.e.*, a form that can be translated into any target language according to conditions relevant to that target language. (The underlying form is derived by structural replacement routines to be discussed in section 7.1.1). This means that all "participants" of the main action (*e.g.*, *agent*, *patient*, *etc.*) of the sentence are identified and placed in a canonical position relative to the main verb. At the level of thematic substitution, the structure-

[1]The *predict* action starts the construction of a possible phrase; the *scan* action advances over a terminal word; and the *complete* action finishes the construction of a phrases in the input.

[2]See chapter 1 (fn. 8).

[3]In fact, the number of phrase structure templates that are generated per language generally does not exceed 150 since there are a limited number of configurations per language that are allowed by the principles of $\overline{\text{X}}$-Theory. Thus, the running time of the parser is not subject to the same slow-downs that are found in other systems. For example, the GPSG formalism allows the description of a language to be expanded into a large database of context-free rules, forcing the running time of a typical GPSG parser to be exponential in the number of rules.

building module simply replaces target language words with their equivalent target language translations, subject to argument structure requirements and tests of thematic divergence. An example of thematic divergence is the translation of the English word *like* to the Spanish word *gustar*. Although these two verbs are semantically equivalent, the argument structures of these two verbs are not identical: the subject of *like* is the *agent*, whereas the subject of *gustar* is the *patient*. Because of such cases of thematic divergence, the argument structure of a source language verb must be matched with the argument structure of the corresponding target language verb before substitution takes place.

In the third stage, generation, the sentence is transformed into a grammatically acceptable form with respect to the target language (*e.g.*, in English the underlying form *was called John* would be transformed into the surface form *John was called*).

Figure 4.1 gives the overall design of the system, including access to GB components. Figure 4.2 shows the input/output of the UNITRAN components using the sample sentence *Vi al hombre*. We will briefly examine how one of the translation components, the parser, operates in tandem with the GB modules. Consider the problem of parsing (53).[4]

(53) Comí una manzana.
'I ate an apple.'

Table 4.2 contains a procedural description of the GB tasks associated with parsing a phrase. We will follow snapshots of the parser in action during the processing of (53).

The Earley parser predicts that the sentence (`I-MAX`) has a specifier (`I-SPEC`), a head (`I`), and a complement (`I-COMPLEMENT`), the order of which is determined by the "constituent order" parameter at precompilation time. It uses this information to predict that a noun phrase is in specifier position. Figure 4.3 displays the resulting stack configuration.[5] Since the next input word (*comí*) is not a noun, the (`N-MAX`) must be expanded in some other way. There are two possibilities: first, the pro-drop parameter allows the `N-MAX` to be *pro*; second, the `N-MAX` might

[4]The assumption is that preprocessing and morphological analysis have already taken place.

[5]The direction of stack growth is *upwards* (*i.e.*, new stack items are added to the *top* of the stack). Although the stack items are actually tree structures, throughout this report they will be represented in a bracketed form that is closer to the internal Lisp representation used by the system. For example, the form `[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]` corresponds to a tree with `I-MAX` as its root and `I-SPEC`, `I`, and `I-COMPLEMENT` as its three daughters.

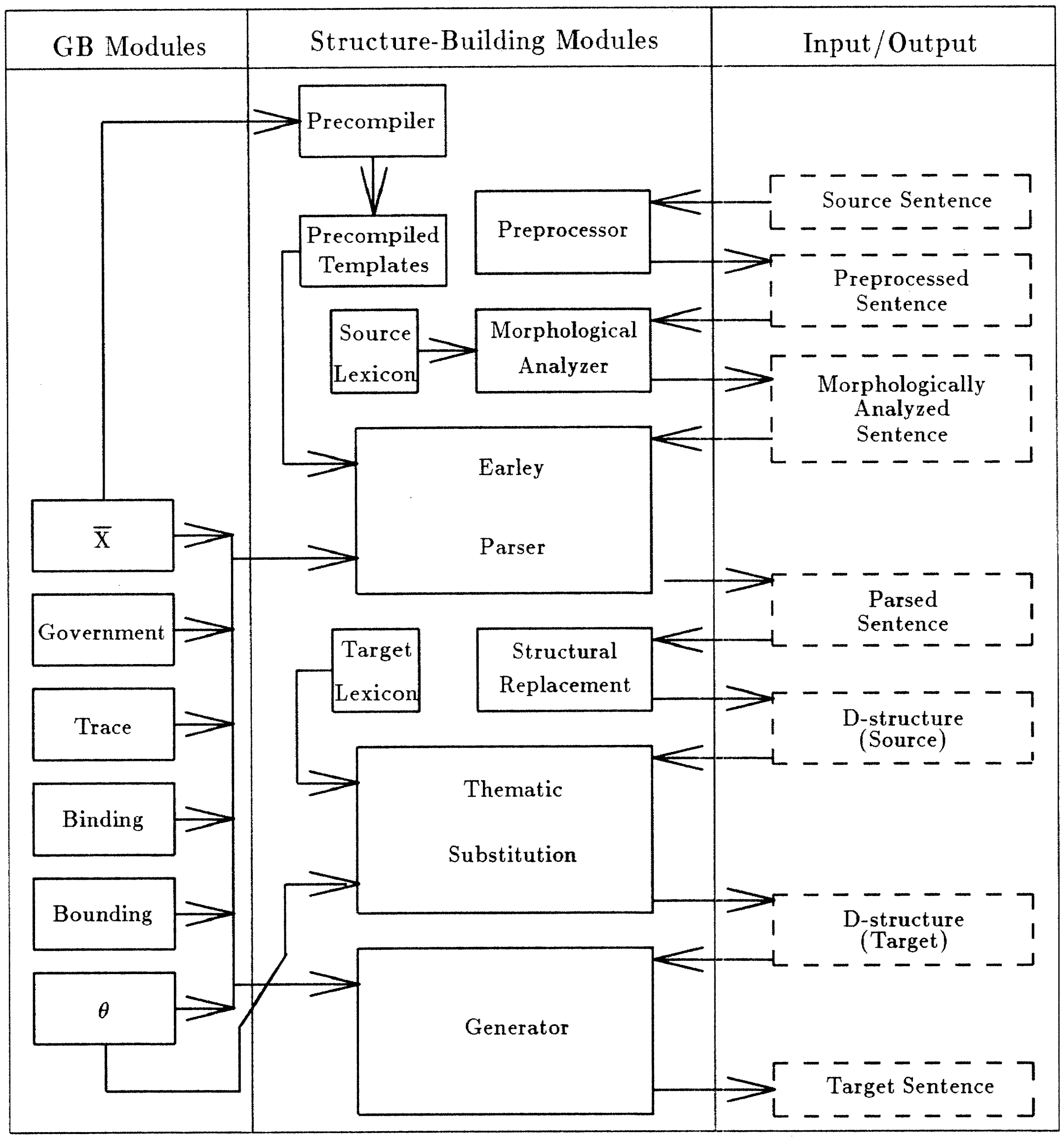


Figure 4.1: UNITRAN Design: Interacting GB and Structure-Building Modules with Associated Input/Output

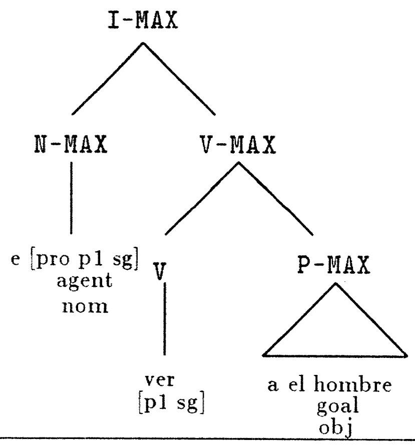
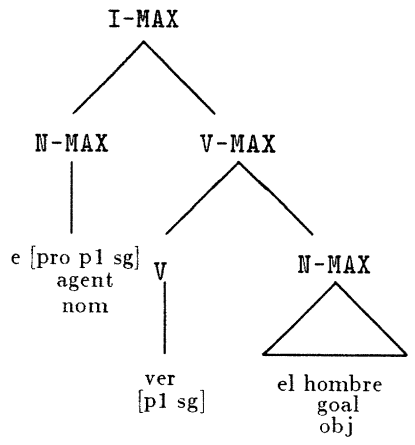
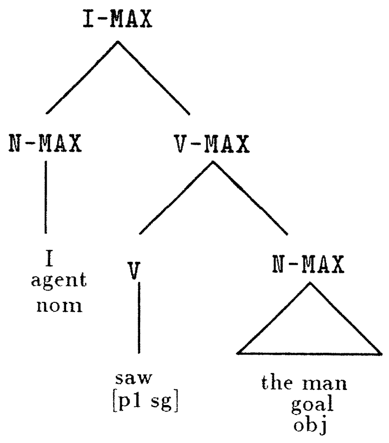
Input/Output	*Example*
Source Sentence	Vi al hombre.
Preprocessed Sentence	Vi a el hombre.
Morphologically Analyzed Sentence	((ver v past p1 sg) (a p) (el det sg masc) (hombre n sg masc))
Parsed Sentence	 I-MAX N-MAX V-MAX e [pro p1 sg] agent nom V P-MAX ver [p1 sg] a el hombre goal obj
D-structure (Source)	 I-MAX N-MAX V-MAX e [pro p1 sg] agent nom V N-MAX ver [p1 sg] el hombre goal obj
D-structure (Target)	 I-MAX N-MAX V-MAX I agent nom V N-MAX saw [p1 sg] the man goal obj
Target Sentence	I saw the man.

Figure 4.2: Input/Output of the UNITRAN Components

1. If there are unlinked traces, link them to antecedents.
   - (a) If there is no antecedent, or the antecedent is too far away, reject the parse.
   - (b) If trace-antecedent linking is successful, continue.
2. If there are newly introduced features of heads, percolate features up to maximal projections.
   - (a) If specifier-max agreement fails, reject the parse.
   - (b) If specifier-max agreement is successful, continue.
3. If a verb phrase is complete, percolate tense and agreement features up to I(nfl).
4. If the phrase contains an unexpanded complement, then:
   - (a) Predict complements, pushing them onto the stack.
   - (b) Associate heads with complements by storing the subcategorization information in the head of the phrase.
5. If the head of the phrase has been parsed, then:
   - (a) Instantiate features including person, gender, number, tense, *etc.*
   - (b) Set up θ-roles that the head assigns to external and internal arguments.
6. If the phrase is complete, then:
   - (a) Set up A and $\overline{\text{A}}$ positions.
   - (b) Link traces to antecedents.
   - (c) Set up Government relations.
   - (d) Check that all traces are properly governed.
   - (e) Assign Case.
   - (f) Assign θ-roles.
   - (g) Reject parse if there is an ECP, Binding, or θ-Criterion violation.

Table 4.2: Procedural Description of the Parser

[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]

Figure 4.3: Earley Prediction of `N-MAX` in Specifier Position of `I-MAX`

be a trace. (These possibilities are established at precompilation time.) The GB component rejects the trace possibility because of step 1 from table 4.2 (*i.e.*, there is no possible antecedent for the trace). By contrast, the *pro* possibility is accepted (see figure 4.4).

Now the top of stack element (`N-MAX`) is complete and ready to be dropped under `I-SPEC` in the second stack element. The completed `N-MAX` is in turn dropped under `I-MAX` in the third stack element. The resulting configuration is in figure 4.5.

The Earley parser determines that `I` is an empty feature holder, and that `I-COMPLEMENT`

[N-MAX e [pro]]
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]

Figure 4.4: Earley Expansion of N-MAX into *pro*

[I-MAX [I-SPEC [N-MAX e [pro]]] [I] [I-COMPLEMENT]]

Figure 4.5: Earley Completion of N-MAX containing *pro*

expands to the default complement value of I, which is V-MAX. The V-MAX is then expanded so that it contains a head and a complement (since the specifier of V-MAX is optional in Spanish). Figure 4.6 gives the result.

Now the Earley parser scans the word *comer* (the root form of comí) and attaches it under the V node. Figure 4.7 shows the result. Because the Earley module cannot proceed any further, the GB module takes over again. The first 4 steps from table 4.2 do not apply, so step 5 is executed: since a head has just been parsed, the features of the head [past sg p1] are instantiated, and the θ-roles assigned by the head are associated with the head node. Next, step 2 applies since new features have been introduced: the features [past sg p1] are percolated up to V-MAX. Finally, there is an unexpanded complement (V-COMPLEMENT), so step 4 is applicable: the subcategorization information associated with the head *comer* allows V-COMPLEMENT to be expanded as N-MAX; then this complement is associated with the V node (see figure 4.8).[6]

The Earley parser takes over again, parsing the N-MAX *una manzana*, and attaching it under the V-COMPLEMENT node (see figure 4.9). This completes V-COMPLEMENT, which in turn completes V-MAX. The completion of V-MAX allows steps 3 and 6 to be executed: first, features [past sg p1] are percolated up to I(nfl); next, objective (OBJ) case is assigned to *una manzana*; finally, θ-role *goal* is assigned internally from the verb. (See figure 4.10.) Because θ-role has been discharged successfully to N-MAX, the parse continues.

[6]Actually, since *comer* may be either intransitive or transitive, the parser predicts both the elimination *and* the expansion of V-COMPLEMENT. However, the elimination of V-COMPLEMENT leads to a dead end immediately because the parse is completed before all the input words have been read. Thus, this possibility is not shown here.

[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e [pro]]] [I E] [I-COMPLEMENT]]

Figure 4.6: Earley Expansion of `I-COMPLEMENT` into `V-MAX`

[V-MAX [V comer] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e [pro]]] [I E] [I-COMPLEMENT]]

Figure 4.7: Earley Scanning of the Input Word *comer*

The Earley parser attaches `I-COMPLEMENT` under `I-MAX`, thus completing the phrase and invoking a final call to the GB component. First, step 2 is executed since new tense and agreement features appear in `I-MAX` position: the maximal features `[past sg p1]` are tested against the features of *pro*; however, the features of *pro* have not yet been determined, so feature matching automatically succeeds and the agreement features `[sg p1]` are associated with *pro*. Figure 4.11 shows the parse at this point. Second, step 6 is executed since `I-MAX` is now complete. Because I(nfl) has tense features (`past`), it assigns nominative case to the specifier `N-MAX` of the phrase; thus, the specifier is visible for θ-marking and it receives a θ-role of *agent* externally from the verb. Finally, the θ-Criterion is checked: all arguments have been given θ-roles, so the phrase is successfully parsed. Figure 4.12 shows the completed phrase.

The following three chapters describe the pretranslation routines (*i.e.*, the preprocessing, morphological analysis, and precompilation components); the parsing component (*i.e.*, phrase-structure construction routines); and the final translation routines (*i.e.*, thematic substitution and generation routines). The top-level translation routines are TRANSLATE, READ-INPUT-SENTENCE, RUN-THROUGH-TRANSLATION-LOOP and PREPROCESS-MORPH-PARSE-TRANSLATE (see appendix J).

[V-COMPLEMENT [N-MAX]]
[V-MAX [past sg p1] [V comer [int:goal ext:agent past sg p1]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e [pro]]] [I E] [I-COMPLEMENT]]

Figure 4.8: GB Feature Instantiation and Prediction of Verbal Complement `N-MAX`

[V-COMPLEMENT [N-MAX [N-SPEC [DET una]] [N manzana]]]
[V-MAX [past sg p1] [V comer [int:goal ext:agent past sg p1]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e [pro]]] [I E] [I-COMPLEMENT]]

Figure 4.9: Earley Completion of Verbal Complement `una manzana`

```
[I-COMPLEMENT
  [V-MAX [past sg p1]
    [V comer [int:goal ext:agent past sg p1]]
    [V-COMPLEMENT
      [N-MAX [goal obj] [N-SPEC [DET una]] [N manzana]]]]]
-----------------------------------------------------------
[I-MAX [I-SPEC [N-MAX e [pro]]]
  [I [past sg p1] E] [I-COMPLEMENT]]
```

Figure 4.10: GB Feature Percolation, and Internal Case and θ-Role Assignment

```
[I-MAX
  [I-SPEC [N-MAX e [pro sg p1]]]
  [I [past sg p1] E]
  [I-COMPLEMENT
    [V-MAX [past sg p1]
      [V comer [int:goal ext:agent past sg p1]]
      [V-COMPLEMENT
        [N-MAX [goal obj] [N-SPEC [DET una]] [N manzana]]]]]]
```

Figure 4.11: Earley Completion of `I-COMPLEMENT` and GB Feature Matching

```
[I-MAX
  [I-SPEC [N-MAX e [pro sg p1 agent nom]]]
  [I [past sg p1] E]
  [I-COMPLEMENT
    [V-MAX [past sg p1]
      [V comer [int:goal ext:agent past sg p1]]
        [V-COMPLEMENT
          [N-MAX [goal obj] [N-SPEC [DET una]] [N manzana]]]]]]
```

Figure 4.12: GB Case Marking and External θ-Role Assignment

Chapter 5

Pretranslation Routines

This chapter gives a detailed description of the components that operate on the input before parsing, thematic substitution, and generation take place. These components include the preprocessor, the morphological analyzer, and the precompiler.

5.1 Preprocessing Routines

The preprocessor is the first module accessed when an input sentence has been entered for translation. Prior to preprocessing, all lexicons and parameter settings have been loaded. Two standard operations are performed: the first is division (splitting) of contractions into constituents (*e.g.*, `al` $\leftrightarrow$ `a el` in Spanish, and `don't` $\leftrightarrow$ `do not` in English); and the second is combining (merging) of closely related words into single units (*e.g.*, `as soon as` $\leftrightarrow$ `as_soon_as` in English and `en cuanto` $\leftrightarrow$ `en_cuanto` in Spanish).[1] Appendix D shows SPLITS-AND-MERGES parameter values for Spanish and English.[2] In order for a language to be translated by the system, the user must specify values for this parameter. At parse time, the merged words are accessed as a single unit from the lexicon; thus, they must be stored as a single unit. Similarly, the split constituents are accessed individually from the lexicon; thus, they must be

[1]The splitting and merging operations are similar to the collocation and contraction operations in Sharp's translator (1985), and are standard in many natural language systems, including ATN's. However, in this implementation, splits and merges are specified as a parameter setting.

[2]This parameter is external to GB theory; it is one of three user-modifiable parameters specific to the translation implementation. The other two parameters will be described in later sections.

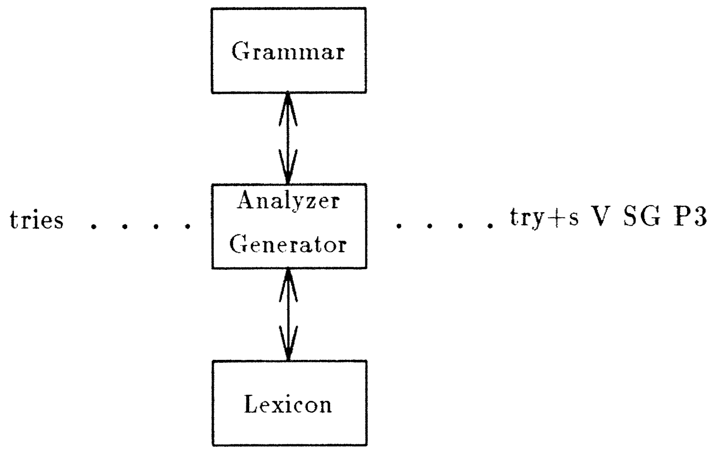


Figure 5.1: Organization of Kimmo

stored individually.

5.2 Morphological Analysis

The morphological analysis is performed by Kimmo Koskenniemi's two-level analyzer-synthesizer as implemented by Barton (1985). The organization of the Kimmo system is in figure 5.1 (taken from Karttunen and Wittenburg (1983)). Note that the model is reversible: the same grammar description and lexicon are used both for recognition and for generation. The grammar, or morphological rules, are represented as finite-state transducers. In the lexicon, every entry consists of a string of features expressing the syntactic or semantic properties of the morpheme and a symbol for its continuation class.[3] The morphological rule formalism and lexicon will be discussed in turn in the following sections.

5.2.1 Morphological Rule Formalism

The morphological rules for each language are represented in the processor as automata that express a one-to-one correspondence between lexical and surface forms (*e.g.*, TRY+S $\leftrightarrow$ TRIES). Appendix B.1 lists the English automata and appendix B.2 lists the Spanish automata.[4]

[3] A continuation class is used to determine what suffixes are applicable to a root form.

[4] The English automata are taken directly from Karttunen and Wittenburg (1983), and the Spanish automata were written by the author. The inspiration for the design of the Spanish automata came from Nassi, *et. al.*

The automata are encoded in matrix form. Rows correspond to states and columns show the transitions for particular input pairs. A colon (:) marks a state as final, and a period (.) marks a state as nonfinal. In order for languages to be translated by the system, the user must supply automata for each source and target language. Each of the English and Spanish automata will be discussed in turn.

Before describing the morphological rules, we must introduce some notation. The lexical characters are those that occur in the lexicon (to be discussed shortly) and the surface characters are those in the surface form of the word to be synthesized or analyzed. A plus marker (+) indicates a suffix appended to a morpheme. The context of the rule application specifies the character occurrences to the left and right of the lexical/surface characters in the rule. (An underscore (__) specifies the position of the lexical/surface characters.) The set notation {..,..} indicates that one of the characters in the set must occur. If zero (0) is included in this set, then no characters need to appear in the specified position. A capital letter or characters inside angle brackets (<..>) indicates that any of a set of characters is possible in a specified position. (The sets represented by capital letters will be introduced as they come up in the examples below.)

Five of the English automata encode morphological rules.[5] The first English morphological rule changes **s** to **es** before making certain roots plural or third person singular:[6]

EPENTHESIS

lexical	*surface*	*context*
+s	es	{ch, sh, s, x, z}_

EXAMPLES

lexical	*surface*
fox+s	foxes
church+s	churches
ski+s	skis
boy+s	boys

(1965), and Stockwell, *et. al.* (1965).

[5]The first of the six English automata maps every character to itself (this automaton is required for technical reasons internal to the Kimmo implementation), and the remaining five encode morphological rules.

[6]Since the Kimmo system is set up for both analysis and synthesis, the morphological rules can be applied in either direction. However, the above textual description of the rules puts them in the context of synthesis in order to simplify the explanation.

The second rule doubles the final consonant of a root before adding certain suffixes (`ing`, `ed`, *etc.*):

GEMINATION

lexical	*surface*	*context*
+	<C1>	CV<C1> _V

where: C = set of all consonants, V = set of all vowels,
and <C1> = {b, d, f, g, l, m, n, p, r, s, t}

EXAMPLES

lexical	*surface*
big+er	bigger
stop+ing	stopping
cool+er	cooler
travel+ing	traveling

The third rule changes `ie` to `y` before adding suffixes beginning with `i`:

Y-SPELLING

lexical	*surface*	*context*
ie+	y	_i

EXAMPLES

lexical	*surface*
die+ing	dying
die+ed	died

The fourth rule changes `y` to `i` before adding suffixes beginning with `e` or `l`:

Y-REPLACEMENT

lexical	*surface*	*context*
y+	i	C_{e, l}

EXAMPLES

lexical	*surface*
spy+ed	spied
happy+ly	happily
spy+ing	spying
day+s	days

The fifth and final English rule deletes an `e` in the following contexts: after any non-vowel except `c` or `g` and before a suffix beginning with a vowel; after a vowel and before a suffix beginning with `e`; and after `c` or `g` and before a suffix beginning with `e` or `i`. Thus, we have:

ELISION

lexical	*surface*	*context*
e+	0	<C2> _V; V_e; or {c, g}_{e, i}

where: <C2> = all consonants except c or g

EXAMPLES

lexical	*surface*
die+ed	died
move+able	movable
agree+ed	agreed
move+s	moves
race+able	raceable

The first of the ten Spanish automata maps every character to itself (as in the English automata), and the remaining nine encode morphological rules. The first rule is infinitival removal. All Spanish infinitive verbs end in **er**, **ar**, or **ir** (thus partitioning all verbs into 3 classes). In order to extract the root form of a verb (and add suffixes), the infinitive ending must be removed:

INFINITIVE-REMOVAL

lexical	*surface*	*context*
{a, e, i}r+	0	{C, V}_

EXAMPLES

lexical	*surface*
haber+	hab
ver+	v
ir+	ir+

The second rule handles (er and ir) verbs that are irregular in the present subjunctive and also in the present first person singular.[7] For these verbs a **g** must be inserted after **en**, or a **g** must be softened (changed to **j**) after anything else:

PRESENT-SUBJUNCTIVE (ER/IR) G-SOFTEN, ADD G

lexical	*surface*	*context*
r+o	go	en{e, i}_
r+a	ga	en{e, i}_{0, n, mos, s}
r+o	jo	g{e, i}_
r+a	ja	g{e, i}_{0, n, mos, s}

[7]In general, morphological rules for the present first person singular form coincide with rules for the present subjunctive form for *all* types of verbs.

EXAMPLES

lexical	*surface*
venir+o	vengo
venir+amos	vengamos
coger+o	cojo
comer+o	como

The third rule handles additional (er) verbs that are irregular in the present subjunctive (and present first person singular). The rule changes `c` to `cz` after a vowel and `c` to `z` after `n` or `r`:

PRESENT-SUBJUNCTIVE (ER) C↔ZC, C↔Z

lexical	*surface*	*context*
+o	zco	V_
+a	zca	V_{0, n, mos, s}
+o	zo	{n, r}_
+a	za	{n, r}_{0, n, mos, s}

EXAMPLES

lexical	*surface*
conocer+o	conozco
vencer+amos	venzamos

The fourth rule handles (ir) verbs that are irregular both in the present and in the present subjunctive. The rule changes `gu` to `g` after a vowel and `ui` to `uy` after a consonant other than `g` (except for the first person plural, *i.e.*, the `mos` form):

PRESENT AND SUBJUNCTIVE (ER) GU↔G, UI↔UY

lexical	*surface*	*context*
guir+o	go	V_
guir+a	ga	V_{0, n, mos, s}
uir+o	uy	<C1> _
uir+{a,e}	uy	<C1> _{0, n, s}

where: C1 = all consonants except g

EXAMPLES

lexical	*surface*
distinguir+o	distingo
distinguir+amos	distingamos
distinguir+imos	distinguimos
concluir+e	concluye
huir+imos	hui^mos

The fifth rule adds an accent (represented as the symbol ^) to **u** or **i** syllable for (ar) verbs that are irregular both in the present and in the present subjunctive (except for the first person plural, *i.e.*, the **mos** form):[8]

PRESENT AND SUBJUNCTIVE (AR) U↔U^, I↔I^

lexical	*surface*	*context*
ar+o	^o	{u, i}_
ar+{a, e}	^a	{u, i}_{0, n, s}

EXAMPLES

lexical	*surface*
enviar+o	envi^a
enviar+amos	enviamos
continuar+a	continu^a
continuar+e	continu^e

The sixth rule operates on preterit (ar) verbs. It changes **c** to **qu**, **g** to **gu**, and **z** to **c** in the first person singular form:

PRETERIT (AR) C↔QU G↔GU Z←C

lexical	*surface*	*context*
car+e	que	_^#
gar+e	gue	_^#
zar+e	ce	_^#

where: # = end of morpheme marker

EXAMPLES

lexical	*surface*
atacar+e^	ataque^
llegar+e^	llegue^
cruzar+e^	cruce^

The seventh rule changes plural ending **s** to **es** for all nouns ending in a consonant other than **z**. The consonant **z** gets changed to **c** before adding the plural ending, and all other nouns are pluralized without any changes:

PLURALIZE

lexical	*surface*	*context*
<C2> +	<C2>e	_*s*
z+	ce	_*s*

where: <C2> = all consonants except z

[8]An accent occurs *after* the syllable that is stressed.

EXAMPLES

lexical	*surface*
ciudad+s	ciudades
lapiz+s	la^pices
carro+s	carros

The eighth rule removes an accent mark from a syllable if adding a suffix changes the stress properties of a syllable. The general rule for stress in Spanish is the following: the second to last syllable of a word must be stressed if the word ends in a vowel, **n**, or **s**; any exception to this stress rule forces an accent mark to occur on the stressed syllable. Thus, *reunión* (ending in **n**) carries an accent on the last syllable since the stress is not on the second to last syllable; similarly, *jóvenes* (ending in **s**) carries an accent on the first syllable since the stress is not on the second to last syllable. The reason the accent needs to be removed in certain cases is that adding a suffix adds a *new* syllable to the word, thus changing stress properties so that the stress rule is no longer violated (and consequently, the accent mark is no longer required). For example, reuniones (plural for *reunión*) does not require an accent mark since adding the +s suffix changes the stress so that it is on the second to last syllable. The accent removal rule is the following:

REMOVE ACCENT

lexical	*surface*	*context*
^C+	C	V_V{0, n, s}

EXAMPLES

lexical	*surface*
reunio^n+s	reuniones

The ninth rules adds an accent mark to a vowel if adding a suffix changes the stress properties. Again, the stress rule applies. If adding a suffix forces stress to be "moved up" to a syllable that is before the second to last syllable, an accent mark must be placed on the stressed syllable:

ADD ACCENT

lexical	*surface*	*context*
CVC+	^CVC	V_V{0, n, s}

EXAMPLES

lexical	*surface*
lapiz+s	la^pices
examen+s	exa^menes

One thing to note about the eighth and ninth morphological rules is that they do not directly correspond to the automata. The right context {0, n, s} is not part of the automata since the +`V` must crucially to be followed by a terminal **`n`**, **`s`**, or `0`.

5.2.2 The Lexicon

In order for languages to be translated by the system, the user must supply a lexicon for each source and target language. The lexicon must be in the format required by the Kimmo morphological system. Only the Kimmo recognizer accesses the lexicon.

The lexicon consists of several smaller lexicons, each of which contains either root forms or continuations (suffixes). There is a single *root lexicon*, which is the first lexicon searched when a word is being recognized. After the root form is identified, the system determines the *continuation lexicons* to be searched. This information is determined from the lexical entry of the root form, which includes a list of continuation lexicons that contain applicable suffixes for the root. For example, the Spanish word *libro* is listed as **`libr /N`** in the root lexicon (`/N` is a continuation class for nouns); the ending **`o`** is then found in the continuation lexicon N (pointed to by `/N` in the ALTERNATIONS list). Thus, the lexical form **`libr`**+**`o`** corresponds to the surface form **`libro`**. If no continuation lexicon is listed in the lexical entry (*i.e.*, no suffixes are applicable), the marker **#** is used. For example, the Spanish determiner **`el`** has no continuation lexicon listed. The English lexicon is in appendix C.1 and the Spanish lexicon is in appendix C.2.[9]

Each lexical entry is associated with a set of features. Tables 5.1 and 5.2 show the features associated with the suffix lexicon entries for Spanish and English respectively.[10] In order for any feature to be *legally* listed in a lexical entry, the feature must be included in the FEATURES parameter setting supplied by the user. (This parameter is also used by the $\overline{\text{X}}$ module for feature instantiation during parsing, as will be discussed further in section 6.2.1.)

As shown in appendix E, each preterminal and basic category (from the PRE-TERMINALS and BASIC-CATEGORIES values) must have features associated with it. The assumption is

[9]Parts of the English lexicon were taken from Karttunen and Wittenburg (1983). The Spanish lexicon was written by the author.

[10]Not all of the Spanish features are included in the table since they are too numerous to list here. Refer to appendix C.2 for more details.

Lexicon	*Suffix*	*Feature Specifications*
N	0	n p3 sg obj nom
	+s	n p3 pl obj nom
	+y	a
PROP	0	n prop p3 sg
	+s	n prop p3 pl
MN	0	mass n
C1	%s	poss
C2	%	poss
P3S	+s	v pres p3 sg
P3P	0	v pres p3 pl
P12	0	v pres p1 p2 sg pl
IP3S	0	v pres p3 sg
PS	+ed	v past p1 p2 p3 sg pl
IPS	0	v past p1 p2 p3 sg pl
PPD	+ed	v perf
PPN	+en	v perf
IPP	0	v perf
PR	+ing	v prog
I	0	v inf
AG	+er	agent neut
PA	0	a neut sg pl
CA	+er	comparative a neut sg pl
CS	+est	superlative a neut sg pl
LY	+ly	adv
AB	+able	a condition v-able neut sg pl
DET	0	det neut
WH-PHRASE-A	0	n wh-phrase-a neut p3 pl sg wh
MODAL-PAST	0	i past modal p1 p2 p3 sg pl
MODAL-PRES	0	i pres modal p1 p2 p3 sg pl
PREP	0	p
C-FIN	0	c
C-INF	0	c

Table 5.1: Features associated with English Suffix Lexicons

that the features can be categorized into "universal" classifications (GENDER, PERS, NUMBER, CASE, and TENSE) and language-specific classifications (those that are in a one-to-one correspondence with the categories of BASIC-CATEGORIES and PRE-TERMINALS). A "universal" feature classification can be included as an element of a language-specific classification (this is shorthand for directly enumerating the features that occur in the universal feature classification as elements of the language-specific classification). During the parsing of

Lexicon	*Suffix*	*Feature Specifications*
N	0	n p3 sg
	+s	n p3 pl
	+a	n fem p3 sg
	+o	n masc p3 sg
	+as	n fem p3 pl
	+os	n masc p3 pl
	+e	n neut p3 sg
PROP	0	n proper p3
LY	+amente	adv
A	0	a sg
	+s	a pl
	+a	a fem sg
	+o	a masc sg
	+as	a fem pl
	+os	a masc pl
	+e	a neut sg
DET	0	det sg
	+s, +es	det pl
	+a	det fem sg
	+o	det neut sg
	+os	det masc pl
	+as	det fem pl
	+e	det masc sg
WH-PHRASE-A	0	n neut p3 wh wh-phrase-a
	+s, +es	n p3 pl wh wh-phrase-a
	+a	n fem p3 sg wh wh-phrase-a
	+o	n neut p3 sg wh wh-phrase-a
	+os	n masc p3 pl wh wh-phrase-a
	+as	n fem p3 pl wh wh-phrase-a
	+e	n masc p3 sg wh wh-phrase-a
AR-MODAL	0	i modal
PREP	0	p
INF	+ar, +er, +ir	inf
C-FIN	0	c
C-INF	0	c
AR-PRES	+o	pres p1 sg
	+as	pres p2 sg
	+a	pres p3 sg
	+amos	pres p1 pl
	+an	pres p3 pl
AR-PRET	+é	past p1 sg
	+aste	past p2 sg
	+ó	past p3 pl
	+amos	past p1 pl
	+aron	past p3 pl
AR-PROG	+ando	prog
PERF	+ado, +ido	perf

Table 5.2: Features associated with Spanish Suffix Lexicons

a source language sentence, the value of FEATURES corresponding to the language is stored in *CURRENT-FEATURES. Appendix D shows the English and Spanish instantiations of the FEATURES parameter. Table 5.3 and table 5.4 contain descriptions of the features for Spanish and English respectively.

In addition to feature information, lexical entries contain subcategorization and external argument information. That is, lexical elements that might be parsed as heads of phrases must have argument information stored in their lexical entries. For example, the English verb *expect* has the following information in its lexical entry:

```
(subcat (proposition))
(subcat (goal) (proposition))
(external (agent animate))
```

Thus, *expect* can take a *proposition*, or a *goal* and a *proposition* as its internal arguments, and it must have an external *agent*. Note that feature information (*e.g.*, ±*animate*) can be included in argument frames. Also, the arguments are in fact unordered, although they appear to be ordered in the argument frames. For example, the frame `(subcat (goal) (proposition))` requires there to be a *goal* and a *proposition*, but it does not require that the surface order be the order specified in the frame. The ordering of arguments is not the task of the lexical processing routines, but of other components (*e.g.*, the Case module). This means that as far as the lexicon is concerned, the verb `expect` can take a *goal* and a *proposition* as arguments in any order (either `expect [the man] [to go]`, or `expect [to go] [the man]`), and it is up to the later stages of translation to determine which of these orders is (are) correct.

Each root form must include *equivalent* translation forms from all source languages (that the user desires to translate into) in its lexical entry. This introduces much ambiguity into the system. For example, the English word *know* translates to either *conocer* or *saber* in Spanish. Both of these translations must be included in the lexical entry:

KNOW /IV2 (spanish ((saber) (conocer))) ...

It is the duty of the thematic substitution routine to determine the correct translation (via subcategorization information or feature matching). For example, if the argument structure of the word *know* includes a *proposition*, the thematic substitution routine will determine that the translation is *saber* (since *conocer* can only take a *goal* with *animate* features). On the other

Feature	*Description*	*Example*
p1	first person	go
p2	second person	go
p3	third person	goes
sg	singular number	man
pl	plural number	men
masc	masculine gender	him
fem	feminine gender	her
neut	neuter gender	it
obj	objective case	me
nom	nominative case	I
poss	possessive case	my
pronoun	pronominal noun	he
anaphor	anaphoric noun	himself
pleonastic	pleonastic element	it
proper	proper noun	John
inf	infinitive tense	go
perf	perfect tense	gone
prog	progressive tense	going
past	past tense	went
pres	present tense	go
fut	future tense	will (go)
cond	conditional tense	would (go)
comparative	comparative adjective	bigger
superlative	superlative adjective	biggest
location	locative noun or prepositional phrase	at home
duration	quantified temporal noun	two months
quantity	noun specifying quantity	dollars
access	noun specifying route	door
method	noun specifying method	car
exchange	type of exchange	payment
time	temporal noun	month
fact	clausal fact	that he eats well
animate	animate noun	woman
v-able	able to v	parsable
descript	descriptive adjective	ugly
condition	condition adjective	sick
sdel	s-bar deletion verb	seem
intrans	intransitive verb	run
wh	wh-word	where
wh-phrase	argument wh-word	what

Table 5.3: Description of Features for English

Feature	*Description*	*Example*
p1	first person	voy
p2	second person	vas
p3	third person	va
sg	singular number	hombre
pl	plural number	hombres
masc	masculine gender	el
fem	feminine gender	la
neut	neuter gender	lo
obj	objective case	mí
nom	nominative case	yo
poss	possessive case	mi
pronoun	pronominal noun	le
anaphor	anaphoric noun	se
proper	proper noun	Juan
inf	infinitive tense	ir
perf	perfect tense	ido
prog	progressive tense	yendo
past	past tense	fue
pres	present tense	voy
pres-subj	present subjunctive tense	vaya
past-subj	past subjunctive tense	fuera
fut	future tense	iré
cond	conditional tense	iría
location	locative noun or prepositional phrase	en casa
duration	quantified temporal noun	dos meses
quantity	noun specifying quantity	dolares
access	noun specifying route	puerta
method	noun specifying method	carro
exchange	type of exchange	pagamiento
time	temporal noun	mes
fact	clausal fact	que come bien
animate	animate noun	hombre
descript	descriptive adjective	feo
condition	condition adjective	enfermo
sdel	s-bar deletion verb	parecer
intrans	intransitive verb	correr
wh	wh-word	dónde
wh-phrase	argument wh-word	qúe

Table 5.4: Description of Features for Spanish

hand, if the argument structure of the word *know* includes only *goal* with *animate* features, the thematic substitution routine will determine that the translation is *conocer*.

If a root form could be part of an idiomatic expression, the translation of that idiom must be included in the lexical entry. For example, the expression *kick the bucket* can be translated either to *morir* (die) or an equivalent idiomatic expression *estirar la pata*. Both of these translations of the idiomatic expression are included in the lexical entry of the verb *kick* (as well as the "normal" translation, *patear*):

```
KICK /V
  ... (spanish ((patear))
      (expression ((the bucket) (spanish ((morir) (estirar la pata))))) ...
```

A lexical entry may also contain additional information: noun-forms, for verbs or adjectives that can be nominalized (*e.g.*, `know` $\leftrightarrow$ `knowledge`); INTRANS, for optional intransitivity;[11] and a root form (for irregular verbs), which is shorthand for listing the information that appears in the lexical entry of the root form. Note that irregular forms (like `knew`) need to be stored in the lexicon since they do not conform to the morphological rules encoded by the automata (*e.g.*, `know+ed` $\leftrightarrow$ `*knowed`).

5.3 Precompilation

The purpose of the precompilation stage is to construct skeletal phrase structures (based on $\overline{\text{X}}$-theory) that will be used to guide the Earley algorithm parser. The structure that is built is underspecified; that is, subcategorization information, antecedent-trace relations, case, and θ marking are not multiplied into the skeletal phrase structures. Rather, this information is determined on-line (only as needed) through access to the principles of GB.

The function that performs precompilation, SET-UP-XBAR (see appendix F.1.1), performs two top-level actions: (1) it generates all the context-free rules (into the variable FULL-BLOWN-RULES) required to satisfy $\overline{\text{X}}$ constraints for a given language (*e.g.*, `(C-MAX` $\Rightarrow$ `C-SPEC C C-COMPLEMENT)`, `(C-SPEC` $\Rightarrow$ `N-MAX)`, *etc.*); and (2) it generates (via DEFGRAMMAR) the corresponding $\overline{\text{X}}$ structures required by the Earley parser (*e.g.*, `[C-SPEC [N-MAX]]`,

[11] If a lexical entry does not include any subcategorization information and does not include the INTRANS feature, it is assumed to be *obligatorily* intransitive.

*CURRENT-CONSTITUENT-ORDER	constituent order for the current language
*CURRENT-BASIC-CATEGORIES	basic categories for the current language
*CURRENT-ADJUNCTION	adjunction configurations for the current language
*CURRENT-TRACES	trace possibilities for the current language
*CURRENT-EMPTY-FEATURE-HOLDERS	non-lexical heads and their default complements for the current language
*CURRENT-PRO-DROP	pro-drop setting for the current language
*CURRENT-OPTIONAL-SPECIFIERS	optional specifiers for the current language

Table 5.5: Global Variables Set up Prior to Precompilation

`[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]`, *etc.*). Thus, there are two results of the precompilation stage: the first is simply a set of context-free rules (solely for recognition, not for structure-building), and the second is a set of phrase-structure templates (for building structure during the parsing process). The parameter values that are accessed at precompilation time are in appendix E.1.

The rules and templates generated by SET-UP-XBAR are both unconstrained and underspecified. They are unconstrained in that they have no information about other possible linguistic violations. For example, the templates may contain traces of moved elements in positions that are not legal according to Trace theory. Consequently, the structures overgenerate wildly during parsing. The templates are underspecified in that they do not contain information about subcategorization of basic categories (except those specified in the EMPTY-FEATURE-HOLDERS parameter). Thus, complements are left empty until parsing begins, at which time lexical category information will be accessed on-line to fill in complement information as heads of phrases are encountered.

Just prior to execution of the SET-UP-XBAR function, certain global variables are set according to the parameter settings of the language. (Appendix I lists all global variables for the translation system.)[12] The global variables that are set up before SET-UP-XBAR is called are in table 5.5.

These variables are used by SET-UP-XBAR to create the rules required to parse the lan-

[12]If the $\overline{X}$ parameter settings have not changed since the last execution of SET-UP-XBAR, then the function is not called (since the rules and templates remain the same).

Variable(s) (and *Associated Function*)	*Use of Variable*
*CURRENT-CONSTITUENT-ORDER *CURRENT-BASIC-CATEGORIES (CREATE-BASE-RULES)	Generate X-MAX $\Rightarrow$ ... X ... for each basic category X (with specifier and complement positioned according to constituent order)
*CURRENT-ADJUNCTION (CREATE-ADJUNCTS)	Generate rules of form: (1) X-MAX $\Rightarrow$ ADJUNCT X-MAX, (2) X-MAX $\Rightarrow$ X-MAX ADJUNCT, (3) X $\Rightarrow$ ADJUNCT X, (4) X $\Rightarrow$ X ADJUNCT instantiating X with specified HEAD(s), and instantiating ADJUNCT with the corresponding NODE(s) (all permutations)
*CURRENT-EMPTY-FEATURE-HOLDERS (MAKE-EMPTY-RULES)	Generate Z $\Rightarrow$ E and Z-COMPLEMENT $\Rightarrow$ X-MAX$_1$ ... X-MAX$_n$ for each head Z and each list of X$_i$'s corresponding to head Z
*CURRENT-TRACES (ADD-TRACE-RULES)	Generate Y $\Rightarrow$ Y-TRACE for each Y-TRACE specified
*CURRENT-PRO-DROP (ADD-EMPTY-NPS)	Generate N-MAX $\Rightarrow$ PRO (automatically) and N-MAX $\Rightarrow$ pro (if AGR-RICH is T)
*CURRENT-CHOICE-OF-SPECIFIERS (MAKE-SPEC-RULES)	Generate M-SPEC $\Rightarrow$ X-MAX$_1$ ... X-MAX$_n$ for each head M, and each list of X$_i$'s corresponding to head M
*CURRENT-OPTIONAL-SPECIFIERS (DROP-OPTIONAL-SPECIFIERS)	Generate X-MAX $\Rightarrow$ α β for each rule X-MAX $\Rightarrow$ α X-SPEC β, where X-SPEC is optional

Table 5.6: How Global Variables Are Used by SET-UP-XBAR

guage. Table 5.6 shows how SET-UP-XBAR uses these variables.

Note that two variables (*CURRENT-TRACES and *CURRENT-PRO-DROP) associated with the trace module are included in the precompilation of $\overline{\text{X}}$ templates. These variables are accessed only to establish trace (and other empty category) possibilities for a particular language, not to check empty category conditions that are in the domain of the trace module. For example, the templates include **N-MAX-TRACE** wherever an **N-MAX** category is allowed, but they do not include coindexing or proper government information. In an earlier version of the system, the trace possibilities were not multiplied into the rules, but were added on-line as needed. This turned out to slow down the system at execution time since the parser had to make frequent checks to see if a trace (or other empty category) could be dropped at any point during the parse; it then had to check whether the trace satisfied empty category conditions. The current version allows traces to be inserted automatically at precompilation time (thus eliminating time-consuming searches through trace possibilities at execution time); it then checks the validity of the trace at execution time (in the same way as in the original version).

In contrast to the processing of trace possibilities, the processing of subcategorization information is not multiplied into the rules at precompilation time. This is because the space of possibilities for complements of a given category could be as large as the number of lexical entries for that category in a given language. For example, if every verb in the language had a unique subcategorization frame, then there would be n possibilities, where n is the number of verbs in the language. Performing this processing at execution time means that subcategorization information need only be checked a small number of times during execution, in particular, as many times as there are words in the sentence. Multiplying out all the subcategorization possibilities for a given language ahead of time would lose by a wide margin because all the subcategorization rules generated would have to be searched at execution time at each step of the parse.

Tables 5.7 and 5.8 show the FULL-BLOWN-RULES generated by SET-UP-XBAR for english and Spanish respectively.

In the last stage of precompilation, SET-UP-XBAR applies DEFGRAMMAR to the rules generated by the earlier stages. This function compiles all the rules into a grammar for a given language, and sets up $\overline{\text{X}}$ templates corresponding to these rules. These templates are generated from the rules, starting with the distinguished start symbol (`C-MAX` in both Spanish and English) and building a tree structure using the symbol as the root. As each maximal projection under the root is encountered, a new tree is spawned using the maximal projection as the root (unless the maximal projection has already been expanded). The process is repeated until no more trees can be constructed. Finally, all untouched rules are expanded. (This includes specifiers, adjunction constructions, non-lexical terminals and maximal projections not yet expanded). Because the number of maximal projections and preterminals is finite, this process is finite (*i.e.*, there is not an infinite number of trees). In fact, the templates are in a one-to-one

Rules Added By Create-Base-Rules
(START ⇒ C-MAX PUNC)
(C-MAX ⇒ C-SPEC C C-COMPLEMENT) (I-MAX ⇒ I-SPEC I I-COMPLEMENT)
(V-MAX ⇒ V-SPEC V V-COMPLEMENT) (N-MAX ⇒ N-SPEC N N-COMPLEMENT)
(P-MAX ⇒ P-SPEC P P-COMPLEMENT) (A-MAX ⇒ A-SPEC A A-COMPLEMENT)
Rules Added By Create-Adjuncts
(N ⇒ A-MAX N) (N-MAX ⇒ N-MAX P-MAX) (V-MAX ⇒ V-MAX P-MAX)
(V-MAX ⇒ V-MAX ADV) (N-MAX ⇒ N-MAX C-MAX) (V ⇒ BE-AUX V)
(I-MAX ⇒ DO-AUX I-MAX) (I-MAX ⇒ BE-AUX I-MAX)
(I-MAX ⇒ HAVE-AUX I-MAX)
Rules Added By Add-Trace-Rules
(P-MAX ⇒ P-MAX-TRACE) (N-MAX ⇒ N-MAX-TRACE)
Rules Added By Make-Empty-Rules
(C-COMPLEMENT ⇒ I-MAX) (C ⇒ E)
(I-COMPLEMENT ⇒ V-MAX) (I ⇒ E)
Rules Added By Add-Empty-NPs
(N-MAX ⇒ PRO)
Rules Added By Make-Spec-Rules
(P-SPEC ⇒ ADV) (A-SPEC ⇒ ADV) (I-SPEC ⇒ N-MAX)
(N-SPEC ⇒ N-MAX) (V-SPEC ⇒ DO-AUX) (V-SPEC ⇒ HAVE-AUX)
(C-SPEC ⇒ ADV) (C-SPEC ⇒ P-MAX) (C-SPEC ⇒ N-MAX)
Rules Added By Drop-Optional-Specifiers
(N-MAX ⇒ N N-COMPLEMENT) (P-MAX ⇒ P P-COMPLEMENT)
(C-MAX ⇒ C C-COMPLEMENT) (A-MAX ⇒ A A-COMPLEMENT)
(V-MAX ⇒ V V-COMPLEMENT)

Table 5.7: English Rules Generated by SET-UP-XBAR

correspondence with the FULL-BLOWN-RULES. The $\overline{\text{X}}$ templates for English and Spanish are in tables 5.9 and 5.10 respectively. Although these templates are actually tree structures, they are represented internally as list forms similar to the bracketed notation used here. Once these templates are built, the parse is driven top-down (with some bottom-up processing due to access of subcategorization information) by these structures.

A question under current investigation is what is the "optimal" balance of linguistic principle clustering between the precompilation and later processing phases. On the one hand, incorporating a large amount of linguistic information (*e.g.*, subcategorization information, agreement information *etc.*) into the precompilation phase causes the grammar size to become explosive, thus slowing down grammar search time during the structure-building phase of the

Rules Added By Create-Base-Rules
(START ⇒ C-MAX PUNC) (C-MAX ⇒ C-SPEC C C-COMPLEMENT) (I-MAX ⇒ I-SPEC I I-COMPLEMENT) (V-MAX ⇒ V-SPEC V V-COMPLEMENT) (N-MAX ⇒ N-SPEC N N-COMPLEMENT) (P-MAX ⇒ P-SPEC P P-COMPLEMENT) (A-MAX ⇒ A-SPEC A A-COMPLEMENT)
Rules Added By Create-Adjuncts
(N ⇒ A-MAX N) (N ⇒ N A-MAX) (N-MAX ⇒ N-MAX P-MAX) (V-MAX ⇒ V-MAX P-MAX) (V-MAX ⇒ V-MAX ADV) (N-MAX ⇒ N-MAX C-MAX) (V ⇒ CL-DAT CL-ACC V) (V ⇒ CL-DAT V) (V ⇒ CL-ACC V) (V ⇒ CL-REF V) (V ⇒ CL-REF CL-ACC V) (V ⇒ BE-AUX V) (I-MAX ⇒ I HAVE-AUX BE-AUX V I-MAX) (I-MAX ⇒ V I-MAX) (V-MAX ⇒ V-MAX N-MAX)
Rules Added By Add-Trace-Rules
(P-MAX ⇒ P-MAX-TRACE) (V ⇒ V-TRACE) (I ⇒ I-TRACE) (HAVE-AUX ⇒ HAVE-AUX-TRACE) (BE-AUX ⇒ BE-AUX-TRACE) (N-MAX ⇒ N-MAX-TRACE)
Rules Added By Make-Empty-Rules
(C-COMPLEMENT ⇒ I-MAX) (C ⇒ E) (I-COMPLEMENT ⇒ V-MAX) (I ⇒ E)
Rules Added By Add-Empty-NPs
(N-MAX ⇒ pro) (N-MAX ⇒ PRO)
Rules Added By Make-Spec-Rules
(P-SPEC ⇒ ADV) (A-SPEC ⇒ ADV) (N-SPEC ⇒ N-MAX) (I-SPEC ⇒ N-MAX) (N-SPEC ⇒ DET) (V-SPEC ⇒ HAVE-AUX) (C-SPEC ⇒ ADV) (C-SPEC ⇒ P-MAX) (C-SPEC ⇒ N-MAX)
Rules Added By Drop-Optional-Specifiers
(P-MAX ⇒ P P-COMPLEMENT) (N-MAX ⇒ N N-COMPLEMENT) (C-MAX ⇒ C C-COMPLEMENT) (A-MAX ⇒ A A-COMPLEMENT) (V-MAX ⇒ V V-COMPLEMENT)

Table 5.8: Spanish Rules Generated by SET-UP-XBAR

parse; on the other hand, eliminating too much linguistic information from precompilation (*e.g.*, trace possibilities, specifier possibilities, *etc.*) forces a high cost during the linguistic constraint verification phase of the parse. In the present incarnation of the system, only those principles relating to $\overline{\text{X}}$ (plus trace possibilities, but not trace principles) are accessed at precompilation time, leaving many of the principles to apply at execution time.

Base Templates
[START [C-MAX] [PUNC]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[V-MAX [V-SPEC] [V] [V-COMPLEMENT]]
[N-MAX [N-SPEC] [N] [N-COMPLEMENT]]
[P-MAX [P-SPEC] [P] [P-COMPLEMENT]]
[A-MAX [A-SPEC] [A] [A-COMPLEMENT]]
Adjunct Templates
[N [A-MAX] [N]] [N-MAX [N-MAX] [P-MAX]] [V-MAX [V-MAX] [P-MAX]]
[V-MAX [V-MAX] [ADV]] [N-MAX [N-MAX] [C-MAX]] [V [BE-AUX] [V]]
[I-MAX [DO-AUX] [I-MAX]] [I-MAX [BE-AUX] [I-MAX]]
[I-MAX [HAVE-AUX] [I-MAX]]
Trace Templates
[P-MAX [P-MAX-TRACE]] [N-MAX [N-MAX-TRACE]]
Non-Lexical Head Templates
[C-COMPLEMENT [I-MAX]] [C [E]]
[I-COMPLEMENT [V-MAX]] [I [E]]
Empty NP Templates
[N-MAX [PRO]]
Specifier Templates
[P-SPEC [ADV]] [A-SPEC [ADV]] [I-SPEC [N-MAX]]
[N-SPEC [N-MAX]] [V-SPEC [DO-AUX]] [V-SPEC [HAVE-AUX]]
[C-SPEC [ADV]] [C-SPEC [P-MAX]] [C-SPEC [N-MAX]]
Optional Specifier Templates
[N-MAX [N] [N-COMPLEMENT]] [P-MAX [P] [P-COMPLEMENT]]
[C-MAX [C] [C-COMPLEMENT]] [A-MAX [A] [A-COMPLEMENT]]
[V-MAX [V] [V-COMPLEMENT]]

Table 5.9: English Templates Generated by SET-UP-XBAR

Base Templates
[START [C-MAX] [PUNC]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[V-MAX [V-SPEC] [V] [V-COMPLEMENT]]
[N-MAX [N-SPEC] [N] [N-COMPLEMENT]]
[P-MAX [P-SPEC] [P] [P-COMPLEMENT]]
[A-MAX [A-SPEC] [A] [A-COMPLEMENT]]
Adjunct Templates
[N [A-MAX] [N]] [N [N] [A-MAX]]
[N-MAX [N-MAX] [P-MAX]] [V-MAX [V-MAX] [P-MAX]]
[V-MAX [V-MAX] [ADV]] [N-MAX [N-MAX] [C-MAX]]
[V [CL-DAT] [CL-ACC] [V]] [V [CL-DAT] [V]] [V [CL-ACC] [V]]
[V [CL-REF] [V]] [V [CL-REF] [CL-ACC] [V]]
[V [BE-AUX] [V]] [I-MAX [I] [HAVE-AUX] [BE-AUX] [V] [I-MAX]]
[I-MAX [V] [I-MAX]] [V-MAX [V-MAX] [N-MAX]]
Trace Templates
[P-MAX [P-MAX-TRACE]] [V [V-TRACE]] [I [I-TRACE]]
[HAVE-AUX [HAVE-AUX-TRACE]] [BE-AUX [BE-AUX-TRACE]]
[N-MAX [N-MAX-TRACE]]
Non-Lexical Head Templates
[C-COMPLEMENT [I-MAX]] [C [E]]
[I-COMPLEMENT [V-MAX]] [I [E]]
Empty NP Templates
[N-MAX [pro]]
[N-MAX [PRO]]
Specifier Templates
[P-SPEC [ADV]] [A-SPEC [ADV]] [N-SPEC [N-MAX]] [I-SPEC [N-MAX]]
[N-SPEC [DET]] [V-SPEC [HAVE-AUX]] [C-SPEC [ADV]] [C-SPEC [P-MAX]]
[C-SPEC [N-MAX]]
Optional Specifier Templates
[P-MAX [P] [P-COMPLEMENT]] [N-MAX [N] [N-COMPLEMENT]]
[C-MAX [C] [C-COMPLEMENT]] [A-MAX [A] [A-COMPLEMENT]]
[V-MAX [V] [V-COMPLEMENT]]

Table 5.10: Spanish Templates Generated by SET-UP-XBAR

Chapter 6

Parsing Component

The parser consists of two types of procedures (corresponding to the two boxes in figure 1.3 of chapter 1): (1) those that build structure (pushing, scanning, popping), relying primarily on phrase-structure templates generated at precompilation time; and (2) those that verify GB constraints (θ-Criterion, Empty Category Principle, *etc.*), acting as well-formedness tests on phrase-structures built by structure-building procedures.

When the parser is activated, the structure-building module (actually an augmented implementation of the Earley algorithm) draws upon the $\overline{\text{X}}$ templates, processing each word of input until no more structure-building actions apply. The GB constraint verification module takes over after each step (PUSH, SCAN and POP) of the structure-building module, modifying or eliminating structures derived thus far. The final resulting trees are stored in the *PARSE-TREES variable. Table 6.1 shows the actions the Earley and GB components must perform in tandem. The next two sections will describe the structure-building component and the linguistic constraint components in more detail.

6.1 Structure-Building Component: Augmented Earley Algorithm

The structure-building module uses the precompiled $\overline{\text{X}}$ context-free rules to *recognize* (accept) the input while simultaneously accessing the precompiled $\overline{\text{X}}$ templates to *parse* (assign structure

Action	*Earley Parser Tasks*	*GB Module Tasks*
PUSH	Expand nonterminal	Link traces to antecedents (and check Bounding) Predict complements Associate complements with heads Perform feature percolation
SCAN	Traverse terminal	Perform feature instantiation Determine argument structure Perform feature percolation
POP	Complete nonterminal	Link traces to antecedents (and check Bounding) Set up A and $\overline{\text{A}}$ positions Check Binding conditions Perform feature percolation Set up Government relations Check ECP Assign case Assign θ-roles Perform θ-role transmission Check θ-criterion

Table 6.1: Tasks of Earley and GB during Parsing

to) the input. In order to process the $\overline{\text{X}}$ rules and build $\overline{\text{X}}$ structure simultaneously, a stack must be used. Expanding a nonterminal symbol of an $\overline{\text{X}}$ rule corresponds to pushing the equivalent $\overline{\text{X}}$ template onto the stack; traversing a terminal symbol of an $\overline{\text{X}}$ rule corresponds to attaching the word to the open terminal symbol at the top of the stack; and completing a nonterminal symbol of an $\overline{\text{X}}$ rule corresponds to dropping the top of stack element under an open nonterminal in the stack element below it. For example, suppose we had the sequence of PUSH-SCAN-POP actions shown in table 6.2.[1]

The corresponding snapshots of the structure built in the stack is in figure 6.1. Snapshot (a) shows the state of the stack after the first three $\overline{\text{X}}$ templates (corresponding to the first three $\overline{\text{X}}$ rules) are pushed. Snapshot (b) shows the result of attaching the word **he** under **N** (*i.e.*, scanning the symbol **N** in the fourth $\overline{\text{X}}$ rule). Snapshot (c) shows the result of dropping

[1]The reader is assumed to be familiar with the Earley algorithm. In short, if the dot (represented as an asterisk (*)) precedes a nonterminal, that nonterminal is about to be pushed; if the dot precedes a terminal, that terminal is about to be scanned; and if the dot is not before any symbol, the nonterminal on the left-hand side of the rules is about to be popped. Each time a word is about to be scanned, a new *state set* is entered. Thus, a successful parse involves $n + 1$ state sets, where n is the number of words in the input.

PUSH	I-MAX ⇒ * I-SPEC I I-COMPLEMENT
	I-SPEC ⇒ * N-MAX
	N-MAX ⇒ * N N-COMPLEMENT
SCAN	N-MAX ⇒ N * N-COMPLEMENT (current word = he)
POP	I-SPEC ⇒ N-MAX *
	I-MAX ⇒ I-SPEC * I I-COMPLEMENT

Table 6.2: Sequence of PUSH-SCAN-POP Actions During Earley Parsing

(a) Push I-MAX
[N-MAX [N] [N-COMPLEMENT]]
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]

(b) Scan N
[N-MAX [N he] [N-COMPLEMENT]]
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]

(c) Pop N-MAX
[I-SPEC [N-MAX [N he]]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]

(d) Pop I-SPEC
[I-MAX [I-SPEC [N-MAX [N he]]] [I] [I-COMPLEMENT]]

Figure 6.1: Snapshots of Structure Built During PUSH, SCAN and POP

N-MAX under I-SPEC (*i.e.*, completing the N-MAX symbol in the fifth rule). Snapshot (d) shows the result of dropping I-SPEC under I-MAX (*i.e.*, completing the I-SPEC symbol in the sixth $\overline{X}$ rule).

Appendix G gives the bare Earley parser functions (no structure-building). PARSE-SENTENCE contains the main parsing loop that calls PREDICT, SCAN and COMPLETE. The function that calls PARSE-SENTENCE is EARLEY, which is among the top-level translation routines in appendix J. Appendix H (sections 1-3) shows the functions required for on-line structure-building. PUSH-STRUCT (called by PREDICT) builds structure at PUSH time, SCAN-STRUCT (called by SCAN) builds structure at SCAN time, and POP-STRUCT (called by COMPLETE) builds structure at POP time.

Because there are several choice-points in the derivation of a tree structure (*e.g.*, N-SPEC could expand to DET or N-MAX in English), multiple stacks must be maintained at all times. Thus, several structures may be considered at any point during parsing. One way that multiple structures are pruned without the use of GB constraints is by using a one-word lookahead facility at PUSH time. PUSH-STRUCT calls the CHECK-TERMINALS function (appendix H.1) that weeds out those templates that will not derive any of the categories of the current word.

Left-recursion presents a problem for building structure on-line. The danger of left-recursion is that it can force an infinite cycle of PUSH actions. For example, in English the possessive construction is a left-recursive structure:

[N-MAX [N-SPEC [N-MAX [N-SPEC [DET the]] [N man's]]] [N dog]]

In the stack scheme described above, it appears that PUSH-STRUCT will push for an **N-MAX**, and then an **N-SPEC**, and then an **N-MAX** again, and so on, without ever seeing the word *the*.

The way the bare Earley algorithm (without structure-building) handles this problems is that it never adds the same rule twice to the same state set. However, the analogous trick for structure-building (*i.e.*, refraining from pushing a structure onto the stack) will not necessarily work since the structure might need to arbitrarily branch left before reaching a terminal symbol. In order to handle this problem, the following trick has been devised: (1) during the PUSH stage, a given template is pushed onto the stack only *once*; (2) during the POP stage (presumably in a later state set) an empty left-recursive template is pushed onto the stack *before* the completed structure is dropped. This method is guaranteed to work because pushing recursive structure onto the stack *before* dominated structure is complete is ultimately equivalent to pushing recursive structure onto the stack *after* the dominated structure is complete. This "deferral" of the PUSH stage does not alter the outcome of the structure that is built. Furthermore, this method is guaranteed to terminate (*i.e.*, it does not cause an infinite loop). This is because a left-recursive structure may only be pushed once in a given state set, and the number of state sets is finite (assuming the sentence length is finite).

An example will make the above process clearer. Suppose we want to parse the **N-MAX** *the man's dog* as the subject of a sentence. Just before scanning *the*, the stack and state set look like this:

Stack	*State Set*
[N-MAX [N-SPEC [DET]] [N]]	I-MAX ⇒ * I-SPEC I I-COMPLEMENT
[I-SPEC [N-MAX]]	I-SPEC ⇒ * N-MAX
[I-MAX[I-SPEC] [I] [I-COMPLEMENT]]	N-MAX ⇒ * N-SPEC N
	N-SPEC ⇒ * DET

Note that we do not push for the `N-MAX` template again (under `N-SPEC`). Now a new state set is entered and the word *the* is scanned and attached:

Stack	*State Set*
[N-MAX [N-SPEC [DET the]] [N]]	N-SPEC ⇒ DET *
[I-SPEC [N-MAX]]	N-MAX ⇒ N-SPEC * N
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]	

Now the word *man's* can be scanned:

Stack	*State Set*
[N-MAX [N-SPEC [DET the]] [N man's]]	N-MAX ⇒ N-SPEC N *
[I-SPEC [N-MAX]]	
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]	

This completes the `N-MAX`; thus, it is time to POP. But before doing so, the parser notices that the popped symbol is part of a left-recursive cycle (this information is determined at precompilation time and saved in a variable called *LEFT-DERIVES-CYCLES). Now the stack is split into two possibilities; the first allows for the case where no left-recursion is required (*e.g.*, if the sentence were *the man* this parse would pan out); and the second allows for left-recursion:

Nonrecursive Stack	*Recursive Stack*	*State Set*
[I-MAX [I-SPEC [N-MAX [N-SPEC [DET the]] [N man's]]] [I] [I-COMPLEMENT]]	[N-MAX [N-SPEC [DET the]] [N man's]]	I-SPEC ⇒ N-MAX *
	[N-SPEC [N-MAX]]	
	[N-MAX [N-SPEC] [N] [N-COMPLEMENT]]	
	[I-SPEC [N-MAX]]	
	[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]	

In the first case the `N-MAX` is dropped "normally." However, this parse is ruled out as soon as it is determined that the next word is not of category I(nfl). (In fact, the aforementioned one-word lookahead facility rules out such parses well ahead of time.) In the second case, the `[N-SPEC [N-MAX]]` and `[N-MAX [N-SPEC] [N] [N-COMPLEMENT]]` templates have been dropped between the first and second stack elements. Now the top of stack element is attached under the `N-SPEC` of the empty template, and then the completed `N-SPEC` node is attached under `N-MAX`:

Stack	*State Set*
[N-MAX [N-SPEC [N-MAX [N-SPEC [DET the]] [N man's]]] [N] [N-COMPLEMENT]]	N-SPEC ⇒ N-MAX * N-MAX ⇒ N-SPEC * N N-COMPLEMENT
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]	

The DROP-INTO-NEXT-LOWER and DROP-RECURSE-NEXT-LOWER functions perform the "normal" and recursive popping actions respectively. (These are given in appendix H.)

If this provision for left-recursion were not part of the implementation, there would be no way to build structure on-line. An alternative approach to recovering the parse would be to build the structure *after* the input has been recognized.[2] Unfortunately, this approach does not allow on-line structure to be built, thus disallowing the GB component to work in tandem with the Earley parser. Instead, GB principles would need to be applied as well-formedness conditions on the final output of the parser.[3] However, the motivation for building structure on-line in the first place is to allow ungrammatical parses to be weeded out long before the end of input is reached. The structure-building routines act as an interface between the bare Earley parser and the GB component. The next section describes how the GB component performs the task of updating and eliminating structure during parsing.

6.2 Linguistic Constraints of Parser

The task of the GB constraint module is three-fold: first, it weeds out bad parses (*e.g.*, a parse in which a trace has no antecedent, or in which an antecedent is too far away); second, it tries possibilities unavailable to the structure-building module (*e.g.*, it expands complements by projecting maximal projections from the subcategorization frames of lexical heads); and third, it extends the parse to the point where all GB constraints are satisfied, and the structure-building module can take over again (*e.g.*, it ensures that arguments are θ-marked and it assigns indices to trace-antecedent pairs).

[2]This can be done by tracing back through pointers associated with state set rules, thus computing the parse tree(s) in time proportional to the length of the sentence.

[3]Sharp (1985) takes this approach (*i.e.*, his system uses GB principles as well-formedness conditions on final output).

Slot	*Possible Values*
CAT	$\{$X, X-SPEC, X-COMPLEMENT $\mid$ X $\in$ *CURRENT-BASIC-CATEGORIES $\cup$ *CURRENT-PRE-TERMINALS$\}$
WORD	morphologically analyzable word
GENDER	language-particular gender (*e.g.*, masc, fem, neut)
PERS	language-particular person (*e.g.*, p1 p2 p3)
NUMBER	language-particular number (*e.g.*, pl, sg, mass)
TENSE	language-particular tense (*e.g.*, pres, past)
CASE	language-particular case assigned to node (*e.g.*, obj, nom)
ROLE	language-particular role assigned to node (*e.g.*, agent, goal)
LANGUAGE-PARTICULAR-FEATURES	$\{$F $\mid$ F $\in$ *CURRENT-FEATURES and F $\notin$ PERS $\cup$ GENDER $\cup$ NUMBER $\cup$ TENSE $\cup$ CASE $\cup$ ROLE$\}$
TRANSLATION	target language translation of word
SUBCATEGORIZATION	$\{$(X-MAX$_1$ $\ldots$ $X - MAX_n$) $\mid$ X $\in$ *CURRENT-BASIC-CATEGORIES$\}$
EXTERNAL-CATS	$\{$(X-MAX$_1$ $\ldots$ $X - MAX_n$) $\mid$ X $\in$ *CURRENT-BASIC-CATEGORIES$\}$
THETA-ROLES	lists of roles from *CURRENT-CANONICAL-SEMANTIC-MAPPINGS (*e.g.*, goal, patient)
EXTERNAL-ROLES	lists of roles from *CURRENT-CANONICAL-SEMANTIC-MAPPINGS (*e.g.*, agent)
C-GOVERNED?	list of nodes (c-governing the node)
S-GOVERNED?	list of nodes (s-governing the node)
PROPER-GOVERNED?	list of nodes (proper-governing the node)
C-GOVERNS?	list of nodes (c-governed by the node)
S-GOVERNS?	list of nodes (s-governed by the node)
PROPER-GOVERNS?	list of nodes (proper-governed by the node)
A-POSITION?	t or nil
A-BAR-POSITION?	t or nil
TRACE	a node (antecedent that binds a node)
ANTECEDENT	a node (trace bound to node)
COMPLEMENTS-FILLED	t or nil

Table 6.3: Description of a Tree Node

Before describing the tasks associated with the GB component we first examine the type of information included in the nodes of the tree structures built by the structure-building module. Each node has several slots associated with values. For example, the slot CAT specifies the category of the node in the tree. CAT is the only slot which must have a value other than NIL at all times; the other slots may or may not be filled during the parse of a sentence. Table 6.3 shows the slots and possible values.

The tasks associated with each module of the GB component are described below. (Appendices E and F contain the Lisp representation of the parameters and principles that comprise the GB component.) The reader may refer to table 6.1 for an overview of the order in which the routines of the GB modules are accessed.

6.2.1 $\overline{\text{X}}$ Module

During parsing, the $\overline{\text{X}}$ module is responsible for four tasks: predicting complements; associating complements with heads; determining feature information and argument structure of lexical heads; and feature percolation. These four tasks are discussed in the following sections. The $\overline{\text{X}}$ routines accessed at parsing time are in appendices F.1.2–F.1.6.

6.2.1.1 Predicting Complements

Recall that the constituent order parameter allows complement to occur in one of three positions: before the specifier and head, between the specifier and head, or after the specifier and head. From the point of view of parsing, it would seem that the easiest configurations to handle are those in which the head precedes the complement.[4] In such configurations the head can be used to predict complements before the complements are actually seen. By contrast, if the complement precedes the head, it is difficult to predict the head that is to follow a given complement during parsing.[5]

[4]These configurations are called *head-initial* configurations; however, this term is misleading since the specifier-head-complement ordering is included among the *head-initial* configurations, even though the head is not really initial. Throughout this report I will refer to any language that requires the head to precede the complement configuration as head-initial (regardless of where the specifier is positioned); similarly, any language that requires the complement to precede the head is referred to as head-final.

[5]To date, there is no translation system that includes a uniform method of handling the head-initial/head-final distinction. Sharp does not address this issue (possibly because both languages translated by the system are head-initial), and other translation systems (*e.g.*, METAL (1984a)) use language-specific rules to handle

If it could be shown that human processing of head-final languages is "harder" (*e.g.*, takes longer) than processing of head-initial languages, this might justify the use of two different methods of parsing complements. However, since this does not appear to be the case, a uniform method of parsing complements has been constructed. The task of complement prediction is performed at PUSH time. If an unexpanded complement symbol is encountered, it must be expanded according to the subcategorization frame of the *closest head* of the appropriate category, regardless of whether the head has already been parsed.[6] Furthermore, the complements that are predicted must be associated with the corresponding head. If the head already has been parsed, the argument structure information is already stored in the head (in the SUBCATEGORIZATION slot) and must be matched one-to-one with the predicted complements. On the other hand, if the head has not been parsed, the argument structure of the predicted complement is inserted directly in the head (in the SUBCATEGORIZATION slot) and must be matched one-to-one with the subcategorization frame of the head once it is parsed. (The details of associating complements with heads will be discussed in the next section.)

The function LOCATE-COMPLEMENTS (in appendix F.1.2) determines the complements of a head by looking in the input to the left of the current word in a head-initial language, or by looking in the input to the right of the current word in a head-final language. For example, suppose a verbal complement needs to be expanded during the parse of a sentence. Figure 6.2 shows the state of the stack just before processing the phrase *the boy* for the English sentence (54) and the Navajo translation sentence (55).[7]

(54) the boy saw the girl

(55) ashkii at'ééd yiyiiltsą́

In figure 6.2 (a) the head (= *see*) of `V-COMPLEMENT` has already been processed since the language is head-initial. By contrast, in figure 6.2 (b) the head of `V-COMPLEMENT` has not yet

complement prediction for all languages. By contrast, Abney's GB parser (1987) *does* handle the complement-prediction problem since all phrases of a sentence must be *licensed* (regardless of ordering requirements) before the sentence is accepted. The notion of *licensing* will not be discussed here, but for our purposes it is sufficient to equate licensing with θ-marking.

[6]The prediction of complements using the *closest-head* algorithm does not always work. This issue will be addressed in chapter 9.

[7]The Navajo sentence literally translates to *the boy the girl saw* since Navajo is a head-final language.

(a) English	(b) Navajo
[V-MAX [V see [p1 past]] [V-COMPLEMENT]]	[V-MAX [V-COMPLEMENT] [V]]
[I-COMPLEMENT [V-MAX]]	[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX the girl]] [I E] [I-COMPLEMENT]]	[I-MAX [I-COMPLEMENT] [I-SPEC] [I]]
[C-MAX [C E] [C-COMPLEMENT [I-MAX]]]	[C-MAX [C-COMPLEMENT [I-MAX]] [C]]

Figure 6.2: State of Parse Prior to Expansion of a Verbal Complement

(a) English	(b) Navajo
[N-MAX [N-SPEC] [N] [N-COMPLEMENT]]	[N-MAX [N-COMPLEMENT] [N-SPEC] [N]]
[V-COMPLEMENT [N-MAX]]	[V-COMPLEMENT [N-MAX]]
[V-MAX [V see [past p3 sg]] [V-COMPLEMENT]]	[V-MAX [V-COMPLEMENT] [V]]
[I-COMPLEMENT [V-MAX]]	[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX the girl]] [I E] [I-COMPLEMENT]]	[I-MAX [I-COMPLEMENT] [I-SPEC] [I]]
[C-MAX [C E] [C-COMPLEMENT [I-MAX]]]	[C-MAX [C-COMPLEMENT [I-MAX]] [C]]

Figure 6.3: State of Parse After Expansion of a Verbal Complement

been seen. In both cases, the LOCATE-COMPLEMENTS routine determines that *see* is the head corresponding to **V-COMPLEMENT**. It does this by looking to the left of the current word *the* in the head-initial case, and to the right of the current word *the* in the head-final case; when it finds a word of category V, the subcategorization frame of the word is turned into a tree structure and pushed onto the stack. Since *see* takes an **N-MAX** complement, the **N-MAX** template is pushed (see figure 6.3).[8] Also, an additional complement template (not shown above) could be pushed onto the stack (thus splitting the stack into two stacks kept in parallel) since *see* takes a *p-goal* (i.e., a *preposition-goal* or a *goal* instantiated as **P-MAX**). However, this

[8]Actually, *see* takes a *goal*, which is turned into N-MAX via the CSR mapping used by the GET-CATEGORY-SUBCAT function.

(a) English	(b) Navajo
[N-MAX [N-SPEC] [N] [N-COMPLEMENT]]	[N-MAX [N-SPEC] [N]]
[V-COMPLEMENT [N-MAX]]	[V-COMPLEMENT [N-MAX]]
[V-MAX [V see [past p3 sg]] [V-COMPLEMENT]]	[V-MAX [V-COMPLEMENT] [V]]
[I-COMPLEMENT [V-MAX]]	[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX the girl]] [I E] [I-COMPLEMENT]]	[I-MAX [I-COMPLEMENT] [I-SPEC] [I]]
[C-MAX [C E] [C-COMPLEMENT [I-MAX]]]	[C-MAX [C-COMPLEMENT [I-MAX]] [C]]

Figure 6.4: State of Parse After Noun Complement Has Been Eliminated

possibility is ruled out immediately because of the one-word lookahead facility described in section 6.1.

Notice that in figure 6.3 (a) `N-SPEC` is in the correct position to have the determiner *the* attached under it during the next SCAN step. On the other hand, in figure 6.3 (b) `N-SPEC` is not available yet since `N-COMPLEMENT` is the next open symbol. Again, the closest head (of category N this time) must be located in order to determine the expansion of `N-COMPLEMENT`. Since the noun *ashkii* (= boy) is obligatorily intransitive (the LOCATE-COMPLEMENTS function takes care of the intransitive as well as the transitive case), the `N-COMPLEMENT` symbol is eliminated and the `N-SPEC` is now the next open symbol. Figure 6.4 shows the state of the parse after `N-COMPLEMENT` has been eliminated.

The interface between the Earley parser and the LOCATE-COMPLEMENTS routine is the GET-LEFT-MOST-DERIVES function in appendix H.1.

During the prediction of complements, clitics must be taken into account (if the language allows clitics). This is because in certain languages clitics can serve as internal arguments of a head. For example, in the sentence *lo vi* (= I saw him), the clitic *lo* is actually taking the place of an internal argument that would normally occur to the right of the verb *vi*.

The function TAKE-CLITICS-INTO-ACCOUNT (see appendix F.1.2) has the task of generating additional complement structures if clitics are allowed to be associated with the head. These complement structures are *subsets* of the original subcategorization frame found by

(a) C as a Non-Lexical Category	(B) C as a Lexical Category
[C-MAX [C E] [C-COMPLEMENT]]	[C-MAX [C] [C-COMPLEMENT]]

Figure 6.5: Structure Built by a Potentially Non-Lexical Category

LOCATE-COMPLEMENTS. For example, the verb *poner* (= put) takes an N-MAX and a P-MAX; thus, LOCATE-COMPLEMENTS predicts the complement structures `[((N-MAX) (P-MAX)), ((P-MAX) (N-MAX))]` and TAKE-CLITICS-INTO-ACCOUNT generates the additional complement structures `[INTRANS,((N-MAX) (P-MAX))]`. `INTRANS` takes care of the case where clitics replace both arguments; `(N-MAX)` takes care of the case where a clitic replaces `(P-MAX)`; and `(P-MAX)` takes care of the case where a clitic replaces `(N-MAX)`. Thus, the full complement structure to be pushed for the verb *poner* is: `[((N-MAX) (P-MAX)), ((P-MAX) (N-MAX)), INTRANS, (N-MAX) (P-MAX)]`. This allows clitics to replace two, one, or none of the arguments; in addition, if clitic doubling is allowed, the full complement structure *and* clitics may be used. (The clitic-doubling case is checked later by the θ-module.)

Recall that the complement structures for certain heads are expanded at precompilation time; these are the structures associated with heads that are nonlexical (*e.g.*, C and I in both Spanish and English). When such a complement is encountered, it is not always necessary to predict the complement structure in the same way that the complements for lexical heads are predicted. On the other hand, a potentially nonlexical head might correspond to a lexical element (*e.g.*, C could be empty, or it could contain the complementizer *that*). During the lookahead stage mentioned in section 6.1, this possibility is accounted for. CHECK-TERMINALS (see appendix H.1) adds templates in which the head is allowed to contain a lexical element, and the default complement is *not* used. For example, if the template of figure 6.5(a) is passed to CHECK-TERMINALS, the template of figure 6.5(b) is generated via UPDATE-NON-LEXICAL-CATS. Then at PUSH time, the LOCATE-COMPLEMENTS function looks in the input for a word of category C (like *that*) and pushes for a complement structure corresponding to the subcategorization frame of the word, just as if C were an ordinary lexical category.

6.2.1.2 Associating Complements With Heads

Once a complement structure has been pushed onto a stack, the complements must be associated with the appropriate head. This the task of PERFORM-SUBCATEGORIZATION-CHECK (see appendix F.1.4), which is called at PUSH time by PUSH-TEMPLATES-ONTO-STACK (see appendix H.1). This function matches complements to their transitive heads, and establishes the intransitivity of heads without complements.

In a head-initial language, the head will already have been parsed before this subcategorization check takes place; thus, the complements must match the subcategorization frame stored in the SUBCATEGORIZATION slot of the head before the complement can be associated with the head. By contrast, in a head-final language, the complement is associated with the head without the matching check.

For example, in parsing the `V-MAX` of sentences (54) and (55) (our English and Navajo examples of the last section), PERFORM-SUBCATEGORIZATION-CHECK occurs *after* the verb is scanned in (54), and *before* the verb is scanned in (55). In the first case, the subcategorization possibilities for *see* are already in the node:

CAT:	v
WORD:	see
TENSE:	past
THETA-ROLES:	[intrans (goal) (p-goal)]
SUBCATEGORIZATION:	[intrans (n) (p)]

Thus, after the `N-MAX` complement is predicted by LOCATE-COMPLEMENTS, PERFORM-SUBCATEGORIZATION-CHECK must match this to one of the elements in the SUBCATEGORIZATION slot. In this case, the **(n)** entry matches; thus, the SUBCATEGORIZATION slot is modified to contain **(n)**. In the second case, the subcategorization possibilities are not yet available; thus, when PERFORM-SUBCATEGORIZATION-CHECK is executed, the SUBCATEGORIZATION slot is automatically updated to contain **(n)**:

CAT:	v
WORD:	yiyiiłtsą́
TENSE:	past
THETA-ROLES:	nil
SUBCATEGORIZATION:	[(n)]

Once the SUBCATEGORIZATION slot has been filled, the COMPLEMENTS-FILLED slot is set to T (if it has not already been set).

The function that updates the SUBCATEGORIZATION and COMPLEMENTS-FILLED slots is ADD-SUBCAT-INFO (see appendix F.1.4). If a head is determined to be intransitive (*i.e.*, no complements can be pushed), the function MAKE-INTRANSITIVE (also in appendix F.1.4) sets the SUBCATEGORIZATION slot to be INTRANS and the COMPLEMENTS-FILLED slot to be T.

The interface between the Earley parser and PERFORM-SUBCATEGORIZATION-CHECK is PUSH-TEMPLATES-ONTO-STACK (in appendix H.1).

6.2.1.3 Determining Feature Information and Argument Structure of Lexical Heads

At SCAN time, the function INSTANTIATE-FEATURES is called to set up the node slots corresponding to the word that has been scanned. (See appendix F.1.5.) The translation of the word (in the current source language) is stored in the translation slot, the word is stored in the word slot, and the features of the word are stored in the PERS, GENDER, NUMBER, TENSE and the LANGUAGE-PARTICULAR-FEATURES slots of the node. All legal features of a word must be included in the FEATURES parameter setting supplied by the user. This parameter is one of three parameters that is not part of GB theory, but is specific to the UNITRAN system.[9]

Finally, the function SET-INTERNAL-AND-EXTERNAL-ARGUMENTS is called to establish the structure and roles of internal and external arguments. For example, the verb *vio* has the node instantiation in figure 6.6.

The internal and external θ-roles assigned by a lexical element are retrieved by GET-SUBCATEGORIZATION and GET-EXTERNAL, respectively. The internal and external structural realizations are retrieved by GET-CATEGORY-SUBCAT and GET-CATEGORY-EXTERNAL, respectively. (The structural realizations are determined by the CSR mapping in appendix F.2.1.) SET-INTERNAL-AND-EXTERNAL-ARGUMENTS uses the values returned by these functions to set up the θ-ROLES, EXTERNAL-ROLES, SUBCATEGORIZA-

[9]The SPLITS-AND-MERGES parameter discussed in section 5.1 as well as the MATCH-LISTS to be discussed in the next section are also translation parameters that are specific to UNITRAN.

CAT:	v
WORD:	ver
GENDER:	nil
PERS:	p3
NUMBER:	sg
TENSE:	past
LANGUAGE-PARTICULAR-FEATURES:	nil
TRANSLATION:	[(see)]
SUBCATEGORIZATION:	[intrans (n) (p)]
EXTERNAL-CATS:	[(n)]
THETA-ROLES:	[intrans (goal inanimate) (p-goal animate)]
EXTERNAL-ROLES:	[(agent animate)]
COMPLEMENTS-FILLED:	t

Figure 6.6: Node Instantiation of Word *vio*

TION and EXTERNAL-CATS slots of a node.[10]

Note that the SUBCATEGORIZATION and COMPLEMENTS-FILLED slots are set during feature instantiation. This is similar to the outcome of the ADD-SUBCAT-INFO and MAKE-INTRANSITIVE functions discussed in the last section, except that these two slots are filled at SCAN time instead of PUSH time, and they are filled *only* if neither ADD-SUBCAT-INFO nor MAKE-INTRANSITIVE has already filled them (*i.e.*, if the language is a head-initial language). This is tested by first checking to see if the COMPLEMENTS-FILLED slot has been set. If it has not been set, the SUBCATEGORIZATION slot is filled and the COMPLEMENTS-FILLED slot is set; thus, when the complements are later predicted and associated with the head, the subcategorization frame is updated to reflect the structure of the predicted complements (as described in the last section). Otherwise, the SUBCATEGORIZATION and COMPLEMENTS-FILLED slots are left unchanged since they already contain the complement information. (This will be the case in a head-final language.)

The interface between the Earley parser and INSTANTIATE-FEATURES is the SUBSTITUTE-SCANNED-NODE routine in appendix H.2.

[10]The LOCATE-COMPLEMENTS function described in section 6.2.1.1 also uses the GET-CATEGORY-SUBCAT function to predict complement structures. Thus, the subcategorization information may already be present in the node (set by ADD-SUBCAT-INFO or MAKE-INTRANSITIVE) when INSTANTIATE-FEATURES is executed. This point will be addressed presently.

6.2.1.4 Feature Percolation

The PERCOLATE-FEATURES function is called at POP, SCAN and PUSH time. (See appendixF.1.6.) The purpose of this function is to propagate features of elements up to superior nodes, and to test for agreement between specifier and head nodes.[11] For example, in the Spanish phrase *el libro*, the determiner *el* must match in number and gender with its head *libro.* Since both are singular and masculine, feature matching succeeds. Thus, when a word is scanned, its features are percolated up to its superior node: if the word is in specifier position, the features are percolated up to the specifier node; if the word is in complement position, the features are percolated up to the complement node; and if the word is in head position, the features are percolated up to the maximal projection. Propagation to superior nodes is performed by PROPAGATE-FEATURES (also in appendix F.1.6).

When both the specifier and head have been completed, the features of the specifier are tested for agreement against the features in the maximal projection. For example, suppose we are parsing the phrase *have eaten.* First the word *have* is parsed and attached in specifier position of `V-MAX` at SCAN time. Then its features are percolated up to the `V-SPEC` node:

```
[V-MAX [V-SPEC [pres p1 sg] [HAVE-AUX have [pres p1 sg]]]
   [V] [V-COMPLEMENT]]
```

When *eaten* is scanned, its features are percolated up to the maximal projection:

```
[V-MAX [perf] [V-SPEC [pres p1 sg] [HAVE-AUX have [pres p1 sg]]]
   [V eat [perf]] [V-COMPLEMENT]]
```

Now that the phrase is completed, feature matching takes place. At this time a translation parameter, MATCH-LISTS, is accessed to perform agreement testing between the specifier and maximal projection. Appendix D contains the MATCH-LISTS parameter settings for Spanish and English. The value of the MATCH-LISTS for the source language is stored in the *CURRENT-MATCH-LISTS variable. Each match list follows a keyword corresponding to the feature being tested. For example, the tense feature has the following match-list in Spanish:

(((PRES PAST FUT) (PERF)))

The format for a match-list is:

[11] The inspiration for this type of feature matching came from Berwick (1985), p. 75.

$$\begin{array}{ll}(((\alpha_{11}\ \alpha_{12}\ \ldots\ \alpha_{1k}) & (\beta_{11}\beta_{12}\ \ldots\ \beta_{1l})) \\ ((\alpha_{21}\ \alpha_{22}\ \ldots\ \alpha_{2j}) & (\beta_{21}\beta_{22}\ \ldots\ \beta_{2m})) \\ & \ldots \\ ((\alpha_{n1}\ \alpha_{n2}\ \ldots\ \alpha_{ni}) & (\beta_{n1}\ \beta_{n2}\ \ldots\ \beta_{np})))\end{array}$$

where each α corresponds to a feature that a specifier might have, and each β corresponds to a feature a head might have. A valid spec/max match occurs when a specifier has a feature α_{df} and its maximal projection has a feature β_{dg}.

Returning to our example, we will follow the steps of the V-SPEC/V-MAX matching. First the gender of the specifier is matched against the gender of the V-MAX. Since both are NIL, the test succeeds. Now the number of the specifier is matched against the number of the maximal projection. The specifier feature **sg** automatically matches the number of the V-MAX (which is NIL).[12]

Whenever a match succeeds, the maximal projection is updated to contain a combination of the matching features;[13] thus, the NUMBER slot of the **V-MAX** node is updated to contain the **sg** feature. The person test proceeds in a similar fashion, that is, the specifier feature **p1** automatically matches the person feature of the maximal projection; thus, **p1** is inserted in the PERS slot of the maximal projection.

Finally, tense matching takes place. According to the match-list associated with the :TENSE keyword, the **perf** feature of the maximal projection must match either a **pres**, **past**, or **fut** feature in specifier position. Since V-SPEC has the **pres** feature in its TENSE slot, the test succeeds, and the **pres** and **perf** features are inserted into the TENSE slot of the maximal projection. The result is the following:

```
[V-MAX [pres perf p1 sg]
   [V-SPEC [pres p1 sg] [HAVE-AUX have [pres p1 sg]]]
   [V eat [perf]] [V-COMPLEMENT]]
```

The function that performs spec/max agreement is CHECK-SPEC-MAX-AGREEMENT; this function calls CHECK-AGREEMENT, which performs feature matching. Feature combin-

[12]The auxiliary *have* could be analyzed as any of [p1 sg], [p1 pl], [p2 sg], [p2 pl], or [p3 pl]. However, only the [p1 sg] analysis is being shown here.

[13]Also, if the specifier has no agreement features, the agreement features of the max are copied to the specifier node. For example, the parse [I-MAX [e [pro]] [ver p1 sg past]] (corresponding to the sentence *vi* = *I saw*) contains a *pro* in specifier position with no features. However, the head I has tense and agreement features; thus, when spec/max agreement is tested, the [p1 sg] features are copied to the *pro* node. This forces *pro* to be translated as *I* in English. We have already seen an example of this agreement feature copying in the parsing example of chapter 4.

ing is performed by CHECK-MATCH-LIST, and spec/max updating is performed by UPDATE-MAX-AND-SPEC. Appendix F.1.6 contains all of these agreement-checking functions.

Feature percolation occurs at the completion of a phrase (POP) and after analyzing a word (SCAN). Thus, two functions serve as an interface between the Earley parser and PERCOLATE-FEATURES: DROP-INTO-NEXT-LOWER (in appendix H.3) and SUBSTITUTE-SCANNED-NODE (in appendix H.2). In addition, PERCOLATE-FEATURES is called at PUSH time (indirectly) when a trace is predicted. The function CHECK-TRACE-LINKS, which is part of the Bounding module, requires feature percolation to take place because the trace that is predicted might be the head of a phrase. In such a case, the features of the trace are percolated up to the maximal projection just as if the trace were a lexical head. For example, the phenomenon of V-preposing leaves a trace behind in the head position of a `V-MAX`:

[I-MAX
  [V α [tns agr]]$_i$
  [I-MAX ... [V-MAX [V e]$_i$ [V-COMPLEMENT]]]]

Continuing with our example, first the features `[tns agr]` of the preposed verb (`[V` α`]`$_i$) are associated with the trace `[V e]`$_i$, and then during feature percolation these features are percolated up to the `V-MAX`:

[I-MAX
  [V α [tns agr]]$_i$
  [I-MAX ... [V-MAX [tns agr] [V e [tns agr]]$_i$ [V-COMPLEMENT]]]]

The final feature percolation routine that is used is PERCOLATE-TENSE-AND-AGR (see appendix F.1.6). This function is called at POP time (by DROP-INTO-NEXT-LOWER). If the completed item is V-MAX, its features are propagated up to the dominating I(nfl) node. for example, if the phrase *have eaten* (from our last example) were part of the larger phrase *I have eaten*, the stack would appear as follows upon completion of *have eaten*:

[V-MAX [pres perf p1 sg]   [V-SPEC [HAVE-AUX have]] [V eat] [V-COMPLEMENT]]
[I-MAX [I-SPEC [N-MAX [p1 sg] [N I]]]   [I E] [I-COMPLEMENT [V-MAX]]]

The tense (`[pres perf]`) and agreement (`[p1 sg]`) information associated with `V-MAX` is propagated to the I(nfl) node in the second stack element:

```
[V-MAX [pres perf p1 sg]
   [V-SPEC [HAVE-AUX have]] [V eat] [V-COMPLEMENT]]
-------------------------------------------------
[I-MAX [I-SPEC [N-MAX [p1 sg] [N I]]]
   [I E [pres perf p1 sg]] [I-COMPLEMENT [V-MAX]]]
```

After the stack is popped (on the next call to DROP-INTO-NEXT-LOWER), feature percolation takes place:

```
[I-MAX [pres perf p1 sg]
   [I-SPEC [N-MAX [p1 sg] [N I]]]
   [I E [pres perf p1 sg]]
   [I-COMPLEMENT
      [V-MAX [pres perf p1 sg]
          [V-SPEC [HAVE-AUX have]]
          [V eat] [V-COMPLEMENT]]]]
```

The spec/max matching succeeds since the agreement features of `N-MAX` (`[p1 sg]`) match the agreement features of `I-MAX`. Note that the tense features automatically match (since `N-MAX` has no tense features), and the gender features also automatically match (since neither the specifier nor the maximal projection have gender features).

6.2.2 θ Module

The θ module is accessed at POP time by DROP-INTO-NEXT-LOWER (in appendix H.3). The reason POP time is convenient for θ-role assignment is that all arguments of the phrase are complete; thus, they are ready to receive internal and external θ-roles. The tasks this module is responsible for are θ-role assignment (subject to feature requirements and the Visibility Condition), θ-role transmission (for antecedents and clitics), and θ-Criterion checking. The following sections discuss these three tasks.

6.2.2.1 θ-Role Assignment

At the completion of a phrase, it is necessary for θ-role assignment to take place. θ-roles are assigned both internally and externally to arguments. For example, the phrase *gave him a book* is parsed as:

```
[V-MAX [past]
   [V give]
   [V-COMPLEMENT
      [N-MAX [N him]]
      [N-MAX [N-SPEC [DET a]] [N book]]]]
```

The verb *gave* assigns an internal θ-role of *patient* to *him* and *goal* to *a book*. If this phrase were part of a larger phrase *I gave him a book*, the structure would be:

```
[I-MAX [past p1 sg]
  [I-SPEC [N-MAX [p1 sg] [N I]]]
  [I E [past p1 sg]]
  [I-COMPLEMENT
    [V-MAX [past]
      [V give]
      [V-COMPLEMENT
        [N-MAX [N him]]
        [N-MAX [N-SPEC [DET a]] [N book]]]]]]
```

Here *give* assigns an external θ-role of *agent* to *I*.

The internal and external θ-roles assigned by *give* are stored in the THETA-ROLES and EXTERNAL-ROLES slots, respectively. (Recall that INSTANTIATE-FEATURES fills this slot when an assigner is scanned.) When the `V-MAX` node is complete, internal θ-roles are assigned to *him* and *a book*, and when the `I-MAX` node is complete, external θ-role is assigned to `I`.

The function that performs θ-role assignment is PERFORM-THETA-ASSIGNMENT (see appendix F.2.2). This function takes a phrase (the completed top-of-stack element) and determines the θ-roles to be assigned externally to its specifier, and internally to its complement. Thus, two θ-role assignments take place: the first is by an assigner in a lower phrase to elements in specifier position; and the second is by the head of the phrase to elements in complement position.

The function ASSIGN-TO-ARG takes an assigner node and performs both types of θ-role assignment: first it determines the assignees that are to receive θ-roles; then it calls ASSIGN-THETA to discharge θ-roles from the assigner node to the assignee nodes. ASSIGN-THETA first checks if the features associated with the roles to be assigned match the features of the assignee node; then it checks the Visibility Condition (*i.e.*, it ensures that arguments are assigned case); finally it discharges θ-roles to arguments. Both ASSIGN-TO-ARG and ASSIGN-THETA are in appendix F.2.2.

As an example, suppose the completed phrase *I saw* is being popped. The structure of the completed phrase is:

```
[I-MAX [past p1 sg]
   [I-SPEC [N-MAX [p1 sg] [N I]]]
   [I E [past p1 sg]]
   [I-COMPLEMENT [V-MAX [past] [V see]]]]
```

The head of the `I-MAX` phrase, `I`, has no θ-roles to discharge (since it does not contain a lexical element); thus, `I-COMPLEMENT` does not receive a θ-role. However, `I-SPEC` contains a noun phrase; thus, it must receive θ-role externally. PERFORM-THETA-ASSIGNMENT calls FIND-SPEC-ASSIGNER-NODE to search for the head of the phrase in complement position; this node is then used as the assigner for specifier position. In our example, the `V` node corresponding to the verb *see* is established as an external role assigner since it contains **`[agent animate]`** in its EXTERNAL-ROLES slot. ASSIGN-TO-ARG passes the assignee `[N-MAX [N I]]` and assigner `[V see]` down to ASSIGN-THETA, which then checks that the *animate* feature associated with the *agent* θ-role matches the features associated with the assignee node. Since `[N-MAX [N I]]` has an *animate* feature marking, feature-matching succeeds.

Now the Visibility Condition is checked (see CHECK-VISIBILITY-CONDITION in appendix F.2.2). Since nominative case has been assigned to `[N-MAX [N I]]`, the Visibility Condition is satisfied and θ-role assignment takes place. The final instantiation of the `N-MAX` node containing `[N I]` is:

CAT:	n
NUMBER:	sg
LANGUAGE-PARTICULAR-FEATURES:	animate
CASE:	nom
ROLE:	agent

The interface between the Earley parser and the θ-role assignment routines is DROP-INTO-NEXT-LOWER (in appendix H.3).

6.2.2.2 θ-Role Transmission

Once a θ-role has been assigned to an argument, θ-role transmission must take place (to ensure that antecedents of the assigned argument also receive θ-role). For example, in the sentence *he$_i$ seems e$_i$ to eat a lot*, the trace e$_i$ receives external θ-role from the assigner *eat*, and the antecedent receives θ-role via θ-role transmission. Note that in order to receive θ-role from *eat*, e$_i$

must already be assigned case so that Visibility Condition is satisfied. This requirement is satisfied since he$_i$ transmits case to e$_i$.[14] The function TRANSMIT-ROLE-TO-ANTECEDENT performs θ-role transmission for trace-antecedent pairs.[15]

In addition to θ-role transmission from trace to antecedent, there is `clitic/N-MAX` θ-role transmission. If the *CLITIC-THETA-TRANSMIT variable is T (this variable corresponds to the CLITIC-DOUBLING parameter of θ-theory), then θ-transmission applies. The function THETA-TRANSMIT (in appendix F.2.3) tests that the *CLITIC-THETA-TRANSMIT variable is T and that the `N-MAX` node to which θ is to be transmitted has case; it then calls TRANSMIT-THETA-FROM-CLITIC (also in appendix F.2.3) which performs the θ-role transmission. This function locates candidate clitics for θ-role transmission; when it finds a clitic that has been assigned case and θ-role (FIND-CLITICS-WITH-CASE-AND-ROLE), then the role of the clitic is assigned to the `N-MAX` node. Note that the clitic must first receive θ-role (by virtue of its position) from an internal θ-role assigner. For example, suppose the sentence *lo vi a él* (= *I saw him*) is being parsed. The structure of this is:

```
[I-MAX [past p1 sg]
   [I-SPEC [N-MAX e [pro p1 sg]]]
   [I E [past p1 sg]]
   [I-COMPLEMENT
      [V-MAX [past p1 sg]
         [V [CL lo [p3 sg animate masc obj]] [V ver]]
          [V-COMPLEMENT
             [P-MAX [P a]
                [P-COMPLEMENT
                   [N-MAX [p3 sg animate masc obj] él]]]]]]]
```

Prior to assignment of θ-roles, the clitic *lo* is assigned objective case by the verb *ver*, and the `N-MAX` *él* is assigned objective case by the preposition *a*. When ASSIGN-TO-ARG is called, all internal arguments of the verb *ver* are located via LOCATE-SPEC-OR-COMP. If the clitic *lo* were not present, the only argument that would be found is *él*, and θ-role assignment would follow directly. However, both arguments *lo* and *él* are found, and ASSIGN-THETA has the task of assigning θ-role to these arguments. Recall that the internal argument structure of ver

[14] Case transmission will be discussed in section 6.2.4.

[15] Note that θ-role transmission always goes from trace to antecedent (not vice-versa) because the trace is in a D-structure position to which θ-role is assigned automatically. By contrast, case transmission may occur either from trace to antecedent or from antecedent to trace.

is: ((p-goal animate)),[16] *i.e.*, there is only *one* θ-role that can be assigned. This θ-role is assigned to *lo* which is given first priority since it absorbs s-government. (See section 3.4.2 for a description of s-government.) Thus, *lo* gets assigned a θ-role of *goal*, leaving *él* unassigned for the time being.[17]

When the θ-Criterion is checked, TRANSMIT-THETA is applied to the unassigned `N-MAX` *él*. Since the CLITIC-DOUBLING parameter is set to T for Spanish and the `N-MAX` has been assigned case, the TRANSMIT-THETA-FROM-CLITIC procedure is called so that the θ-transmission rule:

$$[\text{CL } +case_i\ +\theta_j] \ldots [\text{NP } +case_i] \Rightarrow [\text{CL } +case_i\ +\theta_j] \ldots [\text{NP } +case_i\ +\theta_j]$$

can fire. The only clitic that has features compatible with *él* is *lo*. Thus, the role of *lo* is copied to the ROLE slot of the `N-MAX` containing *él*. The final result is:

```
[I-MAX [past p1 sg]
   [I-SPEC [N-MAX e [pro p1 sg]]]
   [I E [past p1 sg]]
   [I-COMPLEMENT
      [V-MAX [past p1 sg]
         [V [CL lo [p3 sg animate masc obj goal]] [V ver]]
         [V-COMPLEMENT
            [P-MAX [P a]
               [P-COMPLEMENT
                  [N-MAX [p3 sg animate masc obj goal] él]]]]]]]
```

6.2.2.3 θ-Criterion Checking

The function CHECK-THETA-CRITERION is called if the phrase contains no θ-role assigners or if it contains no θ-role assignees. In the first case, the phrase must be checked for arguments that are not θ-marked, and the parse is rejected if any are found; and in the second case, the phrase must be checked for assigners that have not discharged their θ-roles, and the parse is rejected if any are found. Thus, CHECK-THETA-CRITERION iterates through possible assignees, checking that they are assigned θ-role, or that they *can* be assigned a θ-role via θ-role transmission. (THETA-VIOLATION passes control to the θ-transmission routines described

[16] *p-goal* is a *goal* that is realized as `P-MAX` instead of `N-MAX`.

[17] The *p-goal* θ-role gets turned into the *goal* θ-role by a procedure that is not described here. In a nutshell, this procedure removes the *p-* prefix, and then allows *goal* to be assigned to a noun phrase dominated by a `P` node during θ-role assignment.

in the last section.) Then CHECK-THETA-CRITERION checks that the assigner has no undischarged θ-roles. If there are any unassigned arguments or undischarged θ-roles, CHECK-THETA-CRITERION is called. All of the θ-Criterion functions are in appendix F.2.4.

6.2.3 Government Module

There are two tasks associated with the Government module: the first is to set up government relations between phrases in a stack; and the second is to test for government of a node. The first of these two tasks is performed at POP time (invoked by DROP-INTO-NEXT-LOWER in appendix H.3) and also during θ-role transmission (invoked by TRANSMIT-THETA-FROM-CLITIC in appendix F.2.3); the second task is performed upon request by the θ, Case, and Trace modules. Each of these two tasks will be described in more detail in the following sections.

6.2.3.1 Setting up Government Relations

All phrases of a stack must encode c-government, s-government, and proper-government relations. I will now cover how each of these relations is established in turn. (The functions that establish these relations are in appendix F.3.1.)

SET-UP-GOVERNMENT calls FIND-GOVERNORS to examine each stack item, and set up government relations as follows:

1. The head of the phrase governs its specifiers, complements, and minimally adjoined elements.
2. Each phrase governed by the head must have its government relations set up.

Step 1 allows complements, specifiers, and adjuncts to be governed by the head of a phrase. Step 2 recursively sets up government relations on each phrase within the phrase.

For example, during the parse of the phrase *yo le di libros* (= *I gave him books*), the following structure is set up:

```
[I-MAX [past p1 sg]
   [I-SPEC [N-MAX₁ [p1 sg] [N yo]]]
   [I E [past p1 sg]]
   [I-COMPLEMENT
      [V-MAX [past p1 sg]
         [V₁ [CL le [p3 sg obj animate]] [V₂ dar]]
         [V-COMPLEMENT [N-MAX₂ [pl masc] [N libros]]]]]]
```

When the phrase is popped, the government relations are set up as follows:

1. `I-MAX` has a head `I` that governs the specifier `[N-MAX₁ [N yo]]`, and the complement `[V-MAX [V [CL le] [V dar] [N-MAX₂ libros]]]`.
2. `N-MAX₁` has a head `[N yo]` that does not have a specifier or a complement; thus, `[N yo]` does not govern anything.
3. `V-MAX` has a head `[V₁ [CL le] [V₂ dar]]` that governs its complement `[N-MAX₂ libros]`.
4. `V₁` has a head `[V₂ dar]` that governs an adjoined clitic `[CL le]`.
5. `N-MAX₂` has a head `[N libros]` that does not have a specifier or a complement; thus, `[N libros]` does not govern anything.

The government relations set up by the procedure outlined above are strictly c-government relations. That is, these relations are set up by calling the dispatching function SET-GOVERNORS, which then calls SET-C-GOVERNORS. The c-government relation is established between a head and its specifiers, complements, and minimally adjoined elements. Note that the node must be a governor before it can be allowed to c-govern something. The function GOVERNOR? tests that this condition is satisfied. The *CURRENT-GOVERNORS variable contains the list of governors set by the GOVERNORS parameter. In order to be a governor, the node must fulfill one of two requirements: either the category of the node must occur in the *CURRENT-GOVERNORS variable (*i.e.*, the category is lexical); or AGR must occur in the *CURRENT-GOVERNORS variable and the node must be an I(nfl) node with agreement features.

SET-S-GOVERNORS establishes the s-government relation between a head and those c-governed nodes that can be uniquely paired with the θ-roles assigned by the head. For example, in the ill-formed sentence **lo vimos Guille* (= *we (him) saw Guille*), *lo* gets paired up with the *goal* θ-role and thus is s-governed by *vimos*. This leaves Guille without case since it is not s-governed; thus, the sentence is ruled out by the θ-Criterion (since θ-role cannot be assigned to an element that is not case-marked). On the other hand, *lo vimos a Guille* (= *we (him) saw* *(to) Guille*) is well-formed because *lo* receives case from *vimos*, Guille receives case from *a*, and θ-role is transmitted from *lo* to Guille (thus satisfying the θ-Criterion). UNIQUE-PAIRING determines the nodes that are s-governed by a head. Note that clitics *absorb* s-government; that is, they are given first priority during the matching between θ-roles and c-governed elements. PUT-CLITICS-FIRST prioritizes noun phrases so that clitics are first in line during the unique pairing process.

During the θ-role/NP pairing, features of the θ-role are tested for agreement against the features of the NP. For example, in the sentence *le di el libro* (= *I gave him the book*), the verb *di* s-governs *both* the clitic *le* and the complement *el libro*; the clitic *le* is matched to the role *patient animate* since the *le* has the *animate* feature, and the noun phrase *el libro* is matched to the role *goal*. However, this matching of θ-roles to noun phrases could not be reversed because *el libro* does not have *animate* features.

The third type of government configuration that is established is proper-government. SET-PROPER-GOVERNORS first checks to see if a head node can be a proper governor by calling PROPER-GOVERNOR? In order to be a proper governor, the category of the node must be in the *CURRENT-GOVERNORS variable, or if the node is `I` with agreement features, *AGR-RICH must be T. (*AGR-RICH is the variable associated with the PRO-DROP parameter.)

For example, the parse structure for the sentence *vio* (= {*he, she*} *saw*) is as follows:

```
[I-MAX [past p3 sg]
    [I-SPEC [N-MAX e [pro]]]
    [I E [past p3 sg]]
    [I-COMPLEMENT [V-MAX [past p3 sg] [V ver]]]]
```

In this structure there is a proper-government relation between I(nfl) and *pro* (*i.e.*, `[I [past p3 sg]]` properly governs *pro*) since Spanish is a pro-drop language.

In addition to proper government by the head of a phrase, an element can be properly governed by an antecedent. However, this type of proper-government is not set up at POP time; rather, antecedent-government is included in the proper-government test (to be discussed in the next section) so that it is checked only upon request by the trace module.

6.2.3.2 Tests for Government Relations

All tests for government of a node are in appendix F.3.2. The predicates C-GOVERNED? and S-GOVERNED? are used by the θ and Case modules. Both these predicates return the nodes that govern the node in question. C-GOVERNED? is used (by TRANSMIT-THETA-FROM-CLITIC in appendix F.2.3) to test that a clitic and `N-MAX` are not governed by the same head (since this would mean objective case is being assigned twice, an impossibility due to absorption of s-government by the clitic). Both S-GOVERNED? and C-GOVERNED? are used (by ASSIGN-CASE in appendix F.4) to assign case.

The PROPER-GOVERNED? predicate is used by the trace module (ECP in appendix F.5) and is slightly more complicated than the first two government predicates. Proper government of a node holds if any of the following conditions are satisfied:

1. The node is not a trace.
2. The node has a proper governor (determined by SET-PROPER-GOVERNORS as described in the last section).
3. The node is antecedent-governed by a c-commanding NP.
4. Chain conditions are satisfied.

Suppose we have the sentence *who do you think ate dinner?*. This parses into the following structure:

[C-MAX
  [C-SPEC [N-MAX who]$_i$] [C E]
  [I-MAX [DO-AUX do]
    [I-MAX [N-MAX you]
      [V-MAX [V think]
        [C-MAX
          [C-SPEC [N-MAX e']$_i$
          [I-MAX [N-MAX e'']$_i$
            [V-MAX [V ate] [N-MAX [N dinner]]]]]]]]]

Table 6.4 shows how the elements of the chain (Who, e', e'') are properly governed.

The fourth condition is checked by the function CHAIN-CONDITIONS-SATISFIED? which determines whether all elements of a chain containing a node are c-governed. This test can only be made if the *CHAIN-CONDITIONS variable is set to T. (This variable corresponds to the setting of the ECP parameter). The conditions are satisfied for a trace node if: the node is c-governed; all antecedents of the node are c-governed; and all traces of the node are

Function Call	*Condition Satisfied*
PROPER-GOVERNED?([N-MAX Who]$_i$)	Condition 1: not a trace
PROPER-GOVERNED?([N-MAX e']$_i$)	Condition 2: lexically governed by *think*
PROPER-GOVERNED?([N-MAX e'']$_i$)	Condition 3: antecedent-governed by [N-MAX e']$_i$

Table 6.4: Proper Government of Elements in Chain (Who, e', e'')

c-governed. For example, the structure of the Spanish sentence *qué vio* (= *what did {he, she} see*) is the following:

```
[C-MAX
   [C-SPEC [N-MAX que]$_i$] [C E]
   [I-MAX [V ver]$_j$
      [I-MAX [N-MAX e [pro]]
         [I E [pres p3 sg]]
         [V-MAX [V e]$_j$ [N-MAX e]$_i$]]]]
```

The call PROPER-GOVERNED?(`[N-MAX e]`) returns T since `[N-MAX e]` is c-governed by [V e] and [N-MAX que] is governed by C, thus satisfying the fourth condition.

6.2.4 Case Module

All case assignment functions are in appendix F.4. Case assignment is performed at POP time. Because a phrase has just been completed, this is the most convenient time to perform case assignment: the completed head is now free to assign case to its completed specifier and complement. The function PERFORM-CASE-ASSIGNMENT locates assignee/assigner pairs in the top-of-stack element. Each assignee/assigner pair is passed to the ASSIGN-CASE function, which accesses the CASE-ASSIGNMENT parameter to determine the conditions under which case is assigned. (The setting of this parameter is stored in the *CURRENT-CASE-ASSIGNMENT variable.)

First ASSIGN-CASE determines the assignment-rule corresponding to the assigner node; once this has been determined, the condition associated with the rule must be satisfied. For example, if the assignment rule is S-GOVERNS, then the assigner must s-govern the assignee in order for case to be assigned. Since the assignee node might have inherent case features (*e.g.*, *I* has an inherent nominative case feature), the case to be assigned must be matched for agreement with the case of the assignee node. If this test fails, the CASE-MISMATCH function signals a case mismatch error. Otherwise, case is assigned according to the case required by the assignment rule.

Once case has been assigned, TRANSMIT-CASE-TO-TRACE-OR-ANTECEDENT performs case transmission to traces and antecedents of the assignee node. For example, the structure of the sentence *what did he see* is:

```
[C-MAX
   [C-SPEC [N-MAX what]_i] [C E]
   [I-MAX [DO-AUX do [p3 sg past]]
      [I-MAX [N-MAX he] [V-MAX [V see] [N-MAX e]_i]]]]
```

The trace `[N-MAX e]`$_i$ is assigned objective case by the verb `[V see]`; this case is then transmitted to the antecedent `[N-MAX what]`$_i$.

Case transmission occurs for moved noun phrases as well as moved *Wh*-phrases. For example, the structure of the sentence *he seems to be happy* is:

```
[C-MAX
   [C-SPEC
      [I-MAX [N-MAX he]_i
         [I E [pres p3 sg]]
         [V-MAX [V seems]
            [C-MAX
               [I-MAX [N-MAX e]_i
                  [I to] [V-MAX be happy]]]]]]]
```

Here, `[I E [pres p3 sg]]` assigns nominative case to *he*; the case is then transmitted to the trace `[N-MAX e]`$_i$.

Note the contrast between case transmission, which operates from antecedent to trace or from trace to antecedent, and θ-role transmission, which operates only from trace to antecedent. Intuitively, this asymmetry is understandable since arguments in D-structure position always bear θ-roles, but do not always have case; thus, when an argument is moved, it receives θ-role, but not necessarily case, from the trace it leaves behind.

During case transmission, an error may occur (most likely due to an incorrect linking of a trace to an antecedent). For example, suppose the sentence *qué vio* is parsed as:

```
[C-MAX
   [C-SPEC [N-MAX qué]] [C E]
   [I-MAX [V ver]_j
      [I-MAX [N-MAX e [pro]]_i
         [I E [pres p3 sg]] [V-MAX [V e]_j [N-MAX e]_i]]]]
```

During case assignment, `[N-MAX e]`$_i$ receives objective case (from `[V e]`). However, transmission of this case to the alleged antecedent `[N-MAX e [pro]]`$_i$ will result in an error since `[N-MAX e [pro]]`$_i$ has already been assigned nominative case by `[I E [pres p3 sg]]`. If case transmission fails, the TRANSMISSION-ERROR function signals a case transmission error and the parse is rejected.

DROP-INTO-NEXT-LOWER, in appendix H.3, interfaces between the Earley parser and the Case module.

6.2.5 Trace Module

The trace module consists of the single function ECP (see appendix F.5) accessed at POP time by DROP-INTO-NEXT-LOWER (in appendix H.3). ECP is called after government relations (in particular, proper-government relations) have been established. Recall that all empty categories except PRO must be properly governed. In particular, PRO must be ungoverned, and traces and *pro* must be properly governed. Thus, ECP operates as follows:

1. If the node is PRO, then if it is c-governed, reject the parse.
2. If the node is trace, then if it is not properly governed, reject the parse.
3. If the node is *pro*, then if it is not governed by AGR, reject the parse.
4. Otherwise, accept the parse.

Table 6.5 shows the result of applying ECP to several types of phrases.

6.2.6 Binding Module

The Binding module is accessed at POP time, first to determine A and $\overline{\text{A}}$ positions, and then to check binding conditions. The interface between the Earley parser and the Binding module is the DROP-INTO-NEXT-LOWER function in appendix H.3.

Phrase	*Outcome of Applying ECP*
I believe him PRO to be happy	none
I believe him PRO is happy	PRO is c-governed (by [I +tns])
Who$_i$ do you think e'_i e''_i came	none
Who$_i$ do you think e'_i that e''_i came	e''_i is not properly governed
¿Quién$_i$ piensas tú e'_i que e''_i vino?	none
¿Qué$_i$ vio$_j$ pro v$_j$ e_i?	none
¿Qué$_i$ vio$_j$ e_i v$_j$ pro?	pro is not governed by AGR

Table 6.5: Effect of ECP Application

6.2.6.1 Determining A and $\overline{\text{A}}$ Positions

The function SET-A-AND-A-BAR-POSITIONS (in appendix F.6) is called on each phrase of a stack at POP time. This function operates as follows:

1. Locate all nodes in A-positions (by calling FIND-A-POSITIONS) and set the A-POSITION? slot of each of these to T.
2. Locate all nodes in A-positions (by calling FIND-A-POSITIONS) and set the A-POSITION? slot of each of these to T.
3. Examine each element in the phrase: (a) If the element is in specifier or complement position, SET-A-AND-A-BAR-POSITIONS is called on the element. (b) If the element is minimally or maximally adjoined, SET-A-AND-A-BAR-POSITIONS is called on the element.

Some examples of A-positions are `I-SPEC` and `V-COMPLEMENT`. Some examples of $\overline{\text{A}}$-positions are `C-SPEC`, `I-COMPLEMENT`, and all adjunct positions. The way A and $\overline{\text{A}}$ positions are distinguished is as follows: A-positions are those positions where θ-role could be assigned at D-structure regardless of the choice of lexical element in head position. All `I-SPEC` and `V-COMPLEMENT` positions are A-positions since verbs assign internal and external θ-roles; by contrast, I(nfl) does not assign internal and external θ-roles, so `C-SPEC` and `I-COMPLEMENT` are $\overline{\text{A}}$-positions.

6.2.6.2 Checking Binding Conditions

The function CHECK-BINDING-CONDITIONS (in appendix F.6) applies binding conditions, only as they apply at surface structure. That is, binding conditions are applied in the context

A-Chain (he_i, e_i)	
<NODE NP#1>	
CAT:	n
WORD:	he
GENDER:	masc
PERS:	p3
NUMBER:	sg
CASE:	nom
ROLE:	agent
TRACE:	nil
ANTECEDENT:	<np#2>
<NODE NP#2>	
CAT:	n
WORD:	n-max-trace
GENDER:	masc
PERS:	p3
NUMBER:	sg
CASE:	nom
ROLE:	agent
TRACE:	<np#1>
ANTECEDENT:	nil

$\bar{A}$-*Chain* ($what_i$, e_i)	
<NODE NP#3>	
CAT:	n
WORD:	what
GENDER:	neut
PERS:	p3
NUMBER:	sg
CASE:	obj
ROLE:	goal
TRACE:	nil
ANTECEDENT:	<np#4>
<NODE NP#4>	
CAT:	n
WORD:	n-max-trace
GENDER:	neut
PERS:	p3
NUMBER:	sg
CASE:	obj
ROLE:	goal
TRACE:	<np#3>
ANTECEDENT:	nil

Table 6.6: Nodes Corresponding to A and $\bar{\text{A}}$ Chains

of antecedent-trace relations (*e.g.*, $what_i/e_i$ in *what*$_i$ *did he see* e_i), not in the context of referential dependencies (*e.g.*, he$_i$/himself$_i$ in *he*$_i$ *saw himself*$_i$). Thus, CHECK-BINDING-CONDITIONS first checks that each anaphoric trace is A-bound in its governing category, and that each referential trace is $\bar{\text{A}}$-bound. These two cases cover sentences like (56) and (57), respectively.

(56) He$_i$ seems e$_i$ to be happy.

(57) What$_i$ was he doing e$_i$?

In (56), e$_i$ is A-bound in the domain of its subject (he$_i$). In (57), e$_i$ is $\bar{\text{A}}$-bound (by what$_i$). The nodes in the chains corresponding to (56) and (57) are in table 6.6.

The functions ANAPHORIC? and REFERENTIAL? determine whether an NP is anaphoric or referential by testing whether the *wh* feature is associated with the node. (Anaphoric traces do *not* have the *wh* feature, and referential traces *do* have the *wh* feature.) Note that traces must already be linked to antecedents before binding conditions can be checked.

6.2.7 Bounding Module

The Bounding module is accessed both at PUSH and at POP time. The interface between the Earley parser and this module is PUSH-TEMPLATES-ONTO-STACK (see appendix H.1) and DROP-INTO-NEXT-LOWER (see appendix H.3). When a trace is pushed onto the stack, it must be linked to an antecedent that is not *too far away*. In the case of rightward movement, the trace may remain unlinked until its dominating maximal projection has been completed. Thus, at POP time, the Bounding module is accessed again to link unlinked traces to antecedents. The Bounding module is also accessed at POP time to ensure that certain language-specific movement requirements are fulfilled. The next two sections describe the PUSH and POP trace linking operations, and the third section describes the language-specific movement effect routines.

6.2.7.1 Trace Linking at PUSH Time

The function CHECK-TRACE-LINKS is invoked at PUSH time. Recall that trace possibilities are included in the phrase structure templates generated at precompilation time. Thus, when the structure-building component pushes a template onto the stack, the template may contain a trace which needs to be linked with its antecedent. The function CHECK-TRACE-LINKS performs the trace-antecedent linking when this is the case. The template that is being added to the stack is searched (by SEARCH-TRACE); if there is a trace (there will be at most *one* unlinked trace at PUSH time), the possible antecedents are located (by FIND-TRACE-ANTECEDENTS) and the features of the stack element containing the trace are percolated (by PERCOLATE-FEATURES).[18]

The FIND-TRACE-ANTECEDENTS routine returns all possibilities for antecedents of a trace. For example, the structure corresponding to the sentence *qué vio* is:

[18] An example of how the trace linking and feature percolation actions interact was given in section 6.2.1.4.

```
[C-MAX
   [C-SPEC [N-MAX qué]] [C E]
   [I-MAX [V ver]_j
      [I-MAX [N-MAX e [pro]] [I E [past p3 sg]] [V-MAX [V e]_j [N-MAX e]]]]]
```

If we are parsing the final trace in this structure, FIND-TRACE-ANTECEDENTS returns two antecedent possibilities: `[N-MAX qué]` and `[N-MAX e [pro]]`. Thus, CHECK-TRACE-LINKS returns the two structures shown in figure 6.7.

(a) Trace Linked with *qué*	(b) Trace Linked with *pro*
[C-MAX	[C-MAX
[C-SPEC [N-MAX qué]$_i$] [C E]	[C-SPEC [N-MAX qué]] [C E]
[I-MAX [V ver]$_j$	[I-MAX [V ver]$_j$
[I-MAX [N-MAX e [pro]]	[I-MAX [N-MAX e [pro]]$_i$
[I E [past p3 sg]]	[I E [past p3 sg]]
[V-MAX [V e]$_j$	[V-MAX [V e]$_j$
[N-MAX e]$_i$]]]]	[N-MAX e]$_i$]]]]

Figure 6.7: Two Trace Linking Possibilities

Part of the trace-linking procedure includes transmission of node-slot values from antecedent to trace. Thus, case and θ-role values of the antecedent are copied to the trace. In the case of figure 6.7(a), the antecedent has no case or θ-role. Thus, the trace is left open for case and θ-role assignment later on. (The verbal element `[V e]`$_j$, which receives its case and θ-role assigning properties from its antecedent `vio`$_j$, will assign case and θ-role to this position.) In the case of figure 6.7(b), the trace receives *agent* θ-role and nominative case from its antecedent; this parse will be ruled out later by the case module, which will attempt to assign objective case (from the assigner `[V e]`$_j$) to an `N-MAX` that already has nominative case.

If FIND-TRACE-ANTECEDENTS returns FAILURE, then one of two possibilities are available: either the antecedent has not yet been seen (*e.g.*, in rightward movement); or there is no antecedent possibility. The function ANTECEDENT-NOT-BEYOND-BOUNDING-NODE checks to see which one of these cases applies. For example, in the case of free inversion, a trace is predicted before its antecedent has been seen:

[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT [V-MAX]]]
[C-MAX [I-MAX]]

Figure 6.8: Prediction of Trace of Free Inversion

[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[V-MAX [V wonder] [C-MAX [C-SPEC whether] [I-MAX]]]
[I-MAX did [I-MAX [I-SPEC [N-MAX John]] [I E] [I-COMPLEMENT [V-MAX]]]]
[C-MAX [C-SPEC who] [I-MAX]]

Figure 6.9: Stack with Trace Whose Antecedent is "Too Far Away"

```
[C-MAX
   [I-MAX [N-MAX e]$_i$
      [I e [past p3 sg]]
         [I-COMPLEMENT
            [V-MAX [V-MAX [V ver]] [N-MAX Juan]$_i$]]]]
```

Here, the trace `[N-MAX e]`$_i$ has been left behind by rightward movement of the `[N-MAX Juan]`$_i$; thus, when `[N-MAX e]` is predicted, it must be determined that a position containing the antecedent is still a possibility. Figure 6.8 shows the state of the stack after e$_i$ has been predicted. ANTECEDENT-NOT-BEYOND-BOUNDING-NODE iterates through stack elements counting bounding nodes to determine if there is an unexpanded symbol less than two bounding nodes away from the trace. (The bounding nodes for the current language are stored in *CURRENT-BOUNDING-NODES.) Since `V-MAX` has not been expanded in figure 6.8, there is still a possibility that an antecedent will be found; thus, ANTECEDENT-NOT-BEYOND-BOUNDING-NODE returns T, and the trace is left unlinked.

By contrast, the sentence *who$_i$ did john wonder whether e$_i$ went home* is ruled out by CHECK-TRACE-LINKS because the antecedent *who$_i$* is "too far away" (*i.e.*, beyond 2 bounding nodes) from the trace e$_i$. Figure 6.9 shows the state of the stack after the trace has been predicted for this sentence. The English bounding nodes are `N-MAX` and `C-MAX` (as specified by the BOUNDING-NODES parameter setting). Since there are two `C-MAX` nodes intervening between the `[N-MAX e]` in the top-of-stack element, and the `[C-SPEC who]` in the fourth stack element, *who* cannot be the antecedent for the trace. Thus, FIND-TRACE-ANTECEDENTS returns FAILURE, and when ANTECEDENT-NOT-BEYOND-BOUNDING-NODE is called, there are no unexpanded symbols less than two bounding nodes away from [N-MAX e]. Consequently, ANTECEDENT-NOT-BEYOND-BOUNDING-NODE returns NIL, and the parse is rejected.[19]

The following section describes trace-antecedent linking in the case where an antecedent is encountered after the trace has been parsed.

6.2.7.2 Trace Linking at POP Time

When a trace has been left unlinked at PUSH time, it must be linked to its antecedent later. The FIND-AND-LINK-TRACES routine (in appendix F.7.2) performs trace-antecedent linking at POP time. This routine is similar to the CHECK-TRACE-LINKS function except that there may be more than one unlinked trace to be processed. This is because repeated calls to PUSH may have left more than one unlinked trace (from potential rightward movement) in the stack. Each trace must be linked up to all of its possible antecedents, and all possible trace-linkings are returned. In the free-inversion case (mentioned in the last section), the unlinked trace predicted at PUSH time must be linked up at POP time. Figure 6.10 shows snapshots of the parser in action after a trace has been predicted.

In 6.10(a), the trace `[N-MAX e]`, does not yet have an antecedent, but CHECK-TRACE-LINKS does not reject the parse since `V-MAX` is not yet expanded. In 6.10(b), `[V ver [past p3 sg]]` is dropped under `V-MAX`. Note that the left-recursive adjunction structure `[V-MAX [V-MAX] [N-MAX]]` has been inserted in the second stack element in preparation for dropping the completed top-of-stack element. (The details of how DROP-RECURSE-NEXT-LOWER inserts a left-recursive stack element are in section 6.1.) In 6.10(c), the completed `V-MAX` is dropped under the `[V-MAX]` adjunction structure that contains the unexpanded `[N-MAX]`, and FIND-AND-LINK-TRACES is executed. The trace `[N-MAX e]` still does not have an antecedent, but the parse is not rejected since there is an unexpanded `[N-MAX]` in the top-of-stack element. Finally, `N-MAX` is expanded to contain *Juan* as shown in figure 6.10(d), and it is then dropped under `V-MAX` as shown in figure 6.10(e). Again the FIND-AND-LINK-TRACES

[19] Actually, `[N-MAX John]` could be taken as an antecedent of [N-MAX e], but then the operator `[C-SPEC who]` would not bind a variable; this case is ruled out by the Binding module.

```
(a) Prediction of Trace
-----------------------------
[I-MAX
   [I-SPEC [N-MAX e]]
    [I e]
    [I-COMPLEMENT [V-MAX]]]
-----------------------------
[C-MAX [I-MAX]]
```

```
(b) Expansion of V-MAX
-----------------------------
[V-MAX [V ver [past p3 sg]]]
-----------------------------
[I-MAX
    [I-SPEC [N-MAX e]]
    [I e [past p3 sg]]
    [I-COMPLEMENT
        [V-MAX [V-MAX] [N-MAX]]]]
-----------------------------
[C-MAX [I-MAX]]
```

```
(c) Completion of V-MAX
-----------------------------
[I-MAX
   [I-SPEC [N-MAX e]]
    [I e [past p3 sg]]
    [I-COMPLEMENT
        [V-MAX
            [V-MAX
                [V ver [past p3 sg]]]
            [N-MAX]]]]
-----------------------------
[C-MAX [I-MAX]]
```

```
(d) Expansion of N-MAX
-----------------------------
[N-MAX [N Juan]]
-----------------------------
[I-MAX
   [I-SPEC [N-MAX e]]
   [I e [past p3 sg]]
   [I-COMPLEMENT
       [V-MAX
           [V-MAX [V ver [past p3 sg]]]
           [N-MAX]]]]
-----------------------------
[C-MAX [I-MAX]]
```

```
(e) Completion of N-MAX
-----------------------------
[I-MAX
    [I-SPEC [N-MAX e]i]
    [I e [past p3 sg]]
    [I-COMPLEMENT
        [V-MAX
            [V-MAX [V ver [past p3 sg]]]
            [N-MAX [N Juan]]i]]]
-----------------------------
[C-MAX [I-MAX]]
```

```
(f) Completed Parse
-----------------------------
[C-MAX
   [I-MAX
       [I-SPEC [N-MAX e]i]
       [I e [past p3 sg]]
       [I-COMPLEMENT
           [V-MAX
               [V-MAX [V ver [past p3 sg]]]
               [N-MAX [N Juan]]i]]]]
```

Figure 6.10: Snapshots of Free Inversion Parsing

function is executed; this time an antecedent of [N-MAX e] is found, and the trace/antecedent linking is set up. Now the `I-MAX` is complete. The final parse is shown in figure 6.10(f).

In general FIND-AND-LINK-TRACES is needed for linking traces in rightward movement structures because an unresolved trace is predicted only if the antecedent has not yet been seen in the input. This occurs infrequently in head-initial languages; however, it occurs more frequently in head-final languages (*e.g.*, Japanese). The reason for this difference in direction of movement is that the structure in a head-initial language is primarily right-branching

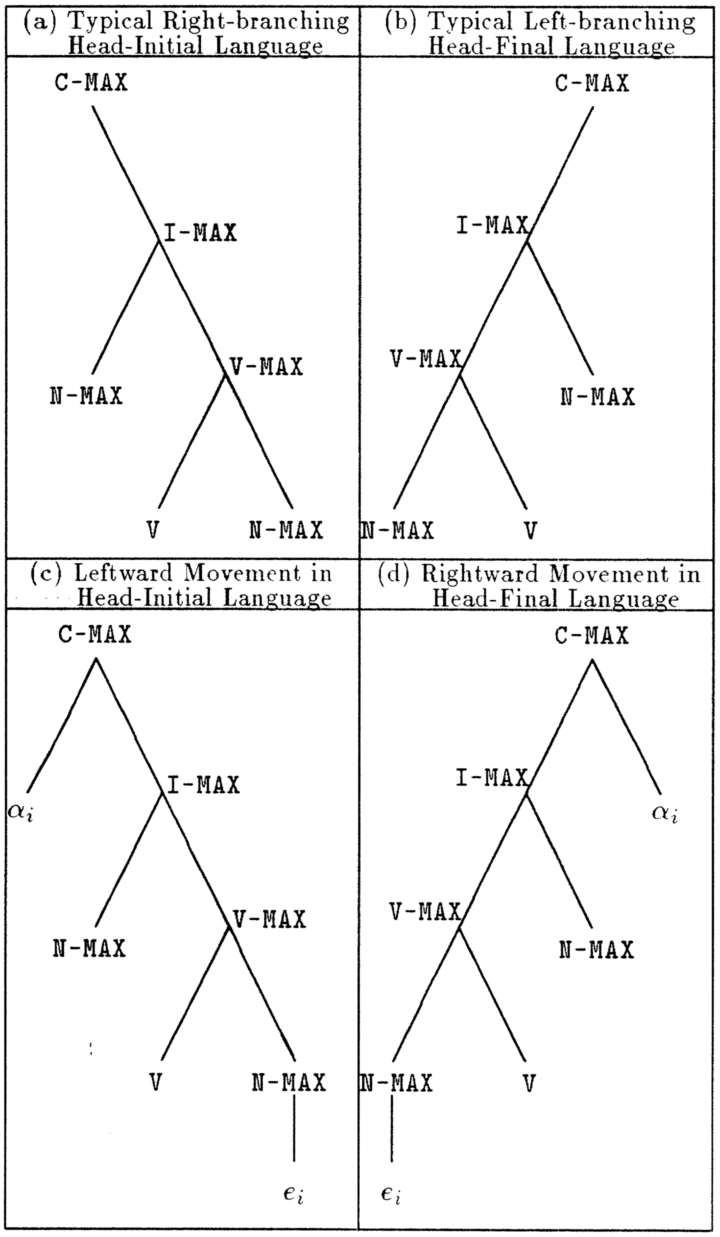

(a) Typical Right-branching Head-Initial Language
(b) Typical Left-branching Head-Final Language
(c) Leftward Movement in Head-Initial Language
(d) Rightward Movement in Head-Final Language
C-MAX
I-MAX
V-MAX
N-MAX
V
α_i
e_i

Figure 6.11: Contrasting Head-Initial and Head-Final Structures

(see figure 6.11(a)); thus, c-commanding landing sites are generally to the left. By contrast, the structure in a head-final language is primarily left-branching (see figure 6.11(b)); thus, c-commanding landing sites are generally to the right. The stack mechanism used in this implementation readily accommodates both types of movement: the Bounding module simply iterates over stack elements, counting bounding nodes along the way, ignoring the direction of branching imposed by the language. Thus, just as complement-prediction is handled uniformly for head-initial and head-final languages, antecedent-linking is also handled uniformly for both types of languages.

6.2.7.3 Checking Language-Specific Effects

Another condition checked by the Bounding module at POP time is a (non-standard) principle that requires certain language-specific movement effects to take place. As mentioned in chapter 1 (fn. 9), the system presented here operates universally only to the extent that GB theory is universally applicable. There are some phenomena not covered by the theory that consequently are not handled by the system using standard GB principles. Thus, there is a parameter associated with the Bounding module called LANGUAGE-SPECIFIC-EFFECTS, which requires certain actions to occur only in the context of other actions. (The value of the LANGUAGE-SPECIFIC-EFFECTS parameter for the current language is stored in *CURRENT-LANGUAGE-SPECIFIC-EFFECTS.) The user is allowed to add as many language-specific effects as he/she feels is necessary for any source or target language. The intent of this parameter is that it serve as an "escape hatch" for only a small number of phenomena that are not handled by the (standard) GB principles.[20]

The LANGUAGE-SPECIFIC parameter setting consists of lists of implications.[21] If the left-hand side of the implication holds, then the right-hand side of the implication must also hold. The left-hand and right-hand sides contain boolean operators (*i.e.*, OR, and AND),

[20] This feature obviously affords considerable computational power; however, the hope is that the user will not use this feature to encode large portions of the system. Using the LANGUAGE-SPECIFIC-EFFECTS parameter in this way would be redundant because the other modules of the system should handle most, or all, of the movement effects for the language. In fact, the author found only one language-specific effect for Spanish, and one language-specific effect for English. The choice of this setting was determined after testing several data and discovering that only one effect was not handled by the (standard) GB principles in each of the two languages. It may be that these effects are derivable from principles that are not yet known or well-understood.

[21] If a two-way implication is required, a double-arrow ($\Leftrightarrow$) is used.

and lists of movement labels (*e.g.*, WH-MOVEMENT) with associated features (*e.g.*, WH-PHRASE-A). Thus, the language-specific-effect (((PREPOSE) ⇔ (WH-MOVEMENT WH-PHRASE-A))) requires that preposing occur if and only if *Wh*-movement of a phrase (with the WH-PHRASE-A feature) has taken place.[22] Thus, the structure of the Spanish sentence *qué vio* (= *what did* {*he, she*} *see*) could not be the following:

```
[C-MAX
   [C-SPEC [N-MAX que]_i] [C E]
   [I-MAX [N-MAX e [pro]]
      [I e [pres p3 sg]] [V-MAX [V ver]_j [N-MAX e]_i]]]
```

Because *Wh*-movement has taken place, V-preposing must occur. However, this is not the case in the above structure (the verb *ver* appears in D-structure position).

Similarly, in English SAI must occur if *Wh*-movement has taken place. Thus, the English sentence **what you have said* is not possible since the auxiliary *have* is in D-structure position:

```
[C-MAX
   [C-SPEC [N-MAX what]_i] [C E]
   [I-MAX [N-MAX you]
      [I e [pres p2 sg]]
      [V-MAX [HAVE-AUX have] [V said]_j [N-MAX e]_i]]]
```

The function CHECK-LANGUAGE-SPECIFIC-EFFECTS, which is called at POP time, performs this check on completed parses. (See appendix F.7.3.)

[22]The construction of this implication was inspired by the following comment in Torrego (1984):

> Note that *Wh*-phrases that require inversion are the thematic arguments of the verb and subject of S ... I will indicate that a *Wh*-word is of the kind that makes inversion obligatory by labeling it a *Wh*-word$_A$. We may then state the following: in Spanish, a *Wh*-word$_A$ in the Comp position of a tensed clause triggers obligatory inversion in both main and embedded clauses.

Chapter 7

Final Translation Routines

The final two stages of translation are replacement and generation. The replacement routines consist of structural movement of elements back into their positions of origin (*i.e.*, the positions of their linked traces), and thematic substitution of equivalent target language words. The generation routines consist of move-α (*i.e.*, movement to both A and $\overline{\text{A}}$ positions) and morphological synthesis. The replacement and generation routines will be discussed in more detail in the following sections.

7.1 Replacement Routines

The STRUCTURAL-REPLACEMENT and THEMATIC-SUBSTITUTION routines are in appendix J. The functions called by each of these routines are in appendix H.4; they will be described in the following two sections. The third section describes the linguistic constraints imposed on the replacement routines. Table 7.1 gives a procedural description of the replacement module.

7.1.1 Structural Replacement

In order to move elements into underlying positions, the structural replacement routines need only know about antecedent-trace relations in the structure returned by the parsing component. The reader may refer to the first part of table 7.1 for an overview of the order in which the

1. Perform structural replacement by: (a) Transferring slot values of each moved element to its base position (b) Evacuating positions of moved elements (c) Removing evacuated positions (d) Eliminating useless structure
2. Perform thematic substitution by: (a) Extracting the head node of a phrase (b) Retrieving target language translations of the head (c) Setting the word slot of the head to be the target language translation (d) Determining the argument structure of the new head (e) Positioning arguments of the head according to internal and external argument requirements (f) Establishing structural realizations of arguments (g) Performing thematic substitution (recursively) on arguments of the head

Table 7.1: Procedural Description of the Replacement Module

structural replacement actions are applied.

The function MOVE-ELEMENTS-BACK takes a parse tree and drops antecedents into the positions occupied by their traces. For example, the parse structure of the sentence *a quién vio juan* is in figure 7.1(a) and the output of MOVE-ELEMENTS-BACK is in figure 7.1(b).

Notice that the positions vacated by the MOVE-ELEMENTS-BACK routine contain NIL. The function DROP-INTO-BASE-POSITION performs the evacuation of moved elements (by the function EVACUATE) leaving NIL behind in the evacuated position. If a moved element is part of a chain with more than one intermediate landing site, each one of the landing sites along the way to the base position is evacuated; hence, the DROP-INTO-BASE-POSITION function is called recursively, until a final destination is reached. For example, in sentence (58), the positions occupied by $what_i$ and e'_i are evacuated, thus leaving NIL in both positions as shown in sentence (59).

(58) $what_i$ did you think e'_i he ate e''_i

(59) NIL did you think NIL he ate what

(a) Parsed Structure	(b) Structure with Replaced Elements
[C-MAX	[C-MAX
[C-SPEC [P-MAX a quién]$_i$]	[C-SPEC NIL]
[I-MAX	[I-MAX
[V ver]$_j$	[V NIL]
[I-MAX	[I-MAX
[I-SPEC [N-MAX Juan]]	[I-SPEC [N-MAX Juan]]
[I e [past p3 sg]]	[I E [past p3 sg]]
[I-COMPLEMENT	[I-COMPLEMENT
[V-MAX	[V-MAX
[V e]$_j$	[V ver]
[P-MAX e]$_i$]]]]]	[P-MAX a quién]]]]]

Figure 7.1: Replacement of Moved Elements

Once the final destination is reached, DROP-INTO-BASE-POSITION transfers slot values of the moved element to the node in base position (by the function TRANSFER-SLOT-VALUES). The only values that are not transferred are those containing antecedent-trace links (since these are no longer required in the base form).

After antecedents have been moved into trace position, a collapsing routine eliminates "useless" structure. The function COLLAPSE-STRUCTURE eliminates null elements (by the function REMOVE-NIL) and merges equivalent structure into a single unit (by the function MERGE-EQUIVALENT-STRUCTURE). For example, when the null elements are removed from the structure of figure 7.1(b), the result is the structure in figure 7.2(a); then the `[I-MAX [I-MAX ...]]` is collapsed into `[I-MAX ...]` as shown in figure 7.2(b).

The result obtained by structural replacement is an *underlying form* that is the basis of the interlingual approach to translation. Essentially this form requires arguments of verbs to appear in their canonical (or "base") positions. That is, the underlying form is the D-structure of the target language. According to Chomsky, D-structures are "an abstract representation of semantically relevant grammatical relations such as subject-verb, verb-object, and so on, one crucial element that enters into semantic interpretation of sentences ..."[1]

In reality, D-structure is only an approximation to the true underlying form that is the

[1]Chomsky (1986b), p. 67.

(a) Structure Without Evacuated Elements	(b) Collapsed Structure
[C-MAX	[C-MAX
[I-MAX	[I-MAX
[I-MAX	[I-SPEC [N-MAX Juan]]
[I-SPEC [N-MAX Juan]]	[I E [past p3 sg]]
[I E [past p3 sg]]	[I-COMPLEMENT
[I-COMPLEMENT	[V-MAX
[V-MAX	[V ver]
[V ver]	[P-MAX a quién]]]]]
[P-MAX a quién]]]]]]	

Figure 7.2: Elimination of "Useless" Structure

action:	ver
agent:	Juan masc sg animate nom
patient:	quién wh p3 sg animate obj
time:	past

Figure 7.3: A Possible Interlingual Form

interlingua of a translation system. If a full interpretive system were implemented here, the interlingual form of figure 7.2(b) would be somewhat different, perhaps resembling the structure in figure 7.3. This interlingual form would be thematically mapped to a target language form from which structure would be generated according to structural requirements of the target language.

In contrast to the underlying form of figure 7.2(b), the structure in figure 7.3 leaves out language-specific requirements (*e.g.*, constituent order and morphological requirements). However, the underlying form of 7.2(b) is the one adopted as the interlingua for the implementation presented here for three reasons: (1) it is compatible with the structure required by the generation stage; (2) it is sufficient for the two languages handled by the system; and (3) it does not require a particularly robust generator, the construction of which is outside of the scope of this report.[2] The underlying form of the source language is stored in *SOURCE-BASE-TREES.

[2]In a later version of the translation system, such a generator may indeed by constructed. However, the details of this tentative plan for future work are not discussed here. In the version presented here, a simple routine is used to reorder constituents prior to generation of the target language sentence.

7.1.2 Thematic Substitution

Just prior to thematic substitution, the target language parameter values replace the source language parameter values. The function REPLACE-SOURCE-WITH-TARGET determines the correct translation of heads and their arguments according to θ-role matching routines. As each maximal projection is analyzed, the translation of the head is chosen, and then each argument of the head is translated, positioned, and structurally realized according to the requirements of the target language word. The reader may refer to the second part of table 7.1 for an overview of the order in which the thematic substitution actions are applied. (The θ-role matching, argument positioning, and structural realization routines are described in the next section.)

The target lexicon is accessed during thematic substitution in order to determine the features and argument structure of the translated head. Only one translation is chosen for each source language head, and all source language arguments are uniquely paired up with the target language arguments.

For example, suppose we are translating the sentence *I knew a man.* The source language D-structure is the following:

```
[C-MAX
  [C E]
  [I-MAX
    [I-SPEC [N-MAX I]]
    [I E [pres I sg]]
    [I-COMPLEMENT [V-MAX [V know] [N-MAX a man]]]]]
```

REPLACE-SOURCE-WITH-TARGET first processes `C-MAX`, determining that the head `C` has no lexical constituent to translate. It then analyzes `I-MAX`, the head of which also has no lexical constituent to translate. Finally, `V-MAX` is analyzed. The head `V` (= *know*) is extracted, the external argument `[N-MAX I]` is located (by GET-EXTERNAL-ARGS), and the internal argument `[N-MAX a man]` is located (by GET-INTERNAL-ARGS). The translation of *know* is determined by looking at the lexical entry of the two target language possibilities, *conocer* and *saber*. The word *saber* requires either a *fact* or a *proposition* as its internal argument, whereas *conocer* requires an animate *p-goal* or an inanimate *goal*. Because `[a man]` is an animate argument that is assigned a θ-role of *goal*, the translation that is chosen is *conocer*.[3]

Finally, each of the maximal projections in the internal and external arguments are analyzed by REPLACE-SOURCE-WITH-TARGET, and the translations for *I* and *a man* are chosen to be *yo* and *un hombre* respectively. (The structural realization of the arguments is determined by the θ-module as discussed in the next section.) The output of thematic-substitution is stored in *TARGET-BASE-TREES.

7.1.3 Linguistic Constraints on Replacement Routines

During structural replacement, virtually no access to the GB component is required since all actions are determined by antecedent-trace relations established during parsing. The only principle that is being acted upon is the Projection Principle of θ-theory which requires that θ-roles be preserved during the transition from S-structure to D-structure. This requirement is implicitly obeyed as a result of the replacement process, which transfers all θ-roles of antecedents to their traces.

By contrast, thematic substitution requires explicit access to the GB component (in particular, to the θ module) as well as access to the target language lexicon. The θ module functions accessed during thematic substitution are in appendix F.2.5.

The function MATCH-TRANSLATION-AND-ARGUMENTS determines the correct translation for a word on the basis of θ-role matching of arguments: first, GET-ARGUMENTS accesses the target language lexicon in order to determine the argument structure of a potential translation; next ARGUMENT-FEATURES-MATCH tests that there is a one-to-one correspondence between θ-roles assigned by the source language word and θ-roles assigned by the target language word, and then returns the θ-roles.[4]

The function SET-UP-ARGUMENT-POSITIONING then places arguments in external and internal positions according to the requirements of the target language word; then the function SET-UP-STRUCTURAL-REALIZATION applies the CSR mapping to external and internal arguments to ensure that the correct structure is obtained. The interface to the

[3]The role *p-goal* (or *prepositional goal*) matches the role *goal* for two reasons: (1) both have the *animate* feature; and (2) both are actually the same role (*goal*) with different structural realizations.

[4]This argument-structure matching routine handles thematic divergence such as is found in the *gustar-like* example mentioned in chapter 4. An example will be given shortly.

thematic substitution routines is the REPLACE-SOURCE-WITH-TARGET function described in section 7.1.2.

An example will clarify the procedure outlined above. If we are translating the phrase *me gusta el libro a mí*, the source language underlying form is:

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX el libro]]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX
            [V [CL-DAT me] [V gusta] [P-MAX a mí]]]]]]
```

MATCH-TRANSLATION-AND-ARGUMENTS attempts to translate the head *gustar* to the target language equivalent *like*. The GET-ARGUMENTS routine accesses the target language lexicon to determine that the argument structure of the target language verb *like* is: `[(external (agent animate)) (subcat (goal))]`. The source language external argument is `[N-MAX el libro]`, which is assigned *agent* θ-role, and the source language internal argument is `[P-MAX a mi]`, which is assigned *patient* θ-role with *animate* features.

The ARGUMENT-FEATURES-MATCH function calls PICK-ARG, which chooses target language candidates for each source language argument (on the basis of feature matching). The function LOCATE-UNIQUE-ARG-STRUCTURE then narrows down the candidates so that there is a one-to-one mapping from source to target language arguments. Thus,

```
the call: (PICK-ARG [P-MAX a mí] PATIENT ANIMATE)
returns: (EXTERNAL [P-MAX a mí] AGENT ANIMATE),

and the call: (PICK-ARG [N-MAX el libro] AGENT NIL)
returns: (INTERNAL [N-MAX el libro] GOAL).

Finally, the call:
  (LOCATE-UNIQUE-ARG-STRUCTURE
     ((EXTERNAL [P-MAX a mí] AGENT ANIMATE)
      (INTERNAL [N-MAX el libro] GOAL))
     ((EXTERNAL (AGENT ANIMATE)) (SUBCAT (GOAL))))
returns:
   ((EXTERNAL [P-MAX a mí] AGENT ANIMATE)
    (INTERNAL [N-MAX el libro] GOAL))
```

Note that in this case there is already a one-to-one mapping since PICK-ARG returned only one target language candidate for each source language argument. In the general case, there will be more than one candidate, and LOCATE-UNIQUE-ARG-STRUCTURE will choose a single unique mapping between arguments.

Now the function SET-UP-ARGUMENT-POSITIONING is called, and the internal and external arguments are replaced so that the base structure is modified to be:

```
[C-MAX
   [I-MAX
      [I-SPEC [P-MAX a mí]]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [V [CL-DAT me] [V like] [N-MAX el libro]]]]]]
```

Finally, the SET-UP-STRUCTURAL-REALIZATION function performs a CSR (canonical structural realization) mapping of the argument θ-roles to their canonical structural realization in the target language. Thus, the external argument structure is changed to **N-MAX** (since the call (CSR AGENT) returns **N-MAX** in English), and the internal argument structure is left as **N-MAX** (since the call (CSR GOAL) returns **N-MAX** in English). The functions UPDATE-INTERNAL-STRUCTURE and UPDATE-EXTERNAL-STRUCTURE perform each of these modifications respectively. In the conversion of **P-MAX** to **N-MAX**, the preposition *a* is dropped, and the **N-MAX** *mí* becomes the head of the phrase. Note that such deletion cannot be arbitrary: only those elements that do not have *semantic content* can undergo deletion.[5]

The structure obtained after execution of SET-UP-STRUCTURAL-REALIZATION is:

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX mí]]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [V [CL-DAT me] [V like] [N-MAX el libro]]]]]]
```

The arguments themselves must now be translated. This is accomplished by performing a recursive call to REPLACE-SOURCE-WITH-TARGET on each of the arguments. The result is:

[5]In this implementation, semantically null elements are those constituents that have 2 properties: (1) they do not serve as arguments of a θ-role assigner; and (2) they are not θ-role assigners. See Chomsky (1986b), pp. 70-71, for details concerning deletion of semantically null elements.

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX me]]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [V [CL-DAT me] [V like] [N-MAX the book]]]]]]
```

Notice that the mapping into the target language is not complete: first, the clitic *me* must be removed since there are no clitics in English; and second, the subject *me* must be transformed into its nominative form *I*. The first of these two tasks is performed by the structural movement component as described in section 7.2.1; the second task is performed by the linguistic constraint component as described in section 7.2.3.

7.2 Generation Routines

Minimal effort has been spent in the area of generation since it is not the focus of the research presented in this report. The STRUCTURAL-MOVEMENT and MORPHOLOGICAL-SYNTHESIS routines are in appendix J. The functions called by each of these routines are in appendix H.5. They will be described in the following two sections. The third section describes the linguistic constraints imposed on the generation routines. A procedural description of the generation routines is given in table 7.2.

7.2.1 Structural Movement (Move-α)

Just as the $\overline{\text{X}}$ module is the main driver of the Earley parser, the Bounding module is the main driver for the generator (in particular, the structural-movement component of the generator). The function MOVE-ALPHA is called on each of the target base trees, and one or more surface trees are generated and returned. The first part of MOVE-ALPHA consists of applying movement functions, while the second part (to be described in section 7.2.3) applies GB constraints on the output of the movement functions. Before elements are moved out of base positions, two functions from the $\overline{\text{X}}$ module are executed: GENERATE-CORRECT-ADJUNCTIONS-AND-SPECIFIERS, and GENERATE-CORRECT-CONSTITUENT-ORDER. (See appendix **??**.) The reader may refer to the first part of table 7.2 for an overview of the order in which structural movement actions are applied.

1. Perform structural movement by:
   - (a) Changing adjunctions and specifiers to be consistent with the target language requirements
   - (b) Modifying the structure to be consistent with the constituent order of the target language
   - (c) Performing movement via adjunction and substitution subject to Bounding constraints and language-specific constraints.
   - (d) Setting up A and $\overline{\text{A}}$ positions
   - (e) Checking Binding conditions
   - (f) Setting up Government relations
   - (g) Checking ECP
   - (h) Checking Case assignment
   - (i) Checking θ-role assignment
2. Perform morphological synthesis by:
   - (a) Lexicalizing empty elements
   - (b) Mapping features of root forms into affixes
   - (c) Applying Kimmo generation to derive the surface sentence

Table 7.2: Procedural Description of the Generation Module

The first function ensures that certain incompatibilities (*e.g.*, clitics in a non-clitic language or *pro* in a non-pro-drop language) are eliminated. The *CURRENT-BASE-SPECIFIERS-AND-ADJUNCTION bounding parameter is used to determine positions where specifier elements (*e.g.*, N-MAX in I-SPEC) and adjoined elements (*e.g.*, clitic to V) are base-generated in the target language. If an element is allowed to occur in a position, two results are returned: one in which it *does* occur (ALLOW-OCCURRENCE), and one in which it does *not* occur. If an element exists in a position where it is *not* allowed to occur, it is either eliminated (REMOVE-UNAVAILABLE), or placed in a position where it *is* allowed to occur (MODIFY-POSITION). Some examples of the operation of this function are in table 7.3. The second $\overline{\text{X}}$ function arranges the order of constituents so that they are compatible with the constituent order of the target language.

Once the structure has been changed to accommodate the target language, all candidates for movement are paired up with positions to which they might move. The function GET-SUBSTITUTION-CANDIDATES returns possible positions to which an element can move via substitution (*e.g.*, *Wh*-movement), and the function GET-ADJUNCTION-CANDIDATES re-

Source Sentence	*Target Base Structure*	*Action*	*Output*
Yo leo libros buenos	I [read+tns] [books good]	Re-Position Adjunct	I [read+tns] [good books]
Parece que Juan vaya	[pro] seem John [go+tns]	Remove *pro*	[N-MAX] seem John [go+tns]
Juan parece ir	[seem+tns] John [go+inf]	Allow Subject	[N-MAX] [seem+tns] John [go+inf] [seem+tns] John [go+inf]
Yo le veo a él	I [le-see+tns] him	Remove Clitic	I [see+tns] him
I hit him	yo [pegar+tns] a él	Allow clitic	yo [cl-pegar+tns] a él yo [pegar+tns] a él
What did you eat	[do+tns] tú comer qué	Remove *do*	tú [comer+tns] qué
Qué comiste tú	you [eat+tns] what	Allow *do*	[do-aux] you [eat+tns] what you [eat+tns] what

Table 7.3: Operation of GENERATE-CORRECT-ADJUNCTIONS-AND-SPECIFIERS

turns all positions to which an element can move via adjunction (*e.g.*, V-preposing). It is assumed that substitution is always made in specifier position; thus GET-SUBSTITUTION-CANDIDATES accesses the *CURRENT-CHOICE-OF-SPEC parameter in order to determine the possible substitution site. The GET-ADJUNCTION-CANDIDATES function accesses the *CURRENT-ADJUNCTION parameter in order to determine the possible adjunction sites. Both of these functions access the *CURRENT-DERIVED-SPECIFIERS-AND-ADJUNCTION to search for possible movement sites, and also the *CURRENT-BOUNDING-NODES parameter so that the search for movement sites does not go beyond 2 bounding nodes.

The PERFORM-MOVEMENT function applies the PERFORM-SUBSTITUTION routine to move elements into specifier positions, and the PERFORM-ADJUNCTION routine to move elements into adjunction positions. The *CURRENT-LANGUAGE-SPECIFIC-EFFECTS parameter is accessed at this time so that obligatory movement will not be overlooked. For example, suppose the target base structure is:

```
[C-MAX
   [COMP-COMPLEMENT
      [I-MAX
         [I-SPEC [N-MAX you]]
         [I E [pres p2 sg]]
         [I-COMPLEMENT [V-MAX [V see] [N-MAX what]]]]]]
```

The PERFORM-SUBSTITUTION routine moves `[N-MAX what]` into COMP-SPEC position, thus deriving the following structure:[6]

```
[C-MAX
   [COMP-SPEC [N-MAX what]ᵢ]
   [COMP-COMPLEMENT
      [I-MAX
         [I-SPEC [N-MAX you]]
         [I E [pres p2 sg]]
         [I-COMPLEMENT [V-MAX [V see] [N-MAX e]ᵢ]]]]]
```

CHECK-LANGUAGE-SPECIFIC-EFFECTS determines that this structure is not complete because of the following the parameter setting for English:

$$(((\text{OR (DO-SAI) (BE-SAI) (HAVE-SAI) (I-SAI)}) \Leftrightarrow \text{(WH-MOVEMENT)}))$$

This parameter setting forces SAI to take place. Thus, the result is not added to the *SURFACE-TREES array, but is temporarily saved (in RESULTS) so that other movement actions can apply.

On the other hand, the target base structure is:[7]

```
[C-MAX
   [COMP-COMPLEMENT
      [I-MAX [DO-AUX]
         [I-MAX
            [I-SPEC [N-MAX you]]
            [I E [pres p2 sg]]
            [I-COMPLEMENT [V-MAX [V see] [N-MAX what]]]]]]]
```

The PERFORM-SUBSTITUTION routine moves *what* into COMP-SPEC position, and the CHECK-LANGUAGE-SPECIFIC-EFFECTS test is satisfied (since both SAI and *Wh*-movement have taken place):

[6]During substitution and adjunction, links between traces and antecedents are maintained.

[7]The DO-AUX constituent is inserted by the GENERATE-CORRECT-ADJUNCTIONS-AND-SPECIFIERS routine described in section 7.1.1.

```
[C-MAX
  [C-SPEC [N-MAX what]$_i$
  [COMP-COMPLEMENT
    [I-MAX [DO-AUX]
      [I-MAX
        [I-SPEC [N-MAX you]]
        [I E [pres p2 sg]]
        [I-COMPLEMENT [V-MAX [V see] [N-MAX e]$_i$]]]]]]]
```

Once the target language surface structure is generated, the linguistic constraints routines apply. Notice that MOVE-ALPHA overgenerates. For example, it might move elements to positions where they cannot receive case or θ-role, or it may leave a trace in a position that is not properly governed. It is the duty of the GB constraints to weed out illegal structures generated by MOVE-ALPHA. This co-routine design is parallel to the design of the parser, which also generates unconstrained structure, and relies on the GB component to rule out ill-formed parses. For example, the source language sentence *Juan parece ir* (= John seems to go) is transformed into the following two structures:

[C-MAX
  [COMP-COMPLEMENT
    [I-MAX
      [I-SPEC [N-MAX John]$_i$]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
        [V-MAX
          [V seem]
          [I-MAX [N-MAX e]$_i$ to go]]]]]]
	[C-MAX
	  [COMP-COMPLEMENT
	    [I-MAX
	      [I-SPEC [N-MAX]]
	      [I E [pres p3 sg]]
	      [I-COMPLEMENT
	        [V-MAX
	          [V seem]
	          [I-MAX [N-MAX John] to go]]]]]]

It is then up to the GB component to determine that the second parse is bad (because *John* does not receive case). Section 7.2.3 discusses the linguistic constraints imposed on MOVE-ALPHA. The output of MOVE-ALPHA is stored in *SURFACE-TREES.

7.2.2 Morphological Synthesis

The morphological synthesis component consists of a mapping from root forms (*e.g.*, write) and affixes (*e.g.*, +en) into surface forms (*e.g.*, written). The Kimmo system provides this facility since it is a two-way system; thus, the morphological rules discussed in section 5.2.1 are applied in reverse to derive surface forms. The GENERATE function first interfaces with the Kimmo generator (by the MORPHER function) and then reads off leaf nodes linearly to

Type of Lexicalization	*Input Form*	*Output Form*
Subject Lexicalization	[[N-MAX p1 sg nom animate] eat dinner]	[I eat dinner]
Pleonastic Lexicalization (it-insertion)	[[N-MAX p3 sg pleon] [seem pres] to be [rain prog]]	[it [seem pres] to be [rain prog]]
Clitic Lexicalization	[[CLITIC acc] [ver pres]]	[lo [ver pres]]
Auxiliary Lexicalization (do-insertion)	what [DO-AUX past] he eat	what [do past] he eat

Table 7.4: Operation of Lexicalization Routine

	English		*Spanish*	
Features	*Affix(es)*	*Example*	*Affix(es)*	*Example*
pres p1 sg	0	walk	+o	camino
pres p2 sg	0	walk	+as	caminas
pres p3 sg	+s	walks	+a	camina
pres p1 pl	0	walk	+an	caminan
pres p3 pl	0	walk	+amos	caminamos
past p1 sg	+ed	walked	+é	caminé
past p2 sg	+ed	walked	+aste	caminaste
past p3 sg	+ed	walked	+ó	caminó
past p1 pl	+ed	walked	+amos	caminamos
past p3 pl	+ed	walked	+aron	caminaron
perf	+en	walked	+ado	caminado
a sg masc	0	good	+o	bueno
a sg fem	0	good	+a	buena
a pl masc	0	good	+os	buenos
a pl fem	0	good	+as	buenas

Table 7.5: Mapping of Features to Affixes

derive the surface sentence (by the SURFACE-FORM function). The final result is stored in the *TARGET-RESULT array. The reader may refer to the second part of table 7.2 for an overview of the order in which morphological synthesis actions are applied.

In preparation for the Kimmo processing, two routines are required: one is the LEXICALIZATION routine, which maps null elements into their lexicalized counterparts, and the other is CHANGE-FEATS-TO-AFFIX, which maps features into appropriate affixes. Table 7.4 shows some example of lexicalization provided by the system. Table 7.5 shows the mapping from features to affixes in Spanish and English.

7.2.3 Linguistic Constraints of Generation Routines

The linguistic constraints of the generation stage are accessed only during structural movement since morphological synthesis operates at the word level, not at the phrase level (as required by the GB constraints). The constraints accessed at structural movement time are parallel to those accessed at parsing time. The second half of MOVE-ALPHA applies the same principles that are applied by DROP-INTO-NEXT-LOWER at POP time. However, there are two differences between the DROP-INTO-NEXT-LOWER routine and the second half of MOVE-ALPHA: (1) access to the θ module is used to test θ-role assignment rather than perform θ-role assignment; and (2) case assignment may modify the constituent to which case is being assigned.

The reason θ-roles are checked, but not assigned, is that θ-roles must already be assigned when the structure is in D-structure form before MOVE-ALPHA applies. A moved element retains the θ-role it is initially assigned, and any positions left behind are left θ-marked. Thus, PERFORM-THETA-ASSIGNMENT is called by MOVE-ALPHA, only to check that the derived structure satisfies the θ-Criterion.

By contrast, case assignment may result in a modification of the structure derived by MOVE-ALPHA. Recall that in the *gustar-like* example of section 7.1.3, the final structure returned by REPLACE-SOURCE-WITH-TARGET-LANGUAGE is the following:

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX me]]
    [I E [pres p3 sg]]
    [I-COMPLEMENT
      [V-MAX [V [CL-DAT me] [V like] [N-MAX the book]]]]]]
```

We noted that the target language form is not complete because the clitic *me* needs to be removed (this is performed by the GENERATE-CORRECT-ADJUNCTIONS-AND-SPECIFIERS routine as shown in table 7.3), and the objective subject *me* must be transformed into its nominative equivalent *I*. It is the job of the PERFORM-CASE-ASSIGNMENT routine (called by MOVE-ALPHA) to perform this second task. Note that this task would not be necessary if the interlingual form of figure 7.3 were used, since morphological considerations are not included in such a form. However, because a more syntactic underlying form is used here, case mapping routines are required.

In the above example, the I(nfl) node assigns nominative case to `[N-MAX me]`, thus changing it to `[N-MAX I]`.[8] After case and θ-role assignment have taken place, the result is:

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX I [nom agent]]]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [V [V like] [N-MAX the book [obj goal]]]]]]]
```

The rest of the linguistic constraints are applied exactly as they are applied during parsing except for those belonging to the Bounding module (these were already taken care of during the first half of the MOVE-ALPHA routine).

The surface form of the above structure is simply a linear reading of the leaf nodes without brackets and feature annotations: *I like the book.*

[8] A similar process occurs for the object `[N-MAX the book]`, which was originally assigned nominative case, but is now assigned objective case. However, this case alternation is not morphologically realizable for non-pronominal noun phrases.

Chapter 8

Example Of Translation

This chapter presents an example of translation of a sentence from Spanish to English. Additional examples of phenomena handled by the system are in appendix K.

Consider the following sentence:

(60) ¿A quién le visitaste? (= who did you visit)

We will examine the processing of this sentence during each of the three translation stages.

8.1 Parsing

Once preprocessing and morphological analysis have taken place, the input to the parser is the following two forms:

```
((a P (SUBCAT (N)))
 (quie^n N ANIMATE NEUT P3 WH WH-PHRASE-A)
 (le CL-DAT NEUT PRONOUN P3 SG OBJ ANIMATE)
 (visitar V SG P2 PAST
  (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (P-GOAL ANIMATE)) (SUBCAT (GOAL INANIMATE))))

((a INF I (SUBCAT (V)))
 (quie^n N ANIMATE NEUT P3 WH WH-PHRASE-A)
 (le CL-DAT NEUT PRONOUN P3 SG OBJ ANIMATE)
 (visitar V SG P2 PAST
  (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (P-GOAL ANIMATE)) (SUBCAT (GOAL INANIMATE))))
```

Note that *a* has been analyzed as both a preposition and an infinitive marker (thus accounting for the two input forms).[1]

The precompilation routine generates the rules and templates shown in tables 5.8 and 5.10. Before scanning the first input word *a*, the parser expands **C-MAX** (the start symbol) as follows:

Input Word: NIL; Next Word: a

Stack #1
[I-SPEC [N-MAX e]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C E] [C-COMPLEMENT]]

Stack #2
[C-SPEC [N-MAX e]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #3
[C-SPEC [P-MAX e]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #4
[P-MAX [P] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #1 corresponds to the case where there is no specifier under **C-MAX**. The stack has been expanded until subject position has been reached. The subject is predicted to be empty (either PRO, *pro*, or trace) since the next word *a* is not derivable from **N-MAX**. There is no antecedent if this element is a trace, but the parse is not yet ruled out in case an antecedent has not yet been seen. Stacks #2, #3, and #4 contain a specifier under **C-MAX**; this specifier expands either to **N-MAX** (stack #2) or to **P-MAX** (stacks #3 and #4). In stack #2, **N-MAX** must be empty (*i.e.*, PRO, *pro*, or trace) for the same reason that **C-SPEC** must be empty in stack #1. In the case of **P-MAX**, the element may either be empty (stack #3) or it may contain a lexical element (stack #4) since the next word in the input is of category **P**.[2]

The top-of-stack element is complete in the first three stacks, so the completed element is dropped, and the parse continues. In stacks #2 and #3, **C-COMPLEMENT** is expanded to be **I-MAX**, and this in turn is expanded into **N-MAX** (which again must be empty). The result is:

[1] The second form will quickly be ruled out, but we will include it in our analysis until it is eliminated.

[2] For brevity, certain steps will not be shown in this example. For instance, the **P-MAX** in stack #4 may or may not have a specifier since P-SPEC is optional; however, only the case where **P-MAX** does *not* have a specifier is shown here. (The case in which **P-MAX** *does* have a specifier is ruled out immediately since **P-SPEC** does not derive the next input word.) In general, specifiers will not be shown if they do not derive the next input word since the one-word lookahead facility rules them out as soon as they are predicted.

Input Word: NIL; Next Word: a

Stack #1
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C E] [C-COMPLEMENT]]

Stack #2
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [N-MAX e]] [C E] [C-COMPLEMENT]]

Stack #3
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX e]] [C E] [C-COMPLEMENT]]

Stack #4
[P-MAX [P] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

All traces remain unlinked, but ECP is not violated since there are still some unexpanded elements (in particular, I(nfl) nodes) that could potentially be proper governors. Notice that the top-of-stack in the first three stacks is the same. I(nfl) is traversed, and `I-COMPLEMENT` is expanded as `V-MAX` in these three stacks. The fourth stack remains as is since there is a terminal symbol (`P`) waiting to be scanned:

Input Word: NIL; Next Word: a

Stack #1
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C E] [C-COMPLEMENT]]

Stack #2
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [N-MAX e]] [C E] [C-COMPLEMENT]]

Stack #3
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX e]] [C E] [C-COMPLEMENT]]

Stack #4
[P-MAX [P] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Now that there is a terminal symbol (either `V` or `P`) at the top of all the stacks, the first word *a* is scanned. Note that the second input form (in which *a* is analyzed as an infinitive marker) is now eliminated since it is not of category `V` or `P`:

Input Word: a; Next Word: quie^n

[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Now that a word has been scanned, its features are instantiated. Thus, the `[P a]` node is set up as follows:

CAT:	p
WORD:	a
TRANSLATION:	[(to) (at)]
SUBCATEGORIZATION:	[(n)]
THETA-ROLES:	nil

This information is percolated up to P-MAX. Since there is no specifier, spec/max agreement automatically succeeds.

Now an unexpanded complement is at the top of the stack. Recall that at PUSH time the subcategorization frame of the *closest head* (of the relevant category) determines complement expansion. The direction of the search for the closest head is to the left since Spanish is a head-initial language. The closest head of category `P` at this point is the word *a*; thus, `P-COMPLEMENT` is expanded as `N-MAX` in accordance with the subcategorization information associated with the word `a`. The new stack configuration is the following:

Input Word: a; Next Word: quie^n

Stack #1
[N-MAX e]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #2
[N-MAX [N] [N-COMPLEMENT]]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Note that `N-MAX` has been expanded to be either empty (*i.e.*, PRO, *pro*, or trace) or lexical. Since this completes the `N-MAX`, stack #1 is popped and stack #2 remains as is:

Input Word: a; Next Word: quie˜n

Stack #1
[P-MAX [P a] [P-COMPLEMENT [N-MAX e]]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #2
[N-MAX [N] [N-COMPLEMENT]]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Now that the parser is in the POP stage, several conditions apply. First the GB component attempts to link `[N-MAX e]` up to an antecedent. Finding no antecedent, it tries to interpret `[N-MAX e]` as PRO or *pro*. However, both of these cause an ECP violation: PRO cannot be governed (but `[P a]` governs `[N-MAX e]`); and *pro* must be governed by AGR (but there is no AGR available for government of `[N-MAX e]`). Thus, the only possibility for `[N-MAX e]` is trace (assuming rightward movement has prevented the antecedent from being seen yet). This possibility is held in reserve (for the time being). Case and θ-role assignment take place at this point, thus marking the `N-MAX` with objective case (but no θ-role since `P` is not an assigner).

Note that even though `[N-MAX e]` does not have a θ-role, the parse is not yet rejected since θ-role might be transmitted to the trace via some other mechanism later in the parse. Now that `P-MAX` is complete, stack #1 is popped. `C` is traversed, and `C-COMPLEMENT` is expanded until the subject position is reached:

Input Word: a; Next Word: quie˜n

Stack #1
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #2
[N-MAX [N] [N-COMPLEMENT]]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Again, `N-MAX` may be expanded to be either empty (*i.e.*, PRO, *pro*, or trace) or lexical; both these possibilities are tried, and in the case where `N-MAX` is empty, the stack is popped:

Input Word: a; Next Word: quie´n

Stack #1
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #2
[N-MAX [N] [N-COMPLEMENT]]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #3
[N-MAX [N] [N-COMPLEMENT]]
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT]]

In stacks #1 and #3, I(nfl) is traversed, and `I-COMPLEMENT` is expanded as `V-MAX`. Now all three stacks contain a terminal symbol (either `V` or `N`) in the top-of-stack element:

Input Word: a; Next Word: quie´n

Stack #1
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #2
[N-MAX [N] [N-COMPLEMENT]]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Stack #3
[N-MAX [N] [N-COMPLEMENT]]
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]]
[C E] [C-COMPLEMENT]]

Stack #1 is ruled out immediately because `V` does not derive the next input word *quién*. (This is determined by the one-word lookahead facility.) Thus, stacks #2 and #3 are the only applicable stacks for scanning the word *quién*:

Input Word: quie^n; Next Word: le

Stack #1
[N-MAX [N quie^n] [N-COMPLEMENT]]
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #2
[N-MAX [N quie^n] [N-COMPLEMENT]]
[P-COMPLEMENT [N-MAX]]
[P-MAX [P a] [P-COMPLEMENT]]
[C-SPEC [P-MAX]]
[C-MAX [C-SPEC] [C] [C-COMPLEMENT]]

Because *quién* does not subcategorize for anything, the $\overline{\text{X}}$ module eliminates `N-COMPLEMENT` in both stacks. This completes the top-of-stack, and it can now be popped:

Input Word: quie^n; Next Word: le

Stack #1
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I] [I-COMPLEMENT]]]]

Stack #2
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C] [C-COMPLEMENT]]

Now I(nfl) is traversed in stack #1 and `C` is traversed in stack #2. This allows stack #1 to expand `I-COMPLEMENT` and stack #2 to expand `C-COMPLEMENT`:

Input Word: quie^n; Next Word: le

Stack #1
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[I-SPEC [N-MAX]]
[I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

In stacks #2 and #3, `V` can either be left as is, or it can be expanded into the adjunct possibility `[V [CL-DAT] [V]]`. In stack #2, `[N-MAX]` may be empty since the next word *le* is not derivable from `[N-MAX]`. The resulting stacks are:

Input Word: quie^n; Next Word: le

Stack #1
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V [CL-DAT] [V]]
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #3
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Stack #4
[V [CL-DAT] [V]]
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Stack #5
[I-SPEC [N-MAX e]] [I-MAX [I-SPEC] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Stacks #1 and #3 are immediately ruled out since the next word *le* is not of category **V**. Stacks #2 and #4 are not tossed out since a clitic category is at the top of the stack. In stack #5, **N-MAX** is complete, so the stack is popped. This mean that traces need to be linked up, and ECP must be checked. Since **N-MAX** is empty, the trace possibility is tried. The only possible antecedent so far is the word *quién*; thus, the trace is linked with *quién* when the stack is popped. The bounding module determines that this trace-antecedent linking is valid since no bounding nodes are crossed. The other possibilities for **N-MAX** (PRO, *pro*, or unlinked trace) are also tried. The result is:

Input Word: quie^n; Next Word: le

Stack #1
[V [CL-DAT] [V]]
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V [CL-DAT] [V]]
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[I-MAX [I-SPEC [N-MAX e]$_i$] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]$_i$]]] [C E] [C-COMPLEMENT]]

Stack #4
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Note that the analysis in stack #3 is that of movement from subject position. This analysis will be ruled out later when nominative case is assigned to [N-MAX e]$_i$ because objective case is assigned to [N-MAX [N quién]]$_i$, and this will force a case clash. Thus, we will no longer consider this parse. On the other hand, in stack #4 I(nfl) is traversed and **I-COMPLEMENT** is expanded. Stacks #1 and #2 remain as is:

Input Word: quie^n; Next Word: le

Stack #1
[V [CL-DAT] [V]]
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V [CL-DAT] [V]]
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Now V in stack #3 can be expanded to the clitic adjunction structure as in stack #1 and #2:

Input Word: quie^n; Next Word: le

Stack #1
[V [CL-DAT] [V]]
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V [CL-DAT] [V]]
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n][obj]]]]] [C E] [C-COMPLEMENT]]

Stack #4
[V [CL-DAT] [V]]
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Note that only stack #2 is correct so far. Stack #1 has incorrectly predicted that *quién* is in subject position and that the verb is not preposed. Stacks #3 and #4 also do not contain a preposed verb. However, when the clitic *le* is scanned, stacks #1, #2, and #4 still remain:

Input Word: le; Next Word: visitar

Stack #1
[V [CL-DAT le] [V]]
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V [CL-DAT le] [V]]
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[V [CL-DAT le] [V]]
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Since the nonterminal `V` is at the top of all three stacks, the verb *visitar* can immediately be scanned:

Input Word: visitar; Next Word: NIL

Stack #1
[V [CL-DAT le] [V visitar]]
[V-MAX [V]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V [CL-DAT le] [V visitar]]
[I-MAX [V] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[V [CL-DAT le] [V visitar]]
[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

The features of *visitar* are instantiated since a head has just been scanned. The `[V visitar]` node is set up as follows:

CAT:	v
WORD:	visitar
GENDER:	nil
PERS:	p2
NUMBER:	sg
TENSE:	past
TRANSLATION:	[(visit)]
SUBCATEGORIZATION:	[(p) (n)]
EXTERNAL-CATS:	[(n)]
THETA-ROLES:	[(p-goal animate) (goal inanimate)]
EXTERNAL-ROLES:	[(agent animate)]

These features are then percolated up to the superior node (the dominating `V`). Then the clitic-V adjunction structure is popped, and, in stacks #2 and #3, the features are percolated up to the maximal level **V-MAX**. Since **V-MAX** has no specifier, spec/max agreement automatically succeeds. The three stacks now appear as follows:

Input Word: visitar; Next Word: NIL

Stack #1
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[I-MAX [V [CL-DAT le] [V visitar]] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

In stacks #1 and #3, **V-COMPLEMENT** is expanded according to the subcategorization frame of the *closest head* of category **V** (*i.e.*, the word *visitar*). Thus, **V-COMPLEMENT** is expanded to be **P-MAX** or **N-MAX**. The result is the following:

Input Word: visitar; Next Word: NIL

Stack #1
[V-COMPLEMENT [P-MAX]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V-COMPLEMENT [N-MAX]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #3
[I-MAX [V [CL-DAT le] [V visitar]] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #4
[V-COMPLEMENT [P-MAX]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #5
[V-COMPLEMENT [N-MAX]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Since there are no more input words to scan, expansions of the top-of-stack in stacks #1–#5 must contain empty elements. Thus `P-MAX` in #1 and #4 must be a trace; `N-MAX` in #2 and #5 must be either PRO, *pro*, or trace; and subject position in stack #3 must be PRO, *pro*, or trace. Thus, we have:

Input Word: visitar; Next Word: NIL

Stack #1
[V-COMPLEMENT [P-MAX e]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #2
[V-COMPLEMENT [N-MAX e]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT]]]]

Stack #3
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #4
[V-COMPLEMENT [P-MAX e]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #5
[V-COMPLEMENT [N-MAX e]]
[V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stacks #1, #2, #4 and #5 are now complete and are popped:

Input Word: visitar; Next Word: NIL

Stack #1
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT [V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT [P-MAX e]]]]]]]

Stack #2
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX e [obj]]]]] [C E] [C-COMPLEMENT [I-MAX [I-SPEC [N-MAX [N quie^n]]] [I E] [I-COMPLEMENT [V-MAX [past p2 sg] [V [CL-DAT le] [V visitar]] [V-COMPLEMENT [N-MAX e]]]]]]]

```
Stack #3
[I-MAX [I-SPEC [N-MAX e]] [I] [I-COMPLEMENT]]
[I-MAX
   [V [CL-DAT le] [V visitar]]
   [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX
   [C-SPEC
     [P-MAX [P a]
         [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]]
   [C E] [C-COMPLEMENT]]
```

```
Stack #4
[C-MAX
   [C-SPEC
     [P-MAX [P a]
       [P-COMPLEMENT
         [N-MAX [N quie^n] [obj]]]]]
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX e]] [I E]
         [I-COMPLEMENT
           [V-MAX [past p2 sg]
             [V [CL-DAT le]
               [V visitar]]
             [V-COMPLEMENT
               [P-MAX e]]]]]]]
```

```
Stack #5
[C-MAX
  [C-SPEC
    [P-MAX [P a]
      [P-COMPLEMENT
        [N-MAX [N quie^n] [obj]]]]]
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC [N-MAX e]] [I E]
        [I-COMPLEMENT
          [V-MAX [past p2 sg]
            [V [CL-DAT le]
              [V visitar]]
            [V-COMPLEMENT
              [N-MAX e]]]]]]]
```

During popping, several GB constraints are checked. First, stacks #1 and #2 are ruled out because `[N-MAX e]` under `P-MAX` in `C-SPEC` is forced to be linked with `[N-MAX [N quién]]` in subject position (since it is the only possible antecedent); this causes a case clash: `[N-MAX [N quién]]` is assigned nominative case by I(nfl) with +tns features, and `[N-MAX e]` is assigned objective case by `[P a]`.

In stacks #4 and #5, `[N-MAX e]` in subject position must be *pro*: it cannot be PRO since it is governed by I(nfl) with +tns features; and it cannot be trace since it has nominative case and `[N-MAX [N quién]]` (its only possible antecedent) has objective case. At this point, stacks #4 and #5 are ruled out, not because of a GB constraint violation (in fact, ECP, Case, and θ conditions are satisfied), but because of a language-specific movement violation. The parameter setting (((PREPOSE) $\Rightarrow$ (WH-MOVEMENT WH-PHRASE-A))) requires that preposing occur if *Wh*-movement (*i.e.*, movement of `P-MAX` or `N-MAX` into `C-SPEC` position) has taken place. Since V-preposing has not taken place, stacks #4 and #5 are eliminated.

We are left with stack #3. I(nfl) is traversed, and `I-COMPLEMENT` is expanded until `V` is at the top of the stack:

Input Word: visitar; Next Word: NIL

[V-MAX [V] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Since there are no more input words, the only possibility for `V` is trace:

Input Word: visitar; Next Word: NIL

[V-MAX [V e] [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]] [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

The only antecedent for `[V e]` is `[V visitar]`. Thus, trace linking takes place, and the features of `[V visitar]` are transferred to `[V e]`. These features are then percolated up to `V-MAX`. Now `V-COMPLEMENT` needs to be expanded. The *closest head* procedure expands the complement to `P-MAX` and `N-MAX` (as dictated by the subcategorization requirements of *visitar*):

Input Word: visitar; Next Word: NIL

Stack #1	Stack #2
[V-COMPLEMENT [N-MAX]]	[V-COMPLEMENT [P-MAX]]
[V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT]]	[V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT]]
[I-COMPLEMENT [V-MAX]]	[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]	[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]	[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]
[C-COMPLEMENT [I-MAX]]	[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]	[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

In either case, an empty element is dropped and the stacks are then popped. During popping, several actions take place. When `V-COMPLEMENT` is popped, trace linking takes place. In the case of stack #1, there are two possibilities of antecedent for the verbal complement: `[N-MAX e]` in subject position and `[N-MAX [N quién]]` in `C-SPEC` position. In the case of stack #2, the only possible antecedent is `[P-MAX a quién]`. Trace linking takes place, and the Bounding module determines that this linking is valid since no bounding nodes are crossed. The result is:

Input Word: visitar; Next Word: NIL

Stack #1
[V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [N-MAX e]$_i$]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]$_i$]]] [C E] [C-COMPLEMENT]]

Stack #2
[V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [N-MAX e]$_i$]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]$_i$] [I E] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]] [C E] [C-COMPLEMENT]]

Stack #3
[V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [P-MAX e]$_i$]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX e]] [I E] [I-COMPLEMENT]]
[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]
[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]$_i$] [C E] [C-COMPLEMENT]]

Next, when `V-MAX` is popped, inflection information is percolated up to the `I-MAX` level, at which point `[N-MAX e]` in subject position is assigned nominative case and forced to be *pro* (it cannot be PRO since it is governed by I(nfl) with +tns features, and it cannot be trace since its only possible antecedent (`[N-MAX [N quién]]`) is assigned objective case). Once this is determined, *pro* is given `[p2 sg]` features due to merging of the I(nfl) constituent with subject position. Finally, objective case is assigned to `[N-MAX e]` under `V-COMPLEMENT` position in stacks #1 and #2. Because there is a case clash in stack #2 (`[N-MAX e]`$_i$, which has objective case, is linked with *pro*, which has nominative case), this stack is eliminated. We are left with two stacks:

Input Word: visitar; Next Word: NIL

Stack #1	Stack #2
[I-MAX [I-SPEC [N-MAX e [pro nom p2 sg animate]]] [I E] [I-COMPLEMENT [V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [N-MAX e [obj]]$_i$]]]]	[I-MAX [I-SPEC [N-MAX e [pro nom p2 sg animate]]] [I E] [I-COMPLEMENT [V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [P-MAX e]$_i$]]]]
[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]	[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]
[C-COMPLEMENT [I-MAX]]	[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]$_i$]]] [C E] [C-COMPLEMENT]]	[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]$_i$] [C E] [C-COMPLEMENT]]

Now that **I-MAX** is complete, it too is popped:

Input Word: visitar; Next Word: NIL

Stack #1	Stack #2
[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX	[I-MAX [V [CL-DAT le] [V visitar]]$_j$ [I-MAX]]
[I-SPEC [N-MAX e [pro nom p2 sg animate]]] [I E] [I-COMPLEMENT [V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [N-MAX e [obj]]$_i$]]]]	[I-SPEC [N-MAX e [pro nom p2 sg animate]]] [I E] [I-COMPLEMENT [V-MAX [past p2 sg] [V e]$_j$ [V-COMPLEMENT [P-MAX e]$_i$]]]]
[C-COMPLEMENT [I-MAX]]	[C-COMPLEMENT [I-MAX]]
[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]$_i$]]] [C E] [C-COMPLEMENT]]	[C-MAX [C-SPEC [P-MAX [P a] [P-COMPLEMENT [N-MAX [N quie^n] [obj]]]]]$_i$] [C E] [C-COMPLEMENT]]

At this point θ-role assignment takes place. External θ-role is assigned to *pro* since *visitar* requires an external agent with *animate* features. Internal θ-role assignment is more complex. Recall that clitics absorb s-government. This means that case assignment to a verbal clitic takes priority over case assignment to a verbal complement. Consequently, `[CL-DAT le]` receives objective case. Then, since no other argument is *visible* (*i.e.*, assigned case) for θ-assignment, the clitic is assigned a θ-role of *goal*.

Now θ-role transmission takes place. In stack #1, neither `[N-MAX [N quién]]` under `C-SPEC` nor `[N-MAX e]` in verbal complement position have a θ-role; however, since `[N-MAX [N quién]]` has the same case as the clitic (both have objective case), θ-role transmission is triggered, and the *goal* θ-role is copied to `[N-MAX [N quién]]`. This leaves `[N-MAX e]` without a θ-role. (Recall that θ-role transferral occurs only from trace to antecedent, not vice-versa.) The result is that stack #1 is ruled out by the θ-Criterion. By contrast, in stack #3 `[CL-DAT le]` transmits θ-role to `[N-MAX [N quién]]`, and the θ-Criterion is satisfied since the verbal complement `[P-MAX e]` does not require θ-role.

Since `C-MAX` is complete, it is popped, and the final parse is:

```
[C-MAX
  [C-SPEC
    [P-MAX [P a]
      [P-COMPLEMENT
        [N-MAX [N quie^n] [obj goal]]]]_i]
  [C E]
  [C-COMPLEMENT
    [I-MAX [V [CL-DAT le [obj goal]] [V visitar]]_j
           [I-MAX
             [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
             [I E]
             [I-COMPLEMENT
               [V-MAX [past p2 sg]
                  [V e]_j
                  [V-COMPLEMENT [P-MAX e]_i]]]]]]]
```

This structure is passed to the replacement module and processed as described in the following section.

8.2 Replacement

During replacement, two actions take place: first moved elements are dropped into D-structure positions; and second, source language heads and arguments are thematically substituted by their target language counterparts. Replacement of moved elements in the parsed structure derived in section 8.1 results in the following form:

```
[C-MAX
  [C-SPEC NIL]
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [V NIL]
      [I-MAX
        [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
        [I E]
        [I-COMPLEMENT
          [V-MAX [past p2 sg]
            [V [CL-DAT le [obj goal]] [V visitar]]
              [V-COMPLEMENT
                [P-MAX [P a]
                  [P-COMPLEMENT
                    [N-MAX [N quie^n] [obj goal]]]]]]]]]]
```

Here the verb `[V visitar]` has moved into its D-structure position, leaving NIL behind in the preposed verbal position. Also, the *wh*-phrase `[P-MAX [P a] [N-MAX [N quié]]]` has been moved into `V-COMPLEMENT` position, leaving NIL behind in `C-SPEC` position. Before thematic substitution can take place, the evacuated elements (*i.e.*, positions that contain NIL) must be removed, and useless structure (*e.g.*, `[I-MAX [I-MAX ...]]`) must be eliminated:

Evacuated Elements Removed

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-MAX
        [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
        [I E]
        [I-COMPLEMENT
          [V-MAX [past p2 sg]
            [V [CL-DAT le [obj goal]] [V visitar]]
            [V-COMPLEMENT
              [P-MAX [P a]
                [P-COMPLEMENT
                  [N-MAX [N quie^n] [obj goal]]]]]]]]]]
```

Useless Structure Eliminated

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
        [V-MAX [past p2 sg]
          [V [CL-DAT le [obj goal]] [V visitar]]
          [V-COMPLEMENT
            [P-MAX [P a]
              [P-COMPLEMENT
                [N-MAX [N quie^n] [obj goal]]]]]]]]]
```

Now thematic substitution takes place. This is done recursively: first a head is translated; then the arguments of the head are translated; finally, if the arguments contain heads, these heads are also translated, and so on. The first lexical head to be translated is *visitar*. The translation is direct in this case (since there is no thematic divergence between the source and target language equivalent); the source language word is *visit*. The result is:

```
[C-MAX
   [C E]
   [C-COMPLEMENT
      [I-MAX
         [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
         [I E]
         [I-COMPLEMENT
            [V-MAX [past p2 sg]
               [V [CL-DAT le [obj goal]] [V visit]]
               [V-COMPLEMENT
                  [P-MAX [P a]
                     [P-COMPLEMENT
                        [N-MAX [N quie^n] [obj goal]]]]]]]]]]
```

The internal argument of *visit*, `[P-MAX [P a] [N-MAX [N quién]]]`, is also translated directly. The source language equivalent is `[P-MAX [P to] [N-MAX [N who]]]`. Although the clitic is eliminated later (during generation), it is translated for now to the source language word *him*. Thus, the following structure is derived:

```
[C-MAX
   [C E]
   [C-COMPLEMENT
      [I-MAX
         [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
         [I E]
         [I-COMPLEMENT
            [V-MAX [past p2 sg]
               [V [CL-DAT him [obj goal]] [V visit]]
               [V-COMPLEMENT
                  [P-MAX [P to]
                     [P-COMPLEMENT
                        [N-MAX [N who] [obj goal]]]]]]]]]]
```

This form is almost ready to be fed to the generation routines; however, the argument structure must be modified since the structural realizations of the source and target internal arguments diverge. The internal argument of *visitar* is a *p-goal*, which is structurally realized as `P-MAX`. By contrast, the internal argument of *visit* is a *goal*, which is structurally realized as `N-MAX`. Thus, the structural realization routines return the following form:

```
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX e [pro nom p2 sg animate agent]]]
       [I E]
       [I-COMPLEMENT
         [V-MAX [past p2 sg]
           [V [CL-DAT him [obj goal]] [V visit]]
           [V-COMPLEMENT [N-MAX [N who] [obj goal]]]]]]]]
```

Note that there is no thematic divergence between the source language structure and the target language structure. Thus, the internal and external argument positioning does not change (even though their structural realizations are not the same).

Now that all lexical constituents have been translated, this form is processed by the generation routines as described in the next section.

8.3 Generation

During generation, two types of procedures process the target language D-structure in turn: the first type concerns movement of elements into S-structure positions, and the second type consists of morphological synthesis into surface forms. Before elements are moved out of base position, the $\overline{X}$ module is accessed to ensure that certain incompatibilities are eliminated. For example, in the source language D-structure derived by the replacement routines in section 8.2, a *pro* subject and a clitic-adjunction structure are present. Since these are both disallowed in the target language (as determined by the BASE-SPECIFIERS-AND-ADJUNCTION parameter setting), they are eliminated.[3] Also, since a DO-AUX adjunct is allowed to occur in English, this structure is generated. The case where DO-AUX adjunction does *not* occur is also included; thus, two structures are generated:

[3]Elimination of *pro* amounts to removal of the pro feature.

```
Stack #1: DO-AUX Adjunction
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [DO-AUX]
         [I-MAX
           [I-SPEC [N-MAX e [nom p2 sg animate agent]]]
           [I E]
           [I-COMPLEMENT
             [V-MAX [past p2 sg]
               [V visit]
               [V-COMPLEMENT [N-MAX [N who] [obj goal]]]]]]]]]
```

```
Stack #2: No DO-AUX Adjunction
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX e [nom p2 sg animate agent]]]
       [I E]
       [I-COMPLEMENT
         [V-MAX [past p2 sg]
           [V visit]
           [V-COMPLEMENT [N-MAX [N who] [obj goal]]]]]]]]
```

At this point, the $\overline{\text{X}}$ module also checks that the constituent order of this structure is compatible with the constituent order of the target language. Because both Spanish and English are head-initial languages, no modifications are necessary. Now movement routines are activated, and all possible substitution and adjunction possibilities are tried (according to the DERIVED-SPECIFIERS-AND-ADJUNCTION parameter setting). The only movement that is applicable is *Wh*-movement (*i.e.*, substitution of `[N-MAX [N who]]` into `C-SPEC` position). Now the two stacks are multiplied into four (two *with* *Wh*-movement, and two *without* *Wh*-movement):

Stack #1

```
[C-MAX
  [C-SPEC
    [N-MAX [N who] [obj goal]]_i]
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [DO-AUX]
        [I-MAX
          [I-SPEC
            [N-MAX e
              [nom p2 sg animate agent]]]
          [I E]
          [I-COMPLEMENT
            [V-MAX [past p2 sg]
              [V visit]
              [V-COMPLEMENT
                [N-MAX e
                  [obj goal]]_i]]]]]]]
```

Stack #2

```
[C-MAX
  [C-SPEC
    [N-MAX [N who] [obj goal]]_i]
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC
        [N-MAX e [nom p2 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
        [V-MAX [past p2 sg]
          [V visit]
          [V-COMPLEMENT
            [N-MAX e
              [obj goal]]_i]]]]]]
```

Stack #3

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [DO-AUX]
        [I-MAX
          [I-SPEC
            [N-MAX e
              [nom p2 sg animate agent]]]
          [I E]
          [I-COMPLEMENT
            [V-MAX [past p2 sg]
              [V visit]
                [V-COMPLEMENT
                  [N-MAX [N who]
                    [obj goal]]]]]]]]]
```

Stack #4

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC
        [N-MAX e
          [nom p2 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
        [V-MAX [past p2 sg]
          [V visit]
          [V-COMPLEMENT
            [N-MAX [N who]
              [obj goal]]]]]]]]
```

The Bounding module determines that this movement is legal since the trace-antecedent linking crosses only one bounding node (**I-MAX**). However, the language-specific-effects constraint rules out stacks #2 and #3 because SAI (adjunction of **DO-AUX** to *I-MAX*) and *Wh-*

movement (substitution of `[N-MAX]` in `C-SPEC`) do not co-occur. Thus, only stacks #1 and #4 remain. Now linguistic constraints are applied. In particular, ECP is checked and assignment of case and θ-role takes place. Since no movement has taken place in stack #4, no traces need to be checked by ECP. By contrast, in stack #1, the trace `[N-MAX e]`$_i$ in `V-COMPLEMENT` position is checked by ECP: no violation is found since `[N-MAX e]`$_i$ is properly governed by `[V visit]`. Case assignment is straightforward: assignment of nominative case to subject position and objective case to object position does not result in a modification of lexical constituents since the cases that were already assigned are compatible. θ-role transmission is equally straightforward since there is no thematic divergence; thus, the θ-Criterion is satisfied.

Stacks #1 and #4 are now ready for morphological synthesis. In preparation for this process, lexicalization and feature-affix mapping must first take place. First the subject is lexicalized:

```
Stack #1
[C-MAX
   [C-SPEC [N-MAX [N who] [obj goal]]_i]    [C E]
   [C-COMPLEMENT
      [I-MAX
         [DO-AUX]
            [I-MAX
               [I-SPEC [N-MAX you [pronoun nom p2 sg animate agent]]]
               [I E]
               [I-COMPLEMENT
                  [V-MAX [past p2 sg]
                     [V visit]
                     [V-COMPLEMENT [N-MAX e [obj goal]]_i]]]]]]]
Stack #2
[C-MAX
   [C E]
   [C-COMPLEMENT
      [I-MAX
         [I-SPEC [N-MAX you [pronoun nom p2 sg animate agent]]]
         [I E]
         [I-COMPLEMENT
            [V-MAX [past p2 sg]
               [V visit]
               [V-COMPLEMENT [N-MAX [N who] [obj goal]]]]]]]]
```

Next the auxiliary is lexicalized:

```
                              Stack #1
[C-MAX
   [C-SPEC [N-MAX [N who] [obj goal]]_i]    [C E]
   [C-COMPLEMENT
     [I-MAX
       [DO-AUX do [past p2 sg]]
         [I-MAX
           [I-SPEC [N-MAX you [pronoun nom p2 sg animate agent]]]
           [I E]
           [I-COMPLEMENT
             [V-MAX [past p2 sg]
               [V visit]
               [V-COMPLEMENT [N-MAX e [obj goal]]_i]]]]]]]
                              Stack #2
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX you [pronoun nom p2 sg animate agent]]]
       [I E]
       [I-COMPLEMENT
         [V-MAX [past p2 sg]
           [V visit]
           [V-COMPLEMENT [N-MAX [N who] [obj goal]]]]]]]]
```

Now features are mapped to the source language affixes:

```
                         Stack #1
[C-MAX
   [C-SPEC [N-MAX [N who+0]]_i]
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [DO-AUX did+0]
         [I-MAX
           [I-SPEC [N-MAX you+0]]
           [I E]
           [I-COMPLEMENT
             [V-MAX
               [V visit+0]
               [V-COMPLEMENT [N-MAX e [obj goal]]_i]]]]]]]
                         Stack #2
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX you+0]]
       [I E]
       [I-COMPLEMENT
         [V-MAX
           [V visit+ed]
           [V-COMPLEMENT [N-MAX [N who+0]]]]]]]]
```

Morphological synthesis by the Kimmo generator takes place at this point:

Stack #1

```
[C-MAX
   [C-SPEC [N-MAX [N who]]_i]   [C E]
   [C-COMPLEMENT
     [I-MAX
       [DO-AUX did]
         [I-MAX
           [I-SPEC [N-MAX you]]
           [I E]
           [I-COMPLEMENT
             [V-MAX
               [V visit]
               [V-COMPLEMENT [N-MAX e [obj goal]]_i]]]]]]
```

Stack #2

```
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX you]]
       [I E]
       [I-COMPLEMENT
         [V-MAX
            [V visited]
            [V-COMPLEMENT [N-MAX [N who]]]]]]]
```

Finally, the surface forms are generated: *Who did you visit?* and *You visited who?*

8.4 Processing Time

Each of the three stages vary in the amount of time taken to process a given sentence. The replacement and generation stages take less time (generally around 20 seconds total) than the parsing stages because most of the structure that is manipulated has already been generated by the parser. By contrast, the parser generally takes a few minutes (with tracing turned on) since it generates the structure from scratch. Most of the slowdown is due to the fact that top-down prediction is used. The Earley parser generates a great deal of unnecessary structure before the linguistic constraints are applied. If the parser were driven by the input and the lexicon instead of by $\overline{X}$ templates, most of the unnecessary structure could be eliminated, and the system would be faster. An alternative parsing method based on a bottom-up projection of structure is discussed in section 9.2.

Chapter 9

Limitations, Future Work, and Conclusions

This chapter discusses some of the limitations of the translation approach presented here, as well as directions for future work and conclusions. The system does not handle several types of syntactic phenomena. Furthermore, it requires certain computationally and linguistically relevant modifications. These problems are addressed in the following two sections; the third section contains the conclusions.

9.1 Limitations

Spanish passivization has been implemented to the extent that the construction coincides with its English counterpart:

(61) Los documentos fueron perdidos por el hombre.
'The documents were lost by the man.'

However, the more common passive construction in Spanish includes the reflexive pronoun *se*, which has no equivalent English form:

(62) Se perdieron los documentos por el hombre.
'The documents were lost by the man.'

Although the reflexive use of *se* has been implemented (along with other clitics), the passive usage is not included.[1]

Coordination has also not been included in this implementation. Recall that $\overline{\text{X}}$ structures allow adjuncts at the minimal and maximal level; however, they do not permit two maximal structures to be conjoined, so that both structures have equal status. Thus, the following structure is disallowed: [X-MAX [X-MAX ...] [α] [X-MAX ...]], where α is a conjunction. See Fong (1986) for more details on a linguistically based approach to parsing conjoined phrases.

Certain adjuncts are not thoroughly handled by the system. For example, the adjunction of BE-AUX to a verb (as in *was eating*) is only parsed at a surface level since feature matching of adjuncts with heads has not been implemented. Similarly, adjectival adjunction is not properly handled for this reason. Thus, the incorrect form **hombre buena* (= man$_{[masc]}$ good$_{[fem]}$) would be considered as acceptable as the correct form *hombre bueno* (= man$_{[masc]}$ good$_{[masc]}$). Feature agreement has also not been implemented for complements of copula verbs since specifier-complement feature-matching is not included. Thus, *él es buena* (= he$_{[masc]}$ is good$_{[fem]}$) would be considered as acceptable as *él es bueno* (= he$_{[masc]}$ is good$_{[masc]}$)

Although thematic divergence has been handled to a certain extent (*e.g.*, the *gustar-like* example of section 7.1.3), cases of thematic divergence in which one or more lexical items are structurally modified have not yet been implemented. For example, the translation of *Tengo calor* to *I am hot* is not handled since the literal translation is *I have heat.* Not only is there different choice of verbs in each language, but there is also a categorial divergence between the predicate complements. The inclusion of θ-roles aids the translation of thematically divergent predicates, but the θ module and lexical processing routines needs to be extended in order to handle conversion of lexical items into structurally distinct but semantically equivalent constituents.[2]

An additional shortcoming of the system is that it requires the specification of language-specific effects. Unfortunately, the specification of these effects is in direct opposition to the

[1]Borer (1984) presents an analysis of *se* within the GB framework.

[2]Because Sharp's translation system does not include θ-role assignment, it cannot handle even simple thematic divergence (such as translation of the word *gustar*). Thus, the model presented here at least approaches the type of processing required for handling more complex cases of thematic divergence. It appears that the θ-grids of lexical items may ultimately be derivable from a suitable representation of their meanings; see Guerssel *et al.* (1985), Levin and Rappaport (1985), and Rappaport and Levin (1986) for discussion.

primary objective of the principle-based approach (*i.e.*, to avoid spelling out language-particular details). Happily, there are only a few language-specific effects for Spanish and English (in fact, one for each language); it is likely that this will be the case in most languages. A later version of the system might incorporate these effects in a principle-based way, depending on how the linguistic theory evolves.[3]

Currently, the system does not attempt to incorporate the language-specific requirements in a principle-based way. Rather, they are considered a part of what Chomsky (1986b) calls *periphery*:[4]

> Suppose we distinguish *core language* from *periphery*, where a core language is a system determined by fixing values for the parameters of UG, and the periphery is whatever is added on in the system actually represented in the mind/brain of a speaker-hearer ... What we learn are the values of the parameters and the elements of the periphery (along with the lexicon, to which similar considerations apply). The language that we then know is a system of principles with parameters fixed, along with a periphery of marked exceptions.

Although a better method of specifying peripheral phenomena might be constructed, the one used in this model is sufficient for the processing of the two languages under consideration.

Disambiguation requiring semantic processing has not been attempted. Any semantically n-way ambiguous sentence is assigned n interpretations. This is because no contextual or world knowledge has been included in the system. Thus, in the *referentially* ambiguous sentence *who bought pictures of his friends*, the pronoun *his* might be coindexed with *who* or it might be dependent on an entity in the external context. (Both interpretations are assigned by the system.) Similarly, for *structurally* ambiguous sentences like *I ate the banana on the table* (where *on the table* is either associated with *I* or with *banana*), all possible choices of prepositional attachment are included in the final analysis.

On the other hand, *lexical* ambiguity is usually resolved. For example, in the sentence *I can the beats*, the word *can* is either a verb or a noun; the correct category (verb) is established

[3]A similar change occurred with the development of $\overline{\text{X}}$ Theory. Before $\overline{\text{X}}$ Theory, there was no principle-based way of characterizing constituent orderings across languages. Thus, in early systems, each language was described by its own set of *ad hoc* rules. Now that $\overline{\text{X}}$ theory has been developed, current systems can uniformly describe constituent orderings of languages without recourse to *ad hoc* rules.

[4]See pp. 147-151.

(a) Unexpanded V-COMPLEMENT	(b) Expanded V-COMPLEMENT
[V-MAX [V think] [V-COMPLEMENT]]	[V-COMPLEMENT [C-MAX]]
[I-COMPLEMENT [V-MAX]]	[V-MAX [V think] [V-COMPLEMENT]]
[I-MAX [I-SPEC [N-MAX I]] [I E] [I-COMPLEMENT]]	[I-COMPLEMENT [V-MAX]]
[C-MAX [C E] [C-COMPLEMENT [I-MAX]]]	[I-MAX [I-SPEC [N-MAX I]] [I E] [I-COMPLEMENT]]
	[C-MAX [C E] [C-COMPLEMENT [I-MAX]]]

Figure 9.1: Expansion of a Verbal Complement in a Head-Initial Language

immediately upon encountering the word *the*.[5] In addition, *word choice* ambiguity is generally resolved. Recall that the choice of *conocer* or *saber* as the translation of *know* can be resolved simply by examining the θ-assigning properties of both possibilities (see section 7.1.2).

Turning to other weaknesses, the prediction of complements is based on the expansion of subcategorized elements corresponding to the *closest head* of the appropriate category (see section 6.2.1.1). In general, this procedure works correctly for head-initial languages, but does not always provide an accurate analysis for head-final languages. For example, suppose the parser is processing the following sentence:

(63) I think he eats chicken

The verb *think* takes a `C-MAX` complement. Thus, at the point when `V-COMPLEMENT` is to be expanded (see figure 9.1(a)), the closest head to the left in the input is *think* and the correct prediction is made (see figure 9.1(b)).

On the other hand, if English happened to be a head-final language, the equivalent sentence for (63) would be:

(64) I he chicken eats think

When `V-COMPLEMENT` is first encountered during parsing, it must be associated with the verb *think*. However, neither *eats* nor *think* have been scanned; thus, *eats* is chosen as the closest

[5]Lexical ambiguity is not resolved when there is no disambiguating lexical item in the input. Thus, the sentence *time flies like an arrow* would be assigned more than one parse structure.

head to the right and the wrong prediction (`N-MAX` instead of `C-MAX`) is made since *eat* takes `N-MAX` as its complement:

[V-COMPLEMENT [N-MAX]]
[V-MAX [V-COMPLEMENT] [V]]
[I-COMPLEMENT [V-MAX]]
[I-MAX [I-SPEC [N-MAX I]] [I-COMPLEMENT] [I]]
[C-MAX [C E] [C-COMPLEMENT [I-MAX]]]

The result is that *he* is erroneously taken as the complement of *eats*. However, as soon as the noun *chicken* is scanned, the parse fails because it is expecting a verb. The parser cannot recover from this error for two reasons: (1) it has no way of knowing what went wrong; and (2) it cannot back up and try to find a different head corresponding to `V-COMPLEMENT` once the complement structure has already been processed. In order to make the complement-prediction routine work correctly, the entire input needs to be searched until all possible heads are found; then several parses would be held in parallel until the bad ones are ruled out. However, this method of complement-prediction is undesirable. Not only does it require wasteful searching of the input, but it also performs extra computation in order to process structure that will eventually be eliminated.

Among the issues presented in the next section is a proposal for an alternative parsing method that resolves the complement prediction problem. Although the implementation does not use this approach, the present incarnation of the parser successfully predicts complements for the languages handled by the system. The uniform treatment of complements is at least a step in the right direction for allowing both head-initial and head-final structure to be parsed systematically.[6]

9.2 Directions for Future Work

The system presented here does not include interpretation of ill-formed parses. Thus, the sentence *who did you wonder whether went* is rejected during parsing even though many people understand this as meaning *for which x you wondered whether x went*.[7] The reason this interpretation is disallowed is that at the level of S-structure a linguistic constraint (Subjacency)

[6]The issue of head-complement order is not addressed in Sharp's translation system (1985), which only translates head-initial languages.

[7]In fact, the equivalent sentence in Spanish is well-formed, and is assigned this interpretation.

is applied, and the structure is eliminated. Note that if the Subjacency requirement were temporarily ignored, the sentence would be successfully parsed, and it could then be assigned an interpretation at the level of logical form. However, the system cannot choose to suppress a given linguistic constraint since it does not know the reason that a sentence might be ill-formed, nor does it know the degree to which the sentence is ill-formed.

An interesting question to be investigated is that of how a parser might diagnose problems underlying ill-formed sentences and determine the severity of the ill-formedness. For example, rather than failing on a parse because an antecedent of a trace is not found, a parser might be designed so that it knows there is an antecedent, but that the antecedent might be too far away for possible interpretation of the sentence. If the system includes LF processing, syntactically ill-formed structures could then be reanalyzed in an attempt to assign semantic interpretations.

Another possible extension of the system is the addition of a facility that allows the "current GB theory" to be tested for validity. The parameters associated with the principles are *not* hard-wired, but the principles themselves *are*. A user of the system has a small degree of leeway with respect to the constructs utilized by the linguistic component: the $\overline{\text{X}}$ templates may be changed to include bar-levels or n-ary branching; the basic categories are modifiable; and the choice of traces and empty categories may be specified. However, the definitions associated with the principles (*e.g.*, government, c-command, *etc.*) cannot be altered.

An approach that would serve as a testing ground for current linguistic theory has been addressed by Thiersch and Kolb (1986):

> We have undertaken to build a first experimental foundation for a parser which adheres as close as possible to these basic premises [strict modularity, projection from the lexicon, principle-based rather than rule-based, and parameterizability], while allowing enough flexibility so that it is not bound, for example, to a particular instantiation of GB theory, but can be used as a testing ground for various theoretical hypotheses in this general framework across a variety of languages.

Further discussion of this proposal is not included in the literature pertaining to the parser built by Thiersch and Kolb. However, one might imagine a method of allowing the GB definitions to be modified: a "meta-language" might be constructed so that definitions can be written from scratch by a user of the system. For example, *c-command* might be defined using the meta-language as follows:[8]

```
(DEFINITION C-COMMAND (X Y)
   (AND (NOT (CONTAIN X Y))
        (FORALL Z (IMPLIES (CONTAIN Z X) (CONTAIN Z Y)))))
```

The predicate `CONTAIN` is a primitive of the meta-language. Once the c-command definition is constructed, it could then be used to build other definitions associated with the theory. If the notion of c-command changes, the above definition can be modified, and the new definition can be tested.

One problem with the construction of such a meta-language is that the choice of primitives might not be a trivial task. For example, should `CONTAIN` also be modifiable by a user of the system? Most likely, definitions at the level of dominance and precedence relations would be chosen as primitives, and all definitions would be built on top of these primitives.

Another drawback in the present incarnation of the system is that the parser is based on the Earley algorithm; thus, context-free rules are still part of the system. Although subcategorization information is not multiplied out, the number of rules required to parse a sentence of a language might still be quite large. Furthermore, because the Earley algorithm is "eager," much unnecessary structure is built; thus, the system is subject to slowdowns due to processing of structure that might never be needed.

An alternative method of parsing is to use the lexicon as the driver for predicting structure rather than $\overline{\text{X}}$ templates. This approach has been taken by Kashket's Walpiri parser (1987), which uses "L-structures" (syntactic manifestations of lexical items) that are projected as $\overline{\text{X}}$ structure during parsing. Abney's parser (1987) also focuses on lexical items for the prediction of $\overline{\text{X}}$ structure; for example, a lexical head *licenses* the construction of its complements. (See Abney (1987) for a discussion of the notion of *licensing.*)

If the system were modified to parse in a more "bottom-up" manner, lexical items could be used for the prediction of $\overline{\text{X}}$ structure. This approach is currently being investigated. The tentative plan is the following: (1) the templates generated at precompilation time are "matched" against input items that constitute lexical heads; (2) maximal projections are linked with the lexical heads that subcategorize for them; (3) preterminal symbols (*i.e.*, non-basic categories)

[8]See section 3.3 for the definition of c-command.

are placed in adjunct or specifier positions according to those allowed by the templates; (4) empty elements and traces are put into unfilled internal and external argument positions; and (5) optional specifiers and unfilled complements of intransitive heads are eliminated. At the completion of each phrase, GB constraints (*e.g.*, Binding, Trace, Case, θ, and Bounding) are applied. Thus, structure-building operates in coordination with linguistic constraints, a design that is similar to the current version of the system except that there is no explicit notion of pushing, scanning, or popping. Instead, items are analyzed in a bottom-up fashion, and complements are linked to heads according to subcategorization information and phrase structure requirements of the language. Since this parsing approach is not top-down, the prediction of unnecessary structure is much more limited. Structure is built only if an element in the input *requires* the structure to be built, not just if the templates *allow* the structure to be built.

Note that this lexically driven method of parsing avoids the problem of complement-prediction found in top-down parsing. The routine that operates on the basis of information found in the *closest head* is eliminated, and erroneous complement-prediction is avoided. Furthermore, there would be more of a definite separation between the $\overline{\text{X}}$ and trace modules since empty elements (*e.g.*, PRO, *pro*, *etc.*) and traces are inserted into positions as required by lexical θ-assigners.

This method of parsing changes the clustering of principles: the $\overline{\text{X}}$ module is no longer used solely at precompilation time, and the trace module is not accessed at all during precompilation. All modules are used to check well-formedness conditions during processing of the input, and the lexicon is now the driver of the parser. Figure 9.2 shows the new design.

This approach moves toward the approaches taken by Abney (1987) and Kashket (1987) in that the GB modules are more distinct, and the $\overline{\text{X}}$ module is not the main driver of the parser.[9]

Because the translation system does not include a semantic analysis, the mapping between the source and target language is primitive. First, there is the problem of the underlying form mentioned in section 7.1.1. This form (which is actually D-structure) is not entirely interlingual: it includes some language-specific information (*e.g.*, constituent order); and it does not contain

[9]To a certain extent, both of these systems incorporate $\overline{\text{X}}$ into the parsing mechanism; however, the $\overline{\text{X}}$ machinery is not the primary controller of the parsing actions.

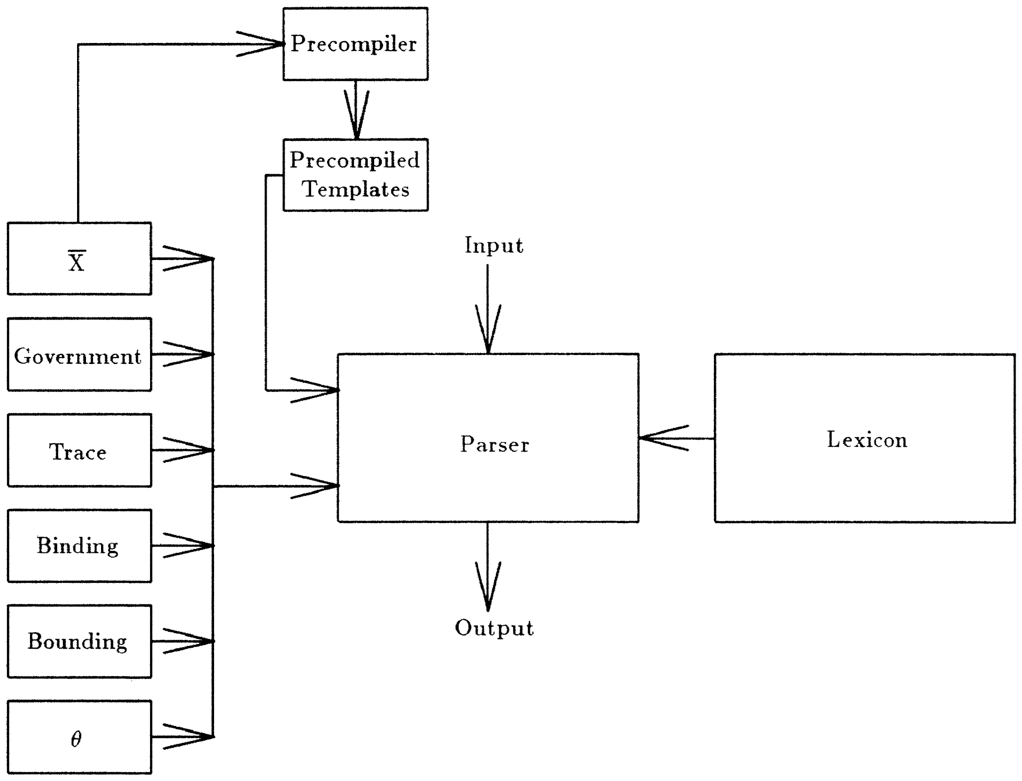


Figure 9.2: Lexically-Driven Parsing

a language-independent specification of how the syntactic constituents are interpreted. Second, there is the problem of generation, which is only handled at a surface level here. No information about context is utilized during the mapping of D-structure to S-structure.

The principles modeled here are relevant to syntax. Thus, the omission of semantic processing is not accidental. However, the inclusion of θ-theory in the model is the first step toward incorporation of a semantic component into the system. The θ-roles assigned to participants of an action might be used as an interface to the semantic component, which could assign an interpretation to the roles.

The system is intended to be the *core* of a more complex translation system that includes semantic interpretation and global contextual understanding. Investigation is currently underway to construct an interlingual form that does not include the language-specific details required by the D-structure form. Once this form is constructed, more sophisticated generation routines can be added to the system. These routines would access the interlingual form as well as contextual knowledge in order to generate the target language form.

9.3 Conclusions

This report has presented an implementation for interlingual translation based on GB theory. The approach taken uses a strategy that is principle-based, in contrast to recent strategies that are based on context-free language-specific rules. Translation is primarily syntactic. Semantic processing has not been attempted, although investigation of semantic issues is currently underway.

Few other GB parsers are currently in operation (see Wehrli (1984), Abney (1987), Thiersch and Kolb (1986), and Kashket (1987)), and only one work on GB translation has been published (see Sharp, (1985)). The system presented here differs from Sharp's system in that language-specific information (in particular, constituent order) is not stipulated. Furthermore, parsing is not based on hardwired rules, but on $\overline{\text{X}}$ templates that are instantiated according to the requirements of the language. Also, the system is designed to operate in a co-routine fashion between the structure-building module (which accesses the $\overline{\text{X}}$ templates) and the linguistic constraint module (which applies GB principles); this differs from Sharp's system, which interleaves the structure-building actions with the GB constraint application. Finally, several syntactic phenomena not handled by Sharp's system are handled by UNITRAN, including clitics, free inversion, and thematic divergence.

The linguistic principles are represented as routines that access user-modifiable parameters. Translation consists of a mapping the source language S-structure to a D-structure representation that is then mapped to the target language S-structure. The interlingual form (i.e., the D-structure representation) is not entirely language-independent; however, investigation of semantic-based interlingua is currently underway.

One of the primary goals of the principle-based approach is to allow translation to be performed without recourse to language-specific rules. A surprisingly small set of parameterized principles have been shown to handle most phenomena; only a handful of peripheral language peculiarities need to be specified. Thus, the high cost induced by grammar searching in alternative approaches has been eliminated. Perhaps most importantly, progress has been made toward the characterization of a universal grammar.

Appendix A

Principles and Parameters

This appendix summarizes the principles and parameters discussed in chapter 3. Section A.1 contains a table of the GB principles and their corresponding parameters, and section A.2 displays the parameter settings for Spanish and English.

A.1 Principles and Parameters of GB Modules

$\overline{X}$ Principles	*$\overline{X}$ Parameters*	*Level of Application*
A phrasal projection ($\overline{\overline{X}}$) has a head (X), a specifier and a complement	Constituent Order, Basic Categories, Pre-terminals, and Specifiers	DS
Left and right adjunction are optional and occur on the X^{max} or X^0 level	Choice of Adjuncts and their positions	DS
Complements of nonlexical heads (*e.g.*, C and I) are determined from default values	default value for nonlexical heads	DS
Specifiers may be optional	optional specifiers	DS
Complements of lexical heads are projected from argument structure information	Subcategorization Information (Lexicon)	DS

Government Principles	*Government Parameters*	*Level of Application*
α governs β if α is a governor and α minimally c-commands β	Choice of governors	SS, DS, LF

θ Principles	*θ Parameters*	*Level of Application*
[CL $+case_i$ $+\theta_j$] ... [NP $+case_i$] $\Rightarrow$ [CL $+case_i$ $+\theta_j$] ... [NP $+case_i$ $+\theta_j$] if language allows clitic doubling	Clitics, Clitic Doubling	SS, DS, LF
θ-Criterion	no parameter	SS, DS, LF
Visibility Condition	no parameter	SS, DS, LF
CSR(*semantic role*) = *syntactic category*	CSR Mapping	SS, DS, LF

Case Principles	*Case Parameters*	*Level of Application*
(A) Objective Case is assigned to object governed by transitive P	Choice of Government	SS
(B) Objective Case is assigned to object governed by transitive V	Choice of Government	SS
(C) Nominative Case is assigned to subject governed by [Infl +*tns*]	Choice of Government	SS
Case Filter (obviated by Visibility Condition)	no parameter	SS

Trace Principles	*Trace Parameters*	*Level of Application*
V, A, N, P, N_i and NP_i are proper governors for ECP (where EC = $[e]_i$)[1]	Pro-drop	SS
An empty category $[e]_i$ with [−V] feature must be properly governed (PROP is ungoverned)	Choice of Traces and ECP Chain Conditions	SS (and possibly LF)

Binding Principles	*Binding Parameters*	*Level of Application*
(A) An anaphor must be bound in its governing governing category	Choice of Governing Category	LF (if lexical) SS (if trace)
(B) A pronoun must be free in its governing category	Choice of Governing Category	LF
(C) An r-expression must be A-free (in the domain of its operator)	no parameter	LF (if lexical) SS (if trace)
V, A, N, P and [Infl +*tns*] are governors for Binding	no parameter	SS, LF
A variable must be $\bar{A}$-bound	no parameter	SS

Bounding Principles	*Bounding Parameters*	*Level of Application*
Move-α may cross at most one bounding node	Choice of Bounding Nodes	DS $\rightarrow$ SS

[1]Since AGR takes on nominal features when the pro-drop parameter is set, it too can serve as a proper governor in null subject languages.

A.2 Parameter Values for Spanish and English

$\overline{\overline{X}}$ Parameters	*$\overline{\overline{X}}$ Parameter Values*	
	Spanish	English
Constituent Order	spec-head-comp	spec-head-comp
Basic Categories	C, I, V, N, P, A	C, I, V, N, P, A
Pre-terminals	det, adv, wh-phrase, have-aux be-aux cl-acc, cl-dat, cl-ref	det, adv, wh-phrase, have-aux be-aux, do-aux
Specifiers	V: have-aux; N: det; A: adv; P: adv; I: $\overline{\overline{N}}$; C: $\overline{\overline{N}}$	V: have-aux, do-aux; N: det, $\overline{\overline{N}}$ A: adv; P: adv; I: $\overline{\overline{N}}$; C: $\overline{\overline{N}}$
Adjuncts[2]	$\overline{\overline{A}}$ to N: 3, 4 $\overline{\overline{P}}$ to N or V: 2, 4 adv to V: 1, 2 $\overline{\overline{C}}$ to N: 2, 4 be-aux to V: 3 I, have-aux, be-aux, or V to I: 1 clitic to V: 3, 4 $\overline{\overline{N}}$ to V: 2	$\overline{\overline{A}}$ to N: 3 $\overline{\overline{P}}$ to N or V: 2, 4 adv to V: 2 $\overline{\overline{C}}$ to N: 2, 4 be-aux to V: 3 I, have-aux, be-aux, or do-aux to I: 1
Default Value for Non-lexical Heads	I: $\overline{\overline{V}}$ complement C: $\overline{\overline{I}}$ complement	I: $\overline{\overline{V}}$ complement C: $\overline{\overline{I}}$ complement
Optional Specifiers	V, A, C, P, N	V, A, C, P, N
Subcategorization Information[3]	Spanish lexicon	English lexicon

Government Parameters	*Government Parameter Values*	
	Spanish	English
Choice of Governors	V, A, N, P, AGR	V, A, N, P, AGR

θ Parameters	*θ Parameter Values*	
	Spanish	English
Clitics	dative, accusative, reflexive	none
Clitic Doubling	yes	no
CSR mapping	GOAL $\rightarrow$ N, PATIENT $\rightarrow$ P, PROPOSITION $\rightarrow$ C, *etc.*	GOAL $\rightarrow$ N, PATIENT $\rightarrow$ N, PROPOSITION $\rightarrow$ C or N, *etc.*

[2]The numbers 1, 2, 3 and 4 correspond to the positioning of an adjunct: 1 = adjoin left to X^{max}; 2 = adjoin right to X^{max}; 3 = adjoin left to X^0; and 4 = adjoin right to X^0.

[3]The Spanish and English lexicons are in appendix C.

Case Parameters	*Case Parameter Values*	
	Spanish	English
Choice of Government	(A) c-government (B) s-government (C) c-government	(A) c-government (B) c-government (C) c-government

Trace Parameters	*Trace Parameter Values*	
	Spanish	English
Pro-drop	yes	no
Choice of Traces	$\overline{\overline{\text{N}}}$, $\overline{\overline{\text{P}}}$, be-aux, have-aux, I, V	$\overline{\overline{\text{N}}}$, $\overline{\overline{\text{P}}}$
ECP Chain Conditions	yes	no

Binding Parameters	*Binding Parameter Values*	
	Spanish	English
Governing Category	has a subject	has a subject

Bounding Parameters	*Bounding Parameter Values*	
	Spanish	English
Bounding Nodes	$\overline{\overline{\text{N}}}$ and $\overline{\overline{\text{C}}}$ (= $\overline{\text{S}}$)	$\overline{\overline{\text{N}}}$ and $\overline{\overline{\text{I}}}$ (= S)

Appendix B

Representation of Kimmo Automata

This appendix contains the representation of the Kimmo automata for English and Spanish (described in section 5.2.1).

B.1 English Automata

```
ALPHABET
    a b c d e f g h i j k l m n o p q r s t u v w x y z + . / ? %

    NULL 0
    ANY =
    SUBSET V a e i o u
    SUBSET V1 a e i o u y
    SUBSET C b c d f g h j k l m n p q r s t v w x z
    SUBSET S s x z
    END

"Surface Characters" 1 31
    a b c d e f g h i j k l m n o p q r s t u v w x y z . / ? = %
    a b c d e f g h i j k l m n o p q r s t u v w x y z . / ? = %
1:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

"Epenthesis" 6 8
    c h s S y + + =
    c h s S i e 0 =
1:  2 1 4 3 3 0 1 1
```

```
2:  2 3 3 3 3 0 1 1
3:  2 1 3 3 3 5 6 1
4:  2 3 3 3 3 5 6 1
5.  0 0 1 0 0 0 0 0
6:  1 1 0 1 1 1 1 1

"Gemination" * *
    V1 b  d  f  g  h  l  m  n  p  r  s  t  + + + + + + + + + + + + =
    V1 b  d  f  g  h  l  m  n  p  r  s  t  b d f g l m n p r s t 0 =
1:  4  1  1  1  1  1  1  1  1  1  1  1  1  0 0 0 0 0 0 0 0 0 0 0 1 1
2.  1  0  0  0  0  0  0  0  0  0  0  0  0  0 0 0 0 0 0 0 0 0 0 0 0 0
3:  0  0  0  0  0  0  0  0  0  0  0  1  0  0 0 0 0 0 0 0 0 0 0 0 0 0
4: 16  5  6  7  8 16  9 10 11 12 13 14 15  0 0 0 0 0 0 0 0 0 0 0 1 16
5: 16 16 16 16 16  1 16 16 16 16 16 16 16  2 0 0 0 0 0 0 0 0 0 0 3 1
6: 16 16 16 16 16  1 16 16 16 16 16 16 16  0 2 0 0 0 0 0 0 0 0 0 3 1
7: 16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 2 0 0 0 0 0 0 0 0 3 1
8: 16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 2 0 0 0 0 0 0 0 3 1
9: 16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 0 2 0 0 0 0 0 0 3 1
10:16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 0 0 2 0 0 0 0 0 3 1
11:16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 0 0 0 2 0 0 0 0 3 1
12:16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 0 0 0 0 2 0 0 0 3 1
13:16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 0 0 0 0 0 2 0 0 3 1
14:16 16 16 16 16  1 16 16 16 16 16 16 16  0 0 0 0 0 0 0 0 0 2 0 3 1
15:16 16 16 16 16 16 16 16 16 16 16 16 16  0 0 0 0 0 0 0 0 0 0 2 3 1
16:16 16 16 16 16 16 16 16 16 16 16 16 16  0 0 0 0 0 0 0 0 0 0 0 16 1

"Y-spelling" 6 7
   C y y + i a =
   C y i = i a =
1: 2 1 0 1 1 1 1
2: 2 5 3 1 1 1 1
3. 0 0 0 4 0 0 0
4. 1 1 0 1 0 0 1
5: 1 1 0 6 1 1 1
6. 0 0 0 0 1 1 0

"Elision" 17 9
   V i  e  e  +  g  c = n
   V i  e  0  0  g  c = n
1: 2 2  3  4  1 11 11 1 1
2: 1 1  5  6  1 11 11 1 1
3: 1 1  5  6  9 11 11 1 1
4. 0 0  0  0  8  0  0 0 0
5: 1 1  1  4  7 11 11 1 1
6. 0 0  0  0 10  0  0 0 0
7. 1 1  0  0  0  1  1 1 1
8. 1 1  1  0  0  0  0 0 0
9. 0 0  16 0  0  1  1 1 1
10.0 0  1  0  0  0  0 0 0
11:1 1 14 12  1  1  1 1 1
```

```
12.0 0   0   0 13   0   0 0 0
13.0 1   1   0  0   0   0 0 0
14:1 1   1   0 15  11  11 1 1
15:1 0   0   0  1  11  11 1 1
16.0 0   0   0  0   0   0 0 17
17:0 0   0   0  0   0   0 0 0

"I-Spelling" 7 6
   i e + i e =
   y 0 0 i e =
1: 2 1 1 5 1 1
2. 0 3 0 0 0 0
3. 0 0 4 0 0 0
4. 0 0 0 1 0 0
5: 1 1 1 1 6 1
6: 0 1 7 0 0 1
7. 0 0 0 0 0 1

    END
END
```

B.2 Spanish Automata

```
ALPHABET
    a b c d e f g h i j k l m n o p q r s t u v w x y z ^ + / ? .

    NULL 0
    ANY =
    SUBSET V a e i o u
    SUBSET C b c d f g h j k l m n p q r s t v w x y z
    SUBSET C2 b c d f g h j k l m n p q r s t v w x y
    END


"Surface Characters" 1 31
    a b c d e f g h i j k l m n o p q r s t u v w x y z ^ / ? . =
    a b c d e f g h i j k l m n o p q r s t u v w x y z ^ / ? . =
1:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

"Infinitive Removal" 11 15
    a  a  e  e  i  i  a  e  i  r  +  =  r  +  s
    a  =  e  =  i  =  0  0  0  0  0  =  r  =  s
1.  2  2  2  2  2  2  0  0  0  0  0  2  2  0  2
2:  6  9  6  9  6  9  3  3  3  0  2  2  2  2  2
3.  0  0  0  0  0  0  0  0  0  4  0  0  0  0  0
4.  0  0  0  0  0  0  0  0  0  0  5  0  0  5  0
5.  10 0  10 0  10 0  0  0  0  0  0  10 10 0  11
6:  6  0  6  0  6  0  3  3  3  0  2  2  7  2  2
7:  2  0  2  0  2  0  0  0  0  0  8  2  2  8  2
8.  11 0  11 0  11 0  0  0  0  0  0  11 11 0  10
9.  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0
10: 10 0  10 0  10 0  0  0  0  0  10 10 10 10 10
11. 11 0  11 0  11 0  0  0  0  0  11 11 11 11 11

"Present Subjunctive er-ir G-soften Add-g" 14 16
    e  n  r  g  e  o  a  s  m  +  +  +  =  a  i  i
    e  n  0  j  0  o  a  s  m  g  0  =  =  0  i  0
1:  2  1  1  3  1  1  1  1  1  0  1  1  1  1  2  1
2:  1  4  0  3  1  1  1  1  1  0  1  1  1  1  1  1
3.  0  0  0  0  5  0  0  0  0  0  0  0  0  0  0  5
4:  2  1  0  0  6  1  1  1  1  0  1  1  1  1  2  6
5.  0  0  7  0  0  0  0  0  0  0  0  0  0  0  0  0
6.  0  0  8  0  0  0  0  0  0  0  0  0  0  0  0  0
7.  0  0  0  0  0  0  0  0  0  0  9  0  0  0  0  0
8.  0  0  0  0  0  0  0  0  0  9  0  0  0  0  0  0
9.  0  0  0  0  0  10 11 0  0  0  0  0  0  0  0  0
10: 12 12 0  0  0  12 12 12 12 0  0  0  12 0  12 0
11: 12 10 0  0  0  12 12 10 13 0  0  0  12 0  12 0
12. 12 12 0  0  0  12 12 12 12 0  0  0  12 0  12 0
13. 0  0  0  0  0  14 0  0  0  0  0  0  0  0  0  0
```

```
14. 0  0  0  0  0  0  0  10 0  0  0  0  0  0  0  0

"Present Subjunctive er c/zc, c/z" 19 18
    c  e  n  r  e  o  a  s  m  +  +  +  =  V  =  c  r  i
    c  e  n  0  0  o  a  s  m  0  c  =  =  V  0  z  r  i
1:  1  2  3  1  1  2  2  1  1  1  0  1  1  2  1  0  3  2
2:  16 2  3  1  1  2  2  1  1  1  0  1  1  2  1  4  3  2
3:  16 2  3  1  1  2  2  1  1  1  0  1  1  2  1  5  3  2
4.  0  0  0  0  6  0  0  0  0  0  0  0  0  0  0  0  0  0
5.  0  0  0  0  7  0  0  0  0  0  0  0  0  0  0  0  0  0
6.  0  0  0  8  0  0  0  0  0  0  0  0  0  0  0  0  0  0
7.  0  0  0  9  0  0  0  0  0  0  0  0  0  0  0  0  0  0
8.  0  0  0  0  0  0  0  0  0  0  10 0  0  0  0  0  0  0
9.  0  0  0  0  0  0  0  0  0  10 0  0  0  0  0  0  0  0
10. 0  0  0  0  0  11 12 0  0  0  0  0  0  0  0  0  0  0
11: 15 15 15 0  0  15 15 15 15 0  0  0  15 15 0  0  15 15
12: 15 15 11 0  0  15 15 11 13 0  0  0  15 15 0  0  15 15
13. 0  0  0  0  0  14 0  0  0  0  0  0  0  0  0  0  0  0
14. 0  0  0  0  0  0  0  11 0  0  0  0  0  0  0  0  0  0
15. 15 15 15 0  0  15 15 15 15 0  0  0  15 15 0  0  15 15
16. 1  2  3  0  17 2  2  1  1  1  0  1  1  2  1  0  3  2
17. 0  0  0  18 0  0  0  0  0  0  0  0  0  0  0  0  0  0
18. 0  0  0  0  0  0  0  0  0  19 0  0  0  0  0  0  0  0
19. 15 1  15 0  0  15 15 15 15 0  0  0  15 15 0  0  15 1

"Present and Subjunctive ir gu/g, ui/uy" 23 19
    =  e  V  ^  C  =  u  +  i  u  g  r  i  +  o  a  n  s  m
    =  e  V  ^  C  0  u  =  y  0  g  0  0  0  o  a  n  s  m
1:  1  2  2  1  3  1  2  1  0  0  3  1  1  1  2  2  3  3  3
2:  1  2  2  2  3  1  2  1  0  0  4  1  1  1  2  2  3  3  3
3:  1  2  2  3  3  1  5  1  0  23 3  1  1  1  2  2  3  3  3
4.  1  2  2  3  3  1  2  1  0  6  3  1  1  1  2  2  3  3  3
5.  1  2  2  2  3  1  2  1  7  1  3  1  20 1  2  2  3  3  3
6.  0  0  0  0  0  0  0  0  0  0  0  0  8  0  0  0  0  0  0
7.  0  0  0  0  0  0  0  0  0  0  0  9  0  0  0  0  0  0  0
8.  0  0  0  0  0  0  0  0  0  0  0  10 0  0  0  0  0  0  0
9.  0  0  0  0  0  0  0  0  0  0  0  0  0  11 0  0  0  0  0
10. 0  0  0  0  0  0  0  0  0  0  0  0  0  12 0  0  0  0  0
11. 0  13 0  0  0  0  0  0  0  0  0  0  0  0  15 16 0  0  0
12. 0  0  0  0  0  0  0  0  0  0  0  0  0  0  15 16 0  0  0
13: 19 19 19 19 19 0  19 0  0  0  19 0  0  0  19 19 14 19 19
14: 19 19 19 19 19 0  19 0  0  0  19 0  0  0  19 19 19 19 19
15: 19 19 19 19 19 0  19 0  0  0  19 0  0  0  19 19 15 19 19
16: 19 19 19 19 19 0  19 0  0  0  19 0  0  0  19 19 19 15 17
17. 0  0  0  0  0  0  0  0  0  0  0  0  0  18 0  0  0  0  0
18. 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  15 0
19. 19 19 19 19 19 0  19 0  0  0  19 0  0  0  19 19 19 19 19
20. 0  0  0  0  0  0  0  0  0  0  0  21 0  0  0  0  0  0  0
21. 0  0  0  0  0  0  0  0  0  0  0  0  0  22 0  0  0  0  0
22. 22 22 22 22 22 0  22 0  0  0  22 0  0  0  22 22 22 22 22
```

```
23. 0  0  0  0  0  0  0  0  0  0  0  0  20 0  0  0  0  0  0

"Present and Subjunctive ar u-u^ i-i^" 10 16
    =  +  i  u  ^  a  r  +  o  a  e  n  s  +  =  C
    =  0  i  u  ^  0  0  ^  o  a  e  n  s  =  0  C
1:  1  1  2  2  1  1  1  0  1  1  1  1  1  1  1  1
2:  1  1  1  1  1  3  0  0  1  1  1  1  1  1  1  1
3.  0  0  0  0  0  0  4  0  0  0  0  0  0  0  0  0
4.  0  9  0  0  0  0  0  5  0  0  0  0  0  0  0  0
5.  0  0  0  0  0  0  0  0  6  6  6  0  0  0  0  0
6:  0  0  0  0  0  0  0  0  0  0  0  7  7  0  0  0
7:  8  0  8  8  8  0  0  0  8  8  8  8  8  0  0  8
8.  8  0  8  8  8  0  0  0  8  8  8  8  8  0  0  8
9.  0  0  0  0  0  0  0  0  10 10 10 0  0  0  0  0
10. 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1

"Preterit ar c-qu g-gu z-c" 21 15
    +  +  =  c  g  z  =  a  a  r  +  e  ^  z  c
    =  0  0  q  g  c  =  a  0  0  u  e  ^  z  c
1:  1  1  1  2  19 3  1  1  1  1  0  1  1  4  4
2.  0  0  0  0  0  0  0  0  6  0  0  0  0  0  0
3.  1  0  0  0  0  0  0  0  7  0  0  0  0  0  0
4:  1  1  1  0  1  0  1  1  5  1  0  1  1  1  1
5.  0  0  0  0  0  0  0  0  0  8  0  0  0  0  0
6.  0  0  0  0  0  0  0  0  0  9  0  0  0  0  0
7.  0  0  0  0  0  0  0  0  0  10 0  0  0  0  0
8.  11 11 0  0  0  0  0  0  0  0  0  0  0  0  0
9.  0  0  0  0  0  0  0  0  0  0  12 0  0  0  0
10. 0  12 0  0  0  0  0  0  0  0  0  0  0  0  0
11. 0  0  0  0  16 0  16 16 0  0  0  17 0  16 16
12. 0  0  0  0  0  0  0  0  0  0  0  13 0  0  0
13. 0  0  0  0  0  0  0  0  0  0  0  0  14 0  0
14: 0  0  0  0  15 0  15 15 0  0  0  15 15 15 15
15. 0  0  0  0  15 0  15 15 0  0  0  15 15 15 15
16: 0  0  0  0  16 0  16 16 0  0  0  16 16 16 16
17: 1  1  1  0  1  0  1  0  0  0  0  1  18 1  1
18. 1  1  1  0  1  0  1  0  0  1  0  1  1  1  1
19: 0  0  0  0  0  0  0  0  20 0  0  0  0  0  0
20. 0  0  0  0  0  0  0  0  0  21 0  0  0  0  0
21. 0  0  0  0  0  0  0  0  0  0  12 0  0  0  0

"Pluralize" 8 9
    =  C2 z  +  s  =  +  +  z
    =  C2 c  e  s  0  =  0  z
1:  1  2  5  0  2  1  1  1  8
2:  1  1  5  3  1  1  1  6  1
3.  0  0  0  0  4  0  0  0  0
4:  4  4  0  0  4  4  4  4  4
5.  0  0  0  3  0  0  0  0  0
6.  1  2  0  0  7  0  0  0  8
```

```
7.  1  2  0  3  2  1  1  1  8
8:  1  2  0  6  2  1  1  6  8

"Remove Accent" 8 7
    V  C  ^  ^  =  +  =
    V  C  0  =  =  =  0
1:  2  1  0  1  1  1  1
2:  2  1  5  3  1  1  1
3:  2  3  0  0  1  4  1
4.  2  4  0  0  1  0  0
5.  0  6  0  0  0  0  0
6.  0  6  0  0  0  7  0
7:  8  7  0  0  0  0  0
8.  8  8  0  8  8  8  8

"Add Accent" 8 6
    V  C  +  =  0  =
    V  C  =  =  ^  0
1:  1  2  1  1  5  1
2:  1  2  3  1  0  1
3.  4  3  3  3  0  3
4:  1  2  3  4  0  1
5:  6  5  0  0  0  0
6.  0  6  7  0  0  0
7:  8  7  0  0  0  8
8.  8  8  0  0  0  0

    END
END
```

Appendix C

Representation of Lexicons

This appendix contains the representation of the Kimmo lexicons for English and Spanish (described in section 5.2.2).

C.1 English Lexicon

```
/* ********************************************************************** */
/*                                                                        */
/*                         LEXICON: English                               */
/*                                                                        */
/* ********************************************************************** */

ALTERNATIONS
        ( Root = Root )
        ( /N = N )
        ( /PROP = PROP )
        ( /IN = C1 )
        ( /MN = MN )
        ( /A = PA CA CS LY )
        ( /V = P12 P3S P3P PS PPD PPN PR I AG AB )
        ( /IV1 = PR I AG AB )                    /* pres, perf, past irregular */
        ( /IV2 = P12 P3S P3P PR I AG AB )        /* perf, past irregular */
        ( /IV3 = P12 P3S P3P PS PR I AG AB )     /* perf irregular */
        ( /IP3S = IP3S )
        ( /IP123 = P3P P12 )
        ( /IPS = IPS )
        ( /IPP = IPP )
        ( /PPN = PPN )
        ( /MODAL-PAST = MODAL-PAST )
        ( /MODAL-PRES = MODAL-PRES )
        ( /WH-PHRASE-A = WH-PHRASE-A )
        ( /C-FIN = C-FIN )
        ( /C-INF = C-INF )
        ( /PREP = PREP )
        ( /DET = DET )
        ( C1 = C1 )
        ( C2 = C2 )
        ( # = )
END

LEXICON N       0 C1 "n p3 sg"; +s C2 "n p3 pl"; +y /A "A"
LEXICON PROP    0 C1 "n proper p3 sg"; +s C2 "n proper p3 pl"
LEXICON MN      0 C2 "mass n"
LEXICON C1      0 # ""; %s # "poss"
LEXICON C2      0 # ""; % # "poss"

/* regular pres 1st, 2nd, 3rd person singular and plural */
LEXICON P3S     +s # "v pres p3 sg"
LEXICON P3P     0 # "v pres p3 pl"
LEXICON P12     0 # "v pres p1 p2 sg pl"

/* irregular 3rd person singular */
LEXICON IP3S    0 # "v pres p3 sg"

/* regular and irregular past tense */
LEXICON PS      +ed # "v past p1 p2 p3 sg pl"
LEXICON IPS     0 # "v past p1 p2 p3 sg pl"

/* regular perfective ED, EN */
LEXICON PPD     +ed # "v perf"
LEXICON PPN     +en # "v perf"

/* irregular perfective 0 */
LEXICON IPP     0 # "v perf"

/* progressive */
LEXICON PR      +ing # "v prog"

/* infinitive */
LEXICON I       0 # "v inf"

/* agent (v+er), adjective (0), comparative (a+er), */
/* superlative (a+est), able (v+able)               */
LEXICON AG      +er /N "agent neut"
LEXICON PA      0 # "a neut sg pl"
LEXICON CA      +er # "comparative a neut sg pl"
LEXICON CS      +est # "superlative a neut sg pl"
LEXICON LY      +ly /A "adv"
LEXICON AB      +able # "a condition v-able neut sg pl"
LEXICON DET
 0 # "det neut"

LEXICON MODAL-PAST
 0 # "i past modal (subcat (v)) p1 p2 p3 sg pl"

LEXICON MODAL-PRES
 0 # "i pres modal (subcat (v)) p1 p2 p3 sg pl"

LEXICON PREP
 0 # "p (subcat (n))"

LEXICON WH-PHRASE-A
 0 # "n wh-phrase-a neut p3 pl sg wh"

LEXICON C-FIN
 0 # "c (subcat (i))"

LEXICON C-INF
 0 # "c (subcat (i inf))"

LEXICON Root
 /* DETERMINERS */
 the /DET "(spanish ((el))) sg";
 that /DET "(spanish ((ese))) sg";
 those /DET "(spanish ((esos))) pl";
 this /DET "(spanish ((este))) sg";
 these /DET "(spanish ((estos))) pl";

 /* QUANTIFYING (DETERMINERS) */
 many /DET "(spanish ((mucho))) pl";
 no /DET "(spanish ((ningu^n))) sg";
 another /DET "(spanish ((otro))) sg";
 other /DET "(spanish ((otro))) pl";
 what /DET "(spanish ((que^))) sg pl wh";
 a  /DET "(spanish ((un))) sg";
 an /DET "(spanish ((un))) sg";
 several /DET "(spanish ((varios))) pl";
 some /DET "(spanish ((algu^n))) sg pl";
 each /DET "(spanish ((cada))) sg";
 all /DET "(spanish ((todo))) pl";
 which /DET "(spanish ((cua^l))) sg pl wh";
 how_much /DET "(spanish ((cua^nto))) sg wh";
 how_many /DET "(spanish ((cua^nto))) pl wh";

 /* POSSESSIVE (DETERMINERS) */
 my /DET "(spanish ((mi))) poss pronoun p1 sg pl";
 your /DET "(spanish ((tu) (su))) poss pronoun p2 sg pl";
 his # "(spanish ((su))) poss masc pronoun p3 sg pl det";
 her # "(spanish ((su))) poss fem pronoun p3 sg pl det";
```

```
its /DET "(spanish ((su))) poss pronoun p3 sg pl";
our /DET "(spanish ((nuestro))) poss pronoun p1 sg pl";
their /DET "(spanish ((su))) poss pronoun p3 sg pl";

/* COMPS */
that /C-FIN "(spanish ((que)))";
so_that /C-FIN "(spanish ((para que) (a fin de que) (de manera que)
   (de modo que)))";
unless /C-FIN "(spanish ((a menos que)))";
before /C-FIN "(spanish ((antes de que) (antes que)))";
provided_that /C-FIN "(spanish ((con tal que) (en caso de que)))";
without /C-FIN "(spanish ((sin que)))";
although /C-FIN "(spanish ((aunque)))";
when /C-FIN "(spanish ((cuando)))";
after /C-FIN "(spanish ((despue^s que) (despue^s de que)))";
as_soon_as /C-FIN "(spanish ((luego que) (tan pronto como) (asi^ que)
   (en cuanto)))";
until /C-FIN "(spanish ((hasta que)))";
while /C-FIN "(spanish ((mientras)))";
that /C-FIN "(spanish ((que)))";
for /C-INF "(spanish ((para)))";

/* INFINITIVE MARKER */
to # "(spanish ((a))) inf i (subcat (v))";
to # "(spanish ()) inf i (subcat (v))";

/* VERBS */
have /IV1 "(spanish ((haber))) have-aux (root have)";
have /IV1 "(spanish ((tener))) (external (agent animate))
   (external (agent)) (subcat (n)) (root have)";
has /IP3S "(root have)";
have /IP123 "(root have)";
had /IPS "(root have)";
had /IPP "(root have)";

be /IV1  "(spanish ((ser) (estar))) be-aux (root be)";
be /IV1  "(spanish ((ser) (estar))) (root be) intrans (external (agent))
    (subcat (entity)) (subcat (p)) (subcat (a)) (subcat (p-location))";
be /PPN  "(root be)";
am # "v pres sg p1 (root be)";
are # "v pres sg pl p2 (root be)";
is /IP3S "(root be)";
are # "v pres pl p1 p3 (root be)";
was # "v past p1 p3 sg (root be)";
were # "v past p1 p3 pl (root be)";
were # "v past p2 sg pl (root be)";

believe /V "(spanish ((creer))) (subcat (goal)) (external (agent animate))
   (subcat (p-goal)) (subcat (goal) (proposition)) intrans sdel (root believe)
   (noun-form (belief /N) (subcat (p-goal)) (subcat (proposition)))";
```

```
buy /V "(spanish ((comprar))) (subcat (goal)) (root buy)
   (external (agent animate)) (noun-form (purchase /N) (subcat (p-goal)))";
bought /IPP "(root buy)";

come /IV2 "(spanish ((venir))) (root come) intrans
   (subcat (p-location)) (external (agent))";
came /IPP "(root come)";
came /IPS "(root come)";
conquer /V "(spanish ((vencer))) (root conquer) intrans (subcat (goal))
   (external (agent animate))";

continue /V "(spanish ((continuar))) (root continue) intrans (subcat (goal))
   (external (agent))";

contribute /V "(spanish ((contribuir))) (root contribute) (subcat (goal))
   (external (agent animate))  (subcat (goal) (p-patient))";

cost /IV2 "(spanish ((costar))) (root cost) intrans (subcat (quantity))
   (external (agent)) (noun-form (cost /N) (subcat (p-quantity)))";
cost /IPP "(root cost)";
cost /IPS "(root cost)";

do /IV1 "(spanish ((hacer))) (root do) (subcat (goal)) (external (agent))";
do /IP123 "(root do)";
does /IP3S "(root do)";
did /IPS "(root do)";
done /IPP "(root do)";

die /V "(spanish ((morir))) (root die) (subcat (cause)) intrans
   (noun-form (death /N) (subcat (p-goal animate))) (external (agent animate))
   (adj-form (dead /A) (subcat (cause)))";

eat /IV2 "(spanish ((comer))) intrans (subcat (goal))";
ate /IPS "(root eat)";
eaten /IPP "(root eat)";

expect /V "(spanish ((suponer))) (root expect) (subcat (proposition))
   (external (agent animate)) (subcat (goal) (proposition))";

flee /V "(spanish ((huir))) (root flee) intrans (subcat (p-location))
   (subcat (location)) (external (agent animate))
   (noun-form (flight /N) (subcat (p-location)))";
fled /IPS "(root flee)"; fled /IPP "(root flee)";

give /V "(spanish ((dar))) (root give) (subcat (patient) (goal))
   (external (agent animate)) (subcat (goal) (p-patient)) (subcat (goal))";
gave /IPS "(root give)";

go /IV1 "(spanish ((ir))) intrans (subcat (proposition)) (subcat (p-location))
```

```
   (root go) (external (agent))";
go /IP123 "(root go)";
goes /IP3S "(root go)";
gone /IPP "(root go)";
went /IPS "(root go)";

hate /IV3 "(spanish ((odiar))) (subcat (goal)) (external (agent animate))";
hated /IPP "(root hate)";

kick /V "(spanish ((patear))) (subcat (goal)) (external (agent animate))
   (expression ((the bucket) (spanish ((morir) (estirar la pata)))))";

know /IV2 "(spanish ((saber) (conocer))) intrans (root know)
   (subcat (goal)) (subcat (fact inanimate)) (subcat (proposition))
   (subcat (goal) (proposition))
   (external (agent animate)) sdel (noun-form (knowledge /MN)
   (subcat (p-fact)) (subcat (p-proposition)))";
knew /IPP "(root know)"; known /IPS "(root know)";

leave /IV2 "(spanish ((salir))) (root leave)_intrans  (subcat (p-location))
   (external (agent))";
left /IPP "(root leave)";
left /IPS "(root leave)";

look /V "(spanish ((buscar))) (root look) (subcat (p-goal))
   (external (agent animate))";

like /V "(spanish ((gustar))) (root like) (subcat (goal))
   (external (agent animate))";

make /IV2 "(spanish ((hacer))) (subcat (goal)) (root make)
   (external (agent animate))";
made /IPP "(root make)";
made /IPS "(root make)";

put /IV2 "(spanish ((poner))) (subcat (goal) (p-location)) (root put)
   (external (agent animate))";
put /IPP "(root put)";
put /IPS "(root put)";

read /IV2 "(spanish ((leer))) (external (agent animate))
    (subcat (goal inanimate))"
read /IPP "(root read)";
read /IPS "(root read)";

say /IV2 "(spanish ((decir))) (subcat (proposition)) (root say)
   (external (agent animate))";
said /IPP "(root say)";
said /IPS "(root say)";
```

```
tell /IV2 "(spanish ((decir))) (subcat (goal animate) (proposition))
   (external (agent animate)) (subcat (goal animate) (fact))
   (subcat (goal animate)) (root say)";
told /IPP "(root tell)";
told /IPS "(root tell)";

see /IV3 "(spanish ((ver))) intrans (subcat (goal)) (root see)
   (external (agent animate))";
saw /IPS "(root see)";
seen /IPP "(root see)";

seem /V "(spanish ((parecer))) (subcat (proposition)) (root seem)";

sell /IV2 "(spanish ((vender))) (subcat (goal)) (root sell)
   (external (agent animate)) intrans (noun-form (sale /N)
   (subcat (p-goal)))";
sold /IPP "(root sell)";
sold /IPS "(root sell)";

sleep /IV2 "(spanish ((dormir))) intrans (root sleep)
   (external (agent animate))";
slept /IPP "(root sleep)";
slept /IPS "(root sleep)";

speak /IV2 "(spanish ((hablar))) (subcat (p-goal)) (subcat (p-patient))
   (external (agent animate)) intrans (root speak)";
spoke /IPS "(root speak)";
spoken /IPP "(root speak)";

stretch /V "(spanish ((estirar))) (subcat (n)) intrans (root stretch)
   (external (agent))";

think /IV3 "(spanish ((pensar))) (subcat (proposition)) (subcat (p-goal))
   (external (agent animate)) intrans (root think)";
thought /IPS "(root think)";
thought /IPP "(root think)";

visit /IV3 "(spanish ((visitar))) (external (agent animate))
   (subcat (p-goal animate)) (subcat (goal inanimate))";
visited /IPP "(root visit)";

want /V "(spanish ((querer))) (subcat (goal) (proposition)) (subcat (goal))
   (external (agent animate)) sdel (root want)";

write /IV2 "(spanish ((escribir))) (subcat (goal)) intrans (root write)
   (external (agent animate))";
written /IPP "(root write)";
wrote /IPS "(root write)";

/* PRONOMINALS */
```

```
he # "(spanish ((e^l))) n pronoun p3 sg masc nom animate";
him # "(spanish ((e^l))) n pronoun p3 sg masc obj animate";
she # "(spanish ((ella))) n pronoun p3 sg fem nom animate";
her # "(spanish ((ella))) n pronoun p3 sg fem obj animate";
they # "(spanish ((e^l))) n pronoun p3 pl neut nom animate";
them # "(spanish ((e^l))) n pronoun p3 pl neut obj animate";
we # "(spanish ((nosotros))) n pronoun p1 pl neut nom animate";
us # "(spanish ((nosotros))) n pronoun p1 pl neut obj animate";
you # "(spanish ((tu^) (usted))) n pronoun p2 sg neut nom obj animate";
i # "(spanish ((yo))) n pronoun p1 sg neut nom animate";
me # "(spanish ((mi^))) n pronoun p1 sg neut obj animate";
it # "(spanish ()) n pronoun p3 sg nom neut obj pleonastic";
this # "(spanish ((e^ste))) n pronoun p3 sg neut nom obj animate";
these # "(spanish ((e^ste))) n pronoun p3 pl neut nom obj animate";
that # "(spanish ((e^se))) n pronoun p3 sg neut nom obj animate";
those # "(spanish ((e^se))) n pronoun p3 pl neut nom obj animate";

/* REFLEXIVE */
myself # "(spanish ((se))) n anaphor p1 sg neut obj animate";
yourself # "(spanish ((se))) n anaphor p2 sg neut obj animate";
himself # "(spanish ((se))) n anaphor p3 sg masc obj animate";
herself # "(spanish ((se))) n anaphor p3 sg fem obj animate";
themselves # "(spanish ((se))) n anaphor p3 pl neut obj animate";
yourselves # "(spanish ((se))) n anaphor p2 pl neut obj animate";


/* WH-PHRASES */
what /WH-PHRASE-A "(spanish ((que^)))";
who /WH-PHRASE-A "(spanish ((quie^n))) animate";
whom /WH-PHRASE-A "(spanish ((quie^n))) animate";
which /WH-PHRASE-A "(spanish ((cua^l)))";
when # "(spanish ((cua^ndo))) n wh adv";
where # "(spanish ((do^nde))) n wh adv";
how # "(spanish ((co^mo))) n wh adv";
how_much # "(spanish ((cua^nto))) n wh adv";
how_many # "(spanish ((cua^nto))) n wh adv";

/* MODALS */
can /MODAL-PRES "(spanish ((poder))) (root can)";
could /MODAL-PAST "(root can)";
should /MODAL-PRES "(spanish ((deber))) (root should)";
should /MODAL-PAST "(spanish ((deber))) (root should)";
will # "(spanish ()) i modal fut (subcat (v)) (root will) p1 p2 p3 sg pl";
would # "(spanish ()) i modal cond (subcat (v)) (root would) p1 p2 p3 sg pl";

/* DO-AUX */
do /IV1 "(spanish ()) (root do) do-aux";

/* PREPOSITIONS */
about /PREP "(spanish ((acerca de)))";
```

```
above /PREP "(spanish ((encima de) (sobre)))";
after /PREP "(spanish ((despues de)))";
against /PREP "(spanish ((contra)))";
among /PREP "(spanish ((entre)))";
around /PREP "(spanish ((alrededor de)))";
at /PREP "(spanish ((a)))";
before /PREP "(spanish ((antes de)))";
behind /PREP "(spanish ((detras de)))";
below /PREP "(spanish ((debajo de)))";
beneath /PREP "(spanish ((bajo) (debajo de)))";
beside /PREP "(spanish ((al lado de)))";
between /PREP "(spanish ((entre)))";
by /PREP "(spanish ((por)))";
during /PREP "(spanish ((durante) (por)))";
for /PREP "(spanish ((por) (para)))";
from /PREP "(spanish ((de) (of)))";
in /PREP "(spanish ((en)))";
inside /PREP "(spanish ((dentro de)))";
into /PREP "(spanish ((dentro)))";
near /PREP "(spanish ((cerca de)))";
of /PREP "(spanish ((de)))";
on /PREP "(spanish ((en) (sobre)))";
outside /PREP "(spanish ((fuera de)))";
over /PREP "(spanish ((sobre)))";
per /PREP "(spanish ((por)))";
through /PREP "(spanish ((por)))";
to /PREP "(spanish ((a)))";
toward /PREP "(spanish ((hacia)))";
towards /PREP "(spanish ((hacia)))";
under /PREP "(spanish ((debajo de)))";
with /PREP "(spanish ((con)))";
without /PREP "(spanish ((sin)))";

/* NOUNS */
bed /N "(spanish ((cam))) neut";
book /N "(spanish ((libr))) neut";
boy /N "(spanish ((muchach))) masc animate";
brother /N "(spanish ((herman))) masc animate";
bucket /N "(spanish ((cubet))) neut";
father /N "(spanish ((padre))) masc animate";
film /N "(spanish ((peli^cul))) neut";
cat /N "(spanish ((gat))) neut";
firm /N "(spanish ((empres))) neut";
floor /N "(spanish ((pis))) neut";
friend /N "(spanish ((amig))) neut animate";
girl /N "(spanish ((muchach))) fem animate";
house /N "(spanish ((casa))) neut";
man # "(spanish ((hombre))) masc sg n animate";
meeting /N "(spanish ((reunio^n))) neut";
men # "(spanish ((hombr))) pl masc n animate";
```

```
mother /N "(spanish ((madre))) fem animate";
movie /N "(spanish ((peli^cul))) neut";
pencil /N "(spanish ((lapiz))) neut";
paw /N "(spanish ((pat))) neut";
side /N "(spanish ((lad))) neut";
sister /N "(spanish ((herman))) f animate";
table /N "(spanish ((mes))) neut";
thing /N "(spanish ((cos))) neut";
time /N "(spanish ((tiempo))) neut";
truth /N "(spanish ((verdad))) neut";
water /MN "(spanish ((aqua))) neut";
woman # "(spanish ((mujer))) fem sg n animate";
women # "(spanish ((mujer))) pl fem n animate";

/* PROPER NOUNS */
spain /PROP "(spanish ((espan~a))) sg neut";
john /PROP "(spanish ((juan))) sg masc animate";
mary /PROP "(spanish ((mari^a))) sg fem animate";

/* ADJECTIVES */
certain /A "(spanish ((ciert))) sdel (subcat (proposition))";
likely /A "(spanish ((probabl))) sdel (subcat (proposition))";
possible /A "(spanish ((posibl))) sdel (subcat (proposition))";
probable /A "(spanish ((probabl))) sdel (subcat (proposition))";

tired /A "(spanish ((cansad)))";
proud /A "(spanish ((orgullos))) (subcat (p-goal)) (noun-form (pride /N)
   (subcat (p-goal)))";
eager /A "(spanish ((ansios))) (subcat (c))";
satisfied /A "(spanish ((satisfech))) (subcat (p-goal))";
/. # "(spanish ((/.))) punc";
/? # "(spanish ((/?))) punc"

END
```

C.2 Spanish Lexicon

```
/* ******************************************************************** */
/*                                                                      */
/*                          LEXICON: Spanish                            */
/*                                                                      */
/* ******************************************************************** */

ALTERNATIONS
  (Root = Root)
  (/AR-V = INF AR-PRES AR-SUBJ-PRES AR-SUBJ-PAST
        AR-FUT-COND AR-PRET AR-IMP AR-PROG PERF)
  (/ER-V = INF ER-IR-PRES ER-IR-SUBJ-PRES ER-IR-SUBJ-PAST
        ER-FUT-COND ER-IR-PRET ER-IR-IMP ER-IR-PROG PERF)
  (/IR-V = INF ER-IR-PRES ER-IR-SUBJ-PRES ER-IR-SUBJ-PAST
        IR-FUT-COND ER-IR-PRET ER-IR-IMP ER-IR-PROG PERF)
  (/C-INF = C-INF)
  (/C-FIN = C-FIN)
  (/PREP = PREP )
  (/AR-PRES = AR-PRES)
  (/ER-IR-PRES = ER-IR-PRES)
  (/AR-PRET = AR-PRET)
  (/ER-IR-PRET = ER-IR-PRET)
  (/DET = DET)
  (/WH-PHRASE-A = WH-PHRASE-A)
  (/PRES-IRREG = PRES-IRREG)
  (/AR-PROG = AR-PROG)
  (/ER-IR-PROG = ER-IR-PROG)
  (/AR-MODAL = AR-MODAL)
  (/ER-MODAL = ER-MODAL)
  (/IR-MODAL = IR-MODAL)
  (/PRET-IRREG-1 = PRET-IRREG-1)
  (/PRET-IRREG-2 = PRET-IRREG-2)
  (/PRET-IRREG-PROG = PRET-IRREG-1 PROG-IRREG-1)
  (/PRET-IRREG-SUBJ-1 = PRET-IRREG-1 ER-IR-SUBJ-PAST)
  (/PRET-IRREG-SUBJ-2 = PRET-IRREG-1 SUBJ-IRREG-2)
  (/PRET-IRREG-PROG-SUBJ-1 = PRET-IRREG-1 ER-IR-SUBJ-PAST PROG-IRREG-1)
  (/PRET-IRREG-PROG-SUBJ-2 = PRET-IRREG-2 PROG-IRREG-2 SUBJ-IRREG-1)
  (/PRET-IRREG-PROG-SUBJ-3 = PRET-IRREG-3 PROG-IRREG-2 SUBJ-IRREG-1)
  (/N = N)
  (/PROP = PROP)
  (/PRO = PRO)
  (/A = A LY)
  (INF = INF)
  (/ER-IR-IMP = ER-IR-IMP)
  (/AR-FUT-COND = AR-FUT-COND)
  (/ER-FUT-COND = ER-FUT-COND)
  (/IR-FUT-COND = IR-FUT-COND)
  (/IRREG-FUT-COND = IRREG-FUT-COND)
  (/ER-IR-SUBJ-PAST = ER-IR-SUBJ-PAST)
  (/ER-IR-SUBJ-PRES = ER-IR-SUBJ-PRES)
  (IRREG-FUT = IRREG-FUT)
  (IRREG-COND = IRREG-COND)
  (AR-FUT = AR-FUT)
  (ER-FUT = ER-FUT)
  (IR-FUT = IR-FUT)
  (AR-COND = AR-COND)
  (ER-COND = ER-COND)
  (IR-COND = IR-COND)
  (# =)
END

LEXICON INF
 +ar # "inf";
 +er # "inf";
 +ir # "inf"

LEXICON C-FIN
 0 # "c (subcat (i))"

LEXICON C-INF
 0 # "c (subcat (i inf))"

LEXICON PREP
 0 # "p (subcat (n))"

LEXICON PERF
 +ado # "perf";
 +ido # "perf"

LEXICON AR-PROG
 +ando # "prog"

LEXICON ER-IR-PROG
 +iendo # "prog"

LEXICON PRES-IRREG
 +s # "pres p2 sg";
 0 # "pres p3 sg";
 +mos # "pres p1 pl";
 +n # "pres p3 pl"

LEXICON AR-PRES
 +o # "pres p1 sg";
 +as # "pres p2 sg";
 +a # "pres p3 sg";
 +amos # "pres p1 pl";
 +an # "pres p3 pl"
```

```
LEXICON ER-IR-PRES
 +o # "pres p1 sg";
 +es # "pres p2 sg";
 +e # "pres p3 sg";
 +emos # "pres p1 pl";
 +en # "pres p3 pl"

LEXICON ER-IR-SUBJ-PRES
 +as # "pres-subj p2 sg";
 +a # "pres-subj p1 p3 sg";
 +amos # "pres-subj p1 pl";
 +an # "pres-subj p3 pl"

LEXICON AR-SUBJ-PRES
 +es # "pres-subj p2 sg";
 +e # "pres-subj p1 p3 sg";
 +emos # "pres-subj p1 pl";
 +en # "pres-subj p3 pl"

LEXICON AR-IMP
 +abas # "past p2 sg";
 +aba # "past p1 p3 sg";
 +a^bamos # "past p1 pl";
 +aban # "past p3 pl"

LEXICON ER-IR-IMP
 +i^as # "past p2 sg";
 +i^a # "past p1 p3 sg";
 +i^amos # "past p1 pl";
 +i^an # "past p3 pl"

LEXICON AR-PRET
 +e^ # "past p1 sg";
 +aste # "past p2 sg";
 +o^ # "past p3 pl";
 +amos # "past p1 pl";
 +aron # "past p3 pl"

LEXICON ER-IR-PRET
 +i^ # "past p1 sg";
 +iste # "past p2 sg";
 +io^ # "past p3 pl";
 +imos # "past p1 pl";
 +ieron # "past p3 pl"

LEXICON AR-SUBJ-PAST
 +ara # "past-subj p1 p3 sg";
 +aras # "past-subj p2 sg";
 +a^ramos # "past-subj p2 pl";
 +aran # "past-subj p3 pl"
```

```
LEXICON ER-IR-SUBJ-PAST
 +iera # "past-subj p1 p3 sg";
 +ieras # "past-subj p2 sg";
 +ie^ramos # "past-subj p2 pl";
 +ieran # "past-subj p3 pl"

LEXICON PRET-IRREG-1
 +e # "past p1 sg";
 +iste # "past p2 sg";
 +o # "past p3 sg";
 +imos # "past p1 pl";
 +ieron # "past p3 pl"

LEXICON PRET-IRREG-2
 +i^ # "past p1 sg";
 +i^ste # "past p2 sg";
 +yo^ # "past p3 sg";
 +i^mos # "past p1 pl";
 +yeron # "past p3 pl"

LEXICON PRET-IRREG-3
 +i^ # "past p1 sg";
 +iste # "past p2 sg";
 +yo^ # "past p3 sg";
 +imos # "past p1 pl";
 +yeron # "past p3 pl"

LEXICON PROG-IRREG-1
 +iendo # "prog"

LEXICON PROG-IRREG-2
 +yendo # "prog"

LEXICON SUBJ-IRREG-1
 +yera # "past-subj p1 p3 sg";
 +yeras # "past-subj p2 sg";
 +ye^ramos # "past-subj p2 pl";
 +yeran # "past-subj p3 pl"

LEXICON SUBJ-IRREG-2
 +era # "past-subj p1 p3 sg";
 +eras # "past-subj p2 sg";
 +e^ramos # "past-subj p2 pl";
 +eran # "past-subj p3 pl"

LEXICON AR-FUT-COND
 0 AR-FUT "";
 0 AR-COND ""
```

```
LEXICON ER-FUT-COND
 0 ER-FUT "";
 0 ER-COND ""

LEXICON IRREG-FUT-COND
 0 IRREG-FUT "";
 0 IRREG-COND ""

LEXICON IR-FUT-COND
 0 IR-FUT "";
 0 IR-COND ""

LEXICON AR-COND
 +ari^as # "cond p2 sg";
 +ari^a # "cond p1 p3 sg";
 +ari^amos # "cond p1 pl";
 +ari^an # "cond p3 pl"

LEXICON IR-COND
 +iri^as # "cond p2 sg";
 +iri^a # "cond p1 p3 sg";
 +iri^amos # "cond p1 pl";
 +iri^an # "cond p3 pl"

LEXICON ER-COND
 +eri^as # "cond p2 sg";
 +eri^a # "cond p1 p3 sg";
 +eri^amos # "cond p1 pl";
 +eri^an # "cond p3 pl"

LEXICON IRREG-COND
 +i^as # "cond p2 sg";
 +i^a # "cond p1 p3 sg";
 +i^amos # "cond p1 pl";
 +i^an # "cond p3 pl"

LEXICON AR-FUT
 +are^ # "fut p1 sg";
 +ara^s # "fut p2 sg";
 +ara^ # "fut p3 sg";
 +aremos # "fut p1 pl";
 +ara^n # "fut p3 pl"

LEXICON ER-FUT
 +ere^ # "fut p1 sg";
 +era^s # "fut p2 sg";
 +era^ # "fut p3 sg";
 +eremos # "fut p1 pl";
 +era^n # "fut p3 pl"
```

```
LEXICON IR-FUT
 +ere^ # "fut p1 sg";
 +era^s # "fut p2 sg";
 +era^ # "fut p3 sg";
 +eremos # "fut p1 pl";
 +era^n # "fut p3 pl"

LEXICON IRREG-FUT
 +e^ # "fut p1 sg";
 +a^s # "fut p2 sg";
 +a^ # "fut p3 sg";
 +emos # "fut p1 pl";
 +a^n # "fut p3 pl"

LEXICON AR-MODAL
 0 /AR-V "i modal (subcat (v))"

LEXICON ER-MODAL
 0 /ER-V "i modal (subcat (v))"

LEXICON IR-MODAL
 0 /IR-V "i modal (subcat (v))"

LEXICON DET
 0 # "det sg";
 +s # "det pl";
 +a # "det fem sg";
 +o # "det neut sg";
 +os # "det masc pl";
 +as # "det fem pl";
 +es # "det pl";
 +e # "det masc sg"

LEXICON WH-PHRASE-A
 0 # "n neut p3 wh wh-phrase-a";
 +s # "n p3 pl wh wh-phrase-a";
 +a # "n fem sg p3 wh wh-phrase-a";
 +o # "n neut sg p3 wh wh-phrase-a";
 +os # "n masc pl p3 wh wh-phrase-a";
 +as # "n fem pl p3 wh wh-phrase-a";
 +es # "n pl p3 wh wh-phrase-a";
 +e # "n masc sg p3 wh wh-phrase-a"

LEXICON LY
 +amente # "adv"

LEXICON PRO
 0 # "pronoun p3 sg masc n animate";
 +a # "pronoun p3 sg fem n animate";
 +o # "pronoun p3 sg masc n animate";
 +as # "pronoun p3 pl fem n animate";
 +os # "pronoun p3 pl masc n animate";
 +e # "pronoun p3 sg masc n animate"

LEXICON A
 0 # "a sg";
 +s # "a pl";
 +a # "a fem sg";
 +o # "a masc sg";
 +as # "a fem pl";
 +os # "a masc pl";
 +e # "a neut sg"

LEXICON PROP
 0 # "n proper p3"

LEXICON N
 0 # "n sg p3";
 +s # "n pl p3";
 +a # "n fem p3 sg";
 +o # "n masc p3 sg";
 +as # "n fem p3 pl";
 +os # "n masc p3 pl";
 +e # "n neut p3 sg"

LEXICON Root
 /* DETERMINERS */
 el # "(english ((the))) det masc sg";
 los # "(english ((the))) det masc pl";
 la /DET "(english ((the))) fem";
 es /DET "(english ((that)))";
 est /DET "(english ((this)))";

 /* QUANTIFYING (DETERMINERS) */
 much /DET "(english ((many)))";
 ningun /DET "(english ((no)))";
 vari /DET "(english ((several)))";
 otr /DET "(english ((another)))";
 que^ # "(english ((what))) det neut wh";
 un /DET "(english ((a)))";
 un # "(english ((a))) det masc sg";
 algun /DET  "(english ((some)))";
 algu^n #  "(english ((some))) det masc sg";
 cada # "(english ((each))) det neut sg";
 tod /DET "(english ((all) (every)))";
 cua^l /DET "(english ((which))) neut wh";
 cua^nt /DET "(english ((how much))) wh";

 /* POSSESSIVE (DETERMINERS) */
 mi /DET "(english ((my))) neut poss pronoun p1 sg";
```

```
tu /DET "(english ((your))) neut poss pronoun p2 sg";
nuestr /DET "(english ((our))) poss pronoun p1 pl";
su /DET "(english ((his) (her) (its) (their) (your))) neut poss pronoun p3 sg pl";

/* CLITICS (DATIVE) */
me # "(english ((me))) cl-dat neut pronoun p1 sg obj animate";
te # "(english ((you))) cl-dat neut pronoun p2 sg obj animate";
se # "(english ((him) (her) (them))) cl-dat neut obj pronoun p3 sg pl animate";
nos # "(english ((us))) cl-dat neut pronoun p1 pl obj animate";
le  # "(english ((him) (her) (you))) cl-dat neut pronoun p3 sg obj animate";
les # "(english ((them))) cl-dat neut pronoun p3 pl obj animate";

/* CLITICS (ACCUSATIVE) */
me # "(english ((me))) cl-acc neut pronoun p1 sg obj animate";
te # "(english ((you))) cl-acc neut pronoun p2 sg obj animate";
nos # "(english ((us))) cl-acc neut pronoun p1 pl obj animate";
lo # "(english ((it) (him) (you))) cl-acc masc pronoun p3 sg obj
   inanimate animate";
los # "(english ((them))) cl-acc masc pronoun p3 pl obj inanimate animate";
la # "(english ((it) (her) (you))) cl-acc fem sg pronoun p3 sg obj
   inanimate animate";
las # "(english ((them))) cl-acc fem pronoun p3 pl obj inanimate animate";

/* REFLEXIVE CLITIC */
se # "(english ((himself) (herself) (themselves)))
   anaphor cl-ref obj neut p3 sg pl animate";
se # "(english ((yourselves))) anaphor cl-ref obj neut p2 pl animate";
me # "(english ((myself))) anaphor cl-ref obj neut p1 sg animate";
nos # "(english ((ourselves))) anaphor cl-ref obj neut p1 pl animate";
te # "(english ((yourself))) anaphor cl-ref obj neut p2 sg animate";

/* PRONOMINALS (OBJ) */
ti^ # "(english ((you))) n pronoun p2 sg neut obj animate";
mi^ # "(english ((i))) n pronoun p1 sg neut obj animate";

/* PRONOMINALS (NOM/OBJ) */
e^l /PRO "(english ((he) (it))) nom obj animate";
ell /PRO "(english ((she) (it) (they))) nom obj animate";
nosotros # "(english ((we))) n pronoun p1 pl neut nom obj animate";
nosotras # "(english ((we))) n pronoun p1 pl fem nom obj animate";
tu^ # "(english ((you))) n pronoun p2 sg neut nom animate";
yo # "(english ((i))) n pronoun p1 sg neut nom animate";
usted # "(english ((you))) n pronoun p3 sg neut nom obj animate";
ustedes # "(english ((you))) n pronoun p3 pl neut nom obj animate";
e^s /PRO "(english ((that))) nom obj inanimate";
e^st /PRO "(english ((this))) nom obj inanimate";


/* WH-PHRASES */
que^ /WH-PHRASE-A "(english ((what))) sg pl inanimate";
```

```
quie^n /WH-PHRASE-A "(english ((who))) animate";
cua^l /WH-PHRASE-A "(english ((which))) inanimate animate";
cua^ndo # "(english ((when))) n wh adv inanimate";
do^nde # "(english ((where))) n wh adv inanimate";
co^mo # "(english ((how))) n wh adv inanimate";
cua^nto # "(english ((how much))) n wh adv inanimate animate";

/* FINITE CLAUSE COMPS */
que /C-FIN "(english ((that)))";
para_que /C-FIN "(english ((so that) (for)))";
a_fin_de_que /C-FIN "(english ((so that)))";
a_menos_que /C-FIN "(english ((unless)))";
antes_de_que /C-FIN "(english ((before)))";
antes_que /C-FIN "(english ((before)))";
con_tal_que /C-FIN "(english ((provided that)))";
en_caso_de_que /C-FIN "(english ((provided that)))";
sin_que /C-FIN "(english ((without)))";
aunque /C-FIN "(english ((although)))";
cuando /C-FIN "(english ((when)))";
de_manera_que /C-FIN "(english ((so that)))";
de_modo_que /C-FIN "(english ((so that)))";
despue^s_de_que /C-FIN "(english ((after)))";
despue^s_que /C-FIN "(english ((after)))";
en_cuanto /C-FIN "(english ((as soon as)))";
luego_que /C-FIN "(english ((as soon as)))";
tan_pronto_como /C-FIN "(english ((as soon as)))";
asi^_que /C-FIN "(english ((as soon as)))";
hasta_que /C-FIN "(english ((until)))";
mientras /C-FIN "(english ((while)))";

/* INFINITE CLAUSE COMPS */
que /C-INF "(english ())";
para /C-INF "(english ((so that) (for)))";
a_fin_de /C-INF "(english ((so that)))";
a_menos /C-INF "(english ((unless)))";
antes_de /C-INF "(english ((before)))";
con_tal_de /C-INF "(english ((provided that)))";
en_caso_de /C-INF "(english ((provided that)))";
sin /C-INF "(english ((without)))";
cuando /C-INF "(english ((when)))";
despue^s_de /C-INF "(english ((after)))";
hasta /C-INF "(english ((until)))";
mientras /C-INF "(english ((while)))";

/* INFINITIVE MARKER */
a # "(english ((to))) inf i (subcat (v))";

/* VERBS */
haber /ER-V "(english ((have))) have-aux (root haber)";
he # "pres p1 sg (root haber)";
```

```
ha /PRES-IRREG "(root haber)";
hemos # "pres p1 pl (root haber)";
hub /PRET-IRREG-SUBJ-1 "(root haber)";
habr /IRREG-FUT-COND "(root haber)";

estar /AR-V "(english ((be))) v be-aux
 (external (agent)) (subcat (a)) (subcat (p-location)) (root estar)";
estuv /PRET-IRREG-SUBJ-1 "(root estar)";
estoy # "pres p1 sg (root estar)";
esta^ /PRES-IRREG "(root estar)";

poder /ER-MODAL "(english ((can))) (root poder) (external (agent))";
pued /ER-MODAL "(root poder)";
pud /PRET-IRREG-PROG-SUBJ-1 "(root poder)";
podr /IRREG-FUT-COND "(root poder)";

deber /ER-MODAL "(english ((should))) (root deber) (external (agent))";

saber /ER-V "(english ((know))) (root saber) v intrans (subcat (fact inanimate))
   (subcat (proposition)) (noun-form (sabimient /N) (subcat (p-proposition))
   (subcat (p-fact))) (external (agent))";
sabr /IRREG-FUT-COND "(root saber)";
se^ # "pres p1 sg (root saber)";
sup /PRET-IRREG-SUBJ-1 "(root saber)";

buscar /AR-V "(english ((look))) (root buscar) v
   (subcat (goal)) (external (agent animate))";

comer /ER-V "(english ((eat))) intrans (subcat (goal))";

comprar /AR-V "(english ((buy))) (root comprar) v (subcat (goal inanimate))
   (noun-form (compr /N) (subcat (p-goal inanimate))) (external (agent animate))";

conocer /ER-V "(english ((know))) (root conocer) v (subcat (p-goal animate))
   (subcat (goal inanimate)) (external (agent animate))";

continuar /AR-V "(english ((continue))) intrans v (subcat (goal inanimate))
   (root continuar) (external (agent))";

contribuir /IR-V "(english ((contribute))) (root contribuir) v
   (subcat (goal inanimate)) (subcat (goal inanimate) (patient))
   (external (agent animate))";
contribu /PRET-IRREG-PROG-SUBJ-2 "(root contribuir)";

costar /AR-V  "(english ((cost))) (root costar) v intrans (subcat (quantity))
   (noun-form (cuest /N) (subcat (p-quantity))) (external (agent))";
cuest /AR-V "(root costar)";

creer /ER-V "(english ((believe))) (root creer) v (subcat (p-goal inanimate))
   (subcat (goal inanimate) (proposition)) (subcat (proposition))
   (external (agent animate)) (noun-form (creenci /N)
   (subcat (p-goal inanimate)) (subcat (proposition)))";
cre /PRET-IRREG-PROG-SUBJ-2 "(root creer)";

dar /AR-V "(english ((give))) v (root dar) (subcat (goal inanimate))
   (subcat (goal inanimate) (patient animate)) (external (agent animate))";
doy # "pres p1 sg (root dar)";
di # "past p1 sg (root dar)";
dio # "past p3 sg (root dar)";

decir /IR-V "(english ((say) (tell))) (root decir) v (subcat (proposition))
   (subcat (patient animate) (proposition)) (subcat (patient animate))
   (external (agent animate))";
dic /ER-IR-PRES "(root decir)";
dic /ER-IR-PROG "(root decir)";
digo # "pres p1 sg (root decir)";
dij /PRET-IRREG-SUBJ-2 "(root decir)";
dicho # "perf (root decir)";
dir /IRREG-FUT-COND "(root decir)";

dormir /IR-V "(english ((sleep))) (root dormir) v intrans
   (external (agent animate))";
duerm /ER-IR-PRES "(root dormir)";
durm /ER-IR-PROG "(root dormir)";
durm /ER-IR-PRET "(root dormir)";
durm /ER-IR-SUBJ-PAST "(root dormir)";

leer /ER-V "(english ((read))) (external (agent animate))
    (subcat (goal inanimate))"
le /PRET-IRREG-PROG-SUBJ-2 "(root leer)";

ser /ER-V "(english ((be))) v be-aux (root ser)";
ser /ER-V "(english ((be))) v (subcat (a)) (subcat (entity))
   (subcat (p)) (root ser) (external (agent))";
soy # "pres p1 sg (root ser)";
eres # "pres p2 sg (root ser)";
es # "pres p3 sg (root ser)";
somos # "pres p1 pl (root ser)";
son # "pres p3 pl (root ser)";
fui # "past p1 sg (root ser)";
fueron # "past p3 pl (root ser)";
fu /PRET-IRREG-SUBJ-2 "(root ser)";
fue # "past p3 sg (root ser)";
era # "past p1 p3 sg (root ser)";
e^ramos # "past p1 pl (root ser)";
eran # "past p3 pl (root ser)";
eras # "past p2 sg (root ser)";

escribir /IR-V "(english ((write))) (root escribir) v intrans
   (subcat (goal inanimate)) (subcat (patient animate))
   (external (agent animate))";
escrito # "perf (root escribir)";

estirar /AR-V "v (expression ((la pata) (english ((die) (kick the bucket))))
   english ((stretch))) (root estirar) (subcat (goal inanimate))
   (subcat (p-goal animate))";

gustar /V "(english ((like))) (root gustar) (subcat (patient animate))
   (external (agent))";

hablar /AR-V "v (english ((speak))) (root hablar) intrans
   (subcat (p-goal inanimate)) (subcat (patient animate))
   (external (agent animate))";

hacer /AR-V "(english ((do) (make))) (root hacer) v (subcat (a))
   (subcat (goal inanimate)) (external (agent))";
hago # "pres p1 sg (root hacer)";
hecho # "perf (root hacer)";
hic /PRET-IRREG-SUBJ-1 "(root hacer)";
hizo # "past p1 sg (root hacer)";
har /IRREG-FUT-COND "(root hacer)";

huir /IR-V "(english ((flee))) (root huir) v intrans (subcat (p-location))
   (noun-form (huid /N) (subcat (p-location))) (external (agent animate))";
hu /PRET-IRREG-PROG-SUBJ-3 "(root huir)";

ir # "v inf (english ((go))) (root ir) intrans (subcat (proposition))
   (subcat (p-location)) (external (agent))";
voy # "pres p1 sg (root ir)";
va /PRES-IRREG "(root ir)";
fui # "past p1 sg (root ir)";
fueron # "past p3 pl (root ir)";
fu /PRET-IRREG-SUBJ-2 "(root ir)";
fue # "past p3 sg (root ir)";
iba # "past p1 p3 sg (root ir)";
i^bamos # "past p1 pl (root ir)";
iban # "past p3 pl (root ir)";
ibas # "past p2 sg (root ir)";
yendo # "prog (root ir)";
ido # "perf (root ir)";

morir /IR-V "(english ((die))) (root morir) v intrans (subcat (cause))
   (external (agent animate)) (noun-form (muerte /N fem)
   (subcat (patient animate)))  (adj-form (muert /A) (subcat (cause)))";
muer /ER-IR-PRES "(root morir)";
mur /ER-IR-PROG "(root morir)";
mur /ER-IR-PRET "(root morir)";
mur /ER-IR-SUBJ-PAST "(root morir)";
muerto # "perf (root morir)";
```

```
odiar /AR-V "(english ((hate))) (subcat (goal)) (external (agent animate))";

parecer /ER-V "(english ((seem))) (root parecer) v intrans
   (subcat (proposition))";

patear /AR-V "(english ((kick))) (root patear) v (subcat (goal inanimate))
   (external (agent animate)) (subcat (patient))";

pensar /AR-V "(english ((think))) (root pensar) v intrans (subcat (proposition))
   (subcat (p-goal inanimate)) (external (agent animate))";
piens /AR-PRES "(root pensar)";

poner /ER-V "(english ((put))) (root poner) v (external (agent animate))
   (subcat (goal inanimate) (p-location))";
puesto # "perf (root poner)";
pus /PRET-IRREG-SUBJ-1 "(root poner)";
pondr /IRREG-FUT-COND "(root poner)";

querer /ER-V "(english ((want))) (root querer) v (subcat (patient animate))
   (subcat (p-goal animate)) (subcat (goal inanimate)) (subcat (proposition))
   (external (agent animate))";
quier /ER-IR-PRES "(root querer)";
quis /PRET-IRREG-SUBJ-1 "(root querer)";
querr /IRREG-FUT-COND "(root querer)";

salir /IR-V "(english ((leave))) (root salir) v intrans (subcat (p-location))
   (external (agent))";
salgo # "pres p1 sg (root salir)";
saldr /IRREG-FUT-COND "(root salir)";

suponer /ER-V "(english ((expect))) (root suponer) v (subcat (proposition))
   (external (agent))";
supuesto # "perf (root suponer)";
supus /PRET-IRREG-SUBJ-1 "(root suponer)";
supondr /IRREG-FUT-COND "(root suponer)";

tener /ER-V "(english ((have))) (root tener) v (subcat (goal inanimate))
   (external (agent animate))";
tien /ER-IR-PRES "(root tener)";
tendr /IRREG-FUT-COND " (root tener)";
tuv /PRET-IRREG-SUBJ-1 "(root tener)";

ver /ER-V "(english ((see))) (root ver) v intrans (subcat (p-goal animate))
   (subcat (goal inanimate)) (external (agent animate))";
veo # "pres p1 sg (root ver)";
ve /ER-IR-IMP "(root ver)";
ve /ER-IR-SUBJ-PRES "(root ver)";
vi # "past p1 sg (root ver)";
vio # "past p3 sg (root ver)";
visto # "perf (root ver)";
```

```
vencer /ER-V "(english ((conquer))) (root vencer) v intrans
   (subcat (goal inanimate)) (external (agent animate))";

vender /ER-V  "(english ((sell))) (root vender) v (external (agent animate))
   intrans (subcat (goal inanimate))
   (noun-form (vent /N) (subcat (p-goal inanimate)))";
vendr /IRREG-FUT-COND "(root vender)";

venir /IR-V "(english ((come))) (root venir) v intrans (subcat (p-location))
   (external (agent))";
vin /PRET-IRREG-PROG-SUBJ-1 "(root venir)";
vien /ER-IR-PRES "(root venir)";

visitar /AR-V "(english ((visit))) (external (agent animate))
   (subcat (p-goal animate)) (subcat (goal inanimate))";

/* PREPOSITIONS */
a /PREP "(english ((to) (at)))";
al_lado_de /PREP "(english ((beside)))";
acerca_de /PREP "(english ((about)))";
antes_de /PREP "(english ((before)))";
alrededor_de /PREP "(english ((around)))";
bajo # "(english ((beneath)))";
cerca_de /PREP "(english ((near)))";
con /PREP "(english ((with)))";
contra /PREP "(english ((against)))";
de /PREP "(english ((from) (of)))";
debajo_de /PREP "(english ((below) (beneath) (under)))";
dentro_de /PREP "(english ((inside)))";
despue^s_de /PREP "(english ((after)))";
detra^s_de /PREP "(english ((behind)))";
durante /PREP "(english ((during)))";
en /PREP "(english ((on)))";
encima_de /PREP "(english ((above)))";
entre /PREP "(english ((among) (between)))";
fuera_de /PREP "(english ((outside)))";
hacia /PREP "(english ((toward)))";
para /PREP "(english ((for)))";
por /PREP "(english ((for) (by) (through) (per)))";
sin /PREP "(english ((without)))";
sobre /PREP "(english ((over) (on) (above)))";

/* NOUNS */
agua # "(english ((water))) n masc sg inanimate";
amig /N "(english ((friend))) animate";
cam /N "(english ((bed))) inanimate";
cart /N "(english ((letter))) inanimate";
cas  /N "(english ((house))) inanimate";
cine /N "(english ((movie))) masc inanimate";
```

```
cos /N "(english ((thing))) inanimate";
cubet /N "(english ((bucket))) inanimate";
empres /N "(english ((firm))) inanimate";
gat /N "(english ((cat)))";
muchach /N "(english ((boy) (girl))) animate";
hombre /N "(english ((man))) masc animate";
herman /N "(english ((brother) (sister))) animate";
lad /N "(english ((side))) inanimate";
libr /N "(english ((book))) inanimate";
madre /N "(english ((mother))) fem animate";
mes /N "(english ((table))) inanimate";
mujer /N "(english ((woman))) fem animate";
padre /N "(english ((father))) masc animate";
pat /N "(english ((paw))) inanimate";
peli^cul /N "(english ((movie))) inanimate";
pis /N "(english ((floor))) inanimate";
reunio^n /N "(english ((meeting))) fem inanimate";
verdad /N "(english ((truth))) fem inanimate";
tiempo /N "(english ((time))) masc inanimate";
lapiz /N "(english ((pencil))) masc inanimate";

/* PROPER NOUNS */
espan~a /PROP "(english ((spain))) sg neut";
juan /PROP "(english ((john))) sg masc animate";
mari^a /PROP "(english ((maria))) sg fem animate";

/* ADJECTIVES */
ciert /A "(english ((certain))) intrans (subcat (proposition))";
posibl /A "(english ((possible))) intrans (subcat (proposition))";
probabl /A "(english ((probable) (likely))) intrans
   (subcat (proposition))";

cansad /A "(english ((tired)))";
orgullos /A "(english ((proud))) intrans (subcat (p-goal inanimate))
        (noun-form (orgull /N) (subcat (p-goal inanimate)))";
ansios /A "(english ((eager)))";
satisfech /A "(english ((satisfied)))";

/. # "(english ((/.))) punc";
/? # "(english ((/?))) punc"

END
```

Appendix D

Translation System Parameters

This appendix contains some of the parameter settings that are specific to the UNITRAN system (*i.e.*, they are not included in the parameters of GB Theory).

```
;;; Features are used for instantiating the slots of a node at scan time.  The
;;; are also used as a check on lexical entries (each feature in a lexical
;;; entry must be specified in the FEATURES parameter).
(DEF-PARAM FEATURES
  :SPANISH
  (:GENDER (FEM MASC NEUT)
   :PERS (P1 P2 P3)
   :NUMBER (PL SG)
   :CASE (OBJ NOM POSS)
   :TENSE (INF PERF PROG PAST PRES PRES-SUBJ PAST-SUBJ FUT COND)
   :N-FEATURES (:CASE :GENDER :NUMBER PROPER ANAPHOR PRONOUN
                LOCATION DURATION QUANTITY ACCESS METHOD
                EXCHANGE TIME ANIMATE INANIMATE WH WH-PHRASE-A)
   :A-FEATURES (:PERS :NUMBER :GENDER DESCRIPT CONDITION)
   :P-FEATURES  (P-DESCRIPT P-CONDITION P-LOCATION P-FACT P-GOAL P-QUANTITY)
   :V-FEATURES (:PERS :NUMBER :TENSE SDEL INF INTRANS)
   :I-FEATURES (MODAL :PERS :NUMBER :TENSE INF)
   :C-FEATURES (WH WH-PHRASE FACT)
   :DET-FEATURES (:GENDER :NUMBER :PERS :CASE PRONOUN WH)
   :CL-ACC-FEATURES (:GENDER :NUMBER :PERS :CASE PRONOUN INANIMATE ANIMATE)
   :CL-DAT-FEATURES (:GENDER :NUMBER :PERS :CASE PRONOUN ANIMATE)
   :CL-REF-FEATURES (:GENDER :NUMBER :PERS :CASE PRONOUN ANIMATE))
  :ENGLISH
   (:GENDER (FEM MASC NEUT)
    :PERS (P1 P2 P3)
    :NUMBER (PL SG MASS)
    :CASE (OBJ NOM POSS)
    :TENSE (INF PERF PROG PAST PRES FUT COND)
    :N-FEATURES (:GENDER :NUMBER :CASE PROPER ANAPHOR PRONOUN PLEONASTIC
                 LOCATION DURATION QUANTITY ACCESS METHOD
                 EXCHANGE TIME ANIMATE WH WH-PHRASE-A)
    :A-FEATURES (DESCRIPT V-ABLE CONDITION COMPARATIVE SUPERLATIVE)
    :P-FEATURES (P-DESCRIPT P-CONDITION P-LOCATION P-FACT P-GOAL P-QUANTITY)
    :V-FEATURES (:PERS :NUMBER :TENSE SDEL INTRANS)
    :I-FEATURES (:PERS :TENSE MODAL)
    :C-FEATURES (WH WH-PHRASE FACT)
    :DET-FEATURES (:GENDER :NUMBER :PERS :CASE PRONOUN WH)))

;;; Match-lists are employed by the feature-matching routine to ensure that
;;; Heads and Specs are compatible.
(DEF-PARAM MATCH-LISTS
  :SPANISH
  (:PERSON (((P1) (P1)) ((P2) (P2)) ((PROPER P3) (P3)))
   :NUMBER (((PL) (PL)) ((MASS PROPER SG) (SG)))
   :GENDER (((FEM NEUT) (FEM NEUT)) ((MASC NEUT) (MASC NEUT)))
   :TENSE (((PRES FUT PAST) (PERF))))
  :ENGLISH
  (:PERSON (((P1) (P1)) ((P2) (P2)) ((PROPER P3) (P3)))
   :NUMBER (((PL) (PL)) ((MASS PROPER SG) (SG)))
   :GENDER (((FEM NEUT) (FEM NEUT)) ((MASC NEUT) (MASC NEUT)))
   :TENSE (((PRES PAST FUT) (PERF)))))
```

```
;;; Splits-and-merges are accessed by the preprocessor to split contracted
;;; forms, and join words that function as a single unit
(DEF-PARAM SPLITS-AND-MERGES
  :SPANISH
  (:SPLITS
   (((AL) (A EL))
    ((DEL) (DE EL))
    ((*RME) (*R ME))
    ((*RME) (*R ME))
    ((*RTE) (*R TE))
    ((*RSE) (*R SE))
    ((*RLA*) (*R LA*))
    ((*RLO*) (*R LO*))
    ((*RLE*) (*R LE*))
    ((*OME) (*O ME))
    ((*OTE) (*O TE))
    ((*OSE) (*O SE))
    ((*OLA*) (*O LA*))
    ((*OLO*) (*O LO*))
    ((*OLE*) (*O LE*))
    ((*A^RSELO*) (*AR SE LO*))
    ((*A^RSELA*) (*AR SE LA*))
    ((*A^RMELO*) (*AR ME LO*))
    ((*A^RMELA*) (*AR ME LA*))
    ((*A^RTELO*) (*AR TE LO*))
    ((*A^RTELA*) (*AR TE LA*))
    ((*A^RNOSLO*) (*AR NOS LO*))
    ((*A^RNOSLA*) (*AR NOS LA*))
    ((*A^NDOSELO*) (*ANDO SE LO*))
    ((*A^NDOSELA*) (*ANDO SE LA*))
    ((*A^NDOMELO*) (*ANDO ME LO*))
    ((*A^NDOMELA*) (*ANDO ME LA*))
    ((*A^NDOTELO*) (*ANDO TE LO*))
    ((*A^NDOTELA*) (*ANDO TE LA*))
    ((*A^NDONOSLO*) (*ANDO NOS LO*))
    ((*A^NDONOSLA*) (*A^NDO NOS LA*))
    ((*I^RSELO*) (*IR SE LO*))
    ((*I^RSELA*) (*IR SE LA*))
    ((*I^RMELO*) (*IR ME LO*))
    ((*I^RMELA*) (*IR ME LA*))
    ((*I^RTELO*) (*IR TE LO*))
    ((*I^RTELA*) (*IR TE LA*))
    ((*I^RNOSLO*) (*IR NOS LO*))
    ((*I^RNOSLA*) (*IR NOS LA*))
    ((*IE^NDOSELO*) (*IENDO SE LO*))
    ((*IE^NDOSELA*) (*IENDO SE LA*))
    ((*IE^NDOMELO*) (*IENDO ME LO*))
    ((*IE^NDOMELA*) (*IENDO ME LA*))
    ((*IE^NDOTELO*) (*IENDO TE LO*))
((*IE^NDOTELA*) (*IENDO TE LA*))
((*IE^NDONOSLO*) (*IENDO NOS LO*))
((*IE^NDONOSLA*) (*IENDO NOS LA*))
((*E^RSELO*) (*ER SE LO*))
((*E^RSELA*) (*ER SE LA*))
((*E^RMELO*) (*ER ME LO*))
((*E^RMELA*) (*ER ME LA*))
((*E^RTELO*) (*ER TE LO*))
((*E^RTELA*) (*ER TE LA*))
((*E^RNOSLO*) (*ER NOS LO*))
((*E^RNOSLA*) (*ER NOS LA*))
((*YE^NDOSELO*) (*YENDO SE LO*))
((*YE^NDOSELA*) (*YENDO SE LA*))
((*YE^NDOMELO*) (*YENDO ME LO*))
((*YE^NDOMELA*) (*YENDO ME LA*))
((*YE^NDOTELO*) (*YENDO TE LO*))
((*YE^NDOTELA*) (*YENDO TE LA*))
((*YE^NDONOSLO*) (*YENDO NOS LO*))
((*YE^NDONOSLA*) (*YENDO NOS LA*)))
```

Appendix E

Lisp Representation of GB Parameters

This appendix contains the Lisp representation used for the setting of parameters to the principles. Parameters are grouped according to subtheories of GB (as shown in appendix A.2).

E.1 $\overline{\text{X}}$-Theory

```
;;; Constituent-order is used to determine the positioning of heads, specifiers
;;; and complements in order to set up the XBAR templates, and to perform
;;; generation.
(DEF-PARAM CONSTITUENT-ORDER :SPANISH (SPEC HEAD COMP) :ENGLISH (SPEC HEAD COMP))

;;; Basic-categories are inserted into the XBAR templates at precompilation
;;; time.  The distinguished start symbol (root node) must be the the first
;;; category specified (e.g., C in Spanish and English).
(DEF-PARAM BASIC-CATEGORIES :SPANISH (C I V N P A) :ENGLISH (C I V N P A))

;;; Pre-terminal symbols are inserted into specifier and adjunct positions of
;;; XBAR templates as dictated by the CHOICE-OF-SPEC and ADJUNCTION parameters
;;; (see below).
(DEF-PARAM PRE-TERMINALS
   :SPANISH (DET ADV PUNC HAVE-AUX BE-AUX CL-ACC CL-DAT CL-REF)
   :ENGLISH (DET ADV DO-AUX HAVE-AUX BE-AUX PUNC))
```

```
;;; Choice-of-spec elements are inserted into specifier positions in XBAR
;;; templates.
(DEF-PARAM CHOICE-OF-SPEC
   :SPANISH ((:NAME WH-MOVEMENT (C ((N-MAX) (P-MAX) (ADV))))
             (:NAME PERFECT (V ((HAVE-AUX))))
             (:NAME DETERMINER (I ((DET))))
             (:NAME POSSESSIVE (N ((N-MAX))))
             (:NAME SUBJECT (I ((N-MAX))))
             (:NAME ADVERBIAL (A ((ADV)) P ((ADV)))))
   :ENGLISH ((:NAME WH-MOVEMENT (C ((N-MAX) (P-MAX) (ADV))))
             (:NAME PERFECT (V ((HAVE-AUX))))
             (:NAME DO-SUPPORT (V ((DO-AUX))))
             (:NAME DETERMINER (I ((DET))))
             (:NAME POSSESSIVE (N ((N-MAX))))
             (:NAME SUBJECT (I ((N-MAX))))
             (:NAME ADVERBIAL (A ((ADV)) P ((ADV))))))

;;; Adjunction possibilities include:
;;;   1 = adjoin left at maximal level, 2 = adjoin right at maximal level
;;;   3 = adjoin left at minimal level, 4 = adjoin right at minimal level
(DEF-PARAM ADJUNCTION
   :ENGLISH ((:NAME A-TO-N :NODE (A-MAX) :TYPE (3) :HEAD (N))
             (:NAME P-TO-N-V :NODE (P-MAX) :TYPE (2 4) :HEAD (N V))
             (:NAME ADV-TO-V :NODE (ADV) :TYPE (2) :HEAD (V))
             (:NAME C-TO-N :NODE (C-MAX) :TYPE (2 4) :HEAD (N))
             (:NAME PASS-PROG :NODE (BE-AUX) :TYPE (3) :HEAD (V))
             (:NAME DO-SAI :NODE (DO-AUX) :TYPE (1) :HEAD (I))
             (:NAME BE-SAI :NODE (BE-AUX) :TYPE (1) :HEAD (I))
             (:NAME HAVE-SAI :NODE (HAVE-AUX) :TYPE (1) :HEAD (I))
             (:NAME I-SAI :NODE (I) :TYPE (1) :HEAD (I)))
   :SPANISH ((:NAME A-TO-N :NODE (A-MAX) :TYPE (3 4) :HEAD (N))
             (:NAME P-TO-N-V :NODE (P-MAX) :TYPE (2 4) :HEAD (N V))
             (:NAME ADV-TO-V :NODE (ADV) :TYPE (1 2) :HEAD (V))
             (:NAME C-TO-N :NODE (C-MAX) :TYPE (2 4) :HEAD (N))
             (:NAME PASS-PROG :NODE (BE-AUX) :TYPE (3) :HEAD (V))
             (:NAME OBJ-CLITIC-TO-V :NODE (CL-DAT CL-ACC) :TYPE (3 4) :HEAD (V))
             (:NAME REF-CLITIC-TO-V :NODE (CL-REF) :TYPE (3 4) :HEAD (V))
             (:NAME REF-OBJ-CLITIC-TO-V
                    :NODE (CL-REF CL-ACC) :TYPE (3 4) :HEAD (V))
             (:NAME PREPOSE :NODE (I HAVE-AUX BE-AUX V) :TYPE (1) :HEAD (I))
             (:NAME FREE-INVERSION :NODE (N-MAX) :TYPE (2) :HEAD (V))))

;;; Default complements for heads that are empty.
(DEF-PARAM EMPTY-FEATURE-HOLDERS
    :SPANISH (I ((V)) C ((I))) :ENGLISH (I ((V)) C ((I))))
```

```
;;; Optional specifiers may be dropped in XBAR templates.
(DEF-PARAM OPTIONAL-SPECIFIERS
   :SPANISH (V-SPEC A-SPEC C-SPEC P-SPEC N-SPEC)
   :ENGLISH (V-SPEC A-SPEC C-SPEC P-SPEC N-SPEC))
```

E.2 θ-Theory

```
;;; Values of clitics (if there are any).
(DEF-PARAM CLITICS :SPANISH (CL-DAT CL-ACC CL-REF) :ENGLISH NIL)

;;; Clitic-doubling forces the rule:
;;; [ CL +CASE{i} + THETA{j} ] ... [NP +CASE{i} ] ==>
;;; [ CL +CASE{i} + THETA{j} ] ... [NP +CASE{i} +THETA{j} ] to fire if clitics
;;; are allowed.
(DEF-PARAM CLITIC-DOUBLING
   :SPANISH (SETQ *CLITIC-THETA-TRANSMIT T)
   :ENGLISH (SETQ *CLITIC-THETA-TRANSMIT NIL))

;;; Canonical-semantic-mappings map between semantic roles and canonical
;;; structural representations.
(DEF-PARAM CANONICAL-SEMANTIC-MAPPINGS
   :SPANISH (:AGENT N :PATIENT P :PROPOSITION C :P-PROPOSITION P :LOCATION N
             :P-LOCATION P :FACT N :P-FACT P :QUANTITY N :P-QUANTITY :GOAL N
             :P-GOAL P :ENTITY N :CAUSE P)
   :ENGLISH (:AGENT N :PATIENT N :P-PATIENT P :PROPOSITION C :P-PROPOSITION P
             :LOCATION N :P-LOCATION P :FACT N :P-FACT P :QUANTITY N :P-QUANTITY
             :GOAL N :P-GOAL P :ENTITY N :CAUSE P))
```

E.3 Government Theory

```
;;; Governing categories (used for the Case and Binding modules).
(DEF-PARAM GOVERNORS :SPANISH (N V A P AGR) :ENGLISH (N V A P AGR))
```

E.4 Case Theory

```
;;; Case assignment parameter specifies the type of government and the case
;;; assigned.
(DEF-PARAM CASE-ASSIGNMENT
   :SPANISH (V (S-GOVERNS? OBJ) P (C-GOVERNS? OBJ) AGR (C-GOVERNS? NOM))
   :ENGLISH (V (C-GOVERNS? OBJ) P (C-GOVERNS? OBJ) AGR (C-GOVERNS? NOM)))
```

E.5 Trace Theory

```
;;; Pro-drop is used by the trace module to determine two things: whether a
;;; subject can be null, and whether AGR is allowed to be a proper governor.
(DEF-PARAM PRO-DROP :SPANISH (SETQ *AGR-RICH T) :ENGLISH (SETQ *AGR-RICH NIL))

;;; Traces are inserted into nodes of XBAR templates at precompilation time, but
;;; they are checked by ECP at POP time and generation time.
(DEF-PARAM TRACES
   :SPANISH (N-MAX-TRACE P-MAX-TRACE BE-AUX-TRACE HAVE-AUX-TRACE I-TRACE V-TRACE)
   :ENGLISH (N-MAX-TRACE P-MAX-TRACE))

;;; ECP is used by GB to determine whether chain conditions are required for
;;; checking proper government of an empty category.
(DEF-PARAM ECP :SPANISH (SETQ *CHAIN-CONDITIONS T)
               :ENGLISH (SETQ *CHAIN-CONDITIONS NIL))
```

E.6 Bounding Theory

```
;;; According to the principle of Subjacency, an antecedent may not be beyond
;;; more than one bounding node. (Bounding-nodes are used to limit the search
;;; for an antecedent of a trace.)
(DEF-PARAM BOUNDING-NODES :SPANISH (C-MAX N-MAX) :ENGLISH (I-MAX N-MAX))

;;; Language-specific effects is used in the Bounding module to determine when
;;; certain actions are obligatory during parsing and generation.
(DEF-PARAM LANGUAGE-SPECIFIC-EFFECTS
   :SPANISH (((PREPOSE) <==> (WH-MOVEMENT WH-PHRASE-A)))
   :ENGLISH (((OR (DO-SAI) (BE-SAI) (HAVE-SAI) (I-SAI)) <==> (WH-MOVEMENT))))

;;; Derived-specifiers-and-adjunction are used during move-alpha in order to
;;; determine the positions to which elements may move.
(DEF-PARAM DERIVED-SPECIFIERS-AND-ADJUNCTION
   :SPANISH (WH-MOVEMENT PREPOSE FREE-INVERSION)
   :ENGLISH (WH-MOVEMENT DO-SAI BE-SAI HAVE-SAI I-SAI))

;;; Base-specifiers-and-adjunction are used during move-alpha in order to
;;; determine the base-generated specifiers and adjunction.
(DEF-PARAM BASE-SPECIFIERS-AND-ADJUNCTION
   :SPANISH (WH-MOVEMENT PREPOSE FREE-INVERSION)
   :ENGLISH (WH-MOVEMENT DO-SAI BE-SAI HAVE-SAI I-SAI))
```

Appendix F

Lisp Representation of GB Principles

This appendix contains the Lisp representation of the principles and their incorporation of parameter values. Principles are grouped according to subtheories of GB (as shown in appendix A).

F.1 $\overline{\text{X}}$-Theory

F.1.1 Precompilation Routines

The $\overline{\text{X}}$ module is accessed during precompilation, parsing (PUSH, SCAN and POP), and generation. During precompilation, $\overline{\text{X}}$ templates are generated for the Earley parser. During parsing, the $\overline{\text{X}}$ module performs several tasks: at PUSH time, complement templates are predicted, complements are associated with heads, and features (of pushed traces) are percolated; at SCAN time, features are instantiated, argument structure is determined, and feature percolation takes place; and at POP time, feature percolation takes place again. At generation time, the $\overline{\text{X}}$ module is accessed in order to determine positions where specifiers and adjoined elements are base-generated in the target language, and to modify the structure accordingly. Also, the generator accesses the $\overline{\text{X}}$ component to order constituents according to the constituent order of the target language.

```
;;; Set-up-xbar is invoked at precompilation time.  It sets up the x-bar
;;; templates required by the source and target languages, according to the
;;; parameter values of constituent order, possible-empty-feature-holders,
;;; traces, empty, choice-of-spec and optional-specifiers.
(DEFUN SET-UP-XBAR (LANGUAGE)
  ;; The full blown rules include unconditionally added rules as well as rules
  ;; derived from the base constituent order
  (LET ((OBLIGATORY-RULES) (OPTIONAL-RULES) (FULL-BLOWN-RULES NIL))
    (SETQ
      OBLIGATORY-RULES
      (APPEND
        ;; Create the base x-bar rules.
        (CREATE-BASE-RULES *CURRENT-CONSTITUENT-ORDER *CURRENT-BASIC-CATEGORIES)
        ;; Create adjunction rules.
        (CREATE-ADJUNCTS *CURRENT-ADJUNCTION)
        ;; Add trace rules (Y ==> Y-TRACE)
        (ADD-TRACE-RULES *CURRENT-TRACES)
        ;; Add empty rules (Z ==> E) and (Z-COMPLEMENT ==> <cat>) for all
        ;; <cat>'s specified in the EMPTY-FEATURE-HOLDERS parameter.
        (MAKE-EMPTY-RULES *CURRENT-EMPTY-FEATURE-HOLDERS)
        ;; Add rules corresponding to the empty (NP) elements (big pro and also
        ;; little pro if AGR is rich).
        (ADD-EMPTY-NPS *CURRENT-PRO-DROP)
        ;; Add rules corresponding to possible choices of spec.
        (MAKE-SPEC-RULES *CURRENT-CHOICE-OF-SPEC))
      ;; Allow optional specifiers to be dropped.
      OPTIONAL-RULES
      (DROP-OPTIONAL-SPECIFIERS *CURRENT-OPTIONAL-SPECIFIERS OBLIGATORY-RULES)
      ;; Combine obligatory and optional rules.
      FULL-BLOWN-RULES (APPEND OBLIGATORY-RULES OPTIONAL-RULES))
    ;; Define the grammar for the language.
    (EVAL (CONS 'DEFGRAMMAR (CONS LANGUAGE FULL-BLOWN-RULES)))))
```

F.1.2 Complement Prediction Routines

```
;;; Locate-complements is invoked at PUSH time when a complement needs to be
;;; expanded.  The complement templates are generated by locating the closest
;;; head to the left or right (depending on whether the language is
;;; head-initial or head-final).  The corresponding rules are added later (so
;;; that the state-set will be consistent with the structure that is being
;;; built).
(DEFUN LOCATE-COMPLEMENTS (HEAD-CAT COMPLEMENT-SYMBOL HEAD-FIRST?)
  (LET ((INTRANS NIL) (SUBCAT-LIST) (MAX-CATS) (CURRENT-RESULT))
    ;; Determine the subcategorization frames for the closest head to the left
    ;; or to the right.
    (SETQ SUBCAT-LIST (LOCATE-SUBCAT-LIST HEAD-CAT HEAD-FIRST?))
    ;; Take care of intransitive possibility.
    (WHEN (OR (NULL SUBCAT-LIST) (EQ (CAR SUBCAT-LIST) 'INTRANS))
      (SETQ INTRANS '(INTRANS) SUBCAT-LIST (CDR SUBCAT-LIST)))
    ;; Iterate through subcategorization frames pushing the left-most
    ;; subcategorized maximal projection onto the stack base.
    (LOOP FOR SUBCAT IN SUBCAT-LIST DO
      ;; Determine maximal-projections of subcategorized elements.
      (SETQ MAX-CATS
            (TURN-INTO-MAX (MAPCAR 'CAR (GET-CATEGORY-SUBCAT SUBCAT))))
      ;; Turn the subcategorization frame into a stack element.
      (PUSH (MAPCAR 'LIST MAX-CATS) CURRENT-RESULT))
    ;; Finally return the result.
    (CONS COMPLEMENT-SYMBOL (APPEND INTRANS CURRENT-RESULT))))

;;; Locate-subcat-list finds all possible subcategorization frames for the
;;; closest head (of category HEAD-CAT) to the left or to the right.
(DEFUN LOCATE-SUBCAT-LIST (HEAD-CAT HEAD-FIRST?)
  (IF HEAD-FIRST?
      ;; Head initial
      (CLOSEST-HEAD? 'LEFT HEAD-CAT)
      ;; Head final: check for movement first.
      (IF (MEMBER (ADD-ENDING HEAD-CAT '-TRACE) *CURRENT-TRACES :TEST #'EQ)
          ;; Possibility of leftward movement as well as head appearing to
          ;; right.
          (UNION (CLOSEST-HEAD? 'RIGHT HEAD-CAT)
                 (CLOSEST-HEAD? 'LEFT HEAD-CAT) :TEST #'EQUAL)
          ;; No possibility of leftward movement.
          (CLOSEST-HEAD? 'RIGHT HEAD-CAT))))
```

```
;;; Closest-head? locates the closest morphologically analyzed word (in a
;;; particular direction) corresponding to a head category. (The assumption is
;;; that the head has not moved "too far away," if indeed it has moved at all.)
;;; It then returns the subcategorization frame of the word. Note: clitics need
;;; to be accounted for here so that the subcategorization frame is adjusted
;;; accordingly.
(DEFUN CLOSEST-HEAD? (DIRECTION HEAD-CAT)
  (LET ((SUBCATEGORIZATION NIL))
    (LOOP FOR WORD IN (LOCATE-SECTION-OF-SENTENCE DIRECTION) DO
      (WHEN (MEMBER HEAD-CAT (GET-CAT WORD) :TEST #'EQ)
        (SETQ SUBCATEGORIZATION
              (TAKE-CLITICS-INTO-ACCOUNT
                HEAD-CAT (GET-SUBCATEGORIZATION (SECOND WORD))))
        (RETURN (OR SUBCATEGORIZATION '(INTRANS)))))))

;;; Take-clitics-into-account creates additional subcategorization frames for a
;;; head-cat if clitics are allowed.  This is because certain complements can
;;; be encoded as clitics rather than internal arguments.  So the job of this
;;; function is to add to the subcategorization possibilities by eliminating
;;; one or more elements of SUBCAT.
(DEFUN TAKE-CLITICS-INTO-ACCOUNT (HEAD-CAT SUBCAT)
  (LET ((INTRANS? NIL) (NEW-SUBCAT) (NEW-RESULTS NIL))
    ;; First check if the language allows clitics.
    (IF (NOT *CURRENT-CLITICS)                  ; clitics not allowed.
        SUBCAT                                  ; clitics allowed.
        ;; If the head is intransitive, set the INTRANS? flag.
        (WHEN  (EQ (CAR SUBCAT) 'INTRANS)
          (SETQ INTRANS? T NEW-SUBCAT (CDR SUBCAT)))
        ;; If the head is transitive, process its complements. First check if
        ;; clitic adjunction to the head-cat is allowed.
        (WHEN (AND NEW-SUBCAT (CLITIC-ADJOINED-TO-HEAD? HEAD-CAT))
          ;; Set INTRANS? flag since one possibility is for all internal
          ;; arguments to be eliminated.
          (SETQ INTRANS? T)
          ;; Run through each of the subcategorization frames removing
          ;; elements, unless there is only one element in the subcat frame.
          (LOOP FOR SUB IN NEW-SUBCAT DO
            (SETQ NEW-RESULTS
                  (APPEND NEW-RESULTS
                          (WHEN (> (LENGTH SUB) 1)
                            (REMOVE-SUBCATEGORIZED-COMPLEMENTS SUB))))))
        ;; Return a UNION of the old subcategorization frames and the new ones
        ;; (with subcategorized elements missing).
        (IF INTRANS?
            (CONS 'INTRANS (UNION NEW-RESULTS NEW-SUBCAT :TEST #'EQUAL))
            (UNION NEW-RESULTS NEW-SUBCAT :TEST #'EQUAL)))))
```

```
;;; Clitic-adjoined-to-head? checks to see if clitic adjunction to the head-cat
;;; is allowed.
(DEFUN CLITIC-ADJOINED-TO-HEAD? (HEAD)
  (LOOP FOR ADJUNCTION IN *CURRENT-ADJUNCTION DO
    (WHEN (AND (MEMBER HEAD (GET-VALUE ADJUNCTION :HEAD) :TEST #'EQ)
               (INTERSECTION
                 (GET-VALUE ADJUNCTION :NODE) *CURRENT-CLITICS :TEST #'EQ))
      (RETURN T))))

;;; Locate-section-of-sentence returns the portion the sentence preceding
;;; (LEFT) or following (RIGHT) the *CURRENT-MORPH-WORD depending on the
;;; DIRECTION that is passed to it.
(DEFUN LOCATE-SECTION-OF-SENTENCE (DIRECTION)
  (IF (EQ DIRECTION 'RIGHT)
      (MEMBER *CURRENT-MORPH-WORD
              (REVERSE *CURRENT-MORPH-SENTENCE) :TEST #'EQUAL)
      (MEMBER *CURRENT-MORPH-WORD *CURRENT-MORPH-SENTENCE :TEST #'EQUAL)))

;;; Get-category-subcat turns a subcategorization frame into its corresponding
;;; structural frame (via CSR). If INTRANS is in the subcategorization frame,
;;; it will be consed onto the front of the result.
(DEFUN GET-CATEGORY-SUBCAT (SUBCAT)
  (COND ((EQ SUBCAT 'INTRANS) 'INTRANS)
        ((MEMBER 'INTRANS SUBCAT :TEST #'EQ)
         (CONS 'INTRANS (TURN-INTO-CAT (REMOVE 'INTRANS SUBCAT))))
        (T (TURN-INTO-CAT SUBCAT))))

;;; Turn-into-cat turns a subcategorization (role) frame into its corresponding
;;; structural representation (via CSR).
(DEFUN TURN-INTO-CAT (SUBCATS)
  (LOOP FOR SUBCAT IN SUBCATS
        COLLECT (MAPCAR '(LAMBDA (X) (LIST (CSR X))) (HEADERS-OF SUBCAT))))

;;; Get-subcategorization retrieves the theta-roles of internal arguments from
;;; the features associated with a word.  All permutations of the
;;; subcategorization frames with multiple arguments will be included in the
;;; result.
(DEFUN GET-SUBCATEGORIZATION (FEATURES)
  (LET ((SUBCATEGORIZATION
          (LOOP FOR ELT IN FEATURES
            ;; When a subcategorization frame is encountered, collect it.
                WHEN (AND (LISTP ELT) (EQUAL (CAR ELT) 'SUBCAT)) COLLECT ELT)))
    ;; If INTRANS is in the set of features, include it in the
    ;; subcategorization frame.  Otherwise, just include the subcategorization
    ;; frame (with permuted arguments, of course).
    (IF (MEMBER 'INTRANS FEATURES)
        (CONS 'INTRANS
              (WHEN SUBCATEGORIZATION
                (LOOP FOR SUBCAT IN SUBCATEGORIZATION
                  APPEND (PERMUTE-SUBCAT (CDR SUBCAT)))))
    (WHEN SUBCATEGORIZATION
      (LOOP FOR SUBCAT IN SUBCATEGORIZATION
            APPEND (PERMUTE-SUBCAT (CDR SUBCAT)))))))
```

F.1.3 Routines for Associating Complements with Heads

```
;;; Perform-subcategorization-check links complements up to heads at PUSH time.
;;; All the complements and their corresponding heads and subcat frames are
;;; collected, and then the subcategorization information is stored in the head
;;; nodes. Also, intransitive heads are taken care of here.
(DEFUN PERFORM-SUBCATEGORIZATION-CHECK (STACKS)
  (LET ((TEMP-HEAD) (COMPLEMENTS NIL) (HEADS NIL) (INTRANSITIVE-HEADS NIL))
    (LOOP FOR STACK IN STACKS DO
      ;; Collect complement symbols, transitive heads and intransitive heads.
      (LOOP FOR ELT IN STACK DO
        (COND
          ;; Collect complement symbols.
          ((COMPLEMENT? (CAR ELT)) (PUSH (CDR ELT) COMPLEMENTS))
          ;; Collect transitive heads (i.e., ones with complements).
          ((SETQ TEMP-HEAD (COMPLEMENT-PRESENT? ELT)) (PUSH TEMP-HEAD HEADS))
          ;; Collect intransitive heads (i.e., ones with no complements).
          ((SETQ TEMP-HEAD (COMPLEMENT-NOT-PRESENT? ELT))
           (PUSH TEMP-HEAD INTRANSITIVE-HEADS))))
      ;; Iterate over transitive heads dropping subcategorization information
      ;; into each one (unless the information is already there).
      (LOOP FOR HEAD IN HEADS FOR COMPLEMENT IN COMPLEMENTS DO
        (ADD-SUBCAT-INFO HEAD COMPLEMENT))
      ;; Iterate over intransitive heads ensuring that INTRANS is dropped into
      ;; the subcat slot.
      (LOOP FOR INTRANS-HEAD IN INTRANSITIVE-HEADS DO
        (MAKE-INTRANSITIVE INTRANS-HEAD)))
    STACKS))

;;; Add-subcat-info locates the head (symbol) in the stack and drops the
;;; subcategorization information into place.
(DEFUN ADD-SUBCAT-INFO (HEAD COMPLEMENTS)
  (LET ((ARGUMENT-STRUCTURE
          (APPLY
            '(LAMBDA (LIST)
               (LIST (REMOVE-ENDING (CAR LIST) '-MAX))) COMPLEMENTS)))
    ;; Fill the SUBCATEGORIZATION and COMPLEMENTS-FILLED slot of the head.
    (UNLESS (NODE-COMPLEMENTS-FILLED HEAD)
      (SETF (NODE-COMPLEMENTS-FILLED HEAD) T)
      (SETF (NODE-SUBCATEGORIZATION HEAD) ARGUMENT-STRUCTURE))))

;;; Make-intransitive helps perform-subcategorization-check (at PUSH time).
;;; When a head has has no complements being pushed, it is made intransitive.
(DEFUN MAKE-INTRANSITIVE (HEAD)
  ;; Fill the SUBCATEGORIZATION and COMPLEMENTS-FILLED slot of the head.
  (UNLESS (NODE-COMPLEMENTS-FILLED HEAD)
    (SETF (NODE-COMPLEMENTS-FILLED HEAD) T)
    (SETF (NODE-SUBCATEGORIZATION HEAD) 'INTRANS)))
```

F.1.4 Routines for Determining Feature Information and Argument Structure

```
;;; Instantiate-features is called when at SCAN time.  The node corresponding to
;;; the category of the scanned word is filled up with the features specified in
;;; the word's lexical entry. (The translation and word slots of the node are
;;; also filled.)  Furthermore, the argument structure for the word is
;;; established:  internal and external arguments are associated with the head
;;; node.
(DEFUN INSTANTIATE-FEATURES (SOURCE-LANGUAGE TARGET-LANGUAGE FEATURES NODE WORD)
  (LET ((NODE-SUBCAT (GET-SUBCATEGORIZATION FEATURES))  ; internal theta roles
        (NODE-EXTERNAL (GET-EXTERNAL FEATURES)))        ; external theta roles
    ;; Set the translation slot of the node with target-language translation of
    ;; the word.
    (SETF (NODE-TRANSLATION NODE)
          (COPY-TREE
            (GET-TRANSLATION SOURCE-LANGUAGE TARGET-LANGUAGE WORD FEATURES)))
    ;; Since we are at a terminal node, set the word of the terminal node to be
    ;; the scanned word.
    (SETF (NODE-WORD NODE) WORD)
    ;; Set appropriate slot according to the FEATURES parameter setting of the
    ;; source language.
    (SET-FEATURES NODE FEATURES *CURRENT-FEATURES)
    ;; Set up internal and external arguments.
    (SET-INTERNAL-AND-EXTERNAL-ARGUMENTS NODE NODE-EXTERNAL NODE-SUBCAT)) NODE)

;;; Get-external retrieves the theta-roles of external arguments from the
;;; features associated with a word.  All permutations of the external argument
;;; frames will be included in the result.
(DEFUN GET-EXTERNAL (FEATURES)
  (LET ((EXTERNAL
          (LOOP FOR ELT IN FEATURES
            ;; When an external argument frame is encountered, collect it.
                WHEN (AND (LISTP ELT) (EQUAL (CAR ELT) 'EXTERNAL))
                  COLLECT ELT)))
    ;; Permute the external arguments.
    (WHEN EXTERNAL
      (LOOP FOR EXT IN EXTERNAL APPEND (PERMUTE-SUBCAT (CDR EXT))))))
```

```
;;; Get-translation retrieves the translation of a word in the target language.
;;; If the target language is the same as the source language, the original
;;; word is returned (in a double list).
(DEFUN GET-TRANSLATION (SOURCE-LANGUAGE TARGET-LANGUAGE WORD FEATURES)
  (LET ((TRANSLATION NIL))
    (IF (EQUAL SOURCE-LANGUAGE TARGET-LANGUAGE)
        (LIST (LIST WORD))              ; source language = target language
        (LOOP FOR ELT IN FEATURES DO    ; else, get the translation
          (WHEN (LISTP ELT)
            (SETQ TRANSLATION (GET-VALUE ELT TARGET-LANGUAGE))
            (UNLESS (NULL TRANSLATION) (RETURN TRANSLATION)))))))

;;; Set-internal-and-external-arguments sets up external arguments and
;;; subcategorization slot of a node corresponding to a scanned word. Note: the
;;; INTRANS feature will be included in the subcategorization frame if it is in
;;; the features.
(DEFUN SET-INTERNAL-AND-EXTERNAL-ARGUMENTS (NODE NODE-EXTERNAL NODE-SUBCAT)
  (LET ((CATEGORY-EXTERNAL) (CATEGORY-SUBCAT) (THETA-EXTERNAL) (THETA-SUBCAT))
    ;; First set up the external roles and cats.
    (WHEN NODE-EXTERNAL
      (MULTIPLE-VALUE-SETQ (CATEGORY-EXTERNAL THETA-EXTERNAL)
        (GET-CATEGORY-EXTERNAL NODE-EXTERNAL))
      (SETF (NODE-EXTERNAL-ROLES NODE) (COPY-TREE THETA-EXTERNAL))
      (SETF (NODE-EXTERNAL-CATS NODE) (COPY-TREE CATEGORY-EXTERNAL)))
    ;; Then set up the internal cats (setting to INTRANS if there aren't any).
    ;; (Only fill the complement information in if COMPLEMENTS-FILLED slot is
    ;; NIL.)
    (IF NODE-SUBCAT
        (UNLESS (NODE-COMPLEMENTS-FILLED NODE)
          (MULTIPLE-VALUE-SETQ
            (CATEGORY-SUBCAT THETA-SUBCAT) (GET-CATEGORY-SUBCAT NODE-SUBCAT))
          (SETF (NODE-SUBCATEGORIZATION NODE) (COPY-TREE CATEGORY-SUBCAT)))
        (SETF (NODE-SUBCATEGORIZATION NODE) 'INTRANS))
    ;; Finally, fill the theta roles (unless intransitive).
    (IF (OR (NULL NODE-SUBCAT) (EQ (CAR NODE-SUBCAT) 'INTRANS))
        (SETF (NODE-THETA-ROLES NODE) (COPY-TREE (CDR THETA-SUBCAT)))
        (SETF (NODE-THETA-ROLES NODE) (COPY-TREE THETA-SUBCAT)))
    (SETF (NODE-COMPLEMENTS-FILLED NODE) T)))

;;; Get-category-external turns an external argument frame into its
;;; corresponding structural frame (via CSR).  The second value returned is the
;;; corresponding theta-roles: if NODE-EXTERNAL is a set of categories, this
;;; value will be NIL; otherwise it will be the same value as SUBCAT.
(DEFUN GET-CATEGORY-EXTERNAL (NODE-EXTERNAL)
  (LET ((CATEGORY-EXTERNAL (WHEN NODE-EXTERNAL (TURN-INTO-CAT NODE-EXTERNAL))))
    (VALUES
      CATEGORY-EXTERNAL
      (WHEN CATEGORY-EXTERNAL
        (IF (EQUAL NODE-EXTERNAL CATEGORY-EXTERNAL) NIL NODE-EXTERNAL)))))
```

F.1.5 Feature Percolation Routines

```
;;; Percolate-features is invoked at POP time to ensure that all constituents
;;; of a completed phrase agree.  Features of Heads are "percolated" to their
;;; maximal projections and then features of Specifiers are checked for
;;; compatibility with maximal projections.  Percolate-features is also called
;;; at SCAN time (by the function Substitute-scanned-node, to percolate
;;; features of a scanned element up to its superior node) and at PUSH time (by
;;; the function Check-trace-links, to percolate features of head traces up to
;;; their maximal projections).
(DEFUN PERCOLATE-FEATURES (LIST-OF-NODES)
  (LET ((HEAD-CATEGORY) (HEAD-NODE) (HEAD-NODE-INCOMPLETE NIL)
        (SPEC-NODE) (SPEC-NODE-INCOMPLETE NIL)
        (SUPERIOR-NODE (CAR LIST-OF-NODES)))
    ;; Propagate Head features up to the maximal projection (if there is a
    ;; head), and check Spec-Max compatibility.  If either the Spec or the Head
    ;; has not been completed, then the compatibility check is not done.
    (WHEN (SETQ HEAD-CATEGORY
                (CAR (MEMBER (NODE-CAT SUPERIOR-NODE)
                             *CURRENT-BASIC-CATEGORIES :TEST #'EQ)))
      ;; Find the Head and determine whether or not it has been completed.
      (MULTIPLE-VALUE-SETQ (HEAD-NODE HEAD-NODE-INCOMPLETE)
        (FIND-HEAD LIST-OF-NODES HEAD-CATEGORY))
      ;; Find the Spec and determine whether or not it has been completed.
      (MULTIPLE-VALUE-SETQ (SPEC-NODE SPEC-NODE-INCOMPLETE)
        (FIND-SPEC LIST-OF-NODES HEAD-CATEGORY))
      (WHEN HEAD-NODE
        ;; If there is a Head node, check that both the Spec and Head are
        ;; complete.  If complete, propagate Head features to maximal
        ;; projection.  Then check Spec features for compatibility with maximal
        ;; projection (if there is a Spec). Toss the list of nodes if the
        ;; spec-max check fails.
        (UNLESS (OR HEAD-NODE-INCOMPLETE SPEC-NODE-INCOMPLETE)
          ;; Propagate head up to max.
          (PROPAGATE-FEATURES HEAD-NODE SUPERIOR-NODE)
          ;; Now check spec-max agreement (propagating updated information to
          ;; spec).
          (WHEN SPEC-NODE
            (UNLESS
              (CHECK-SPEC-MAX-AGREEMENT SUPERIOR-NODE SPEC-NODE)
              (SETQ LIST-OF-NODES 'FAILURE))))))
    ;; Return the list of nodes after percolation is complete.
    LIST-OF-NODES))
```

```
;;; Propagate-features percolates the features of an inferior node up to a
;;; superior node.  These features include gender, person, number, tense, case,
;;; role, language-particular features and complement information.
(DEFUN PROPAGATE-FEATURES (INFERIOR-NODE SUPERIOR-NODE)
  (SETF (NODE-GENDER SUPERIOR-NODE) (COPY-LIST (NODE-GENDER INFERIOR-NODE)))
  (SETF (NODE-PERS SUPERIOR-NODE) (COPY-LIST (NODE-PERS INFERIOR-NODE)))
  (SETF (NODE-NUMBER SUPERIOR-NODE) (COPY-LIST (NODE-NUMBER INFERIOR-NODE)))
  (SETF (NODE-TENSE SUPERIOR-NODE) (COPY-LIST (NODE-TENSE INFERIOR-NODE)))
  (SETF (NODE-LANGUAGE-PARTICULAR-FEATURES SUPERIOR-NODE)
        (COPY-LIST (NODE-LANGUAGE-PARTICULAR-FEATURES INFERIOR-NODE)))
  (SETF (NODE-CASE SUPERIOR-NODE)
        (UNION (NODE-CASE SUPERIOR-NODE) (NODE-CASE INFERIOR-NODE)))
  (UNLESS (NODE-COMPLEMENTS-FILLED SUPERIOR-NODE)
    (SETF (NODE-COMPLEMENTS-FILLED SUPERIOR-NODE)
          (NODE-COMPLEMENTS-FILLED INFERIOR-NODE)))
  (SETF (NODE-ROLE SUPERIOR-NODE) (COPY-TREE (NODE-ROLE INFERIOR-NODE))))

;;; Check-spec-max-agreement checks that the gender, number, person, tense, of a
;;; specifier agree with those of the maximal projection.  Note: if pro is
;;; found, agreement features are percolated.
(DEFUN CHECK-SPEC-MAX-AGREEMENT (MAX SPEC)
  (AND (CHECK-AGREEMENT 'GENDER (NODE-GENDER MAX) (NODE-GENDER SPEC) MAX SPEC)
       (CHECK-AGREEMENT 'NUMBER (NODE-NUMBER MAX) (NODE-NUMBER SPEC) MAX SPEC)
       (CHECK-AGREEMENT 'PERSON (NODE-PERS MAX) (NODE-PERS SPEC) MAX SPEC)
       (CHECK-AGREEMENT 'TENSE (NODE-TENSE MAX) (NODE-TENSE SPEC) MAX SPEC)))

;;; Check-agreement checks that a particular kind of feature (number, gender,
;;; person, or tense) agrees w.r.t. spec and maximal projections (this takes
;;; place after the head features have been percolated up).  Also, the combined
;;; features are stored in MAX-NODE, and if no agreement features are associated
;;; with SPEC-NODE, they are also stored in SPEC-NODE.
(DEFUN CHECK-AGREEMENT (TYPE MAX SPEC MAX-NODE SPEC-NODE)
  (LET ((CURRENT-MATCH-LIST
          (GET-VALUE *CURRENT-MATCH-LISTS
                     (INTERN (FORMAT NIL "~s" TYPE) :KEYWORD)))
        (NEW-FEATURE-LIST NIL))
    (IF (SETQ NEW-FEATURE-LIST (CHECK-MATCH-LIST CURRENT-MATCH-LIST MAX SPEC))
        (UPDATE-MAX-AND-SPEC MAX-NODE SPEC-NODE NEW-FEATURE-LIST TYPE)
        (FORMAT T "~&Warning: ~a does not agree." TYPE) NIL)))
```

```
;;; Update-max-and-spec ensures that a specifier receives agreement features
;;; (if it does not have them already), and that a maximal projection receives
;;; combined spec-head features.
(DEFUN UPDATE-MAX-AND-SPEC (MAX-NODE SPEC-NODE FEATURES TYPE)
  (ZL::SELECTQ TYPE
    (GENDER
      (UNLESS (NODE-GENDER SPEC-NODE) (SETF (NODE-GENDER SPEC-NODE) FEATURES))
      (SETF (NODE-GENDER MAX-NODE) FEATURES))
    (NUMBER
      (UNLESS (NODE-NUMBER SPEC-NODE) (SETF (NODE-NUMBER SPEC-NODE) FEATURES))
      (SETF (NODE-NUMBER MAX-NODE) FEATURES))
    (PERS
      (UNLESS (NODE-PERS SPEC-NODE) (SETF (NODE-PERS SPEC-NODE) FEATURES))
      (SETF (NODE-PERS MAX-NODE) FEATURES))
    (TENSE (SETF (NODE-TENSE MAX-NODE) FEATURES))))

;;; Loop through the match-lists checking whether the first features (spec)
;;; match the second features (max).  A combination of the max and spec
;;; features is returned.
(DEFUN CHECK-MATCH-LIST (CURRENT-MATCH-LIST MAX-FULL SPEC-FULL)
  (COND ((NULL MAX-FULL) (IF (NULL SPEC-FULL) 'EMPTY SPEC-FULL))
        ((OR (NULL SPEC-FULL) (SET-EQUAL-P MAX-FULL SPEC-FULL)) MAX-FULL)
        (T (LOOP FOR MATCH IN CURRENT-MATCH-LIST DO
             (WHEN (AND (INTERSECTION SPEC-FULL (CAR MATCH))
                        (INTERSECTION MAX-FULL (SECOND MATCH)))
               (RETURN (OR (INTERSECTION SPEC-FULL MAX-FULL)
                           (UNION SPEC-FULL MAX-FULL))))))))

;;; Percolate-tense-and-agr is called at POP time.  If the item being popped is
;;; V-MAX, then its features need to be percolated up to INFL so that HEAD-SPEC
;;; matching will operate correctly.
(DEFUN PERCOLATE-TENSE-AND-AGR (TOS REST-OF-STACK)
  (LET ((INFL-NODES NIL))
    (WHEN (EQ (MAXIMAL-PROJECTION? (NODE-CAT (CAR TOS))) 'V)
      ;; Locate infl and all its projections.
      (LOOP FOR ELT IN REST-OF-STACK DO
        (WHEN (EQ (NODE-CAT (CAR ELT)) 'I-MAX)
          (RETURN (SETQ INFL-NODES (LOCATE-CAT ELT 'I)))))
      ;; Propagate to infl and all its projections.
      (LOOP FOR INFL-NODE IN INFL-NODES DO
        (PROPAGATE-FEATURES (CAR TOS) INFL-NODE)) T)))
```

F.1.6 Structure-Modification Routines

```
;;; Generate-correct-adjunctions-and-specifiers is called during the generation
;;; stage, just prior to executing move-alpha.  The structure is modified to
;;; accommodate the source language by: removing structure unavailable to the
;;; source language (Remove-unavailable); modifying positions of adjuncts
;;; (Modify-position); and allowing the occurrence of elements that are
;;; available in the source language (Allow-occurrence).
(DEFUN GENERATE-CORRECT-ADJUNCTIONS-AND-SPECIFIERS (TREE BASE-SPECS-AND-ADJUNCTS)
  (LET ((NEW-TREE TREE) (RESULTS NIL))
    (LOOP FOR BASE IN BASE-SPECS-AND-ADJUNCTS DO
      (SETQ NEW-TREE (REMOVE-UNAVAILABLE (MODIFY-POSITION NEW-TREE)))
      FINALLY (PUSH NEW-TREE RESULTS))
    (LOOP FOR BASE IN BASE-SPECS-AND-ADJUNCTS DO
      (SETQ RESULTS (APPEND RESULTS (ALLOW-OCCURRENCE BASE RESULTS))))))

;;; Generate-correct-constituent-order is called at generation time in order to
;;; update the structure to be consistent with the constituent order of the
;;; target language.  (Note: no modification is necessary if the source and
;;; target languages have the same constituent order.)
(DEFUN GENERATE-CORRECT-CONSTITUENT-ORDER (TREES ORDER)
  (UNLESS
    ;; Only change the order if it is different from the source language.
    (EQUAL ORDER (GET *CONSTITUENT-ORDER *SOURCE-LANGUAGE))
    ;; Modify each tree.
    (LOOP FOR TREE IN TREES COLLECT (MODIFY-ORDER ORDER TREE))))

;;; Modify-order updates the structure of a single tree to be consistent with the
;;; constituent order of the target language.
(DEFUN MODIFY-ORDER (ORDER TREE)
  (COND ((ATOM TREE) TREE)
        ((AND (MAXIMAL-PROJECTION? (NODE-CAT (CAR TREE)))
              (NOT (ADJUNCTION-STRUCTURE? TREE)))
         ;; Change head-spec-comp ordering.
         (APPEND (LIST (CAR TREE))
                 (MODIFY-ORDER ORDER (GET-CONSTITUENT (FIRST ORDER) TREE))
                 (MODIFY-ORDER ORDER (GET-CONSTITUENT (SECOND ORDER) TREE))
                 (MODIFY-ORDER ORDER (GET-CONSTITUENT (THIRD ORDER) TREE))))
        ;; Go down next level to modify order recursively.
        (T (CONS (MODIFY-ORDER ORDER (CAR TREE)) (MODIFY-ORDER ORDER (CDR TREE))))))

;;; Get-constituent retrieves a particular constituent of a phrase (specifier,
;;; head or complement).
(DEFUN GET-CONSTITUENT (TYPE PHRASE)
  (LOOP FOR ELT IN (CDR PHRASE) DO
    (WHEN (APPLY TYPE (CAR ELT)) (RETURN ELT))))
```

F.2 θ-Theory

The θ module has the job of assigning a structural realization to θ-roles (CSR) during both parsing and generation. At POP time and generation time, θ-role assignment, θ-role transmission, and θ-Criterion checking are performed. Finally, during thematic substitution, θ-role matching is performed.

F.2.1 CSR Mapping

```
;;; CSR turns a role into its canonical structural representation.
(DEFUN CSR (ROLE)
  (LET ((STRUCTURAL-SYMBOL
          (GET-VALUE *CURRENT-CANONICAL-SEMANTIC-MAPPINGS (MAKE-KEYWORD ROLE))))
    (IF (NULL STRUCTURAL-SYMBOL) ROLE STRUCTURAL-SYMBOL)))
```

F.2.2 θ-Role Assignment Routines

```
;;; Perform-theta-assignment assigns theta roles to both external and internal
;;; arguments of a node at POP time and generation time.  Note:  the
;;; theta-criterion is checked so that each element that requires a theta-role
;;; is assigned such a role.
(DEFUN PERFORM-THETA-ASSIGNMENT (TOS NODE-TREE)
  (LET* ((HEAD (FIND-HEAD TOS (NODE-CAT (CAR TOS))))
         ;; Find the element that will assign theta to specifier of the head.
         (SPEC-ASSIGNER-NODE (FIND-SPEC-ASSIGNER-NODE TOS HEAD))
         ;; Find the element that will assign theta to the complement of the
         ;; head.
         (COMP-ASSIGNER-NODE (FIND-COMP-ASSIGNER-NODE TOS HEAD)))
    ;; First assign theta roles to spec if there is one.
    (ASSIGN-TO-ARG 'SPEC SPEC-ASSIGNER-NODE TOS NODE-TREE)
    ;; Next assign theta roles to Comp if there is one.
    (ASSIGN-TO-ARG 'COMP COMP-ASSIGNER-NODE TOS NODE-TREE)))

;;; Assign-to-arg assigns theta-roles to either a comp or a spec of a maximal
;;; projection, according to those required by an assigner node.
(DEFUN ASSIGN-TO-ARG (TYPE ASSIGNER TOS NODE-TREE)
  (LET ((ASSIGNEES))
    (IF (SETQ ASSIGNEES (LOCATE-SPEC-OR-COMP TOS TYPE))
        ;; If there is an argument, make sure it has a theta-role.
        (IF ASSIGNER
            ;; If there is an assigner, discharge its theta role to the
            ;; argument.
            (ASSIGN-THETA ASSIGNEES ASSIGNER TYPE)
            ;; Otherwise, make sure there are no NP's without theta-roles.
            (CHECK-THETA-CRITERION NODE-TREE ASSIGNEES NIL TYPE))
        ;; Argument not found: Make sure there are no unassigned theta-roles
        ;; (i.e., there could be an assigner node which did not discharge its
        ;; theta-roles to a spec or a comp).
        (CHECK-THETA-CRITERION NODE-TREE NIL ASSIGNER TYPE)))

;;; Assign-theta checks that features of an element match the features
;;; corresponding to a theta role before the theta role is assigned.  By the
;;; visibility condition, the theta-role cannot be assigned unless the element
;;; has been assigned case.
(DEFUN ASSIGN-THETA (ASSIGNEES ASSIGNER TYPE)
  (LET ((THETAS
          (IF (EQ TYPE 'EXTERNAL)
              (NODE-EXTERNAL-ROLES ASSIGNER)
              (NODE-THETA-ROLES ASSIGNER))))
    (LOOP FOR ASSIGNEE IN ASSIGNEES FOR THETA IN THETAS DO
      (IF (SET-DIFFERENCE
            (CDR THETA) (NODE-LANGUAGE-PARTICULAR-FEATURES ASSIGNEE))
          ;; Feature mismatch: return failure.
          (RETURN 'FAILURE)
          ;; Successful feature match: assign the theta role unless the
          ;; assignee is not case-marked.
          (IF (CHECK-VISIBILITY-CONDITION ASSIGNEE)
              ;; Assignee is not case-marked: visibility condition failure.
              (RETURN 'FAILURE)
              ;; Assignee is case-marked: Assign a role to the node.  If there
              ;; is a role transmission error (to the trace or antecedent),
              ;; return FAILURE.
              (SETF (NODE-ROLE ASSIGNEE) (COPY-TREE THETA))
              (WHEN (TRANSMIT-ROLE-TO-ANTECEDENT ASSIGNEE)
                (RETURN 'FAILURE)))))))

;;; Check-visibility-condition is called just prior to assigning a theta-role.
;;; The node must first satisfy the visibility condition: it must have case.
;;; (This obviates the need for the case filter.)
(DEFUN CHECK-VISIBILITY-CONDITION (NODE)
  (UNLESS (NODE-CASE NODE) (V-C-VIOLATION) T))

;;; V-C-Violation prints out a visibility condition violation message.
(DEFUN V-C-VIOLATION NIL (FORMAT T "~&VIOLATION OF VISIBILITY CONDITION.") T)
```

F.2.3 θ-Role Transmission Routines

```
;;; Transmit-role-to-antecedent propagates the role of a trace up to its
;;; antecedent (and onward until all antecedents receive the theta-role).
(DEFUN TRANSMIT-ROLE-TO-ANTECEDENT (NODE)
  (LET ((CLASH NIL) (ANTECEDENT-NODE))
    ;; Iterate until no more antecedents or there is a role clash.
    (LOOP
      UNTIL
        (OR CLASH (NOT (SETQ ANTECEDENT-NODE (NODE-TRACE ANTECEDENT-NODE))))
      DO
     (IF (AND (NODE-ROLE ANTECEDENT-NODE)
              (NOT (EQUAL (NODE-ROLE NODE) (NODE-ROLE ANTECEDENT-NODE))))
         ;; Role clash.
         (SETQ CLASH T)
         ;; Assign role if case is assigned.
         (WHEN (NODE-CASE ANTECEDENT-NODE)
           (SETF (NODE-ROLE ANTECEDENT-NODE) (COPY-LIST (NODE-ROLE NODE)))))
         FINALLY (RETURN CLASH))))

;;; Theta-transmit attempts to transmit the theta-role of a clitic to and
;;; NP-node (for languages that require this rule). According to the
;;; theta-transmission rule, the cases of the clitic and NP-node must first be
;;; matched before transmission can occur.
(DEFUN THETA-TRANSMIT (NP-NODE STACK)
  ;; First test to see if language requires clitic-NP theta-transmission, and
  ;; then make sure the NP-node has case assigned to it (visibility condition).
  (WHEN (AND *CLITIC-THETA-TRANSMIT (NODE-CASE NP-NODE))
    (TRANSMIT-THETA-FROM-CLITIC NP-NODE (NODE-CASE NP-NODE) STACK)))

;;; Transmit-theta-from-clitic determines all possible clitic candidates and
;;; then chooses the right one by checking feature agreement.
(DEFUN TRANSMIT-THETA-FROM-CLITIC (NP-NODE CASE STACK)
  (LET ((CLITIC-CANDIDATES (FIND-CLITICS-WITH-CASE-AND-ROLE CASE STACK)))
    ;; Set up government so that s-government tests work.
    (SET-UP-GOVERNMENT STACK)
    ;; Examine each clitic candidate to see if it matches the NP-node features.
    (LOOP FOR CLITIC IN CLITIC-CANDIDATES DO
      ;; Before transmitting theta-role, check that the clitic and NP-node are
      ;; not c-governed by the same node (since this would mean the verb is
      ;; assigning objective case twice, an impossibility due to absorption of
      ;; s-government).
      (UNLESS
        (INTERSECTION (NODE-C-GOVERNED? NP-NODE) (NODE-C-GOVERNED? CLITIC))
        (WHEN (CHECK-CLITIC-NP-AGREEMENT CLITIC NP-NODE)
          ;; Successful match.
          (SETF (NODE-ROLE NP-NODE) (COPY-TREE (NODE-ROLE CLITIC)))
          (RETURN T))))))
```

```
;;; Find-clitics-with-case-and-role locates all clitic candidates that have the
;;; right case and are theta-marked.
(DEFUN FIND-CLITICS-WITH-CASE-AND-ROLE (CASE STACK)
  (COND ((NULL STACK) NIL)
        ((ATOM STACK)
         (WHEN (AND (MEMBER (NODE-CAT STACK) *CURRENT-CLITICS :TEST #'EQ)
                    (NODE-CASE STACK) (NODE-ROLE STACK)) (LIST STACK)))
        (T (APPEND (FIND-CLITICS-WITH-CASE-AND-ROLE CASE (CAR STACK))
                   (FIND-CLITICS-WITH-CASE-AND-ROLE CASE (CDR STACK))))))

;;; Check-clitic-np-agreement ensures that cases match and that all other
;;; features are compatible.
(DEFUN CHECK-CLITIC-NP-AGREEMENT (CLITIC NP-NODE)
  (AND
    (INTERSECTION (NODE-CASE CLITIC) (NODE-CASE NP-NODE))
    (COMPATIBLE-FEATURES (NODE-GENDER CLITIC) (NODE-GENDER NP-NODE))
    (COMPATIBLE-FEATURES (NODE-PERS CLITIC) (NODE-PERS NP-NODE))
    (COMPATIBLE-FEATURES (NODE-NUMBER CLITIC) (NODE-NUMBER NP-NODE))
    (COMPATIBLE-FEATURES
      (NODE-LANGUAGE-PARTICULAR-FEATURES CLITIC)
      (NODE-LANGUAGE-PARTICULAR-FEATURES NP-NODE))))

;;; Compatible-features tests for matching features (via intersection).
(DEFUN COMPATIBLE-FEATURES (CLITIC-FEATURES NP-FEATURES)
  (COND ((NULL CLITIC-FEATURES) NP-FEATURES)
        ((NULL NP-FEATURES) CLITIC-FEATURES)
        (T (INTERSECTION CLITIC-FEATURES NP-FEATURES :TEST #'EQUAL))))
```

F.2.4 θ-Criterion Routines

```
;;; Check-theta-criterion is called if there is an assigner but not assignee,
;;; or an assignee but no assigner.  It returns 'FAILURE if an assigner has
;;; theta-roles that are not discharged, or if an assignee does not receive
;;; theta-roles.
(DEFUN CHECK-THETA-CRITERION (NODE-TREE ASSIGNEES ASSIGNER TYPE)
  (IF ASSIGNEES
      ;; If there are arguments with no theta-roles, then fail.
      (LOOP FOR ASSIGNEE IN ASSIGNEES DO
        (WHEN (THETA-VIOLATION ASSIGNEE NODE-TREE)
          (T-C-VIOLATION) (RETURN 'FAILURE)))
      ;; If there is an assigner that has not discharged its internal or
      ;; external role, then fail.
      (IF (EQ TYPE 'EXTERNAL)
          (WHEN (NODE-EXTERNAL-ROLES ASSIGNER) (T-C-VIOLATION) 'FAILURE)
          (WHEN (NODE-THETA-ROLES ASSIGNER) (T-C-VIOLATION) 'FAILURE))))

;;; Theta-violation checks NP and clitics in a TOS to see that they are all
;;; theta-marked.  It returns T if there is an unmarked NP or clitic.
(DEFUN THETA-VIOLATION (NODE STACK)
  (AND (NOT (NODE-ROLE NODE))
       (OR
         ;; Check for un-marked NP.
         (AND (EQ (NODE-CAT NODE) 'N-MAX)
              ;; First check if a theta role can't somehow be transmitted
              ;; (possibly via a clitic transmission rule).
              (NOT (THETA-TRANSMIT NODE STACK)))
         ;; Check for un-marked Clitic.
         (MEMBER (NODE-CAT NODE) *CURRENT-CLITICS :TEST #'EQ))))

;;; T-C-Violation prints out a theta-criterion violation message.
(DEFUN T-C-VIOLATION NIL (FORMAT T "~&VIOLATION OF THETA CRITERION."))
```

F.2.5 θ-Role Matching Routines

```
;;; Match-translation-and-arguments retrieves the target language translation of
;;; the head and determines the correct argument structure for the target head.
(DEFUN MATCH-TRANSLATION-AND-ARGUMENTS (EXT INT HEAD)
  (LET ((EXTERNAL) (INTERNAL))
    (LOOP FOR TRANS IN (NODE-TRANSLATION HEAD) DO
      (WHEN
        (MULTIPLE-VALUE-SETQ
          (EXTERNAL INTERNAL)
          ;; Test that features of source language arguments match features
          ;; of target language  arguments.
          (ARGUMENT-FEATURES-MATCH EXT INT (GET-ARGUMENTS TRANS)))
        (RETURN (VALUES TRANS EXTERNAL INTERNAL))))))

;;; Get-arguments determines the internal and external arguments of a target
;;; language word by looking at its lexical entry.
(DEFUN GET-ARGUMENTS (TRANSLATION)
  (APPEND (GET-EXTERNAL-ARGS TRANSLATION) (GET-INTERNAL-ARGS TRANSLATION)))

;;; Argument-features-match finds a unique one-one matching between
;;; source-language arguments and target language arguments by checking the
;;; features of each one.  For example, the source language argument structure:
;;; <external:[I agent animate] internal:[the book goal]> matches the target
;;; language argument structure: <internal:[goal animate] external:[agent]>
;;; since there is a one-one mapping between the features of both.
(DEFUN ARGUMENT-FEATURES-MATCH
      (SOURCE-INTERNAL SOURCE-EXTERNAL POSSIBLE-TARGET-ARGUMENTS)
  (LET ((ROLE) (FEATURES) (TARGET-CANDIDATE) (TARGET-CANDIDATES NIL))
    (LOOP FOR SOURCE-ARG IN (APPEND SOURCE-INTERNAL SOURCE-EXTERNAL) DO
      ;; Extract role and features of source argument.  Then pick a matching
      ;; candidate from the argument frame of the target language head.
      (SETQ ROLE (NODE-ROLE SOURCE-ARG)
            FEATURES (NODE-LANGUAGE-PARTICULAR-FEATURES SOURCE-ARG)
            TARGET-CANDIDATE
            (PICK-ARG SOURCE-ARG ROLE FEATURES POSSIBLE-TARGET-ARGUMENTS))
      (PUSH TARGET-CANDIDATE TARGET-CANDIDATES))
    ;; Determine unique pairing of source language arguments to target language
    ;; arguments.
    (LOCATE-UNIQUE-ARG-STRUCTURE
      TARGET-CANDIDATES POSSIBLE-TARGET-ARGUMENTS)))
```

```
;;; Set-up-argument-positioning determines the positioning of internal and
;;; external arguments according to the requirements of the target language
;;; head.
(DEFUN SET-UP-ARGUMENT-POSITIONING (MAX TARGET-EXTERNAL TARGET-INTERNAL)
  ;; Place external arguments.
  (LOOP FOR TARGET-ARG IN TARGET-EXTERNAL DO
    (PLACE-ARGUMENT-EXTERNAL TARGET-ARG MAX))
  ;; Place internal arguments.
  (LOOP FOR TARGET-ARG IN TARGET-INTERNAL DO
    (PLACE-ARGUMENT-INTERNAL TARGET-ARG MAX)))

;;; Set-up-structural-realization determines the structure of internal and
;;; external arguments according to the requirements of the target language
;;; head.
(DEFUN SET-UP-STRUCTURAL-REALIZATION (MAX TARGET-EXTERNAL TARGET-INTERNAL)
  (LET ((EXTERNAL-STRUCTURE (MAPCAR 'CSR TARGET-EXTERNAL))
        (INTERNAL-STRUCTURE (MAPCAR 'CSR TARGET-INTERNAL)))
    ;; Modify the structure of external arguments.
    (UPDATE-EXTERNAL-STRUCTURE MAX EXTERNAL-STRUCTURE)
    ;; Modify the structure of internal arguments.
    (UPDATE-INTERNAL-STRUCTURE MAX INTERNAL-STRUCTURE)))
```

F.3 Government Theory

The Government module is accessed at POP time and during generation to set up government relations and test for certain types of government relations.

F.3.1 Routines for Setting up Government Relations

```
;;; Set-up-government is invoked at POP time and generation time just prior to
;;; ECP checking and CASE marking (and also during theta-role transmission (for
;;; clitics)).
(DEFUN SET-UP-GOVERNMENT (NODE-STACK)
  (FIND-GOVERNORS NODE-STACK 'C)          ; c-government
  (FIND-GOVERNORS NODE-STACK 'S)          ; s-government
  (FIND-GOVERNORS NODE-STACK 'PROPER))    ; proper government
```

```
;;; Find-governors locates governors of all types (C, S, and Proper).  It then
;;; sets up all the nodes that are governed by the governors that it has found.
(DEFUN FIND-GOVERNORS (NODE-STACK TYPE)
  (LET ((HEAD-NODE) (MAX-CAT))
    ;; Iterate over each stack item setting up government relations.
    (LOOP FOR STACK-ITEM IN NODE-STACK DO
      ;; Governor is a head.
      (WHEN (SETQ HEAD-NODE (FIND-HEAD STACK-ITEM MAX-CAT))
        (SET-GOVERNORS HEAD-NODE STACK-ITEM TYPE))
      ;; Go through other elements too (specs, adjunctions, complements).
      (LOOP FOR NEW-NODE-STACK IN (CDR STACK-ITEM) DO
        (FIND-GOVERNORS NEW-NODE-STACK TYPE)))))

;;; Set-governors sets up all nodes that are c-governed, s-governed and
;;; properly governed by a head.
(DEFUN SET-GOVERNORS (HEAD-NODE NODE-STACK TYPE)
  (COND ((EQ TYPE 'C) (SET-C-GOVERNORS HEAD-NODE NODE-STACK))
        ((EQ TYPE 'S) (SET-S-GOVERNORS HEAD-NODE))
        (T (SET-PROPER-GOVERNORS HEAD-NODE))))

;;; Set-c-governors sets up all nodes c-commanded by a governing head node.
;;; (This includes specifiers, complements and elements adjoined to lexical
;;; categories (e.g., clitics)).
(DEFUN SET-C-GOVERNORS (HEAD-NODE NODE-STACK)
  (WHEN (GOVERNOR? HEAD-NODE)
    ;; Only set up c-government if the head-node is a governor.
    (LOOP FOR POTENTIAL-GOVERNEE IN (CDR NODE-STACK) DO
      (IF (OR (SPECIFIER? (NODE-CAT (CAR POTENTIAL-GOVERNEE)))
              (COMPLEMENT? (NODE-CAT (CAR POTENTIAL-GOVERNEE))))
          ;; Case 1: Set slots corresponding to specifiers and complements of
          ;; the head node.
          (LOOP FOR MAX IN (CDR POTENTIAL-GOVERNEE) DO
            (SETF (NODE-C-GOVERNED? (CAR MAX))  ; set up the c-governed node
                  (CONS HEAD-NODE (NODE-C-GOVERNED? (CAR MAX))))
            (SETF (NODE-C-GOVERNS? HEAD-NODE)   ; set up the c-governor
                  (CONS (CAR MAX) (NODE-C-GOVERNS? HEAD-NODE))))
          ;; Case 2: Set slots corresponding to adjuncts of the head node.
          (SETF (NODE-C-GOVERNED?
                  (CAR POTENTIAL-GOVERNEE))     ; set up c-governed node
                (CONS HEAD-NODE (NODE-C-GOVERNED? (CAR POTENTIAL-GOVERNEE))))
          (SETF (NODE-C-GOVERNS? HEAD-NODE)     ; set up c-governor
                (CONS (CAR POTENTIAL-GOVERNEE)
                      (NODE-C-GOVERNS? HEAD-NODE))))))
```

```
;;; Set-s-governors sets up all nodes s-governed by a c-commanding node (i.e.,
;;; it checks if there is a unique subcategorization pairing for c-commanded
;;; nodes in the subcategorization frame of the c-commanding node.)
(DEFUN SET-S-GOVERNORS (HEAD-NODE)
  (LET ((ROLES (NODE-THETA-ROLES HEAD-NODE)) (S-GOVERNED))
    ;; Make sure there is an exact match between c-commandees and subcategorized
    ;; elements.  The nodes that match are s-governed.
    (LOOP FOR ROLE-SET IN ROLES DO
      (WHEN (SETQ S-GOVERNED (UNIQUE-PAIRING ROLE-SET (NODE-C-GOVERNS? HEAD-NODE)))
        (SETF (NODE-S-GOVERNS? HEAD-NODE) S-GOVERNED)
        (LOOP FOR GOVERNED-NODE IN S-GOVERNED DO
          (SETF (NODE-S-GOVERNED? GOVERNED-NODE)
                (CONS HEAD-NODE (NODE-S-GOVERNED? GOVERNED-NODE))))))))

;;; Unique-pairing determines whether there is a unique subcategorization
;;; pairing corresponding to the c-governed-nodes.
(DEFUN UNIQUE-PAIRING (ROLE-SET C-GOVERNED-NODES)
  (LET ((MATCH NIL) (NEW-C-GOVERNED))
    (OR
      ;; First check whether the categories match before checking the clitic
      ;; case.
      (SET-EQUAL-P
        (MAPCAR '(LAMBDA (X) (CSR (CAR X))) ROLE-SET)
        (MAPCAR '(LAMBDA (X) (MAXIMAL-PROJECTION? (NODE-CAT X))) C-GOVERNED-NODES)
        :TEST #'EQ)
      ;; If the above fails, check that there is a one-one match between roles
      ;; and c-governed elements (including clitics) and that all features
      ;; match.
      (WHEN *CURRENT-CLITICS
        (LOOP FOR ROLE IN ROLE-SET DO
          ;; Since clitics absorb s-government, order them so that they are
          ;; first in the list of c-governed elements.
          (SETQ NEW-C-GOVERNED
                (PUT-CLITICS-FIRST
                  (SET-DIFFERENCE C-GOVERNED-NODES MATCH :TEST #'EQ)))
          (UNLESS
            (LOOP FOR C-GOVERNED IN C-GOVERNED-NODES DO
              ;; Check feature matching.
              (IF (SET-EQUAL-P
                    (NODE-LANGUAGE-PARTICULAR-FEATURES C-GOVERNED)
                    (CDR ROLE))
                  ;; Node matches.
                  (PUSH C-GOVERNED MATCH)
                  ;; Node mismatch.
                  (RETURN NIL)))
            (RETURN NIL))
              FINALLY (RETURN MATCH)))))
```

```
;;; Put-clitics-first prioritizes the nodes so that clitics come first.  (This
;;; is because clitics absorb s-government.)
(DEFUN PUT-CLITICS-FIRST (NODES)
  (LET ((CLITICS NIL) (NON-CLITICS NIL))
    ;; Collect clitics and non-clitics.
    (LOOP FOR NODE IN NODES DO
      (IF (MEMBER (NODE-CAT NODE) *CURRENT-CLITICS :TEST #'EQ)
          (PUSH NODE CLITICS)
          (PUSH NODE NON-CLITICS))
      ;; Put clitics ahead of non-clitics in the final result.
      FINALLY (APPEND CLITICS NON-CLITICS))))

;;; Set-proper-governors sets up all nodes properly governed by a proper
;;; governor.
(DEFUN SET-PROPER-GOVERNORS (HEAD-NODE)
  (WHEN (PROPER-GOVERNOR? HEAD-NODE)
    (LOOP FOR C-GOVERNED-NODE IN (NODE-C-GOVERNS? HEAD-NODE) DO
      (SETF (NODE-PROPER-GOVERNED? C-GOVERNED-NODE)
            (CONS HEAD-NODE (NODE-PROPER-GOVERNED? C-GOVERNED-NODE)))
      (SETF (NODE-PROPER-GOVERNS? HEAD-NODE)
            (CONS C-GOVERNED-NODE (NODE-PROPER-GOVERNS? HEAD-NODE))))))

;;; Governor? determines whether a node is a governor.
(DEFUN GOVERNOR? (NODE)
  (LET ((NODE-CAT (NODE-CAT NODE)))
    (OR
      ;; Lexical governor.
      (MEMBER NODE-CAT
              (SET-DIFFERENCE *CURRENT-GOVERNORS '(AGR) :TEST #'EQ) :TEST #'EQ)
      ;; AGR (Infl with agreement features).
      (AND (EQ NODE-CAT 'I)
           (OR (NODE-PERS NODE) (NODE-NUMBER NODE) (NODE-TENSE NODE))
           (MEMBER 'AGR *CURRENT-GOVERNORS :TEST #'EQ)))))

;;; Proper-governor? determines whether a node is a proper governor.  That is,
;;; it must be a lexical category (not including AGR, unless AGR is rich, i.e.,
;;; unless AGR is NP{i}).
(DEFUN PROPER-GOVERNOR? (NODE)
  (LET ((NODE-CAT (NODE-CAT NODE)))
    ;; A proper governor cannot be a trace.
    (AND (NOT (NODE-TRACE NODE))
         (OR
           ;; Lexical governor.
           (MEMBER NODE-CAT *CURRENT-GOVERNORS :TEST #'EQ)
           ;; AGR (Infl with agreement features).
           (AND *AGR-RICH
                (EQ (NODE-CAT NODE) 'I)
                (OR (NODE-PERS NODE) (NODE-TENSE NODE) (NODE-NUMBER NODE))
                (MEMBER 'AGR *CURRENT-GOVERNORS :TEST #'EQ))))))
```

F.3.2 Routines for Testing Government Relations

```
;;; A node is c-governed if it is c-commanded by a governor.
(DEFUN C-GOVERNED? (NODE) (NODE-C-GOVERNED? NODE))

;;; A node is s-governed if it is c-governed and there is a unique
;;; subcategorization pairing for the node in the subcategorization frame of
;;; the c-governing node.
(DEFUN S-GOVERNED? (NODE) (NODE-S-GOVERNED? NODE))

;;; Proper-governed? determines whether a node is properly-governed either by a
;;; proper-governor, or by chain conditions (if the language has them).
(DEFUN PROPER-GOVERNED? (NODE)
  (OR
    ;; Non-trace automatically properly governed.
    (NOT (NODE-TRACE NODE))
    ;; Properly governed by c-commanding governor.
    (NODE-PROPER-GOVERNED? NODE)
    ;; Properly governed by c-commanding NP{i}.
    (AND (NODE-TRACE NODE)
         (MEMBER (NODE-TRACE NODE) (NODE-C-GOVERNED? NODE) :TEST #'EQ))
    ;; Chain conditions must be satisfied (if applicable).
    (CHAIN-CONDITIONS-SATISFIED? NODE)))

;;; Chain-conditions-satisfied? determines whether all elements of the chain
;;; which the trace node is a part of are c-governed.
(DEFUN CHAIN-CONDITIONS-SATISFIED? (TRACE-NODE)
  (AND *CHAIN-CONDITIONS
       (NODE-C-GOVERNED? TRACE-NODE)
       (ANTECEDENTS-C-GOVERNED? (NODE-TRACE TRACE-NODE))
       (TRACES-C-GOVERNED? (NODE-ANTECEDENT TRACE-NODE))))
```

F.4 Case Theory

Case assignment is performed at POP time after government relations have been determined, and at generation time after move-α has executed. Note that the Case Filter need not be included in the Case module since the Visibility Condition of the θ module ensures that all noun phrases are assigned case.

```
;;; Perform-case-assignment assigns case at POP time and generation time.  Each
;;; TOS node and all its c-governees are passed to Assign-case in an attempt to
;;; assign case.
(DEFUN PERFORM-CASE-ASSIGNMENT (TOS NODE-TREE)
  (LET ((NEW-TOS (FLATTEN TOS)) (CASE-ASSIGNED-RESULT))
    ;; Retrieve all assignee-nodes.
    (LOOP FOR NODE1 IN NEW-TOS DO
      ;; Retrieve all assigner-nodes.
      (LOOP FOR NODE2 IN (NODE-C-GOVERNED? NODE1) DO
        ;; Discharge case from assigner to assignee unless there is a case
        ;; mismatch.
        (SETQ CASE-ASSIGNED-RESULT
              (ASSIGN-CASE NODE2 NODE1 NODE-TREE))
        (WHEN (EQ CASE-ASSIGNED-RESULT 'FAILURE) (RETURN 'FAILURE)))
      (WHEN (EQ CASE-ASSIGNED-RESULT 'FAILURE) (RETURN 'FAILURE)))))

;;; Assign-case assigns allows the assignee-node to be assigned by the
;;; assigner-node unless there is a case mismatch.
(DEFUN ASSIGN-CASE (ASSIGNER-NODE ASSIGNEE-NODE NODE-TREE)
  (LET* ((CAT (NODE-CAT ASSIGNER-NODE))
         (ASSIGNMENT-RULE (GET-VALUE *CURRENT-CASE-ASSIGNMENT CAT))
         (ASSIGNEE-CASE (NODE-CASE ASSIGNEE-NODE)))
    ;; Don't do anything unless there is an assignment rule and the condition
    ;; of the rule is satisfied.
    (WHEN (AND ASSIGNMENT-RULE
               (APPLY (CAR ASSIGNMENT-RULE) (LIST ASSIGNER-NODE ASSIGNEE-NODE)))
      ;; Assign case to the node unless there is a case mismatch.  Then transmit
      ;; case to the trace or antecedent.
      (IF (OR (NULL ASSIGNEE-CASE)
              (INTERSECTION ASSIGNEE-CASE (CDR ASSIGNMENT-RULE)))
          ;; Assign case to the node.  If there is a case transmission error
          ;; (to the trace or antecedent), return FAILURE.
          (PROGN
            (SETF (NODE-CASE ASSIGNEE-NODE) (COPY-LIST (CDR ASSIGNMENT-RULE)))
            (WHEN (TRANSMIT-CASE-TO-TRACE-OR-ANTECEDENT ASSIGNEE-NODE)
              (TRANSMISSION-ERROR ASSIGNEE-NODE NODE-TREE)
              'FAILURE))
          ;; Case mismatch: return FAILURE.
          (CASE-MISMATCH ASSIGNEE-NODE NODE-TREE)
          'FAILURE))))
```

```
;;; Transmit-case-to-trace-or-antecedent propagates the case of a node to all
;;; of its antecedents and traces unless there is a case clash.
(DEFUN TRANSMIT-CASE-TO-TRACE-OR-ANTECEDENT (ASSIGNEE)
  (LET ((LAST-ASSIGNEE ASSIGNEE) (NEXT-ASSIGNEE) (CLASH NIL))
    ;; Propagate to antecedents (in a-bar position: What did he see t?).
    (LOOP UNTIL
            (OR CLASH
                (NOT (SETQ NEXT-ASSIGNEE (NODE-TRACE LAST-ASSIGNEE)))) DO
      (IF (AND (NODE-CASE NEXT-ASSIGNEE)
               (NOT (EQUAL (NODE-CASE NEXT-ASSIGNEE)
                           (NODE-CASE LAST-ASSIGNEE))))
          ;; Case clash.
          (SETQ CLASH T)
          ;; No case clash.
          (SETF (NODE-CASE NEXT-ASSIGNEE) (NODE-CASE LAST-ASSIGNEE))
          (SETQ LAST-ASSIGNEE NEXT-ASSIGNEE)))
    ;; Propagate to traces (in a position: "He seemed t to be eating.").
    (SETQ LAST-ASSIGNEE ASSIGNEE)
    (LOOP UNTIL
            (OR CLASH
                (NOT (SETQ NEXT-ASSIGNEE (NODE-ANTECEDENT LAST-ASSIGNEE)))) DO
      (IF (AND (NODE-CASE NEXT-ASSIGNEE)
               (NOT (EQUAL (NODE-CASE NEXT-ASSIGNEE)
                           (NODE-CASE LAST-ASSIGNEE))))
          ;; Case clash.
          (SETQ CLASH T)
          ;; No case clash.
          (SETF (NODE-CASE NEXT-ASSIGNEE) (NODE-CASE LAST-ASSIGNEE))
          (SETQ LAST-ASSIGNEE NEXT-ASSIGNEE)))
    CLASH))
```

F.5 Trace Theory

The Trace module checks ECP at POP time after government relations have been determined, and at generation time after move-α has executed.

```
;;; ECP checks conditions of empty categories at POP time and generation time:
;;; 1. PRO cannot be c-governed; 2. A trace must be properly governed; and
;;; 3. pro must be c-governed (hence properly governed) by AGR.
(DEFUN ECP (TOS)
  (LOOP FOR TOS-ELEMENT IN TOS DO
    (WHEN (EQ (NODE-CAT TOS-ELEMENT) 'N-MAX)
      (COND
        ;; Trace
        ((EQ (NODE-WORD TOS-ELEMENT) 'PRO)
         (WHEN (C-GOVERNED? TOS-ELEMENT)
           (FORMAT T "~&ECP VIOLATION: PRO is governed.")
           (RETURN 'FAILURE)))
        ;; PRO
        ((MEMBER (NODE-WORD TOS-ELEMENT) *CURRENT-TRACES :TEST #'EQ)
         (UNLESS (PROPER-GOVERNED? TOS-ELEMENT)
           (FORMAT T "~&ECP VIOLATION: Trace not properly governed.")
           (RETURN 'FAILURE)))
        ;; pro
        ((EQ (NODE-WORD TOS-ELEMENT) 'SMALL-PRO)
         (UNLESS
           (MEMBER 'I (CONVERT-TO-CATS (NODE-C-GOVERNED? TOS-ELEMENT)))
           (FORMAT
             T "~&ECP VIOLATION: pro is not governed by AGR.")
           (RETURN 'FAILURE)))
        (T NIL)))))
```

F.6 Binding Theory

The Binding module is accessed at POP time and at generation time first to establish A-positions and $\overline{\text{A}}$-positions, and then to check Binding conditions.

```
;;; Set-a-and-a-bar-positions sets up A and A-BAR positions at all levels of a
;;; phrase at POP and generation time.
(DEFUN SET-A-AND-A-BAR-POSITIONS (PHRASE)
  ;; Set up all a-positions of the phrase.
  (LOOP FOR NODE IN (FIND-A-POSITIONS PHRASE) DO
    (SETF (NODE-A-POSITION? NODE) T))
  ;; Set up all a-bar-positions of the phrase.
  (LOOP FOR NODE IN (FIND-A-BAR-POSITIONS PHRASE) DO
    (SETF (NODE-A-BAR-POSITION? NODE) T))
  ;; Set up all a-positions and a-bar-positions of the phrases in complement,
  ;; specifier, and adjunction positions.
  (LOOP FOR ELT IN (CDR PHRASE) DO
    ;; Set up the complements of the phrase.
    (WHEN (AND (LISTP ELT) (COMPLEMENT? (NODE-CAT (FIRST ELT))))
      (LOOP FOR NEW-PHRASE IN (CDR ELT) DO
        (SET-A-AND-A-BAR-POSITIONS NEW-PHRASE)))
    ;; Set up the specifiers of the phrase.
    (WHEN (AND (LISTP ELT) (SPEC? (NODE-CAT (FIRST ELT))))
      (LOOP FOR NEW-PHRASE IN (CDR ELT) DO
        (SET-A-AND-A-BAR-POSITIONS NEW-PHRASE)))
    ;; Set up the adjoined elements (maximal or minimal) of the phrase
    ;; including the element (maximal or minimal) to which adjunction is taking
    ;; place.
    (WHEN (OR
            ;; Maximal adjunction.
            (MAXIMAL-PROJECTION? (NODE-CAT (FIRST ELT)))
            ;; Minimal adjunction.
            (NOT (OR (SPEC? (NODE-CAT (FIRST ELT)))
                     (COMPLEMENT? (NODE-CAT (FIRST ELT))))))
      (SET-A-AND-A-BAR-POSITIONS ELT))))
```

```
;;; Check-binding-conditions ensures that anaphoric traces are bound in their
;;; governing category by an element in a-position, and that referential traces
;;; are bound by an element in a-bar position.
(DEFUN CHECK-BINDING-CONDITIONS (TOS)
  (LET ((ANTECEDENT))
    (LOOP FOR ELT IN (FLATTEN TOS) DO
      (WHEN (AND (EQ (NODE-CAT ELT) 'N-MAX) (SETQ ANTECEDENT (NODE-TRACE ELT)))
        (COND
          ;; Check NP-traces.
          ((ANAPHORIC? ELT)
           (UNLESS (AND (NODE-A-POSITION ANTECEDENT)
                        (MEMBER ANTECEDENT (C-GOVERNED? ELT) :TEST #'EQ))
             (RETURN 'FAILURE)))
          ;; Check WH-traces.
          ((REFERENTIAL? ELT)
           (UNLESS (NODE-A-BAR-POSITION ANTECEDENT) (RETURN 'FAILURE)))
          (T NIL))))))
```

F.7 Bounding Theory

The Bounding module is accessed during parsing at both PUSH time and POP time. When a trace is pushed onto the stack, it must be linked to an antecedent that is not too far away. In the case of rightward movement, the trace may remain unlinked until its dominating maximal projection has been completed. Thus, at POP time, the Bounding module must be accessed again to link unlinked traces to antecedents. The Bounding module is also accessed to allow language-specific movement requirements to be checked (both at POP time and at generation time). Finally, the Bounding module is accessed during generation to ensure that an element does not move "too far" when move-α is executed.

F.7.1 Routines for Trace Linking at PUSH Time

```
;;; Check-trace-links is invoked at at PUSH time.  Since a trace may have been
;;; added to a stack it must be linked up to its antecedent.  There are two
;;; cases:  either a trace is *not* found, or it *is* found.  If it is *not*
;;; found, the added portion will be pushed (following feature percolation). If
;;; it *is* found, the features are percolated, trace-antecedent linking takes
;;; place, and the added portion will be pushed.
(DEFUN CHECK-TRACE-LINKS (ADD-ON STACK)
  (LET* ((TRACE-NODE NIL) (ANTECEDENT-NODES NIL) (RESULTS NIL)
         (CONVERTED-ADD-ON (CONVERT-TO-NODES ADD-ON)) (FINAL-RESULTS NIL))
    ;; Locate a trace (there should be at most one unlinked trace).
    (IF (NOT (SETQ TRACE-NODE (FIRST (SEARCH-TRACE (TOS CONVERTED-ADD-ON)))))
        ;; If there is no trace or antecedents weren't found, just push the
        ;; unmodified template onto the stack.
        (SETQ FINAL-RESULTS (LIST (COPY-NODES (APPEND CONVERTED-ADD-ON STACK))))
        ;; Otherwise, find possible antecedents of the trace.
        (SETQ
          ANTECEDENT-NODES
          (FIND-TRACE-ANTECEDENTS
            (NODE-CAT TRACE-NODE) (APPEND-CONVERTED-ADD-ON STACK) TRACE-NODE))
        ;; Only PUSH the templates and link antecedents if feature-percolation
        ;; hasn't eliminated the add-on.
        (WHEN CONVERTED-ADD-ON
          ;; Push the templates.
          (SETQ STACK (APPEND CONVERTED-ADD-ON STACK))
          ;; If an antecedent was not found, check to see if there is a
          ;; possibility of the antecedent not yet being seen (for head-final
          ;; or rightward movement case).
          (IF (EQ ANTECEDENT-NODES 'FAILURE)
              ;; Head-final or rightward movement
              (WHEN (ANTECEDENT-NOT-BEYOND-BOUNDING-NODE STACK)
            (PUSH STACK FINAL-RESULTS))
          ;; Antecedents found.
          (SETQ RESULTS
                (LINK-TRACES-AND-ANTECEDENTS
                  ANTECEDENT-NODES TRACE-NODE STACK))
          ;; Only accept the result if feature percolation has not
          ;; eliminated the ADD-ON.
          (LOOP FOR RESULT IN RESULTS DO
            (WHEN (PERCOLATE-FEATURES (TOS RESULT))
              (PUSH RESULT FINAL-RESULTS))))))
    ;; Return the final results unless trace-linking or feature percolation has been
    ;; unsuccessful.
    (OR FINAL-RESULTS 'FAILURE)))


;;; Find-trace-antecedents is used for finding the antecedent of a trace.  It
;;; locates all possible antecedents of a trace, without going beyond two
;;; bounding nodes.  When a node already has its antecedent slot filled, then
;;; it is not considered a possible antecedent.
(DEFUN FIND-TRACE-ANTECEDENTS (TRACE-CAT STACK NODE-TO-LINK)
  (LET ((NODES NIL) (COUNT 0) (FINAL-RESULT NIL))
    ;; Find all traces and their possible antecedents.
    (LOOP FOR STACK-ITEM IN STACK DO
      ;; If the antecedent is found, return the corresponding node unless its
      ;; antecedent slot is filled with something.
      (WHEN (SETQ NODES
                  (FIND-TRACE-AT-SOME-LEVEL TRACE-CAT STACK-ITEM NODE-TO-LINK))
        (LOOP FOR NODE IN NODES DO
          (UNLESS (NODE-ANTECEDENT NODE) (PUSH NODE FINAL-RESULT))))
      ;; If the search goes beyond more than one bounding node, return FAILURE.
      (WHEN (AND (MEMBER (NODE-CAT (CAR STACK-ITEM)) *CURRENT-BOUNDING-NODES)
                 (EQUAL (SETQ COUNT (1+ COUNT)) 2))
        (RETURN (OR FINAL-RESULT 'FAILURE)))
          FINALLY (RETURN (OR FINAL-RESULT 'FAILURE)))))
```

```
;;; Antecedent-not-beyond-bounding-node searches through a stack for an
;;; incomplete item without going beyond two bounding nodes.  If such an item
;;; is found, then the search ends and the result is T.  Otherwise, the result
;;; is NIL.  (Note: the trace is assumed to be in the top of stack item.)
(DEFUN ANTECEDENT-NOT-BEYOND-BOUNDING-NODE (STACK)
  (LET ((COUNT 0) (LAST-PUSHED-SYMBOL NIL))
    (LOOP FOR STACK-ITEM IN STACK DO
      ;; If an unexpanded item is found, stop searching, and return NIL.
      (WHEN (NEXT-EMPTY-SYMBOL STACK-ITEM LAST-PUSHED-SYMBOL) (RETURN T))
      ;; If the search goes beyond more than one bounding node, return T.
      (WHEN (AND (MEMBER (NODE-CAT (CAR STACK-ITEM))
                         *CURRENT-BOUNDING-NODES :TEST #'EQ)
                 (EQUAL (SETQ COUNT (1+ COUNT)) 2)) (RETURN NIL))
      ;; Update the last expanded symbol.
      (SETQ LAST-PUSHED-SYMBOL (NODE-CAT (CAR STACK-ITEM)))
        FINALLY (RETURN T))))

;;; Search-trace locates all unlinked trace nodes (i.e., traces with no
;;; antecedents) dominated by a node.
(DEFUN SEARCH-TRACE (NODES)
  (COND ((NULL NODES) NIL)
        ((ATOM NODES) (WHEN (EQ (NODE-TRACE NODES) T) (LIST NODES)))
        (T (APPEND (SEARCH-TRACE (CAR NODES)) (SEARCH-TRACE (CDR NODES))))))
```

F.7.2 Routines for Trace Linking at POP Time

```
;;; Find-and-link-traces is invoked at POP time: it links traces to
;;; antecedents, unless the corresponding antecedent is already linked.  First
;;; all traces are found, and then antecedent possibilities are found.  These
;;; are then matched up by Link-traces-and-antecedents.  The assumption is that
;;; a constituent has moved and now must be linked up with the trace it left
;;; behind.  However, if this requirement is not met, the parse is either
;;; rejected (if the antecedent is too far away), or allowed to hang around (if
;;; the antecedent has not been encountered during the parse yet).
(DEFUN FIND-AND-LINK-TRACES (STACK)
  (LET ((TRACE-NODES NIL) (ANTECEDENT-NODES NIL) (RESULTS NIL))
      ;; Locate unlinked trace-nodes.
      (IF (SETQ TRACE-NODES (SEARCH-TRACE (TOS STACK)))
          ;; If there are trace nodes, locate the possible antecedents.
          (LOOP FOR TRACE-NODE IN TRACE-NODES DO
            (SETQ ANTECEDENT-NODES
                  (FIND-TRACE-ANTECEDENTS
                    (NODE-CAT TRACE-NODE) STACK TRACE-NODE))
            ;; If an antecedent was not found, check to see if there is a
            ;; possibility of the antecedent not yet being seen (for head-final
            ;; or rightward movement case).
            (IF (EQ 'FAILURE ANTECEDENT-NODES)
                ;; Head-Final or Rightward Movement
                (WHEN (ANTECEDENT-NOT-BEYOND-BOUNDING-NODE STACK)
                  (PUSH STACK RESULTS))
                ;; Antecedents found.
                (SETQ RESULTS
                      (APPEND RESULTS
                              (LINK-TRACES-AND-ANTECEDENTS
                                ANTECEDENT-NODES TRACE-NODE STACK) RESULTS)))
            FINALLY (RETURN (OR RESULTS 'FAILURE)))
          ;; No trace nodes.
          (LIST STACK))))
```

F.7.3 Routines for Checking Language-Specific Effects

```
;;; Check-language-specific-effects is called at POP time and at generation
;;; time.  It tests that the left-hand side of a language-specific implication
;;; holds if the right-hand side holds.
(DEFUN CHECK-LANGUAGE-SPECIFIC-EFFECTS (FINAL-RESULT)
  (IF (AND *CURRENT-LANGUAGE-SPECIFIC-EFFECTS (PARSE-COMPLETE FINAL-RESULT))
      ;; If there are language-specific effects, make sure they hold before
      ;; returning the result.
      (LOOP FOR EFFECT IN *CURRENT-LANGUAGE-SPECIFIC-EFFECTS DO
        (UNLESS
          (MOVEMENT-EFFECT?
            FINAL-RESULT (LHS EFFECT) (RHS EFFECT) (IMPLICATION EFFECT))
          (RETURN 'FAILURE))
        FINALLY (RETURN FINAL-RESULT))
      ;; Otherwise, just return the final-result.
      FINAL-RESULT))
```

F.7.4 Routines for Performing Move-α at Generation Time

```
;;; Perform-movement is called by move-alpha at generation time.  It takes all
;;; substitution and adjunction possibilities and performs all permutations of
;;; these movements.
(DEFUN PERFORM-MOVEMENT (SUBSTITUTION ADJUNCTION TREE)
  ;; Case 1. No movement.
  (VECTOR-PUSH-EXTEND TREE *SURFACE-TREES)
  ;; Case 2. Adjunction possibilities tried before substitution possibilities.
  (PERFORM-ADJUNCTION ADJUNCTION (PERFORM-SUBSTITUTION (LIST TREE) SUBSTITUTION))
  ;; Case 3. Substitution possibilities tried before adjunction possibilities.
  (PERFORM-SUBSTITUTION SUBSTITUTION (PERFORM-ADJUNCTION (LIST TREE) ADJUNCTION)))

;;; Perform-substitution moves elements into specifier positions included in the
;;; SUBSTITUTION list.  It includes all n! combinations of substitutions, where
;;; n is the number of possible substitutions.
(DEFUN PERFORM-SUBSTITUTION (SUBSTITUTION TREES)
  (LET ((NEW-TREE) (RESULTS NIL))
    (LOOP FOR TREE IN TREES DO
      (LOOP FOR SUBST IN SUBSTITUTION DO
        (SETQ NEW-TREE (SUBSTITUTE-NODE-FOR-ELT SUBST TREE))
        (UNLESS (EQ (CHECK-LANGUAGE-SPECIFIC-EFFECTS NEW-TREE) 'FAILURE)
          (VECTOR-PUSH-EXTEND NEW-TREE *SURFACE-TREES))
        (PUSH NEW-TREE RESULTS))))
```

```
;;; Perform-adjunction moves elements into specifier positions included in the
;;; ADJUNCTION list.  It includes all n! combinations of adjunctions, where
;;; n is the number of possible adjunctions.
(DEFUN PERFORM-ADJUNCTION (ADJUNCTION TREES)
  (LET ((NEW-TREE) (RESULTS NIL))
    (LOOP FOR TREE IN TREES DO
      (LOOP FOR ADJ IN ADJUNCTION DO
        (SETQ NEW-TREE (ADJOIN-NODE-TO-ELT ADJ TREE))
        (UNLESS (EQ (CHECK-LANGUAGE-SPECIFIC-EFFECTS NEW-TREE) 'FAILURE)
          (VECTOR-PUSH-EXTEND NEW-TREE *SURFACE-TREES))
        (PUSH NEW-TREE RESULTS)))))
```

Appendix G

Earley Parser Routines

This appendix shows all of the functions required for parsing a sentence using the PREDICT, SCAN and COMPLETE actions of the Earley parser. PARSE-SENTENCE contains the Earley parser main loop. Note that the bare Earley parser (without structure-building) may be used if the *RECOGNIZE-ONLY variable is set to T. Furthermore, if structure is being built, it may either be built with access to the GB component (PUSH-STRUCT, SCAN-STRUCT, and POP-STRUCT) or without access to the GB-COMPONENT (PUSH-STRUCT-1, SCAN-STRUCT-1, and POP-STRUCT-1).

G.1 Main Earley Parser Loop

```
;;; Parse-sentence loops over an input sentence creating a state-set for each
;;; word in the sentence.  Each state-set is scanned, pushed, popped and then
;;; pushed again.  The parse(s) are held in the *SET-OF-STACKS variable. The
;;; user has the option of only recognizing (i.e., not returning building any
;;; structure) by setting the global variable *RECOGNIZE-ONLY to T.
;;; Furthermore, the GB-component is consulted only if parse-sentence is called
;;; with the optional parameter GB-COMPONENT? set to T.
(DEFUN PARSE-SENTENCE (SENTENCE &OPTIONAL (GB-COMPONENT? NIL))
  ;; Initialize parse structures
  (LET* ((PUNC (CAR (LAST SENTENCE)))
         (FIRST-STATE-SET-RULE
           (CONS (CAR *INITIAL-RULE)
                 (CONS (CADR *INITIAL-RULE)
                       (CONS '* (CDDR *INITIAL-RULE)))))
         (NEXT-WORDS SENTENCE)          ; list of next words for lookahead
```

```
      (LAST-STATE NIL)                 ; previous state-set name
      (CURRENT-STATE NIL))             ; current state-set name
  ;; Nilify all elements of *SET-OF-STACKS and *PARSE-TREES
  (SET-ARRAY-NIL *SET-OF-STACKS)
  ;; Put initial template into *SET-OF-STACKS unless *RECOGNIZE-ONLY is set
  ;; to T.
  (UNLESS *RECOGNIZE-ONLY
    (VECTOR-PUSH-DEFNODES
      (LIST (CONS (CAR *INITIAL-RULE) (MAPCAR #'LIST (CDDR *INITIAL-RULE))))
      *SET-OF-STACKS))
  ;; Initialize state-sets. Define S0 to contain a single rule.
  (SETQ *STATE-SET-NUMBER 0
        *STATE-SET
        (DEFINE-STATE 'S0 :NUMBER 0
                      :LIST-OF-RULES (LIST FIRST-STATE-SET-RULE)))
  ;; Push state-set S0
  (PREDICT 'S0 (CAR NEXT-WORDS) GB-COMPONENT?)
  ;; Do a pop/push closure (Note: this loop is necessary only because of
  ;; lambda rules since there is no input word yet).
  (POP-PUSH-CLOSURE 'S0 (CAR NEXT-WORDS) GB-COMPONENT?)
  ;; Prepare to process the rest of the state-sets, starting with S1.
  (SETQ *STATE-SET-NUMBER 1
        NEXT-WORDS (CDR NEXT-WORDS)
        LAST-STATE 'S0 CURRENT-STATE 'S1)
  ;; Loop over each word in the sentence creating a state-set corresponding
  ;; to each word.  Each state-set will be scanned (SCAN), pushed (PREDICT),
  ;; popped (COMPLETE) and then pushed again. Then, because could be lambda
  ;; rules, a pop/push closure is invoked.  This ensures that any rules not
  ;; requiring input have a chance to fire. If a state-set becomes empty,
  ;; reject the sentence.  Otherwise continue until *STATE-SET-NUMBER is
  ;; equal to the number of words in the sentence. Check this state-set for
  ;; the initial state set rule with RETURN = 0. If this state is present,
  ;; accept. Otherwise, reject.
  (LOOP FOR WORD IN SENTENCE DO
    (SETQ *STATE-SET
          (DEFINE-STATE
            CURRENT-STATE :NUMBER *STATE-SET-NUMBER
            :WORD-CAT (GET-CAT WORD) :WORD WORD))
    (SCAN LAST-STATE CURRENT-STATE PUNC NIL GB-COMPONENT?)    ; scan
    (PREDICT CURRENT-STATE (CAR NEXT-WORDS) GB-COMPONENT?)    ; push
    (COMPLETE CURRENT-STATE (CAR NEXT-WORDS) GB-COMPONENT?)   ; pop
    (PREDICT CURRENT-STATE (CAR NEXT-WORDS) GB-COMPONENT?)    ; push
    ;; Do a pop/push closure (to take care of lambda rules).
    (POP-PUSH-CLOSURE CURRENT-STATE (CAR NEXT-WORDS) GB-COMPONENT?)
    (COND
      ;; If state set is empty, reject.
      ((NULL (STATE-SET-LIST-OF-RULES (EVAL CURRENT-STATE))) (RETURN 'REJECTED))
      ;; If the state-set number equals the number of words in sentence, then
      ;; if the state-set includes the state initial state-set rule with
      ;; RETURN = 0, accept.  Else, reject.  Final parse(s) are in the
```

```
        ;; *SET-OF-STACKS variable.
        ((EQUAL *STATE-SET-NUMBER *NUMBER-OF-WORDS)
         (IF (RULE-IN-STATE-SET?
               (REVERSE (CONS '* (REVERSE *INITIAL-RULE))) 0 (EVAL CURRENT-STATE))
             ;; Accept.
             (RETURN 'ACCEPTED)
             ;; Reject.
             (RETURN 'REJECTED))))
      ;; Update *STATE-SET-NUMBER and LAST-STATE.  Move to next state and input
      ;; word.
      (SETQ *STATE-SET-NUMBER (1+ *STATE-SET-NUMBER)
            LAST-STATE CURRENT-STATE
            CURRENT-STATE
    (INTERN (STRING-UPCASE (FORMAT NIL "s~A" *STATE-SET-NUMBER)))
            NEXT-WORDS (CDR NEXT-WORDS)))))
```

G.2 Predict Action

```
;;; Predict adds a dotted rule to a state-set whenever the parser predicts a
;;; nonterminal in the parse (i.e., when the dot appears to the left of a
;;; nonterminal symbol in the current state-set). Each predicted nonterminal
;;; appears on the left-hand side of one or more dotted rules in the state-set.
;;; When a nonterminal is predicted in a state-set, the return address of all
;;; corresponding rules is the current state-set number.
(DEFUN PREDICT (STATE-SET WORD GB-COMPONENT?)
  (LET  ((SET (EVAL STATE-SET))            ; figure out which state-set to push
         (RULES-ADDED NIL)                 ; rules added to state set
         (OLD-RULE NIL)                    ; rule that will not be explicitly added
         (NOT-TO-BE-EXPLICITLY-ADDED NIL) ; rules that have already been added
         (RHS)                             ; right-hand side of unpushed rule
         (SYMBOL-TO-CHECK))                ; symbol to right of dot
    ;; Loop until no more unpushed rules.
    (LOOP FOR RULE IN (STATE-SET-LIST-OF-RULES SET) DO
      (SETQ RHS (STATE-SET-RULE-RHS RULE) ; extract right-hand side of unpushed rule
            SYMBOL-TO-CHECK               ; extract symbol to right of dot
            (WHEN (CDR (MEMBER '* RHS :TEST #'EQ))
              (CADR (MEMBER '* RHS :TEST #'EQ))))
      ;; If symbol to the right of the dot is a non-terminal, loop over grammar
      ;; rules looking for candidates for pushing.
      (WHEN (AND SYMBOL-TO-CHECK (MEMBER SYMBOL-TO-CHECK *NON-TERMINALS :TEST #'EQ))
        (LOOP FOR GRAMMAR-RULE IN *CURRENT-GRAMMAR DO
          (IF
            (SETQ OLD-RULE
                  (RULE-IN-STATE-SET?
                    (APPEND (LIST SYMBOL-TO-CHECK '==> '*) (CDDR GRAMMAR-RULE))
                    *STATE-SET-NUMBER SET))
            ;; If the rule is already in the state set, don't explicitly
            ;; (re-)add it, but put it in a list to be passed to PUSH-STRUCT.
            (PUSH OLD-RULE NOT-TO-BE-EXPLICITLY-ADDED)
            ;; Otherwise, add the new scanned rule to the state-set.
            (SETQ RULES-ADDED
                  (APPEND RULES-ADDED
                          (LAST (ADD-RULE
                                  (APPEND (LIST SYMBOL-TO-CHECK '==> '*)
                                          (CDDR GRAMMAR-RULE))
                                  *STATE-SET-NUMBER SET))))))))
    ;; If parsing, as opposed to recognition only, call the structure-building
    ;; component which will push nonterminal template structures onto the
    ;; stack.  Be sure to pass rules that were not explicitly added to the
    ;; state-set down to the structure-building component.
    (UNLESS *RECOGNIZE-ONLY
      (SETQ RULES-ADDED (APPEND RULES-ADDED NOT-TO-BE-EXPLICITLY-ADDED))
      (IF GB-COMPONENT?
          ;; GB component is accessed.
          (PUSH-STRUCT (LIST-OUT-ADDED-RULES RULES-ADDED) (GET-CAT WORD))
        ;; GB component is not accessed.
        (PUSH-STRUCT-1 (LIST-OUT-ADDED-RULES RULES-ADDED) (GET-CAT WORD))))))
```

G.3 Scan Action

```
;;; Scan adds a dotted rule to a state-set whenever the parser encounters a
;;; terminal which can be scanned during the parse (i.e., when the dot appears
;;; to the left of a terminal symbol in the previous state-set).  Each scanned
;;; terminal appears to the left of the dot on the right-hand side of one or
;;; more dotted rules in the state-set. When a rule corresponding to a scanned
;;; symbol is added to a state-set, the return address is retrieved from the
;;; corresponding unscanned rule in the previous state-set. Note: the GB
;;; component will be consulted if GB? is T and there is a block in the parse
;;; somewhere.
(DEFUN SCAN (LAST-STATE CURRENT-STATE PUNC GB-COMPONENT?)
  (LET ((LAST-SET (EVAL LAST-STATE))          ; state-set to scan from
        (CURRENT-SET (EVAL CURRENT-STATE))    ; state-set to scan into
        (RULES-ADDED NIL)                     ; rules added to state set
        (RHS)                                 ; RHS of unscanned-rule
        (REP)                                 ; entire unscanned rule
        (RET)                                 ; return address of unscanned rule
        (SYMBOL-TO-CHECK))                    ; symbol to right of dot
    ;; Loop over unscanned rules in state-set.
    (LOOP FOR RULE IN (STATE-SET-LIST-OF-RULES LAST-SET) DO
      (SETQ RHS (STATE-SET-RULE-RHS RULE)     ; extract RHS of unscanned rule
            REP (STATE-SET-RULE-RULE RULE)    ; extract entire unscanned rule
            RET (STATE-SET-RULE-RETURN RULE); extract ret addr of unscanned rule
            SYMBOL-TO-CHECK                   ; extract symbol to right of dot
            (WHEN                             ; ignore lambda rules
              (AND (NOT (EQUAL RHS '(*))) (CDR (MEMBER '* RHS :TEST #'EQ)))
              (CADR (MEMBER '* RHS :TEST #'EQ))))
      ;; If symbol to right of dot is a terminal symbol which matches the
      ;; current word category, and if the scanned rule is not already in the
      ;; current state-set, scan the rule and add it to the current state-set
      (WHEN (AND
              (MEMBER SYMBOL-TO-CHECK *TERMINALS :TEST #'EQ)
              (MEMBER SYMBOL-TO-CHECK         ; matches current word category?
                      (STATE-SET-WORD-CAT CURRENT-SET) :TEST #'EQ)
              (NOT                            ; not already in state-set?
                (RULE-IN-STATE-SET?
                  (MOVE-DOT-OVER-SYMBOL REP RHS SYMBOL-TO-CHECK)
                  RET CURRENT-SET)))
        ;; If yes to the above, add the new (scanned) dotted rule to the
        ;; state-set.
        (SETQ RULES-ADDED
              (APPEND RULES-ADDED
                      (LAST
                        (ADD-RULE
                          (MOVE-DOT-OVER-SYMBOL REP RHS SYMBOL-TO-CHECK)
                          RET CURRENT-SET)))
              *LAST-SCANNED (1+ *LAST-SCANNED))))
    ;; If parsing, as opposed to recognition only, call the structure-building
;; component which will attach terminals to nonterminals in lower positions
;; on the stack.
(UNLESS *RECOGNIZE-ONLY
  (IF GB-COMPONENT?
      ;; GB component is accessed.
      (SCAN-STRUCT
        (STATE-SET-WORD CURRENT-SET) (LIST-OUT-ADDED-RULES RULES-ADDED))
      ;; GB component is not accessed.
      (SCAN-STRUCT-1
        (STATE-SET-WORD CURRENT-SET) (LIST-OUT-ADDED-RULES RULES-ADDED))))))
```

G.4 Complete Action

```
;;; Complete adds a dotted rule to a state-set whenever the parser encounters a
;;; nonterminal which can be popped during the parse (i.e., when the dot
;;; appears at the end of the right-hand side of a rule in the current
;;; state-set). Each popped nonterminal appears to the left of the dot on the
;;; right-hand side of one or more dotted rules in the state-set.  When a rule
;;; corresponding to a popped symbol is added to a state-set, the return
;;; address is retrieved from the corresponding unpopped rule in the state-set
;;; pointed to by the return address.
(DEFUN COMPLETE (STATE-SET GB-COMPONENT?)
  (LET ((SET (EVAL STATE-SET))                 ; pop current state set
        (RULES-ADDED NIL)                      ; rules added to state set
        (NEW-RULE NIL)                         ; rule that may potentially be added
        (NOT-TO-BE-EXPLICITLY-ADDED NIL)       ; rules that have already been added
        (LHS)                                  ; LHS of unpopped rule
        (RHS)                                  ; RHS of unpopped rule
        (REP)                                  ; rule rep of unpopped rule
        (RHS-PRIOR)                            ; RHS of prior rule
        (REP-PRIOR)                            ; entire prior rule
        (SYMBOL-TO-CHECK))                     ; symbol to right of dot in prior rule
      (LOOP FOR RULE IN (STATE-SET-LIST-OF-RULES SET) DO
        (SETQ LHS (STATE-SET-RULE-LHS RULE)       ; extract LHS of unpopped rule
              RHS (STATE-SET-RULE-RHS RULE)       ; extract RHS of unpopped rule
              REP (STATE-SET-RULE-RULE RULE))     ; extract rep of unpopped rule
        ;; If symbol to the right of the dot is non-nil, loop over rules in
        ;; state-set corresponding to return address looking for candidates for
        ;; popping.
        (UNLESS (CDR (MEMBER '* RHS :TEST #'EQ))
          (LOOP FOR
            PRIOR-RULE
            IN
            (STATE-SET-LIST-OF-RULES
              (EVAL
                (INTERN
                  (STRING-UPCASE
                    (FORMAT NIL "s~A" (STATE-SET-RULE-RETURN RULE)))))) DO
            (SETQ
              ;; Extract RHS of prior rule.
              RHS-PRIOR (STATE-SET-RULE-RHS PRIOR-RULE)
              ;; Extract entire prior rule.
              REP-PRIOR (STATE-SET-RULE-RULE PRIOR-RULE)
              ;; Extract symbol to right of dot in prior rule.
              SYMBOL-TO-CHECK
              (WHEN (CDR (MEMBER '* RHS-PRIOR :TEST #'EQ))
                (CADR (MEMBER '* RHS-PRIOR :TEST #'EQ))))
            ;; If symbol to right of dot in the prior rule matches the
            ;; left-hand side of the rule in the current state-set, and if the
            ;; popped rule is not already in the current state set, pop the
            ;; rule and add it to the current state-set
            (WHEN (EQ SYMBOL-TO-CHECK LHS) ; matches LHS of current rule?
              (SETQ NEW-RULE
                    (MOVE-DOT-OVER-SYMBOL REP-PRIOR RHS-PRIOR SYMBOL-TO-CHECK))
              (IF (RULE-IN-STATE-SET?
                    NEW-RULE (STATE-SET-RULE-RETURN PRIOR-RULE) SET)
                  ;; If the rule is already in the state-set, it will not be
                  ;; explicitly added, but it will be passed on to the
                  ;; structure building stage for reference.
                  (SETQ NOT-TO-BE-EXPLICITLY-ADDED
                        (CONS NEW-RULE NOT-TO-BE-EXPLICITLY-ADDED))
                  ;; Otherwise, add the new (popped) rule to the state set.
                  (SETQ RULES-ADDED
                        (APPEND
                          RULES-ADDED
                          (LAST
                            (ADD-RULE
                              NEW-RULE
                              (STATE-SET-RULE-RETURN PRIOR-RULE) SET)))))))))
      ;; If parsing, as opposed to recognition only, drop completed terminals
      ;; into lower positions on stack and pop the stack.
      (UNLESS *RECOGNIZE-ONLY
        (IF GB-COMPONENT?
            ;; GB component is accessed.
            (POP-STRUCT
              (LIST-OUT-ADDED-RULES
                (APPEND NOT-TO-BE-EXPLICITLY-ADDED RULES-ADDED)))
            ;; GB component is not accessed.
            (POP-STRUCT-1
              (LIST-OUT-ADDED-RULES
(APPEND NOT-TO-BE-EXPLICITLY-ADDED RULES-ADDED))))))
```

Appendix H

Interface Between Structure-Building and GB

This appendix contains the interface between the structure-building routines and the GB component. The three main structure-building routines of the parser are: PUSH-STRUCT (called by PREDICT), SCAN-STRUCT (called by SCAN), and POP-STRUCT (called by COMPLETE). Each of these routines builds structure according to the rules processed by Earley; this structure is then examined by various modules of the GB component. The GB component may do one of three things: reject the structure (if there is a violation of a constraint), accept the structure (if all constraints are satisfied), or update the structure (if there is a possibility unavailable to the Earley algorithm). The three main structure-building routines of the replacement module are: MOVE-ELEMENTS-BACK, COLLAPSE-STRUCTURE, and REPLACE-SOURCE-WITH-TARGET. These functions are used to derive the target base form by dropping moved elements into base positions, collapsing redundant structure and replacing source language words with target language equivalents. The structure-building routine of the generation module is MOVE-ALPHA. This function restructures trees according to requirements of the target language and moves elements out of base positions into possible landing sites.

H.1 Interface During PUSH Stage

```
;;; PUSH-STRUCT examines the rules added to the current state via the PREDICT
;;; function.  It pushes XBAR templates onto the stack wherever they are
;;; predicted unless the lookahead dictates otherwise.  Ambiguity is
;;; accommodated by keeping multiple stacks in the *SET-OF-STACKS variable.
;;; Also installed into the parser is a one-word lookahead facility so that a
;;; template is only pushed if the next word can be scanned using the template.
;;; Furthermore, trace linking and feature percolation are performed here.
(DEFUN PUSH-STRUCT (LIST-OF-RULES WORD-CATS)
  (LET* ((INCOMPLETE NIL)
         (STACK NIL)
         (LIST-OF-SYMBOLS (HEADERS-OF LIST-OF-RULES))
         (PUSHED
           (CHECK-TERMINALS
             (GET-LEFT-MOST-DERIVES LIST-OF-SYMBOLS)
             LIST-OF-SYMBOLS WORD-CATS)))
    ;; Iterate over each stack in *SET-OF-STACKS, pushing results onto
    ;; *TEMP-STACKS.
    (LOOP WHILE (NOT (ZEROP (FILL-POINTER *SET-OF-STACKS))) DO
      (SETQ STACK (VECTOR-POP *SET-OF-STACKS)
            INCOMPLETE (ITEM-INCOMPLETE-NODES (TOS STACK)))
      ;; If the top of the stack is a lambda rule, a non-expandable terminal,
      ;; or there is no expansion of the node (probably a terminal which has
      ;; already been taken care of in the last push), just add the stack
      ;; unconditionally; otherwise, push the appropriate lists onto the stack.
      (IF (OR (NULL (CDR (TOS STACK)))
              (MEMBER INCOMPLETE *TERMINALS :TEST #'EQ)
              (NULL (GET-VALUE PUSHED INCOMPLETE)))
          (VECTOR-PUSH-EXTEND STACK *TEMP-STACKS 50)
          (PUSH-TEMPLATES-ONTO-STACK STACK (GET-VALUE PUSHED INCOMPLETE)))
      ;; Push *TEMP-STACKS elements back onto *SET-OF-STACKS.
      (UNLESS (EMPTY-ARRAY? *TEMP-STACKS)
        (LOOP WHILE (NOT (ZEROP (FILL-POINTER *TEMP-STACKS))) DO
          (VECTOR-PUSH-EXTEND
            (VECTOR-POP *TEMP-STACKS) *SET-OF-STACKS 50))))))
```

```
;;; Push-templates-onto-stack pushes XBAR templates onto a stack. Because
;;; complements may be expanded here, a subcategorization check is used to link
;;; complements up with heads.  Also, since the list-of-templates may contain
;;; traces, the traces must be linked up before they are pushed onto STACK (via
;;; Check-trace-links).
(DEFUN PUSH-TEMPLATES-ONTO-STACK (STACK LIST-OF-TEMPLATES)
  (LET* ((MODIFIED-NODE-STACKS NIL) (CAT NIL) (NODE NIL) (TEMP-STACKS NIL))
    ;; Find the incomplete node in the top of stack.
    (MULTIPLE-VALUE-SETQ (CAT NODE) (ITEM-INCOMPLETE-NODES (TOS STACK)))
    ;; Iterate over all possible ADD-ON.
    (LOOP FOR ADD-ON IN LIST-OF-TEMPLATES DO
      (IF (EQ ADD-ON 'INTRANS)
          ;; If intransitive complement is being pushed, remove the complement
          ;; symbol from the stack and check that subcategorization information
          ;; matches up.
          (WHEN (SETQ MODIFIED-NODE-STACKS
                      (PERFORM-SUBCATEGORIZATION-CHECK
                        (LIST (CONS (REMOVE NODE (TOS STACK)) (CDR STACK)))))
            (UNLESS (EQ MODIFIED-NODE-STACKS 'FAILURE)
              (LOOP FOR MODIFIED-STACK IN MODIFIED-NODE-STACKS DO
                (PUSH MODIFIED-STACK TEMP-STACKS))))
          ;; Otherwise, process normally: Check that traces are linked and that
          ;; subcategorization information matches up. Note: during trace
          ;; linking, features will be percolated for each ADD-ON element.
          (WHEN (SETQ MODIFIED-NODE-STACKS
                      (PERFORM-SUBCATEGORIZATION-CHECK
                        (CHECK-TRACE-LINKS ADD-ON STACK)))
            (UNLESS (EQ MODIFIED-NODE-STACKS 'FAILURE)
              (LOOP FOR MODIFIED-STACK IN MODIFIED-NODE-STACKS DO
                (PUSH MODIFIED-STACK TEMP-STACKS))))))
    ;; Add the stacks that are considered legitimate.
    (LOOP FOR NEW-STACK IN TEMP-STACKS DO
      (VECTOR-PUSH-EXTEND NEW-STACK *TEMP-STACKS))))
```

```
;;; Get-left-most-derives extracts all possible templates to push for the given
;;; list of symbols.  If a symbol corresponding to a complement arises, the
;;; phrase-structure expansions of the complement are determined (according to
;;; the closest head to the left of the given category in a head-first
;;; language, or the closest head to the right in a head-final language).
;;; Otherwise, the templates are found in *LEFT-DERIVES-RULES.
;;; Add-complement-templates-and-rules ensures that the complement templates
;;; are appended to stacks containing the complement symbol.  It also takes
;;; care of intransitive heads (by eliminating the complement symbol from TOS
;;; elements). It then adds appropriate complement and non-complement rules to
;;; the state set.
(DEFUN GET-LEFT-MOST-DERIVES (LIST-OF-SYMBOLS)
  (LET ((FINAL-RESULT NIL) (HEAD-CAT NIL)
        (COMPLEMENT-TEMPLATES) (TEMP-COMPLEMENT-TEMPLATES))
    (LOOP FOR SYMBOL IN LIST-OF-SYMBOLS DO
      (IF
        ;; If a complement is to be expanded in a head-first or head-final
        ;; construction, generate the complement templates (locate the closest
        ;; head to the left or right) and add the corresponding rules.
        (AND (SETQ HEAD-CAT (COMPLEMENT? SYMBOL))
             ;; Generate template for each subcategorization frame of the head
               (SETQ TEMP-COMPLEMENT-TEMPLATES
                     (APPEND TEMP-COMPLEMENT-TEMPLATES
                             (LOCATE-COMPLEMENTS
                               HEAD-CAT SYMBOL *CURRENT-HEAD-FIRST?))))
          ;; Otherwise, find the templates the "normal" way.
          (SETQ FINAL-RESULT
                (APPEND FINAL-RESULT
                        (CONS SYMBOL
                              (LIST
                                (GET-VALUE *LEFT-DERIVES-RULES SYMBOL))))))
          FINALLY
            (RETURN (ADD-COMPLEMENT-TEMPLATES-AND-RULES
                       COMPLEMENT-TEMPLATES FINAL-RESULT)))))
```

```
;;; Check-terminals weeds out those templates that do not derive any of the
;;; word categories as the left-most terminal symbol. This enables the facility
;;; of one-word look-ahead.
(DEFUN CHECK-TERMINALS (TEMPLATES SYMBOLS WORD-CATS)
  (LET ((CURRENT-PUSHED) (CURRENT-RESULT) (NEW-TEMPLATES))
    ;; Loop through all the symbols checking whether the templates
    ;; corresponding to the symbol will derive the word-cats.
    (LOOP FOR SYMBOL IN SYMBOLS DO
      ;; Fix up the templates so that non-lexical heads are allowed to *not* be
      ;; empty, and default complements are removed.
      (SETQ CURRENT-PUSHED
            (UPDATE-NON-LEXICAL-CATS (GET-VALUE TEMPLATES SYMBOL)))
      ;; Loop through each pushed-rule checking that it derives the word-cats.
      (LOOP FOR PUSHED IN CURRENT-PUSHED DO
        ;; If the pushed template derives the next word, add it to the list.
        (IF (TEST-DERIVES-NEXT-CATS (TOS PUSHED) WORD-CATS PUSHED)
            (PUSH PUSHED CURRENT-RESULT)                ; add the rule
            (WHEN (NULL PUSHED) (PUSH NIL CURRENT-RESULT))))
      ;; If some (or all) of the pushed-rules have been added, update the final
      ;; answer (new-templates).
      (WHEN CURRENT-RESULT
        (SETQ NEW-TEMPLATES
              (APPEND (CONS SYMBOL (LIST CURRENT-RESULT)) NEW-TEMPLATES))
        (SETQ CURRENT-RESULT NIL))
          FINALLY (RETURN NEW-TEMPLATES))))
```

H.2 Interface During SCAN Stage

```
;;; Scan-struct examines the rules added to the current state set via the SCAN
;;; function. It inserts the input word into the appropriate slot in top of
;;; stack item. When an input word is dropped into the appropriate slot, the
;;; resulting stack is pushed onto *TEMP-STACKS.  Those stacks which are not
;;; modified are marked (by a $) and pushed onto NEW-MARKED-STACKS.  At the end
;;; of SCAN-STRUCT, *SET-OF-STACKS is replaced by *TEMP-STACKS (i.e., the
;;; successfully scanned parses).  During the check for whether the current
;;; rule corresponds to the top-of-stack configuration
;;; (Check-for-correct-position), the GB component is consulted
;;; (Check-for-correct-position) because argument structure needs to be
;;; established and features need to be instantiated if a head is scanned.
(DEFUN SCAN-STRUCT (PUNC WORD LIST-OF-SCANNED-RULES)
  (WHEN LIST-OF-SCANNED-RULES
    (LET ((NEW-MARKED-STACKS NIL)          ; unmodified (marked) stacks
          (OLD-MARKED-STACKS               ; previously unmodified (marked)
            (MARK-ALL-STACKS *SET-OF-STACKS)); stacks
          (SCANNED-SYMBOL)                 ; scanned terminal symbol
          (EXAMINE-STACK NIL)              ; old-marked-stack to examine
          (TOS))                           ; top-of-stack to examine
      ;; Loop through each rule in the list of scanned rules (produced by
      ;; SCAN), checking each stack against the scanned symbol.
      (LOOP FOR RULE IN LIST-OF-SCANNED-RULES DO
        (SETQ SCANNED-SYMBOL (COMPLETED-OR-SCANNED-SYMBOL RULE))
        ;; Loop through stacks that are marked (i.e., those preceded by a $).
        ;; Drop the scanned element into place if the current rule corresponds
        ;; to the top-of-stack configuration.
        (LOOP FOR STACK IN OLD-MARKED-STACKS DO
          (SETQ EXAMINE-STACK (CDR STACK) TOS (TOS EXAMINE-STACK))
          ;; Check that the current rule corresponds to the top-of-stack
          ;; configuration, and drop scanned word into place.  This is where
          ;; the GB component is consulted.  Since a head may have been
          ;; scanned, argument structure needs to be established, and feature
          ;; instantiation is required.
          (SETQ EXAMINE-STACK
                (CHECK-FOR-CORRECT-POSITION
                  (GET-CAT WORD) PUNC RULE WORD
                  SCANNED-SYMBOL TOS (CDR EXAMINE-STACK)))
          ;; If the stack has not been modified (i.e., the current rule does
          ;; not correspond to the top-of-stack item on the stack), then add
          ;; the marked ($) version to NEW-MARKED-STACKS.  Otherwise, add the
          ;; modified version to *TEMP-STACKS.
          (IF (EQ (CAR EXAMINE-STACK) '$)
              ;; unmodified
              (PUSH EXAMINE-STACK NEW-MARKED-STACKS)
              ;; modified (or null)
              (WHEN EXAMINE-STACK
                (VECTOR-PUSH-EXTEND EXAMINE-STACK *TEMP-STACKS 50))))
        ;; Reset the marked stacks for next iteration through loop
        (SETQ OLD-MARKED-STACKS NEW-MARKED-STACKS NEW-MARKED-STACKS NIL))
      ;; Set *SET-OF-STACKS to the successfully modified stacks and print out
      ;; all intermediate parses if *STRUCTURE-TRACE is T.
      (LOOP WHILE (NOT (ZEROP (FILL-POINTER *TEMP-STACKS))) DO
        (VECTOR-PUSH-EXTEND (VECTOR-POP *TEMP-STACKS) *SET-OF-STACKS)))))

;;; Check-for-correct-position helps SCAN-STRUCT determine whether it is okay
;;; to drop a scanned-word into the corresponding scanned-slot at the top of
;;; the stack. This decision is dependent on the configuration of the
;;; top-of-stack item, as well as the rule that has been scanned.
;;; Substitute-scanned-node ensures argument structure is established and that
;;; feature percolation from Head to Maximal projection is performed.
(DEFUN CHECK-FOR-CORRECT-POSITION
       (CATS PUNC SCANNED-RULE SCANNED-WORD SCANNED-SYMBOL TOS REST)
  (LET* ((RHS (CDDR SCANNED-RULE))              ; RHS of rule
         (NUMBER-TO-SCAN-RHS                     ; number of scanned-symbols to
           (COUNT-TO-SCAN-RHS SCANNED-SYMBOL     ; right of dot in rhs of rule
                              (CDR (MEMBER '* RHS))))
         (NUMBER-TO-SCAN-TOS                     ; number of empty scanned-symbols
           (COUNT-TO-SCAN-TOS                    ; in top-of-stack item
             SCANNED-SYMBOL
             (CDR (MEMBER (LIST SCANNED-SYMBOL)
                          (CONVERT-TO-TREE (CDR TOS)) :TEST #'EQUAL))))
         (NUMBER-OF-SCANNED                      ; number of scanned-symbols to
           (COUNT-SCANNED SCANNED-SYMBOL         ; left of dot in rhs of rule
                          (REVERSE (CDR (MEMBER '* (REVERSE RHS))))))
         (NUMBER-OF-FILLED-CATS                  ; number of filled scanned-symbols
           (COUNT-FILLED-CATS SCANNED-SYMBOL     ; in top-of-stack item
                              (CONVERT-TO-TREE (CDR TOS))))
         (RESULT NIL))

    ;; If the number of scanned-symbols to the right of the dot in the rhs of
    ;; the rule match the number of empty scanned-symbols in the top-of-stack
    ;; item and if the number of scanned-symbols to the left of the dot in the
    ;; rhs of the rule is one more than the number of filled scanned-symbols in
    ;; top-of-stack item, drop the scanned-word into the appropriate slot in
    ;; the top-of-stack item. Otherwise, mark the stack (with a $).
    (IF
      (AND (EQUAL NUMBER-TO-SCAN-RHS NUMBER-TO-SCAN-TOS)
           (EQUAL NUMBER-OF-FILLED-CATS (1- NUMBER-OF-SCANNED))
           (EQUAL (ITEM-INCOMPLETE-NODES TOS) SCANNED-SYMBOL))
      ;; Rule and stack-item match: drop the scanned element into place (set up
      ;; argument structure and perform feature instantiation/percolation).
      (SETQ
        TOS
        (SUBSTITUTE-SCANNED-NODE
          CATS
          (SECOND SCANNED-WORD)
          (LIST SCANNED-SYMBOL (INTERN (STRING-UPCASE (CAR SCANNED-WORD))))
          (LIST SCANNED-SYMBOL)
          TOS)
        RESULT (CONS TOS REST))
      ;; rule and stack-item do not match
      (CONS '$ (CONS TOS REST))))

;;; Substitute-scanned-node substitutes NEW (a list) for first occurrence of
;;; OLD (a list) in the NODE-STACK-ITEM. Before a result is returned, the
;;; scanned word has its argument structure set up, and its features
;;; instantiated (from the features in the lexical entry).  Also, the features
;;; of the word are percolated up to the maximal projection via
;;; Percolate-features.
(DEFUN SUBSTITUTE-SCANNED-NODE (CATS FEATURES NEW OLD NODE-STACK-ITEM)
  ;; Run over each element in the node-stack until the first unscanned one is
  ;; found.  Make a node out of the word and instantiate the features of the
  ;; word in the node corresponding to the lexical category.  Finally,
  ;; percolate the features of the word up to the maximal projection and drop
  ;; the word-node into the position where the unscanned node was found.
  (LOOP FOR ELT IN NODE-STACK-ITEM DO
    (WHEN (AND (LISTP ELT) (EQUAL (LENGTH ELT) 1)
               (EQUAL (CAR OLD) (NODE-CAT (CAR ELT))))
      ;; Instantiate features of the word in the unscanned node (which
      ;; corresponds to the lexical category of the word).
      (INSTANTIATE-FEATURES
        *SOURCE-LANGUAGE *TARGET-LANGUAGE FEATURES (CAR ELT) (SECOND NEW))
      ;; Percolate features up to the maximal projection and complete the
      ;; substitution.
      (RETURN
        (PERCOLATE-FEATURES
          (SUBSTITUTE (LIST (CAR ELT) (MAKE-NODE :WORD (SECOND NEW))) ELT
                      NODE-STACK-ITEM :TEST #'EQUAL :COUNT 1))))))
```

H.3 Interface During POP Stage

```
;;; Pop-struct examines the rules added to the current state set via the
;;; COMPLETE function.  It drops completed template elements corresponding to
;;; the added rules into the second item on the stack.  If the completed symbol
;;; is part of a left-recursive cycle, two resulting stacks are generated: one
;;; constructed by dropping the left-recursive structure between the
;;; top-of-stack and second item before the completed portion is popped; and
;;; the other constructed by dropping the completed portion without adding a
;;; left-recursive structure.  The GB component is consulted by
;;; drop-into-next-lower in order to link traces, percolate features,
;;; check-empty category conditions, assign case and assign theta.
(DEFUN POP-STRUCT (LIST-OF-RULES)
  (LET ((RECURSIVE-RULES) (TEMP-ITEM) (TOS) (COMPLETED-SYMBOL NIL)
        (COMPLETED-SYMBOLS
          (REMOVE-DUPLICATES (GET-COMPLETED-SYMBOLS LIST-OF-RULES)))
        (STACK))
    ;; Loop over each parse-stack in *SET-OF-STACKS checking each one against
    ;; the list of completed symbols until nothing more can be popped.
    (LOOP WHILE (NOT (ZEROP (FILL-POINTER *SET-OF-STACKS))) DO
      ;; Pop off a stack and initialize temp-item and top-of-stack (feature
      ;; percolation and trace linking will be done in dropping stage).
      (SETQ STACK (VECTOR-POP *SET-OF-STACKS)
            TEMP-ITEM STACK TOS (TOS TEMP-ITEM))
      ;; Iterate over current stack until no more completed-symbols or no more
      ;; items can be popped.
      (DO* ((ITEM-POPPED T))
           ((OR (ITEM-INCOMPLETE-NODES TOS) (NOT ITEM-POPPED)))
        ;; Initialization for completed-symbol loop
        (SETQ ITEM-POPPED NIL COMPLETED-SYMBOL (NODE-CAT (CAR TOS)))
        ;; Recursive rules check: Drop a recursive rule in between TOS and
        ;; SECOND before popping if the completed-symbol is part of a recursive
        ;; cycle.
        (WHEN (SETQ RECURSIVE-RULES (POP-GET-RECURSE COMPLETED-SYMBOL))
          (LOOP FOR STACK IN
                     (DROP-RECURSE-INTO-NEXT-LOWER
                       TOS RECURSIVE-RULES TEMP-ITEM) DO
            (VECTOR-PUSH-EXTEND STACK *TEMP-STACKS 50)))
        ;; Now drop into SECOND without dropping a recursive rule first.
        (IF (AND (EQ COMPLETED-SYMBOL (CONVERT-TO-TREE (CAR TOS)))
                 (NOT (ITEM-INCOMPLETE-NODES TOS))
                 (MEMBER (CONVERT-TO-TREE (CAR TOS))
                         (HEADERS-OF (CONVERT-TO-TREE (SECOND-ITEM TEMP-ITEM)))))
            ;; Drop TOS into place performing feature percolation, case, theta
            ;; and trace tests.
            (SETQ TEMP-ITEM (DROP-INTO-NEXT-LOWER COMPLETED-SYMBOL TEMP-ITEM)
                  ITEM-POPPED T)
            ;; Put the temp-item into a list if it hasn't been processed.
            (SETQ TEMP-ITEM (LIST TEMP-ITEM))
        ;; Add the modified stack onto the final *TEMP-STACKS list and prepare
        ;; to iterate over the next stack.
        (WHEN TEMP-ITEM (VECTOR-PUSH-EXTEND TEMP-ITEM *TEMP-STACKS 50))))
    ;; Set *SET-OF-STACKS to the successfully modified stacks and print out all
    ;; intermediate parses if *STRUCTURE-TRACE is T.
    (LOOP WHILE (NOT (ZEROP (FILL-POINTER *TEMP-STACKS))) DO
      (VECTOR-PUSH-EXTEND (VECTOR-POP *TEMP-STACKS) *SET-OF-STACKS))))

;;; Drop-into-next-lower drops a completed non-terminal into the appropriate
;;; slot in the second item of the stack. It also performs trace linking,
;;; feature percolation, ECP checking, case assignment and theta-assignment.
(DEFUN DROP-INTO-NEXT-LOWER (COMPLETED-SYMBOL TEMP-STACK)
  (LET ((TOS NIL) (ELEMENT-TO-REPLACE NIL) (FINAL-RESULT NIL) (RESULTS))
    ;; Set up a-positions and a-bar-positions.
    (LOOP FOR PHRASE IN TEMP-STACK DO (SET-A-AND-A-BAR-POSITIONS PHRASE))
    ;; Drop the replacement element into place. Note: we must first link all
    ;; traces to antecedents.  Any that are unlinked will be discarded.
    (SETQ RESULTS (FIND-AND-LINK-TRACES TEMP-STACK))
    ;; If there are results (i.e., if trace-linking succeeds), then check
    ;; Binding conditions, perform feature percolation, ECP checking, case
    ;; assignment and theta assignment.  Finally, pop the top of the stack.
    (LOOP FOR RESULT IN RESULTS DO
      ;; Examine the top of stack element.
      (SETQ TOS (TOS RESULT))
      (UNLESS
        (OR
          ;; Check Binding conditions.
          (EQ (CHECK-BINDING-CONDITIONS TOS) 'FAILURE)
          ;; Do feature percolation.
          (EQ (PERCOLATE-FEATURES TOS) 'FAILURE)
          ;; Check language-specific effects.
          (EQ (CHECK-LANGUAGE-SPECIFIC-EFFECTS RESULT) 'FAILURE))
        ;; Percolate Tense and AGR up to INFL.
        (PERCOLATE-TENSE-AND-AGR TOS (CDR RESULT))
        ;; Set up all Government relationships
        (SET-UP-GOVERNMENT RESULT)
        (UNLESS
          (OR
            ;; Check empty-category requirements.
            (EQ (ECP TOS) 'FAILURE)
            ;; Perform case and theta assignment.
            (EQ (PERFORM-CASE-ASSIGNMENT TOS RESULT) 'FAILURE)
            ;; Perform theta assignment.
            (EQ (PERFORM-THETA-ASSIGNMENT TOS RESULT) 'FAILURE))
          ;; Drop TOS into place (into first incomplete element of second stack
          ;; item).
          (SETQ ELEMENT-TO-REPLACE
                (FIND-COMPLETED-SYMBOL-NODE COMPLETED-SYMBOL (SECOND RESULT)))
          (PUSH (SUBST TOS ELEMENT-TO-REPLACE RESULT :TEST #'EQ) FINAL-RESULT)))
          FINALLY (RETURN FINAL-RESULT))))
```

```
;;; Drop-recurse-into-next-lower creates a new stack for each left-recursive
;;; template. Each stack is constructed by inserting a recursive template
;;; between the first and second stack elements.  Then drop-into-next-lower is
;;; called to drop the completed item into place and perform GB constraint
;;; checking.
(DEFUN DROP-RECURSE-INTO-NEXT-LOWER (TOP-OF-STACK RECURSIVE-TEMPLATES TEMP-STACK)
  (LET ((RESULTS NIL))
    ;; Iterate over each left-recursive template dropping it into place unless
    ;; the top-of-stack element already has the recursive template on top.
    (LOOP FOR CYCLE IN RECURSIVE-TEMPLATES DO
      (UNLESS
        ;; Check that the cycle has not already been added
        (CHECK-CYCLE-EQUALITY CYCLE (CDR (CONVERT-TO-TREE TEMP-STACK)))
        (SETQ RESULTS
              (APPEND
                RESULTS
                (DROP-INTO-NEXT-LOWER
                  (NODE-CAT (CAR TOP-OF-STACK))
                  (CONS TOP-OF-STACK
                        (APPEND
                          (CONVERT-TO-NODES CYCLE) (CDR TEMP-STACK)))))))
      FINALLY (RETURN RESULTS))))
```

H.4 Interface During Replacement Stage

```
;;; Move-elements-back is called by Structural-replacement in order to drop
;;; moved elements back into their base positions.
(DEFUN MOVE-ELEMENTS-BACK (PARSE-TREE)
  (LET ((BASE-POSITION) (MOVED-ELEMENT))
    (COND ((NULL PARSE-TREE) NIL)
          ((SETQ MOVED-ELEMENT (ATOM PARSE-TREE))
           (WHEN (SETQ BASE-POSITION (NODE-ANTECEDENT MOVED-ELEMENT))
             (DROP-INTO-BASE-POSITION MOVED-ELEMENT BASE-POSITION)))
          (T (MOVE-ELEMENTS-BACK (CAR PARSE-TREE))
             (MOVE-ELEMENTS-BACK (CDR PARSE-TREE))))))
```

```
;;; Drop-into-base-position moves an element into a base position, unless the
;;; base position is not a *final* base position (i.e., it is an intermediate
;;; landing site on the way to the base position). If this is the case, the
;;; final base position is found via a recursive call to
;;; Drop-into-base-position. Finally, when the true base position is found, the
;;; moved element is placed there via Transfer-slot-values. Then all landing
;;; sites (intermediate and final) are evacuated via the function Evacuate.
(DEFUN DROP-INTO-BASE-POSITION (MOVED-ELEMENT BASE-POSITION)
  (LET ((NEW-BASE-POSITION))
    (UNLESS (NODE-TRACE MOVED-ELEMENT)
      (IF (NOT (SETQ NEW-BASE-POSITION (NODE-ANTECEDENT BASE-POSITION)))
          (TRANSFER-SLOT-VALUES MOVED-ELEMENT BASE-POSITION)
          (DROP-INTO-BASE-POSITION MOVED-ELEMENT NEW-BASE-POSITION)
          (EVACUATE BASE-POSITION))
      (EVACUATE MOVED-ELEMENT))))

;;; Collapse-structure merges equivalent structures by first removing all
;;; evacuated elements intervening between X-bar levels, and then joining the
;;; two equivalent structures into one. For example:
;;;        [INFL-MAX [N-MAX NIL] [INFL-MAX ...]]
;;;              ==> [INFL-MAX [INFL-MAX ...]] (Remove-evacuated)
;;;              ==> [INFL-MAX ...]            (Merge-equivalent-structure)
(DEFUN COLLAPSE-STRUCTURE (PARSE-TREE)
  (MERGE-EQUIVALENT-STRUCTURE (REMOVE-EVACUATED PARSE-TREE)))

;;; Evacuate sets the CDR of the tree to NIL.
(DEFUN EVACUATE (PARSE-TREE) (SETF (CDR PARSE-TREE) NIL))

;;; Removed-evacuated removes all NIL elements intervening between X-bar levels.
;;; For example:
;;; [INFL-MAX [N-MAX NIL] [INFL-MAX ...]]
;;;       ==> [INFL-MAX [N-MAX NIL] [INFL-MAX ...]]
(DEFUN REMOVE-EVACUATED (PARSE-TREE)
  (IF (ATOM PARSE-TREE)
      PARSE-TREE
      (IF (EVACUATED? PARSE-TREE) NIL
          (APPEND (LIST (REMOVE-EVACUATED (CAR PARSE-TREE)))
                  (WHEN (CDR PARSE-TREE)
                    (LIST (REMOVE-EVACUATED (CDR PARSE-TREE))))))))

;;; Merge-equivalent-structure joins two equivalent structures into one.
;;; For example: [INFL-MAX [INFL-MAX ...]] ==> [INFL-MAX ...]
(DEFUN MERGE-EQUIVALENT-STRUCTURE (PARSE-TREE)
  (IF (ATOM PARSE-TREE)
      PARSE-TREE
      (IF (SAME-CATEGORY-UNDER PARSE-TREE)
          (WHEN (CDR PARSE-TREE)
            (LIST (MERGE-EQUIVALENT-STRUCTURE (CDR PARSE-TREE))))
          (APPEND (LIST (MERGE-EQUIVALENT-STRUCTURE (CAR PARSE-TREE)))
                    (LIST (MERGE-EQUIVALENT-STRUCTURE (CDR PARSE-TREE)))))))

;;; Evacuated? determines whether an element has been evacuated.
(DEFUN EVACUATED? (PARSE-TREE)
  (AND (NULL (SECOND PARSE-TREE)) (NOT (THIRD PARSE-TREE))))

;;; Same-category-under determines whether an structure contains two equivalent
;;; structures.
(DEFUN SAME-CATEGORY-UNDER (TOS)
  (LET ((CONVERTED (CONVERT-TO-TREE TOS)))
    (AND (EQ (CAR CONVERTED) (CAR (SECOND CONVERTED))) (= (LENGTH CONVERTED) 2))))

;;; Replace-source-with-target determines the correct translation of heads and
;;; their arguments according to theta-role matching routines.  As each maximal
;;; projection is analyzed, the translation of the head is chosen, and then each
;;; argument of the head is positioned and structurally realized according to
;;; the requirements of the target language head.
(DEFUN REPLACE-SOURCE-WITH-TARGET (BASE-TREE)
  (LET* ((MAX (GET-MAX-ELT BASE-TREE))
         (HEAD (GET-HEAD-ELT MAX))
         (EXT (GET-EXT-ELT MAX BASE-TREE))
         (INT (GET-INT-ELT MAX))
         (TRANSLATION NIL)
         (EXTERNAL NIL)
         (INTERNAL NIL))
    ;; Select the head's translation and internal/external arguments.
    (MULTIPLE-VALUE-SETQ
      (TRANSLATION EXTERNAL INTERNAL)
      (MATCH-TRANSLATION-AND-ARGUMENTS EXT INT HEAD))
    ;; Set up the translation of the head.
    (SETF (NODE-WORD HEAD) TRANSLATION)
    ;; Position the arguments.
    (SET-UP-ARGUMENT-POSITIONING MAX EXTERNAL INTERNAL)
    ;; Structurally realize the arguments.
    (SET-UP-STRUCTURAL-REALIZATION MAX EXTERNAL INTERNAL)
    ;; Translate each of the arguments.
    (LOOP FOR ELT IN MAX DO
      (UNLESS (EQ (CAR ELT) HEAD) (REPLACE-SOURCE-WITH-TARGET ELT)))))
```

H.5 Interface During Generation Stage

```
;;; Move-alpha restructures a tree by:
;;;  1. adding/removing adjoined elements and specifiers.
;;;  2. changing the constituent order.
;;;  3. locating all positions that elements could move or adjoin to and
;;;     moving elements to these positions.
;;; It then applies GB constraints to ensure that the movement(s) is valid.
(DEFUN MOVE-ALPHA (TEST-TREE)
  (LET ((TEST-TREES
          ;; Drop base-generated adjunctions and specifiers into place.
          (GENERATE-CORRECT-ADJUNCTIONS-AND-SPECIFIERS
            *CURRENT-BASE-SPECIFIERS-AND-ADJUNCTION)))
    ;; Restructure the tree to be compatible with the constituent order of the
    ;; language.
    (GENERATE-CORRECT-CONSTITUENT-ORDER TEST-TREES *CURRENT-CONSTITUENT-ORDER)
    ;; Iterate over each test-tree locating substitution and adjunction
    ;; candidates.
    (LOOP FOR TEST-TREE IN TEST-TREES DO
      ;; Locate candidates for movement (i.e., those elements that could
      ;; potentially leave a trace behind).
      (LOOP FOR ELT IN (GET-MOVEMENT-CANDIDATES *CURRENT-TRACES TEST-TREE) DO
        ;; Find positions to which the element can move.
        (LOOP FOR POSSIBLE-SUBSTITUTION
                  IN (GET-SUBSTITUTION-CANDIDATES
                       ELT TEST-TREE
                       *CURRENT-CHOICE-OF-SPEC
                       *CURRENT-DERIVED-SPECIFIERS-AND-ADJUNCTION
                       *CURRENT-BOUNDING-NODES) DO
          (PUSH (LIST ELT POSSIBLE-SUBSTITUTION) SUBSTITUTION-CANDIDATES))
        ;; Find positions to which the element can adjoin.
        (LOOP FOR POSSIBLE-ADJUNCTION
                  IN (GET-ADJUNCTION-CANDIDATES
                       ELT TEST-TREE
                       *CURRENT-ADJUNCTION
                       *CURRENT-DERIVED-SPECIFIERS-AND-ADJUNCTION
                       *CURRENT-BOUNDING-NODES) DO
          (PUSH (LIST ELT POSSIBLE-ADJUNCTION) DERIVED-ADJUNCTION-CANDIDATES)))
      ;; Perform movement and adjunction.
      (PERFORM-MOVEMENT
        SUBSTITUTION-CANDIDATES DERIVED-ADJUNCTION-CANDIDATES TEST-TREE))
    ;; Now check GB constraints.
    (DO ((I 0 (1+ i)) (N (FILL-POINTER *SURFACE-TREES))) ((= I N))
      (SETQ TREE (AREF *SURFACE-TREES I) TOS (TOS TREE))
      ;; Set up a-positions and a-bar-positions.
      (SET-A-AND-A-BAR-POSITIONS PHRASE)
      (UNLESS
        ;; Check Binding conditions.
        (EQ (CHECK-BINDING-CONDITIONS TOS) 'FAILURE)
        ;; Set up all Government relationships
        (SET-UP-GOVERNMENT TREE)
        ;; Check empty-category requirements, perform case and theta assignment
        ;; and perform theta assignment.  If any of these fail, reject the
        ;; structure.
        (WHEN (OR (EQ (ECP TOS) 'FAILURE)
                  (EQ (PERFORM-CASE-ASSIGNMENT TOS RESULT) 'FAILURE)
                  (EQ (PERFORM-THETA-ASSIGNMENT TOS RESULT) 'FAILURE))
          (SETF (AREF *SURFACE-TREES I) NIL))))))

;; GENERATE synthesizes the surface words and then returns the surface sentence.
(DEFUN GENERATE (STACK) (SURFACE-FORM (MORPHER STACK 'GENERATE)))
```

Appendix I

Global Variables for UNITRAN

This appendix contains all global variables required for the Kimmo system, the Earley parser, the precompiler, the translation routines, and the GB component. In general, these variables are *not* user modifiable, but certain of them (the last five) are set up to be toggled by the user at after the system has been loaded.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                        KIMMO                   ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Source language variables for morphological analysis (Kimmo)
(DEFVAR *SOURCE-LANGUAGE)                      ; source language
(DEFVAR *SOURCE-DIC)                           ; source dictionary (for Kimmo)
(DEFVAR *SOURCE-AUT)                           ; source automata (for Kimmo)

;;; Target language variables for morphological analysis (Kimmo)
(DEFVAR *TARGET-LANGUAGE)                      ; target language
(DEFVAR *TARGET-DIC)                           ; target dictionary (for Kimmo)
(DEFVAR *TARGET-AUT)                           ; target automata (for Kimmo)

;;; Source morphological analysis of *CURRENT-SENTENCE
(DEFVAR *CURRENT-MORPH-SENTENCE)               ; morph source sentence
(DEFVAR *CURRENT-MORPH-CATS)                   ; morph cats of source sentence
(DEFVAR *CURRENT-MORPH-WORD)                   ; morph word of source sentence
(DEFVAR *REST-MORPH)                           ; rest of source sentence morph words
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                        EARLEY               ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Global Lists (Earley)
(DEFVAR *GRAMMAR-NAMES NIL)          ; for grammar names
(DEFVAR *DICTIONARY-NAMES NIL)       ; for dictionary names
(DEFVAR *CURRENT-SENTENCE NIL)       ; for sentence currently being parsed

;;; State set variables (Earley)
(DEFVAR *STATE-SET-NAMES NIL)        ; for state-set names
(DEFVAR *STATE-SET-NUMBER 0)         ; running tally of the state sets
(DEFVAR *STATE-SET NIL)              ; current state-set
(DEFVAR *LIST-OF-LEFT-RECURSIONS)    ; left-recursive rules in each state
(DEFVAR *LAMBDA-RULES NIL)           ; lambda-rules in current state set
(DEFVAR *NUMBER-OF-WORDS)            ; number of words in input sentence

;;; Variables for parsing information (Earley)
(DEFVAR *CURRENT-GRAMMAR NIL)        ; for grammar rules (for Earley)
(DEFVAR *DICTIONARY NIL)             ; for dictionary entries (for Earley)
(DEFVAR *INITIAL-RULE NIL)           ; for unique starting rule
(DEFVAR *TERMINALS NIL)              ; for terminal symbols (lex categories)
(DEFVAR *NON-TERMINALS NIL)          ; for non-terminal symbols (phrases)
(DEFVAR *DERIVES-LIST NIL)           ; plist of all derived terminals
(DEFVAR *LEFT-DERIVES-TERMINALS NIL) ; plist of left-most derived terminals
(DEFVAR *LEFT-DERIVES-RULES NIL)     ; stacks pushed from a symbol (for PUSH)
(DEFVAR *LIST-OF-INDIRECT-CYCLES NIL) ; left-recursive cycles (for POP)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   VARIABLES USED PRECOMPILER (SET-UP-XBAR)    ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; If anything in *XBAR-PROCESS-LIST has been modified since the last execution
;;; of the system, the X-BAR preprocessor will need to be invoked for each
;;; language.
(DEFVAR *XBAR-PROCESS-LIST
    '(*CONSTITUENT-ORDER *BASIC-CATEGORIES *PRE-TERMINALS *CHOICE-OF-SPEC
      *ADJUNCTION *EMPTY-FEATURE-HOLDERS *OPTIONAL-SPECIFIERS *PRO-DROP
      *TRACES))

;;; 4 language-independent X-bar adjunction skeletons.
(DEFVAR *ADJUNCTION-SKELETONS
  '(1 (X-MAX ==> ADJUNCT X-MAX)
    2 (X-MAX ==> X-MAX ADJUNCT)
    3 (X ==> ADJUNCT X)
    4 (X ==> X ADJUNCT)))
```

```
;;; Variables set up by SET-UP-XBAR (based on *pro-drop,
;;; *current-empty-feature-holders and *current-traces).
(DEFVAR *CURRENT-EMPTY NIL)
(DEFVAR *CURRENT-POSSIBLE-EMPTY-ELEMENTS)
(DEFVAR *CURRENT-HEAD-FIRST? NIL)

;;; Variables set up when DEFGRAMMAR is executed.
(DEFVAR *CURRENT-SENTENCE-NODE NIL)
(DEFVAR *CURRENT-GRAMMAR-NAME NIL)
(DEFVAR *CURRENT-COMBINED-TERMINALS NIL)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;         VARIABLES FOR TRANSLATION ROUTINES      ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Names of files contain parameter settings.
(DEFVAR *PARAMETER-SETTINGS-FILENAMES NIL)

;;; Parameter names (for file loading only).
(DEFVAR *LANGUAGE-PARAMETER-SETTINGS '(FEATURES MATCH-LISTS SPLITS-AND-MERGES))

;;; Translation system parameter settings.
(DEFVAR *CURRENT-FEATURES NIL)
(DEFVAR *CURRENT-MATCH-LISTS NIL)
(DEFVAR *CURRENT-SPLITS-AND-MERGES NIL)

;;; Changes to T if global parameters have been loaded.
(DEFVAR *GLOBAL-PARAMETER-LOAD? NIL)

;;; Record of all GB parameters to be loaded.
(DEFVAR *GLOBAL-PARAMETERS
    '(*CONSTITUENT-ORDER *BASIC-CATEGORIES *PRE-TERMINALS *CHOICE-OF-SPEC
      *ADJUNCTION *EMPTY-FEATURE-HOLDERS *OPTIONAL-SPECIFIERS
      *CLITICS *CLITIC-DOUBLING *CANONICAL-SEMANTIC-MAPPINGS *GOVERNORS
      *CASE-ASSIGNMENT *PRO-DROP *TRACES *ECP *BOUNDING-NODES
      *LANGUAGE-SPECIFIC-EFFECTS *BASE-SPECIFIERS-AND-ADJUNCTION
      *DERIVED-SPECIFIERS-AND-ADJUNCTION))

;;; Structures used to hold results from each of the translation stages.
(DEFVAR *SET-OF-STACKS
        (MAKE-ARRAY '(1000) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
(DEFVAR *TEMP-STACKS
        (MAKE-ARRAY '(1000) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
(DEFVAR *PARSE-TREES
        (MAKE-ARRAY '(200) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
(DEFVAR *OLD-PARSE-TREES
        (MAKE-ARRAY '(200) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
(DEFVAR *SOURCE-BASE-TREES
        (MAKE-ARRAY '(200) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
```

```
(DEFVAR *TARGET-BASE-TREES
        (MAKE-ARRAY '(200) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
(DEFVAR *SURFACE-TREES
        (MAKE-ARRAY '(200) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))
(DEFVAR *TARGET-RESULT
        (MAKE-ARRAY '(200) :INITIAL-ELEMENT NIL :ADJUSTABLE T :FILL-POINTER 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;          VARIABLES USED BY GB COMPONENT         ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; GB parameters
(DEFVAR *CONSTITUENT-ORDER NIL)               ; constituent-order parameter
(DEFVAR *BASIC-CATEGORIES NIL)                ; basic categories
(DEFVAR *PRE-TERMINALS NIL)                   ; pre-terminal categories
(DEFVAR *CHOICE-OF-SPEC NIL)                  ; choices of specs
(DEFVAR *ADJUNCTION NIL)                      ; choices of adjunctions
(DEFVAR *EMPTY-FEATURE-HOLDERS NIL)           ; feature-holding empty cats
(DEFVAR *OPTIONAL-SPECIFIERS NIL)             ; optional specifiers
(DEFVAR *CLITICS NIL)                         ; clitics (if there are any)
(DEFVAR *CLITIC-DOUBLING NIL)                 ; clitic-doubling parameter
(DEFVAR *CANONICAL-SEMANTIC-MAPPINGS NIL)     ; syntax --> semantics mappings
(DEFVAR *GOVERNORS NIL)                       ; governing nodes parameter
(DEFVAR *CASE-ASSIGNMENT NIL)                 ; case-assignment parameter
(DEFVAR *PRO-DROP NIL)                        ; pro-drop (agr-rich) parameter
(DEFVAR *TRACES NIL)                          ; trace categories
(DEFVAR *ECP NIL)                             ; ECP (chain-conditions) parameter
(DEFVAR *BOUNDING-NODES NIL)                  ; bounding-node parameter
(DEFVAR *LANGUAGE-SPECIFIC-EFFECTS NIL)       ; lang-specific movement effects
(DEFVAR *BASE-SPECIFIERS-AND-ADJUNCTION NIL)  ; base generated elements
(DEFVAR *DERIVED-SPECIFIERS-AND-ADJUNCTION NIL) ; derived elements

;;; Variables that are set up according to the parameter settings of the
;;; language currently being processed.
(DEFVAR *CURRENT-CONSTITUENT-ORDER NIL)
(DEFVAR *CURRENT-BASIC-CATEGORIES NIL)
(DEFVAR *CURRENT-PRE-TERMINALS NIL)
(DEFVAR *CURRENT-CHOICE-OF-SPEC NIL)
(DEFVAR *CURRENT-ADJUNCTION NIL)
(DEFVAR *CURRENT-EMPTY-FEATURE-HOLDERS NIL)
(DEFVAR *CURRENT-OPTIONAL-SPECIFIERS NIL)
(DEFVAR *CURRENT-CLITICS NIL)
(DEFVAR *CURRENT-CLITIC-DOUBLING NIL)
(DEFVAR *CURRENT-CANONICAL-SEMANTIC-MAPPINGS NIL)
(DEFVAR *CURRENT-GOVERNORS NIL)
(DEFVAR *CURRENT-CASE-ASSIGNMENT NIL)
(DEFVAR *CURRENT-PRO-DROP NIL)
(DEFVAR *CURRENT-TRACES NIL)
(DEFVAR *CURRENT-ECP NIL)
(DEFVAR *CURRENT-BOUNDING-NODES NIL)
```

```
(DEFVAR *CURRENT-LANGUAGE-SPECIFIC-EFFECTS NIL)
(DEFVAR *CURRENT-BASE-SPECIFIERS-AND-ADJUNCTION NIL)
(DEFVAR *CURRENT-DERIVED-SPECIFIERS-AND-ADJUNCTION NIL)

;;; Variables accessed by GB modules.
(DEFVAR *AGR-RICH NIL)                       ; trace module
(DEFVAR *CHAIN-CONDITIONS NIL)               ; trace module
(DEFVAR *CLITIC-THETA-TRANSMIT NIL)          ; theta module

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;         Flags that the user may toggle.         ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(DEFVAR *RECOGNIZE-ONLY NIL)          ; flag for recognizing vs. parsing
(DEFVAR *STRUCTURE-TRACE NIL)         ; trace for building structure
(DEFVAR *STATE-SET-TRACE T)           ; trace for printing out state set information
(DEFVAR *PREPROCESS T)                ; turn the preprocessor on
(DEFVAR *GB-TRACE T)                  ; turn the gb-trace on
```

Appendix J

Translation Routines of UNITRAN

This appendix shows all of the functions required for setting up the parameter values of the source language, performing precompilation, and translating a sentence. The translation includes preprocessing, morphological analysis, parsing, structural replacement, thematic substitution, and generation. Because the GB component will be accessed, Earley is called with the GB-COMPONENT? variable set to T.

```
;;; Translate asks for a source and target language and then sets up the source
;;; and target parameter values, as well as the morphological analyzers.  It
;;; then asks for an input sentence until the user says :QUIT.
(DEFUN TRANSLATE NIL
  (LET ((SOURCE NIL) (TARGET NIL) (INPUT NIL))
    ;; Loop until :QUIT initializing the source and target languages.
    (LOOP UNTIL (OR (EQ INPUT :QUIT) (EQ SOURCE :QUIT) (EQ TARGET :QUIT)) DO
      ;; Read source language.
      (SETQ INPUT NIL SOURCE (ENTER-SOURCE))
      (UNLESS (EQ SOURCE :QUIT)
        ;; Read target language.
        (SETQ TARGET (ENTER-TARGET))
        (UNLESS (EQ TARGET :QUIT)
          ;; Initialize source and target files to read in for morphological
          ;; analysis.
          (SET-UP-FILES SOURCE) (SET-UP-FILES TARGET)
          ;; Initialize source and target information for morphological
          ;; analysis.
          (INITIALIZE-SOURCE SOURCE) (INITIALIZE-TARGET TARGET)
          ;; Loop until new source or target language needed or :QUIT (exit
          ;; translation loop).
          (SETQ INPUT (READ-INPUT-SENTENCE SOURCE TARGET))))))))
```

```
;;; Read-input-sentence asks for an input sentence and runs through the
;;; translation loop until the user says :QUIT or :NEW (to enter new source and
;;; target languages)
(DEFUN READ-INPUT-SENTENCE (SOURCE TARGET)
  (LET ((INPUT NIL))
    (LOOP UNTIL (OR (EQ INPUT :QUIT) (EQ INPUT :NEW)) DO
      ;; Ask for input sentence
      (FORMAT
        T
        "~&Enter input sentence,
          :NEW for new source and target, or :QUIT to exit: ")
      (SETQ INPUT (RUN-THROUGH-TRANSLATION-LOOP SOURCE TARGET))
      FINALLY (RETURN INPUT))))

;;; Run-through-translation-loop checks that the sentence is in the correct
;;; format and that all input words are in the dictionary (i.e., are
;;; morphologically analyzable).  The sentence is then put through
;;; preprocessing, morphological analysis, parsing and translation. The user
;;; may continue to translate from source to target language, or may say the
;;; keyword :NEW meaning that a new source and input language are to be
;;; specified.  The keyword :QUIT ends the translation loop, and the keyword
;;; :ABORT allows the source and target language to stay the same, but the
;;; sentence to be re-entered. Also, the preprocessor may be turned on or off
;;; (*PREPROCESS defaults to T, but user may change it to NIL).
(DEFUN RUN-THROUGH-TRANSLATION-LOOP (SOURCE TARGET)
  (LET ((INPUT NIL))
  (SETQ INPUT (READ))
  ;; Loop until new source or target language needed, :QUIT (exit translation
  ;; loop) or :ABORT (enter new sentence).
  (LOOP UNTIL (OR (EQ INPUT :QUIT) (EQ INPUT :NEW) (EQ INPUT :ABORT)) DO
    ;; Prompt until sentence is in correct form or until :QUIT, :ABORT, :NEW
    (COND ((ATOM INPUT)
           (FORMAT
             T
             "~&Please enter input in list form,
             :NEW for new source and target, or :QUIT to exit: ")
           (SETQ INPUT (READ)))
          ;; test for no final punctuation
          ((FINAL-PUNC-MISSING? INPUT)
           (FORMAT
             T
             "~&Please terminate sentence with final punctuation,
             :NEW for new source and target, or :QUIT to exit: ")
           (SETQ INPUT (READ)))
          ;; Preprocess the input, check the spelling (perform morphological
          ;; analysis), and parse the sentence.
          (T (SETQ INPUT
                   (PREPROCESS-MORPH-PARSE-TRANSLATE INPUT SOURCE TARGET))))
      FINALLY (RETURN INPUT))))
```

```
;;; Preprocess-morph-parse-translate sets up grammar according to x-bar, and
;;; then sets up parsing structures based on the grammar.  It then performs
;;; preprocessing based on the language replacements, does a morphological
;;; analysis (checking the spelling), parses and translates the sentence.
(DEFUN PREPROCESS-MORPH-PARSE-TRANSLATE (INPUT SOURCE)
  (LET ((MORPH-INPUT NIL))
    ;; Set up source language-specific variables.
    (SET-UP-CURRENT-GLOBALS SOURCE)
    ;; Preprocess the input if the preprocessor has been turned on.
    (WHEN *PREPROCESS
      (SETQ INPUT (PRE-PROCESS-SPLITS-AND-MERGES INPUT *CURRENT-REPLACEMENTS)))
    ;; Check the spelling of the input by performing a morphological analysis.
    (FORMAT T "~&Checking spelling of ~a ..." INPUT)
    (MULTIPLE-VALUE-SETQ (MORPH-INPUT INPUT)
          (ASK-ABOUT-UNDEFINED-WORD
            INPUT (WORD-NOT-DEFINED-1? *SOURCE-AUT *SOURCE-DIC INPUT)))
    ;; If the user wants to quit, enter a new sentence, or enter new languages,
    ;; just set the input to the response (without doing any processing).
    ;; Otherwise, set up the current sentence, print out the morphological
    ;; analysis and parse the sentence.
    (UNLESS (OR (EQ INPUT :NEW) (EQ INPUT :QUIT) (EQ INPUT :ABORT))
      (FORMAT T "~&Morphological analysis done.")
      (PPRINT MORPH-INPUT)
      ;; Set up XBAR information (i.e., the grammar) and parameter settings of
      ;; the source language.
      (SET-UP-CURRENT-LANGUAGE SOURCE)
      ;; Set up input sentence and morphologically analyzed input sentence.
      (SETQ *CURRENT-SENTENCE INPUT INPUT MORPH-INPUT)
      (FORMAT T "~&Parsing ...")
      ;; Parse the sentence using the gb-component.
      (EARLEY INPUT T)
      ;; Perform structural replacement on the resulting parse-trees.
      (STRUCTURAL-REPLACEMENT *PARSE-TREES)
      ;; Perform thematic substitution on the base forms of the source language.
      (THEMATIC-SUBSTITUTION *SOURCE-BASE-TREES)
      ;; Perform structural movement (move-alpha) on the base forms of the
      ;; target language.
      (STRUCTURAL-MOVEMENT *TARGET-BASE-TREES)
      ;; Set up target language-specific variables.
      (SET-UP-CURRENT-GLOBALS TARGET)
      ;; Set up XBAR information (i.e., the grammar) and parameter settings of
      ;; the target language.
      (SET-UP-CURRENT-LANGUAGE TARGET)
      ;; Perform morphological synthesis on each of the surface trees.
      (MORPHOLOGICAL-SYNTHESIS *SURFACE-TREES)
      ;; Prompt user for new input sentence.
      (FORMAT
        T "~&Enter input sentence,
        :NEW for new source and target, or :QUIT to exit: ")
    (SETQ INPUT (READ)))) INPUT)

;;; Set up XBAR information (i.e., the grammar) and parameter settings of the
;;; current language.
(DEFUN SET-UP-CURRENT-LANGUAGE (LANGUAGE)
  (WHEN (NOT (BOUNDP *CURRENT-GRAMMAR-NAME))
    (FORMAT T "~&Building XBAR processor ...") (SET-UP-XBAR LANGUAGE))
  ;; Set up parameters for current language.
  (SET-UP-PARAMETERS))

;;; Earley loops over all possible morphological analyses of the input
;;; collecting all possible parses.  (Get-all-combination-cats ensures that
;;; ambiguity is accounted for.)
(DEFUN EARLEY (INPUT &OPTIONAL (GB-COMPONENT? NIL))
  (LET ((PARSE-COUNT 0))
    (SETQ *NUMBER-OF-WORDS (LENGTH INPUT))
    (CLEAR-INPUT)
    ;; Parse each possible sentence in the input
    (LOOP FOR SENTENCE IN (GET-ALL-COMBINATION-CATS INPUT) DO
      ;; Initialize parsing structures to NIL.
      (INITIALIZE-SENTENCE-STRUCTURES-TO-NIL)
      ;; Maintain parse count.
      (SETQ PARSE-COUNT (1+ PARSE-COUNT))
      (WHEN (= PARSE-COUNT 1) (SET-ARRAY-NIL *OLD-PARSE-TREES))
      (FORMAT T "~&Parse number ~a:" PARSE-COUNT)
      (FORMAT T "~&~A" (PARSE-SENTENCE SENTENCE GB-COMPONENT?))
      ;; Print out all possible parses for this sentence
      (FORMAT T "~2&There are ~A parses of ~A."
              (ARRAY-LENGTH *PARSE-TREES) *CURRENT-SENTENCE)
      (DO ((I 0 (1+ i)) (N (FILL-POINTER *PARSE-TREES))) ((= I N))
        (WHEN (= PARSE-COUNT 1)
          (VECTOR-PUSH-EXTEND (AREF *SET-OF-STACKS I) *PARSE-TREES))
        (PPRINT-NODES (AREF *PARSE-TREES I)))
      (FORMAT T "~&Hit any key to continue ...")
      (READ-CHAR)
      ;; Carriage return after each set of parses.
      (FORMAT T "~2&"))))

;;; Perform structural replacement on the parse-trees.
(DEFUN STRUCTURAL-REPLACEMENT (PARSE-TREES)
  (COPY-ARRAY-ELEMENTS PARSE-TREES *SOURCE-BASE-TREES)
  (DO ((I 0 (1+ i)) (N (FILL-POINTER *SOURCE-BASE-TREES))) ((= I N))
    (MOVE-ELEMENTS-BACK (AREF *SOURCE-BASE-TREES I))
    (COLLAPSE-STRUCTURE (AREF *SOURCE-BASE-TREES I))))
```

```
;;; Perform thematic substitution on the base forms of the source language.
(DEFUN THEMATIC-SUBSTITUTION (SOURCE-BASE-TREES)
  (COPY-ARRAY-ELEMENTS SOURCE-BASE-TREES *TARGET-BASE-TREES)
  (DO ((I 0 (1+ i)) (N (FILL-POINTER *TARGET-BASE-TREES))) ((= I N))
    (REPLACE-SOURCE-WITH-TARGET (AREF *TARGET-BASE-TREES I))
    (SET-UP-STRUCTURAL-REALIZATION (AREF *TARGET-BASE-TREES I))))

;;; Perform structural movement (move-alpha) on the base forms of the target
;;; language.
(DEFUN STRUCTURAL-MOVEMENT (TARGET-BASE-TREES)
  (COPY-ARRAY-ELEMENTS TARGET-BASE-TREES *TEMP-STACKS)
  (DO ((I 0 (1+ i)) (N (FILL-POINTER *TEMP-STACKS))) ((= I N))
    (MOVE-ALPHA (AREF *TEMP-STACKS I))))

;; Perform morphological synthesis on each of the surface trees.
(DEFUN MORPHOLOGICAL-SYNTHESIS (SURFACE-TREES)
  (COPY-ARRAY-ELEMENTS SURFACE-TREES *TARGET-RESULT)
  (DO ((I 0 (1+ i)) (N (FILL-POINTER *TARGET-RESULT))) ((= I N))
    (GENERATE
      (CHANGE-FEATS-TO-AFFIX
        (LEXICALIZATION (AREF *TARGET-RESULT I))))))
```

Appendix K

Examples

This appendix shows how the system handles various types of phenomena. Each example shows: the source language surface from, the morphologically analyzed input, the source language S-structure(s), the source language D-structure(s), the target language D-structure(s), the target language S-structure(s), and the target language surface form(s).

K.1 Free Inversion

a. Source Language Surface Form: *Vio a María Juan*

b. Morphologically Analyzed Input:

```
((ver V SG P3 PAST INTRANS (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (P-GOAL ANIMATE)) (SUBCAT (GOAL INANIMATE)))
 (a P (SUBCAT (N)))
 (mari^a N SG FEM ANIMATE PROPER P3)
 (juan N SG MASC ANIMATE PROPER P3))
```

c. Source Language S-structure:

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX e [nom masc p3 sg animate agent]]_i]
      [I E]
      [I-COMPLEMENT
         [V-MAX
            [V-MAX ver [past p3 sg]
               [V-COMPLEMENT
                  [P-MAX [P a]
                     [N-MAX [N mari^a]
                         [obj fem p3 sg animate goal]]]]]
            [N-MAX [N juan] [nom masc p3 sg animate agent]]_i]]]]
```

d. Source Language D-structure:

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX [N juan] [nom masc p3 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
         [V-MAX ver [past p3 sg]
            [V-COMPLEMENT
               [P-MAX [P a]
                  [N-MAX [N mari^a]
                     [obj fem p3 sg animate goal]]]]]]]
```

e. Target Language D-structure (and S-structure):

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX [N John] [nom masc p3 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
         [V-MAX see [past p3 sg]
            [V-COMPLEMENT
               [P-MAX [P a]
                  [N-MAX [N mary]
                     [obj fem p3 sg animate goal]]]]]]]
```

f. Target Language Surface Form: *John saw Mary*

K.2 Verb-Preposing

a. Source Language Surface Form: *Qué leyó Juan*

b. Morphologically Analyzed Input:

```
((que^ N INANIMATE NEUT P3 WH WH-PHRASE-A)
 (leer V SG P3 PAST INTRANS (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (GOAL INANIMATE)))
 (juan N SG MASC ANIMATE PROPER P3))
```

c. Source Language S-structure:

```
[C-MAX
  [C-SPEC [N-MAX [N que^] [obj goal inanimate]]_i]
  [C E]
  [C-COMPLEMENT
    [I-MAX [V leer [past p3 sg]]_j
      [I-MAX
        [I-SPEC [N-MAX [N juan] [nom masc p3 sg animate agent]]]
        [I E]
        [I-COMPLEMENT
          [V-MAX [past p3 sg]
            [V e]_j
              [V-COMPLEMENT
                [N-MAX e [obj goal inanimate]]_i]]]]]]]
```

d. Source Language D-structure:

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC [N-MAX [N juan] [nom masc p3 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
        [V-MAX [past p3 sg]
          [leer [past p3 sg]]
            [V-COMPLEMENT
              [N-MAX [N que^] [obj goal inanimate]]]]]]]]
```

e. Target Language D-structure:

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC [N-MAX [N John] [nom masc p3 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
        [V-MAX [past p3 sg]
          [read [past p3 sg]]
            [V-COMPLEMENT
              [N-MAX [N what] [obj goal inanimate]]]]]]]]
```

f. Target Language S-structures:

```
[C-MAX
  [C E]
  [C-COMPLEMENT
    [I-MAX
      [I-SPEC [N-MAX [N John] [nom masc p3 sg animate agent]]]
      [I E]
      [I-COMPLEMENT
        [V-MAX [past p3 sg]
          [read [past p3 sg]]
            [V-COMPLEMENT
              [N-MAX [N what] [obj goal inanimate]]]]]]]]
 [C-MAX
  [C-SPEC [N-MAX [N what] [obj goal inanimate]]_i]
  [C E]
  [C-COMPLEMENT
    [I-MAX [DO-AUX do [past p3 sg]]
      [I-MAX
        [I-SPEC [N-MAX [N John] [nom masc p3 sg animate agent]]]
        [I E]
        [I-COMPLEMENT
          [V-MAX [inf]
            [V read]
              [V-COMPLEMENT
                [N-MAX e [obj goal inanimate]]_i]]]]]]]
```

g. Target Language Surface Forms: *John read what* and *What did John read*

K.3 Subject-Aux Inversion (SAI)

a. Source Language Surface Form: *What did John read*

b. Morphologically Analyzed Input:

```
((what INANIMATE NEUT P3 WH WH-PHRASE-A)
 (do DO-AUX PL SG P3 P2 P1 PAST)
 (john N SG MASC ANIMATE PROPER P3)
 (read V SG P3 PAST INTRANS (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (GOAL INANIMATE))))
```

c. Source Language S-structure:

```
[C-MAX
   [C-SPEC [N-MAX [N what] [obj goal inanimate]]_i]
   [C E]
   [C-COMPLEMENT
     [I-MAX [DO-AUX do [past p3 sg]]
       [I-MAX
         [I-SPEC [N-MAX [N John] [nom masc p3 sg animate agent]]]
         [I E]
         [I-COMPLEMENT
           [V-MAX [inf]
             [V read]
               [V-COMPLEMENT
                 [N-MAX e [obj goal inanimate]]_i]]]]]]]
```

d. Source Language D-structure:

```
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX [DO-AUX do [past p3 sg]]
       [I-MAX
         [I-SPEC [N-MAX [N John] [nom masc p3 sg animate agent]]]
         [I E]
         [I-COMPLEMENT
           [V-MAX [inf]
             [V read]
               [V-COMPLEMENT
                 [N-MAX [N what] [obj goal inanimate]]]]]]]]]
```

e. Target Language D-structure:

```
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX [N juan] [nom masc p3 sg animate agent]]]
       [I E]
       [I-COMPLEMENT
         [V-MAX [past p3 sg]
           [leer [past p3 sg]]
             [V-COMPLEMENT
               [N-MAX [N que^] [obj goal inanimate]]]]]]]]
```

f. Target Language S-structures:

```
[C-MAX
   [C E]
   [C-COMPLEMENT
     [I-MAX
       [I-SPEC [N-MAX [N juan] [nom masc p3 sg animate agent]]]
       [I E]
       [I-COMPLEMENT
         [V-MAX [past p3 sg]
           [leer [past p3 sg]]
             [V-COMPLEMENT
               [N-MAX [N que^] [obj goal inanimate]]]]]]]]
[C-MAX
   [C-SPEC [N-MAX [N que^] [obj goal inanimate]]_i]
   [C E]
   [C-COMPLEMENT
     [I-MAX [V leer [past p3 sg]]_j
       [I-MAX
         [I-SPEC [N-MAX [N juan] [nom masc p3 sg animate agent]]]
         [I E]
         [I-COMPLEMENT
           [V-MAX [past p3 sg]
             [V e]_j
               [V-COMPLEMENT
                 [N-MAX e [obj goal inanimate]]_i]]]]]]]
```

g. Target Language Surface Forms: *Juan leyó qué* and *Qué leyó Juan*

K.4 Thematic Divergence

a. Source Language Surface Form: *El libro me gusta a mí*

b. Morphologically Analyzed Input:

```
((el DET MASC SG)
 (libro N INANIMATE MASC SG)
(me CL-DAT NEUT PRONOUN SG ANIMATE P1)
(gustar V SG P3 PRES (EXTERNAL (AGENT INANIMATE))
 (SUBCAT (PATIENT ANIMATE)))
(a P (SUBCAT (N)))
(mi^ N NEUT PRONOUN SG ANIMATE P1))
```

c. Source Language S-structure (and D-structure):

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX [DET el] [N libro] [p3 sg animate agent nom]]]
    [I E [pres p3 sg]]
    [I-COMPLEMENT
      [V-MAX [pres p3 sg]
        [V [CL-DAT me [p1 sg obj animate]] [V gusta]]
          [P-MAX [P a]
            [N-MAX [N mi^ ] [p1 sg animate patient obj]]]]]]]
```

d. Target Language D-structure:

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX [N me] [p1 sg animate patient obj]]]
    [I E [pres p3 sg]]
    [I-COMPLEMENT
      [V-MAX [pres p3 sg] [V like]
        [N-MAX [DET the] [N book] [p3 sg inanimate goal nom]]]]]]
```

e. Target Language S-structure:

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX [N I] [p1 sg animate agent nom]]]
    [I E [pres p3 sg]]
    [I-COMPLEMENT
      [V-MAX [pres p3 sg] [V like]
        [N-MAX [DET the] [N book] [p3 sg inanimate goal obj]]]]]]
```

f. Target Language Surface Form: *I like the book*

K.5 Raising

a. Source Language Surface Form: *John seems to hate cats*

b. Morphologically Analyzed Input:

```
((john N SG MASC ANIMATE PROPER P3)
 (seem V SG P3 PRES (SUBCAT PROPOSITION))
 (to INF I (SUBCAT (V)))
 (hate V INF (SUBCAT (GOAL)) (EXTERNAL (AGENT)))
 (cats N NEUT PL))
```

c. Source Language S-structure:

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX [N John] [p3 sg masc agent nom]]_i]
    [I E [pres p3 sg]]
    [I-COMPLEMENT
      [V-MAX [pres p3 sg]
        [V seem]
          [C-MAX
            [I-MAX
              [I-SPEC [N-MAX e [p3 sg masc agent nom]]]_i
              [I to [inf]]
              [I-COMPLEMENT
                [V-MAX [inf]
                  [V hate] [N-MAX [N cats] [pl neut goal obj]]]]]]]]]]
```

d. Source Language D-structure:

```
[C-MAX
  [I-MAX
    [I E [pres p3 sg]]
    [I-COMPLEMENT
      [V-MAX [pres p3 sg]
        [V seem]
          [C-MAX
            [I-MAX
              [I-SPEC [N-MAX [N John] [p3 sg masc agent nom]]]
              [I to [inf]]
              [I-COMPLEMENT
                [V-MAX [inf]
                  [V hate] [N-MAX [N cats] [pl neut goal obj]]]]]]]]]]
```

e. Target Language D-structure:

```
[C-MAX
   [I-MAX
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [pres p3 sg]
            [V parecer]
               [C-MAX
                  [I-MAX
                     [I-SPEC [N-MAX [N juan] [p3 sg masc agent nom]]]
                     [I [inf]]
                     [I-COMPLEMENT
                        [V-MAX [inf]
                           [V odiar] [N-MAX [DET los] [N gatos] [pl neut goal obj]]]]]]]]]]
```

f. Target Language S-structure:

```
[C-MAX
   [I-MAX
      [I-SPEC [N-MAX [N juan] [p3 sg masc agent nom]]$_i$]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [pres p3 sg]
            [V parecer]
               [C-MAX
                  [I-MAX
                     [I-SPEC [N-MAX e]$_i$]
                     [I [inf]]
                     [I-COMPLEMENT
                        [V-MAX [inf]
                           [V odiar] [N-MAX [DET los] [N gatos] [pl neut goal obj]]]]]]]]]]
 [C-MAX
   [I-MAX
      [I-SPEC [N-MAX e [pro p3 sg]]]
      [I E [pres p3 sg]]
      [I-COMPLEMENT
         [V-MAX [pres p3 sg]
            [V parecer]
               [C-MAX [C que]
                  [I-MAX
                     [I-SPEC [N-MAX [N juan] [p3 sg masc agent nom]]]
                     [I [pres subj p3 sg]]
                     [I-COMPLEMENT
                        [V-MAX
                           [V odiar] [N-MAX [DET los] [N gatos] [pl neut goal obj]]]]]]]]]]
```

g. Target Language Surface Form: *Juan parece odiar los gatos*, *Parece que Juan odie los gatos*

K.6 Clitic Doubling

a. Source Language Surface Form: *María le dio el libro a Juan*

b. Morphologically Analyzed Input:

```
((mari^a N SG FEM ANIMATE PROPER P3)
 (le CL-DAT NEUT PRONOUN P3 SG OBJ ANIMATE)
 (dar V SG P3 PAST
  (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (GOAL INANIMATE) (PATIENT ANIMATE)))
 (el DET MASC SG)
 (libro N INANIMATE MASC SG)
 (a P (SUBCAT (N)))
 (juan N SG MASC ANIMATE PROPER P3))
```

c. Source Language S-structure (and D-structure):

```
[I-MAX
  [I-SPEC [N-MAX [N mari^a] [p3 sg fem agent nom animate]]]
  [I E [past p3 sg]]
  [I-COMPLEMENT
    [V-MAX [past p3 sg]
      [V [CL-DAT le [p3 sg patient obj animate]] [V dar]]
      [V-COMPLEMENT
        [N-MAX [DET el] [N libro] [sg masc goal obj inanimate]]
        [P-MAX [P a] [N-MAX [N juan] [p3 sg masc patient obj animate]]]]]]]
```

d. Target Language D-structure (and S-structure):

```
[I-MAX
  [I-SPEC [N-MAX [N mary] [p3 sg fem agent nom animate]]]
  [I E [past p3 sg]]
  [I-COMPLEMENT
    [V-MAX [past p3 sg]
      [V give]
      [V-COMPLEMENT
        [N-MAX [DET the] [N book] [sg masc goal obj inanimate]]
        [P-MAX [P to] [N-MAX [N John] [p3 sg masc patient obj animate]]]]]]]
```

e. Target Language Surface Form: *Mary gave the book to John*

K.7 Null Subject

a. Source Language Surface Form: *Vi la película*

b. Morphologically Analyzed Input:

```
((ver V SG P1 PAST INTRANS (EXTERNAL (AGENT ANIMATE))
  (SUBCAT (P-GOAL ANIMATE)) (SUBCAT (GOAL INANIMATE)))
 (la DET FEM SG)
 (peli^cula N INANIMATE FEM SG))
```

c. Source Language S-structure (and D-structure):

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX e [nom pro p1 sg animate agent]]]
    [I E]
    [I-COMPLEMENT
      [V-MAX ver [past p1 sg]
        [V-COMPLEMENT
          [N-MAX
            [DET la]
            [N peli^cula] [obj fem p3 sg animate goal]]]]]]]
```

d. Target Language D-structure:

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX e [nom p1 sg animate agent]]]
    [I E]
    [I-COMPLEMENT
      [V-MAX see [past p1 sg]
        [V-COMPLEMENT
          [N-MAX
            [DET the]
            [N movie] [obj fem p3 sg animate goal]]]]]]]
```

e. Target Language S-structure:

```
[C-MAX
  [I-MAX
    [I-SPEC [N-MAX I [nom p1 sg animate agent]]]
    [I E]
    [I-COMPLEMENT
      [V-MAX see [past p1 sg]
        [V-COMPLEMENT
          [N-MAX
            [DET the]
            [N movie] [obj fem p3 sg animate goal]]]]]]]
```

f. Target Language Surface Form: *I saw the movie*

References

Abney, S. (1987) "Licensing and Parsing," *Proceedings of NELS 17*, University of Massachusetts at Amherst, Amherst, MA.

Barton, G. Edward, Jr. (1984) "Toward a Principle-Based Parser," Massachusetts Institute of Technology, Cambridge, MA, AI Memo 788.

Berwick, Robert C. (1985) *The Acquisition of Syntactic Knowledge*, MIT Press, Cambridge, MA.

Berwick, Robert C. (1985) "Class Notes, Natural Language Processing and Knowledge Representation," Massachusetts Institute of Technology, Cambridge, MA, Course 6.863.

Borer, H. (1984) *Parametric Syntax: Case Studies in Semitic and Romance Languages*, Foris Publications, Dordrecht.

Chomsky, Noam A. (1977) "On *Wh*-Movement," in *Formal Syntax*, Culicover, P. W., and T. Wasow (eds.), Academic Press, New York.

Chomsky, Noam A. (1981a) "Principles and Parameters in Syntactic Theory," in *Explanation in Linguistics*, Hornstein, N. and D. Lightfoot (eds.), Longman, London and New York.

Chomsky, Noam A. (1981b) *Lectures on Government and Binding*, Foris Publications, Dordrecht.

Chomsky, Noam A. (1986a) *Barriers*, MIT Press, Cambridge, MA.

Chomsky, Noam A. (1986b) *Knowledge of Language: Its Nature, Origin and Use*, MIT Press, Cambridge, MA.

Earley, Jay (1970) "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM* 13:2, 94–102.

Dorr, Bonnie J. (1988) "A Lexical Conceptual Approach to Generation for Machine Translation," Massachusetts Institute of Technology, Cambridge, MA, Ph.D. Proposal, AI Memo 1015.

Fong, Sandiway (1986) "Specifying Coordination in Logic," S.M. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Frazier, Lyn (1986) "Natural Classes in Language Processing," presented at the *Cognitive Science Seminar, November*, Cambridge, MA.

Gazdar, G., E. Klein, G. Pullum, and I. Sag (1985) *Generalized Phrase Structure Grammar*, Basil Blackwell, Oxford, England.

Guerssel, Mohamed, Kenneth Hale, Mary Laughren, Beth Levin, and Josie White Eagle (1985) "A Cross-Linguistic Study of Transitivity Alternations," .

Hale, K. (1973) "A Note on Subject-Object Inversion in Navajo," in *Issues in Linguistics: Papers in Honor of Henry and Renee Kahane*, B. Kachrue *et. al.* (eds.), University of Illinois Press, Urbana.

Jaeggli, Osvaldo Adolfo (1980) "On Some Phonologically Null elements in Syntax," Ph.D. thesis, Department of Linguistics and Philosophy, Massachusetts Institute of Technology.

Jaeggli, Osvaldo Adolfo (1981) *Topics in Romance Syntax*, Foris Publications, Dordrecht, Holland/Cinnaminson, USA.

Karttunen, L., and K. Wittenburg (1983) "A Two-Level Morphological Analysis of English," *Texas Linguistic Form* 22, 163–278.

Kashket, Michael B. (1987) "A Government-Binding Based Parser for Warlpiri, a Free-word Order Language," Master of Science thesis, Massachusetts Institute of Technology.

Kayne, Richard S. (1975) *French Syntax: The Transformational Cycle*, MIT Press, Cambridge, MA.

Kayne, Richard S. (1981) "ECP Extensions," *Linguistic Inquiry* 12:1, 93–133.

Levin, Beth, and Malka Rappaport (1985) "The Formation of Adjectival Passives," Center for Cognitive Science, Massachusetts Institute of Technology, Cambridge, MA., Lexicon Project Working Papers #2.

Lightfoot, David (1982) *The Language Lottery: Toward a Biology of Grammars*, MIT Press, Cambridge, MA.

Nassi, Robert J., Bernard Bernstein and Theodore F. Nuzzi (1965) *Review Text in Spanish Three Years*, Amsco School Publications, New York, NY.

Quicoli, C. (1976) "Clitic Movement in French Causatives," *Linguistic Analysis* 6:2.

Rappaport, Malka, and Beth Levin (1986) "What to Do with Theta-Roles," Center for Cognitive Science, Massachusetts Institute of Technology, Cambridge, MA., Lexicon Project Working Papers #11.

Riemsdijk, Henk van and Edwin Williams (1986) *Introduction to the Theory of Grammar*, MIT Press, Cambridge, MA.

Ristad, Eric S. (1986) "Computational Complexity of Current GPSG Theory," Massachusetts Institute of Technology, Cambridge, MA, AI Memo 894.

Rivas, A. (1977) "A Theory of Clitics," Ph.D. thesis, Department of Linguistics, Massachusetts Institute of Technology.

Rizzi, Luigi (1978) "Violations of *Wh* Island Constraint in Italian and the Subjacency Condition," *Montreal Working Papers in Linguistics.*

Sharp, Randall M. (1985) "A Model of Grammar Based on Principles of Government and Binding," M.S. thesis, Department of Computer Science, University of British Columbia.

Slocum, Jonathan (1984a) "METAL: The LRC Machine Translation System," Linguistics Research Center, University of Texas, Austin, Working Paper LRC-84-2.

Slocum, Jonathan (1984b) "Machine Translation: Its History, Current Status, and Future Prospects," Linguistics Research Center, University of Texas, Austin, Working Paper LRC-84-3.

Stockwell, Robert P., J. Donald Bowen and John W. Martin (1965) *The Grammatical Structures of English and Spanish*, University of Chicago Press, Chicago Illinois and London.

Strozer, J. (1976) "Clitics in Spanish," Ph.D. thesis, Department of Linguistics, UCLA.

Thiersch, Craig and Hans-Peter Kolb (1986) "Strict X-Bar Parsing: Prolegomena to a Government-Binding Parser," University of Tilburg and Univeristy of Tuebingen, The Netherlands.

Torrego, Esther (1984) "On Inversion and Some of Its Effects," *Linguistic Inquiry* 15:1, 103–129.

Wehrli, Eric (1984) "A Government-binding Parser for French," Institut pour les Etudes Semantiques et Cognitives, Universite de Geneve, Working Paper 48.

Wexler, Kenneth and M. Rita Manzini (1986) "Parameters and Learnability in Binding Theory," presented at the *Cognitive Science Seminar, September*, Cambridge, MA.