A Minimal GB Parser

Marwan Shaban
shaban@cs.bu.edu

October 26, 1993
BU-CS Tech Report # 93-013

Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215

Abstract:

We describe a GB parser implemented along the lines of those written by Fong [4] and Dorr [2]. The phrase structure recovery component is an implementation of Tomita's generalized LR parsing algorithm (described in [10]), with recursive control flow (similar to Fong's implementation). The major principles implemented are government, binding, bounding, trace theory, case theory, θ-theory, and barriers. The particular version of GB theory we use is that described by Haegeman [5].

The parser is minimal in the sense that it implements the major principles needed in a GB parser, and has fairly good coverage of linguistically interesting portions of the English language.

Contents

1 Introduction

The purpose of this paper is to describe the architecture of a GB (Government-Binding) parser which was recently implemented.

The reader is assumed to be familiar with some version of the linguistic theory of GB. Our primary sources for the linguistic theory were Haegeman's text [5], and the overview by Sells [7].

Appendix D shows the English corpus which was used to test the English language coverage of the parser. Each sentence in the corpus is marked as to whether it is currently parsed correctly.

The parser is implemented in Sun Common Lisp (a variant of Lucid) under SunOS 4.1.3, and also runs on clisp under Linux 0.99pl9.

Given the fact that parsers such as this one already exist, the reader may wonder why another one was implemented. The purpose of implementing our own parser was to garner experience in writing a principle-based parser, and to have in hand a parser that can be easily modified for future research.

2 Overall Architecture of The Parser

The implemented parser follows closely the GB parsing model pioneered in the late 1980s by Dorr [2], Fong [4], Wehrli [11], Kashket [6], and others. The parser consists of independent modules corresponding roughly to the modules of the GB theory of syntax.[1] Thus, there is a "government" module which assigns government relations to phrase structure constituents, a "theta" module which assigns theta roles and performs the theta criterion well-formedness check, and so on. Apart from certain inter-module dependencies, these modules can be applied in any order. A good study of the ordering of principles within a GB parser was done by Fong [4]. Our current parser is "minimal" in the sense that it contains enough components to achieve broad coverage of natural language, and contains most of the components ("principles") usually associated with the current linguistic theory.

Our parser uses fairly standard definitions of the principles. Its core phrase structure recovery component and X-bar module turn out to be fashioned after Fong's, and the rest of the modules are fashioned after the corresponding modules in Dorr's system, since her documentation seemed to be the most elaborate and accessible among the available descriptions of previous GB parsers.

The parser operates as follows. A covering grammar for s-structure is produced using the X-bar template structure and knowledge of what constituents may move (Currently NP, Adv, V, and I are the elements that can move) and

[1] The modules are independent in the sense that each of them corresponds to a distinct subsystem of the GB framework (e.g. the case module corresponds to Case Theory, and can rule out proposed parses based on Case Theory's well-formedness criteria), but are dependent in the sense that some must be applied before others (e.g. the government module must assign government relations before the case module can perform case assignment.)

their possible landing sites. For a sample s-structure grammar, see appendix A. This covering grammar is used to parse an input sentence and produce a set of under-specified parses. The parses produced by the LR parser are under-specified because they don't contain much of the relevant information such as government relations, etc., that relate to the analysis of these parses. These parses are then passed through a sequence of modules which perform three kinds of functions:

1. The first kind of module simply builds structure onto the parse tree that is input to it. It produces one tree for every tree that it takes, and simply makes the tree more specific in some way, such as adding features indicating what elements govern other elements.

2. The second kind of module is called a filter. It examines the input tree and either returns the tree unchanged (in which case the parse tree is accepted by the filter) or returns no tree at all (in which case the parse tree is said to have been rejected by the filter). An example filter is the Case Filter which examines overt NPs to make sure they have proper case features.

3. Finally, the last kind of module is called a generator. It takes a parse tree and returns one or more trees, all of which are based on the input tree, but are expanded in some way. An example generator is the "Form Chains" generator which takes a parse tree without chains and returns the same tree with all possible chain configurations applied to it. Thus, a generator can produce a large number of parses depending on the nature of the input tree.

The modules are applied in a particular sequence, each taking as input all parse trees produced by the previous module. If, after all modules have been applied, we get one or more parse trees out of the last module, we say that the sentence parsed successfully. Otherwise, the parse failed, and the input sentence is ungrammatical.

The modules cannot be applied in any arbitrary order. Some modules must be applied before others. For example, the Case Filter must be applied after the "Case Assignment" module whose job it is to assign structural case to NPs.

3 The LR Parser

Every GB parser needs to contain a context-free parser to recover the basic phrase structure of a sentence. The GB parsing literature varies substantially in this aspect. Fong uses an LR(1) parser, Dorr uses a modified Earley Algorithm, and Wehrli uses a bottom-up chart parser. We decided to implement Tomita's modified LR(1) parsing algorithm, which is the algorithm that Fong uses for his phrase structure recovery. Tomita [10] makes the claim that his algorithm

is more efficient than Earley's algorithm in the case where the grammar being used is "close" to LR.[2] As a simple measure of how close our present English s-structure grammar is to being LR, 12% of its action table entries have multiple actions, which can be considered fairly "close".

The LR parser flow of control that we use is similar to Fong's in that the parser is re-entrant and calls itself at each choice point (i.e., whenever multiple entries are encountered in the action table), and the partial parses (and their state stacks) are kept on the parser's control stack.

The X-bar s-structure grammar that we use is similar to those of Fong and Dorr. Appendix A shows our English s-structure grammar. It is derived from the structure of the basic X-bar phrase, coupled with information about what complements, specifiers, and adjuncts each phrase type can take. Trace rules are added to account for movement.

4 The Government Module

The government module assigns government relations to constituents in each parse tree constructed by the LR parser. Several types of government are distinguished, each having a separate role in the overall parsing process.

4.1 Head Government

Head-Government is set up using the following definition (from [7]):

> α head-governs β iff:
>
> 1. α c-commands β.
> 2. α is a minimal projection.
> 3. Every maximal projection dominating β dominates α.

The c-command relation is defined as follows (from [5]):

> A node α c-commands a node β iff:
>
> 1. α does not dominate β.
> 2. β does not dominate α.
> 3. The first branching node dominating α also dominates β.

We did not implement the above definitions directly. Rather, we deduced the structural relation that exists between a governed constituent and its governor within a phrase. A function was then coded to go through each phrase of a tree and assign the "governs" and "governed-by" features to each applicable constituent. Currently, these features consist of lists of node IDs. Each node in

[2] In general, the less ambiguous a grammar is, the "closer" it is to LR.

the phrase is assigned a unique ID. The procedure follows these steps for each maximal projection node in the tree:

1. Go through the subtree (descendants) of this node (maximal projection) and collect the node IDs of all nodes in this subtree. Ignore everything below a maximal projection as we travel down the subtree. The collected node IDs belong to the nodes to be governed.

2. Go through the subtree again in the same manner, except this time we look for minimal projections and collect their IDs. These minimal projections will be the governors.

3. Go through the subtree again, locating the minimal projections (governors) and setting their government-related features.

4. Go through the subtree again, and set the government-related features of the governed nodes.

4.2 Theta Government

Theta government relations are needed when setting up barriers (see below). The definition of Theta Government is as follows (from [5]):

> When a V governs an element and assigns an internal theta role to it we say that it theta-governs this element.

The theta module (see below) sets up theta government while processing internal theta roles.

4.3 Barriers

Barriers are needed when setting up government relations. The definition of a barrier is as follows (from [5]):

> α is a barrier for β iff at least one of the following two conditions is met:
>
> 1. α is a maximal projection and α immediately dominates γ, γ is a BC (blocking category) for β.
> 2. α is a BC for β, α is not IP.

Blocking categories (BCs) are defined as follows (from [5]):

> γ is a BC for β iff γ is not L-marked and γ dominates β.

L-marking is defined as follows (from [5]):

α L-marks β iff α is a lexical category that theta-governs β.

Three passes are performed on each hypothesized parse tree to set up L-marking relations, blocking categories, and barrier nodes. Currently, the barrier information within a tree is only used in setting up government relations, although it could also be used in the bounding module as well (as discussed in [5]).

4.4 Government

Government is defined as follows (from [5]):

> X governs Y iff:
>
> 1. X is either
>   (a) one of the categies A, N, V, P, or I;
>   (b) or, X and Y are coindexed (i.e. the governor is allowed to be a maximal projection if it is coindexed with the governed element. This includes the possibility of the governor being a trace).
> 2. X c-commands Y.
> 3. No barrier intervenes between X and Y.
> 4. Minimality is respected.

The minimality condition on government is:

> There is no Z such that Z satisfies the first three conditions in the above government definition, and X c-commands Z.

The procedure used to set up government is the same as that used to set up head government, except that in the first step (when collecting the identities of governed nodes), we don't stop descending the subtree at every maximal projection. Rather, we only stop descending the subtree if we encounter a barrier. Also, in the second step (when collecting the identities of the governing nodes) we don't restrict our selection to minimal projections. Maximal projections can also govern. In our system, maximal projections can govern even if they are not coindexed with the governed element. We perform the checking of the coindexation condition later when proper government is set up.

The minimality condition on government is not implemented in the current system. Because of this, certain incorrect parses are generated by the system. For example, the parser accepts "Who do you think that came" (see discussion of *that*-trace effect in [5], pp. 456-7).

4.5 Proper Government

Proper government is used in processing the Empty Category Principle (ECP, see below). It is defined as follows:

> α is properly governed iff:
>
> 1. α is governed.
> 2. α's governor is either lexical (V, A, N, or P) or coindexed with α (they are members of a movement chain).

Since this definition refers to chain indices, proper government cannot be set up until movement chains have been set up.

5 The Case Module

5.1 Case Assignment

Two kinds of case assignment are performed:

1. Lexical information about case is used to assign inherent case.
2. Structural case is assigned according to these rules (from [3]):
   - Objective case is assigned to the object governed by transitive P or V (passive verbs, however, cannot assign case).
   - Possessive case is assigned to the object governed by transitive N (Currently, no transitive nouns are handled by the parser).
   - Nominative case is assigned to the subject governed by [I [+ tns]].

If there is a noun phrase with two different cases being assigned to it, the parse is ruled out due to case conflict.

Note that tense transmission (see below) must be performed before structural case assignment, so that the inflection (I) node gets a tense feature (if appropriate) from the main verb.

5.2 Case Transmission

Each movement chain can transmit case from trace to antecedent or from antecedent to trace. The parser checks to make sure all elements of a chain have the same case. In addition, case clash is checked (as it was during case assignment, see above).

5.3 The Case Filter

A well-formedness check is made on each hypothesized parse tree to make sure the case filter is satisfied. The case filter is defined as follows:

> Every overt (non-empty) noun phrase must possess case.

6 The Theta Module

The theta module operates in much the same way as the case module does. It performs the following three operations on each parse tree.

6.1 Theta Role Assignment

Theta roles are assigned to the internal and external arguments of all phrase heads which act as predicates. Phrase heads which act as predicates include V, A, P, and overt complementizers. We need to consider overt complementizers (C) as predicates so that sentences such as "John said that he is ill" are accepted, where "he is ill" needs to receive a theta role (otherwise it fails the theta criterion check).

Subcategorization information in the lexicon is used to determine what theta roles to assign to constituents in internal and external argument positions.

The "Visibility Condition" dictates that NPs must be assigned case before they are visible for theta role assignment. Thus, we must assign and transmit case before theta roles can be assigned, and check that NPs are visible before assigning them theta roles.

6.2 Theta Role Transmission

Theta roles are transmitted from traces to antecedents. Theta roles are not transmitted from antecedents to traces since the linguistic theory asserts that theta role assignment occurs at d-structure, thus the assignment takes place before movement (Dorr [2]).

6.3 The Theta Criterion

The theta criterion well-formedness check is applied to all hypothesized parse trees. If an argument (of a predicate-like phrase head) receives no theta roles, or receives more than one role, or if a theta role fails to be assigned, then the parse is rejected.

7 The Trace Module

7.1 Trace Insertion

The LR parsing module can produce trees containing empty NPs because the s-structure grammar contains the rule "NP $\rightarrow \epsilon$". As far as the LR parser is concerned, each such NP could be of any of the four possible empty NP types ("pro", "PRO", np-trace, or *wh*-trace). At some point, the trace module is applied to each parse tree to determine the type of each empty NP. We use a simple procedure to determine what type(s) the empty NP could be, and for each hypothesized type of empty NP, produces a version of the parse tree with the empty NP having a child of the hypothesized type. For example, if the module determined that a particular empty NP is either a *wh*-trace or "PRO", it generates two trees, one with the empty NP specified as a *wh*-trace, and one with it specified as "PRO".

The alternative to this procedure is to posit each of the four empty NP types for each empty NP in each tree, and let the trees containing the incorrect NP types be filtered out somewhere later in the parse process. This is undesirable because the number of trees generated by the procedure would be much larger than the case where we make intelligent hypotheses about the possible types of empty NPs before we generate the result trees.

Note that base-generated empty NPs (other than "pro" and "PRO") are not handled by the current system.

The procedure used to determine what type an empty NP could be is the following:

- If the empty NP is governed by the Inflection (I) node, it is either "pro" or "PRO". If the language has "rich agreement" (as flagged by the `*agr-rich?*` parameter, we insert "pro". Otherwise, we insert "PRO".
- If the empty NP is not governed by the Inflection (I) node, then it is either an NP-trace or a *wh*-trace. After movement chains are constructed, we can distinguish between *wh*-traces and NP-traces using the following facts found in Haegeman [5]:
  - An np-trace must not have case.
  - A *wh*-trace must not have case, unless its antecedent is an NP, in which case it must have case.
  - The antecedent of an np-trace must be an NP.
  - The antecedent of a *wh*-trace can be any maximal projection type.

  Based on the above facts, to distinguish between np-traces and *wh*-traces, we wait until case has been transmitted, then apply the following rules:

- If the target (antecedent) is an NP, the case determines the type of the trace (case = *wh*-trace, no case = np-trace).
- If the target is not an NP, the trace must be a *wh*-trace.

The empty NP type determination process occurs in two steps. First, we determine whether the empty NP is "pro", "PRO", or a trace. Second, nodes marked "trace" are determined to be either *wh*-trace or np-trace.

7.2 The Empty Category Principle (ECP)

The Empty Category Principle asserts that all NP traces must be properly governed ("pro" and "PRO" do not count as traces). We distinguish between intermediate traces and traces that head a movement chain. In our system, the ECP does not apply to intermediate traces. This simplification is mostly valid. Haegeman [5] (on p. 465) claims that having intermediate nodes immune to the ECP produces almost correct results. In fact, Haegeman couldn't give an example of a non-sentence that fails purely on the basis of an intermediate trace being ruled out by the ECP.

7.3 Control

The only aspect of the theory of Control that we implement is the condition that "PRO" must be ungoverned (Haegeman [5], p. 251).

8 The Bounding Module

The bounding module performs the following tasks:

- Takes each tree and expands it to include all possible movement chain combinations, producing one or more possible parse trees. A possible movement chain consists of one nonempty NP and one or more empty NPs.
- Checks each tree for possible violation of bounding conditions. The bounding condition is defined as follows: Each two consecutive elements in a movement chain (i.e. one hop of the movement) must not be separated by more than one bounding node. The bounding nodes of a particular language are specified in the parameter `*bounding-nodes*` (the bounding nodes for English are NP and IP). Operationally, to tell whether two constituents are separated by more than one bounding node, we use the following algorithm:

  1. We perform a depth-first traversal of the parse tree.
  2. When one of the two constituents is seen, we start a counter.

3. After the counter is started, bounding nodes which are passed (either going down or up) cause the counter to be incremented.
4. If a bounding node is encountered for the second time, we decrement the counter. This happens when we encounter a bounding node on the way down and then encounter it again on the way back up.
5. When the other constituent is seen, we record the value of the counter. If its value is more than 1, the two constituents are separated by more than one bounding node.

- Checks the landing sites in a tree to make sure that each non-empty landing site is occupied by a member of a chain. If not, the parse is rejected since the element occupying the landing site could not have been base generated. Currently, the only landing site checked in this way is the [spec, CP] position (specifier of CP).
- Checks each chain of each parse tree to make sure that the chain doesn't contain "pro" or "PRO". If such a case is found, the parse is rejected. This helps to weed out bad parses early.

9 The Binding Module

The binding module performs the following tasks:

- Takes each tree and expands it to include all possible NP coindexations, producing one or more possible parse trees. Fong [4] gave an account of the characteristics (and complexity) of the NP coindexation problem. We do not attempt to optimize the coindexation problem to achieve better than exponential performance (by using characteristics of NP coindexation which would allow us to rule out certain coindexations, thereby generating fewer trees with coindexed NPs). Consequently, the time required to perform NP coindexation is quite long if the number of NPs to be coindexed is more than 7 or so.
- Checks each tree for possible violation of binding conditions. The binding conditions checked are the following:

  - An anaphor ([+a]) must be A-bound in its governing category.
  - A pronominal ([+p]) must not be A-bound in its governing category.
  - A referential-expression ([-a,-p]) must be free everywhere.

  The definitions of Governing Category (GC) and A-bound are:

A constituent α's governing category is defined as the smallest NP or IP containing α and a governor of α.

A constituent β is A-bound if it is coindexed with a constituent α in an A-position and α c-commands β.

In our system, A-positions are argument positions of phrase heads. These include the external argument position and all internal argument (complement) positions.

To find an element's governing category, we use the following algorithm:

1. Make a list of NPs and IPs which dominate the node. Do this by going through the tree and keeping a stack of NPs and IPs encountered, and recording the state of the stack when the node is hit. This would also record the order in which the NPs and IPs dominate the node. Note that the tree nodes do not contain back-pointers to their parents, which would simplify this procedure greatly.
2. Make a list similar to the above for each governor of the node.
3. Merge all lists made in step 2.
4. Select from the list made in step 1 and the list made in step 3 the smallest (closest) NP or IP which occurs in both lists.

To determine whether a constituent c-commands another, we perform the following procedure:

1. Find the smallest maximal projection containing the c-commander by performing a depth-first traversal of the tree, and keeping a stack (log) of maximal projections encountered, and seeing who is on top of the stack when the c-commander is hit.
2. Determine whether the c-commanded element is under the maximal projection found in step 1 by going through the subtree of the maximal projection found in step 1.

Finally, to determine whether a constituent is in an A-position, we assume that A-positions correspond one-to-one with theta roles assigned previously in the theta module, and simply look for the presence of a theta role in the constituent's feature list.

10 Miscellaneous Operations

In this section we discuss a few minor parser operations that have not been explained above. These operations are either done for efficiency (to rule out a bad parse at the earliest possible time) or are necessary for correctness (to rule out a parse that would otherwise erroneously be accepted or to perform some operation without which a correct parse would be rejected).

10.1 Tense Transmission

This operation transmits tense from a verb to its corresponding inflection (I) node. Only one of the two may already possess tense before the tense transmission operation. Otherwise the parse is rejected. The tense transmission corresponds roughly to an 'I' node lowering to a verb node to receive tense. The 'I' node needs tense in order to be a valid case assigner (see the above discussion on case assignment).

10.2 Checking Head Movement

This operation checks certain configurations related to head movement. The two cases of head movement that are considered are 'V' to 'I' raising, and 'I' to 'C' raising. The operations performed are the following:

- In every IP, the 'I' and 'V' nodes are checked to make sure that exactly one of them is lexical. Otherwise, the parse is bad. Note that for now, we are ignoring the possibility that 'V' raises to 'I' then moves further.
- If, in a certain IP, 'I' is lexical and 'V' is not, we must coindex them, as this case is not handled by the chain formation mechanism (which currently only deals with NP movement, and I-to-C movement). This coindexation is performed only if there is intervening (overt) material between the 'V' and 'I' nodes. In the case where there is no overt material between the 'V' and 'I' nodes, it doesn't make sense for 'V' to raise to 'I', so we rule out the parse.
- In a manner similar to the above analysis of V-to-I raising, we look within every CP to analyze I-to-C raising. If 'C' contains an auxiliary and 'I' is empty, we coindex them, unless there is no intervening material between the 'C' and 'I' nodes, in which case the parse is rejected.
- Since [spec, CP] and [spec, IP] can both contain NPs, we check if [spec, CP] contains an NP and [spec, IP] is empty, that there is material between the two positions. Otherwise, we rule out the parse in favor of an alternate interpretation where [spec, IP] contains the NP (which would be hypothesized in another parse tree since the LR parser will predict both cases). An exception to this procedure is if the pro-drop parameter is on. In such a case, the fact that [spec, IP] is empty may not necessarily indicate that the it is the source of the movement (it could contain "pro").

The above actions that are performed by the head-movement module (especially the last one) are not based on well-established linguistic theory (as is the case with other modules such as binding, etc.) but rather were coded to remedy certain problems that showed up during system testing. They are based on pragmatic considerations, and as such, should be regarded with more skepticism than other principles whose operation is well understood and accepted.

10.3 Checking NP traces

Each parse tree's NP traces are checked to make sure that each one is part of a movement chain. Otherwise, the parse is rejected.

11 Order of Applying Principles

There are various constraints on the order in which principles are applied when parsing. Here are the constraints that must be observed:

- LR parsing must be done before anything else.
- Head government relations must be set up before L-marking is set up, and before NP traces are inserted since the trace insertion procedure needs to know whether an NP is governed by an inflection (I) node.
- Chains must be formed before bounding conditions are checked, before case and theta roles are transmitted, before distinguishing trace types, before checking landing sites, before transmitting tense (in case the 'I' node has moved to 'C' for example), before checking NP traces, and before binding conditions are checked.
- Tense transmission must be performed before structural case assignment.
- Traces must be inserted before distinguishing trace types, before running the ECP check, and before checking bounding conditions (because the bounding module marks intermediate traces for use by the ECP).
- Case must be assigned and transmitted before the case filter is applied, before assigning theta roles (due to the visibility condition), and before trace types are distinguished.
- Theta roles must be inserted and transmitted before the theta criterion is applied and before theta government is set up.
- Theta government must be set up before L-marking.
- L-marking must be set up before blocking categories are set up.
- Blocking categories must be set up before barriers can be set up.
- Barriers must be set up before government relations can be set up.
- Government relations must be set up before proper government relations can be assigned.
- Trace types must be distinguished before checking head movement, before checking chains, and before checking NP traces.

- NP coindexation must be applied before proper government relations can be set up, and before binding conditions can be checked.
- Proper government relations must be set up before the ECP is applied.

Currently, the system applies the principles in the following order:

1. LR parsing is performed.
2. Head government relations are set up.
3. Movement chains are formed.
4. Landing sites are checked.
5. Traces are inserted into the tree. At this point, we cannot distinguish between NP-traces and WH-traces, because case has not been assigned.
6. Bounding conditions are checked.
7. Tense features are transmitted.
8. Inherent and assigned case are attached to NPs, and transmitted between elements of a movement chain.
9. The case filter is applied to each parse tree.
10. Theta roles are assigned and transmitted from traces to antecedents.
11. The theta criterion is applied to each tree.
12. L-marking is performed.
13. Blocking categories are set up.
14. Barriers are set up.
15. Government relations are set up.
16. Trace types are distinguished since we now have case information.
17. Head movement is checked.
18. Chains are checked.
19. NP traces are checked.
20. NPs are coindexed.
21. Proper government relations are assigned.
22. The ECP is applied to each tree.

23. Binding conditions are checked.

Figure 1 shows the dependencies between the operations.

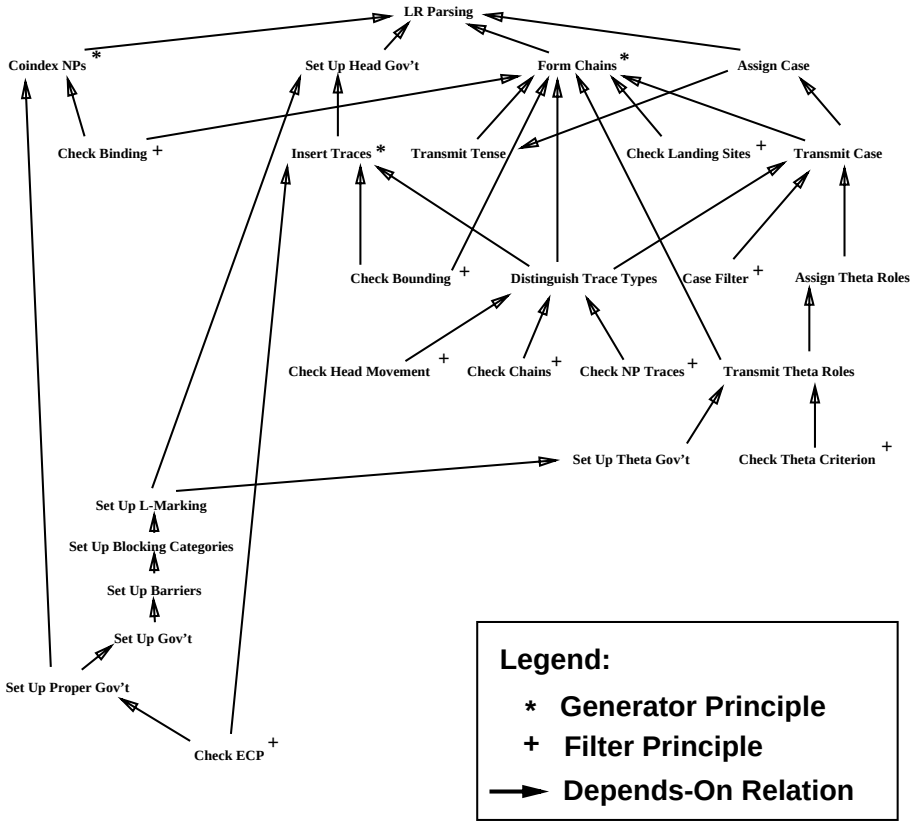

Figure 1. Principle Dependencies

12 Features Used by the Parser

A sample parse is shown in Appendix C. Each parse tree node has a child node with name "features" containing any features the node possesses. The features currently used by the parser are the following:

- NODE-ID : Denotes the unique ID of this node in the tree.
- GOVERNORS : A list of IDs of nodes that govern this node.
- GOVERNS : A list of IDs of nodes that are governed by this node.

- PROPERLY-GOVERNED? : Has the value 'T' if this node is properly governed.
- THETA-ROLE : The theta role that this node bears (e.g. agent, patient, etc.).
- DISCHARGED-ARGS : The theta roles that this node assigns and which have already been assigned (discharged).
- INT-ARG : A list of internal theta roles that this node assigns.
- EXT-ARG : A list of external theta roles that this node assigns (currently, it is only possible for a theta role assigner to assign one external theta role).
- CASE : The case that this node bears (e.g. nominative, accusative, etc.).
- WORD : The lexical word that this node represents (for terminal nodes).
- A-OR-A-BAR : Has the value 'A' if this node is in an argument position. Has the value 'A-BAR' if this node is not in an argument position (i.e. it is in an "A-bar" position).
- PRONOMINAL : Has the value 'T' if this NP or N node is pronominal.
- ANAPHORIC : Has the value 'T' if this NP or N node is anaphoric.
- COINDEXED-WITH : A list of IDs of nodes that are coindexed with this node.
- IN-CHAIN-WITH : A list of IDs of nodes that are in a movement chain along with this node.
- INTERMEDIATE-TRACE : If this feature's value is 'T', this node represents an intermediate trace in a movement chain.
- PASSIVE? : If this feature's value is 'T', the current node represents a passive verb
- TENSE : Tells the tense of the verb represented by the current node.
- HEAD-GOVERNORS : A list of IDs of nodes that head-govern the current node.
- HEAD-GOVERNS : A list of IDs of nodes that are head-governed by the current node.
- BARRIER? : If this feature's value is 'T', this node constitutes a barrier.
- PARENT-IS-BARRIER? : If this feature's value is 'T', the parent of this node is a government barrier for this node and all its descendants.

- THETA-GOVERNED-BY : A list of IDs of nodes that theta-govern this node.
- L-MARKED? : If this feature's value is 'T', this node is L-marked.
- BLOCKING-CATEGORY? : If this feature's value is 'T', this node is a blocking category.
- TRANSMITTED-FEATURES : Tells what features of this node have been transmitted (the possible values of this feature are "case" and "theta")

13 ϵ-grammar Handling

Tomita's algorithm, as described in [10], isn't able to handle arbitrary grammars containing ϵ-productions. In order to handle the presence of such rules in the s-structure grammar, we have implemented the fix used by Fong (described in [4]). The LR parser is augmented with an "environment" stack which holds information about the parse in progress. This new stack is used in two ways:

1. The environment stack is used to count how many CP nodes have been hypothesized since a word was last shifted from the input string. This allows us to avoid hypothesizing empty CPs, which can cause us to loop infinitely.
2. The environment stack is also used to remember shifting a constituent that can move (such as a verb or an adverb), thus licensing the subsequent occurrence of a trace of that constituent (such as a verb trace or an adverb trace). To handle languages where a constituent can move to the right, thus leaving us to encounter the trace before the antecedent in the input stream, the environment stack mechanism is complemented by a scheme where the licensing is achieved by looking forward in the input stream.

In the use of the environment stack as described above, our implementation is tailored after Fong's. For more details, refer to Fong's thesis ([4]).

14 Parser Inadequacies

There are several sentences in our test suite that the parser still cannot handle. The following is a description of each remaining problem, and its proposed solution:

- The Minimality Condition on government (see the definition of Government above) has not yet been implemented. Because of this, ungrammatical sentences such as "Who do you think that came" are accepted by the parser. For an analysis of this sentence and a discussion of the "*that*-trace effect", see Haegeman [5], pp. 456-7.

- The sentence "Poirot said that he is ill" produces three parses, including an incorrect parse. In the incorrect parse, [spec, CP] is empty and coindexed with "he." To fix this problem and others like it, we need a general way of dealing with sentences containing a chain where an element moves from a position where it could not have been base generated.
- Simple Arabic sentences with VSO word order are currently not handled by the system. We first need to find a clear and plausible GB account for such word orders.

15 Planned Improvements for the Parser

Although the parser is minimally adequate for the intended purpose, planned enhancements include the following:

- The current lexicon is a simple list of words and features associated with each word (including syntactic category, subcategorization frames, etc.) The system currently lacks a morphological analyzer. The likely solution to this is to integrate PC-Kimmo, a morphological analyzer written in C, into the system.
- Currently, the only level of representation recovered by our parser is s-structure. Other levels could be recovered if needed.
- Subject to preserving the parser's modularity and maintainability, the parsing needs to be speeded up.
- While Arabic is handled by the current system in a cursory manner, a better treatment of Arabic needs to be sought before the system can be said to parse Arabic correctly. In particular, the problem of flexible word order in Arabic is a difficult one to solve in the current system, as we have not been able to find a clear and convincing GB analysis of this phenomena.

A A Sample S-structure Grammar

This appendix shows our current s-structure grammar for English.

```
(setf *english-ss-grammar*
      (generator::make-grammar-struct
       '(
         ;; first, x-bar rules

         (cp          (c-bar-spec c-bar))
         (c-bar-spec  (possibly-empty-np))
         (c-bar-spec  (adv))
         (c-bar-spec  ())
         (c-bar       (c c-comp))
         (c-comp      (ip))

         (ip          (i-bar-spec i-bar))
         (i-bar-spec  (possibly-empty-np))
         (i-bar-spec  ())
         (i-bar       (i i-comp))
         (i-comp      (vp))

         (vp          (v-bar-spec v-bar))
         (v-bar-spec  ())
         (v-bar       (v v-comp))
         (v-comp      (possibly-empty-np))
         (v-comp      (cp))   ; e.g. "what do you think"
         (v-comp      (ap))   ; e.g. "poirot is ill"
         (v-comp      ())

         (pp          (p-bar-spec p-bar))
         (p-bar-spec  ())
         (p-bar       (p p-comp))
         (p-comp      (np))
         (p-comp      ())

         (ap          (a-bar-spec a-bar))
         (a-bar-spec  ())
         (a-bar       (a a-comp))
         (a-comp      ())

         (np  (n-bar-spec n-bar))
         (n-bar-spec  (det))
         (n-bar-spec  ())
         (n-bar       (n n-comp))
         (n-comp      (cp))
         (n-comp      ())

         ;; empty categories

         (possibly-empty-np (np))
         (possibly-empty-np ())
         (c           ())
         (c           (complementizer))
```

```
(c           (aux))      ; head-to-head movement from 'I'

;; head movement

(v           ())         ; v-to-i raising
(v           (verb))     ; normal case (without raising)
(i           ())         ; normal case (without raising)
(i           (verb))     ; v-to-i raising
(i           (aux))      ; e.g. "poirot will invite me"
(adv         ())         ; for adverb movement
(adv         (adverb))

)))
```

B A Sample Lexicon

This appendix contains part of the English lexicon used by the parser.

```
(setf *english-lexicon*
      '(
        ;; verbs
        (take (verb (parser::ext-arg (parser::agent))
                    (parser::int-arg (parser::goal))))
        (took (verb (parser::ext-arg (parser::agent))
                    (parser::int-arg (parser::goal))
                    (parser::tense parser::past)))

        ;; drink is both a verb and noun
        (drink (verb (parser::ext-arg (parser::agent))
                     (parser::int-arg (parser::goal)))
               (n))
        (believed
                  ;; passive form which possesses no tense
                  (verb (parser::int-arg (parser::goal))
                        (parser::passive? t))
                  ;; active form
                  (verb (parser::ext-arg (parser::agent))
                        (parser::int-arg (parser::goal))
                        (parser::tense parser::past))
                  )
        (believe (verb (parser::ext-arg (parser::agent))
                       (parser::int-arg (parser::goal))))

        (accuse (verb (parser::ext-arg (parser::agent))
                      (parser::int-arg (parser::goal))))
        (accused (verb (parser::ext-arg (parser::agent))
                       (parser::int-arg (parser::goal))
                       (parser::tense parser::past)))

        ;; nouns
        (i (n (parser::case parser::nominative)
              (parser::pronominal t)))
        (myself (n (parser::case parser::objective)
                   (parser::anaphoric t)))
        (him (n (parser::case parser::objective)
                (parser::pronominal t)))
        (himself (n (parser::case parser::objective)
                    (parser::anaphoric t)))
        (he (n (parser::case parser::nominative)
               (parser::pronominal t)))
        (abdul (n))
        (poirot (n))
        (miriam (n))
        (mary (n))
        (john (n))
        (terminal (n))

        ;; prepositions
```

```
(on (p (parser::int-arg (parser::goal))))
(in (p (parser::int-arg (parser::goal))))
(at (p (parser::int-arg (parser::goal))))

;; adjectives
(sleepy (a (parser::int-arg (parser::goal))))
(insecure (a (parser::int-arg (parser::goal))))
(clever (a (parser::int-arg (parser::goal))))
(slow (a (parser::int-arg (parser::goal))))

;; determinants
(a (det))
(the (det))
(an (det))

;; adverbs
(carelessly (adv))
(hungrily (adv))
(endlessly (adv))

;; complementizers
(that (complementizer))

;; auxiliaries
(will (aux (parser::tense parser::future)))
(did (aux (parser::tense parser::past)))
(do (aux (parser::tense parser::present)))
(was (aux (parser::tense parser::past)))
))
```

C A Sample Parse

This appendix contains a sample parse of the sentence "Whom will John invite?" The initial X-bar trees (output by the LR parsing module) are shown, as well as the final parse tree, with all its features. The final tree shows the two movement chains, "will" moving from I to C, and "Who" moving from object position of the verb "invite" to [Spec, CP].

```
> (parse '(whom will john invite) '(x-bar final-trees))
After LR parse, 2 tree(s)
Parse tree(s):

((CP
  (C-BAR-SPEC
   (NP (N-BAR-SPEC)
       (N-BAR (N (FEATURES (CASE OBJECTIVE) (PRONOMINAL T) (WORD WHOM))) (N-COMP))))
  (C-BAR
   (C (AUX (FEATURES (TENSE FUTURE) (WORD WILL))))
   (C-COMP
    (IP (I-BAR-SPEC (NP (N-BAR-SPEC) (N-BAR (N (FEATURES (WORD JOHN))) (N-COMP))))
        (I-BAR
         (I)
         (I-COMP
          (VP (V-BAR-SPEC)
              (V-BAR (V (FEATURES (INT-ARG (GOAL)) (EXT-ARG (AGENT)) (WORD INVITE)))
                     (V-COMP (NP)))))))))
 (CP
  (C-BAR-SPEC
   (NP (N-BAR-SPEC)
       (N-BAR (N (FEATURES (CASE OBJECTIVE) (PRONOMINAL T) (WORD WHOM))) (N-COMP))))
  (C-BAR
   (C (AUX (FEATURES (TENSE FUTURE) (WORD WILL))))
   (C-COMP
    (IP (I-BAR-SPEC (NP (N-BAR-SPEC) (N-BAR (N (FEATURES (WORD JOHN))) (N-COMP))))
        (I-BAR
         (I)
         (I-COMP
          (VP (V-BAR-SPEC)
              (V-BAR (V (FEATURES (INT-ARG (GOAL)) (EXT-ARG (AGENT)) (WORD INVITE)))
                     (V-COMP)))))))))

After head-government, 2 tree(s)
After forming chains, 4 tree(s)
After checking landing sites, 1 tree(s)
After trace insertion, 1 tree(s)
After checking bounding, 1 tree(s)
After transmitting tense features, 1 tree(s)
After case assignment and transmission, 1 tree(s)
After case filter, 1 tree(s)
After theta role assignment and transmission, 1 tree(s)
After theta criterion, 1 tree(s)
After L-marking, 1 tree(s)
After setting up blocking categories, 1 tree(s)
```

```
After setting up barriers, 1 tree(s)
After government, 1 tree(s)
After determining trace types, 1 tree(s)
After checking head movement, 1 tree(s)
After checking tree chains, 1 tree(s)
After checking NP traces, 1 tree(s)
After coindexing NPs, 2 tree(s)
After proper government, 2 tree(s)
After ECP, 2 tree(s)
After checking binding conditions, 1 tree(s)


Final parse tree(s):

((CP
  (FEATURES (NODE-ID 2103) (BLOCKING-CATEGORY? T) (BARRIER? T)
            (GOVERNORS (2100 2081))
            (GOVERNS (2097 2098 2091 2099 2089 2090 2100 2101 2083 2084 2102 2081 2082)))
  (C-BAR-SPEC
   (FEATURES (NODE-ID 2082) (GOVERNORS (2100 2081 2103)))
   (NP
    (FEATURES (NODE-ID 2081) (HEAD-GOVERNORS (2078)) (IN-CHAIN-WITH (2094))
              (CASE OBJECTIVE) (A-OR-A-BAR A-BAR) (THETA-ROLE GOAL)
              (TRANSMITTED-FEATURES (THETA-ROLE CASE)) (BLOCKING-CATEGORY? T)
              (BARRIER? T) (GOVERNORS (2100 2103 2078))
              (GOVERNS (2097 2098 2091 2099 2089 2090 2100 2101 2083 2084 2102 2082 2103 2079 2078 2080 2077))
              (PRONOMINAL T) (COINDEXED-WITH (2094)) (PROPERLY-GOVERNED? T))
    (N-BAR-SPEC
     (FEATURES (NODE-ID 2077) (HEAD-GOVERNORS (2078)) (GOVERNORS (2078 2081))
               (PROPERLY-GOVERNED? T)))
    (N-BAR
     (FEATURES (NODE-ID 2080) (HEAD-GOVERNORS (2078)) (GOVERNORS (2078 2081))
               (PROPERLY-GOVERNED? T))
     (N
      (FEATURES (WORD WHOM) (PRONOMINAL T) (CASE OBJECTIVE) (NODE-ID 2078)
                (HEAD-GOVERNORS NIL) (HEAD-GOVERNS (2079 2080 2077 2081))
                (GOVERNORS (2081)) (GOVERNS (2079 2080 2077 2081))))
     (N-COMP
      (FEATURES (NODE-ID 2079) (HEAD-GOVERNORS (2078)) (GOVERNORS (2078 2081))
                (PROPERLY-GOVERNED? T))))))
  (C-BAR
   (FEATURES (NODE-ID 2102) (GOVERNORS (2100 2081 2103)))
   (C
    (FEATURES (NODE-ID 2084) (GOVERNORS (2100 2081 2103)))
    (AUX
     (FEATURES (WORD WILL) (TENSE FUTURE) (NODE-ID 2083) (IN-CHAIN-WITH (2091))
               (GOVERNORS (2100 2081 2103)))))
   (C-COMP
    (FEATURES (NODE-ID 2101) (GOVERNORS (2100 2081 2103)))
    (IP
     (FEATURES (NODE-ID 2100) (HEAD-GOVERNORS (2091)) (BLOCKING-CATEGORY? T)
               (PARENT-IS-BARRIER? T) (GOVERNORS (2081 2103 2097 2091 2089))
               (GOVERNS (2101 2083 2084 2102 2081 2082 2103 2097 2098 2091 2099 2089 2090)))
     (I-BAR-SPEC
      (FEATURES (NODE-ID 2090) (HEAD-GOVERNORS (2091))
```

```
          (GOVERNORS (2081 2103 2097 2091 2089 2100)))
 (NP
  (FEATURES (NODE-ID 2089) (HEAD-GOVERNORS (2091 2086)) (CASE NOMINATIVE)
            (THETA-ROLE AGENT) (A-OR-A-BAR A) (BLOCKING-CATEGORY? T)
            (BARRIER? T) (PARENT-IS-BARRIER? T)
            (GOVERNS (2097 2098 2091 2099 2090 2100 2087 2086 2088 2085))
            (GOVERNORS (2081 2103 2097 2091 2100 2086))
            (PROPERLY-GOVERNED? T))
  (N-BAR-SPEC
   (FEATURES (NODE-ID 2085) (HEAD-GOVERNORS (2086)) (GOVERNORS (2086 2089))
             (PROPERLY-GOVERNED? T)))
  (N-BAR
   (FEATURES (NODE-ID 2088) (HEAD-GOVERNORS (2086))
             (GOVERNORS (2086 2089)) (PROPERLY-GOVERNED? T))
   (N
    (FEATURES (WORD JOHN) (NODE-ID 2086) (HEAD-GOVERNORS NIL)
              (HEAD-GOVERNS (2087 2088 2085 2089)) (GOVERNORS (2089))
              (GOVERNS (2087 2088 2085 2089))))
   (N-COMP
    (FEATURES (NODE-ID 2087) (HEAD-GOVERNORS (2086))
              (GOVERNORS (2086 2089)) (PROPERLY-GOVERNED? T))))))
(I-BAR
 (FEATURES (NODE-ID 2099) (HEAD-GOVERNORS (2091))
           (GOVERNORS (2081 2103 2097 2091 2089 2100)))
 (I
  (FEATURES (NODE-ID 2091) (HEAD-GOVERNORS NIL)
            (HEAD-GOVERNS (2097 2098 2099 2089 2090 2100))
            (IN-CHAIN-WITH (2083)) (TENSE FUTURE)
            (GOVERNS (2097 2098 2099 2089 2090 2100))
            (GOVERNORS (2081 2103 2097 2089 2100))))
 (I-COMP
  (FEATURES (NODE-ID 2098) (HEAD-GOVERNORS (2091))
            (GOVERNORS (2081 2103 2097 2091 2089 2100)))
  (VP
   (FEATURES (NODE-ID 2097) (HEAD-GOVERNORS (2091 2093))
             (BLOCKING-CATEGORY? T) (BARRIER? T) (PARENT-IS-BARRIER? T)
             (GOVERNS (2098 2091 2099 2089 2090 2100 2130 2094 2095 2093 2096 2092))
             (GOVERNORS (2081 2103 2091 2089 2100 2094 2093))
             (PROPERLY-GOVERNED? T))
   (V-BAR-SPEC
    (FEATURES (NODE-ID 2092) (HEAD-GOVERNORS (2093))
              (GOVERNORS (2094 2093 2097)) (PROPERLY-GOVERNED? T)))
   (V-BAR
    (FEATURES (NODE-ID 2096) (HEAD-GOVERNORS (2093))
              (GOVERNORS (2094 2093 2097)) (PROPERLY-GOVERNED? T))
    (V
     (FEATURES (WORD INVITE) (EXT-ARG (AGENT)) (INT-ARG (GOAL))
               (NODE-ID 2093) (HEAD-GOVERNORS NIL)
               (HEAD-GOVERNS (2094 2095 2096 2092 2097))
               (DISCHARGED-ARGS (AGENT GOAL)) (GOVERNORS (2094 2097))
               (GOVERNS (2130 2094 2095 2096 2092 2097))))
    (V-COMP
     (FEATURES (NODE-ID 2095) (HEAD-GOVERNORS (2093))
               (GOVERNORS (2094 2093 2097)) (PROPERLY-GOVERNED? T))
     (NP
```

```
(FEATURES (NODE-ID 2094) (HEAD-GOVERNORS #1=(2093))
          (IN-CHAIN-WITH (2081)) (CASE OBJECTIVE)
          (TRANSMITTED-FEATURES (CASE)) (THETA-ROLE GOAL)
          (THETA-GOVERNED-BY (2093)) (A-OR-A-BAR A) (L-MARKED? T)
          (GOVERNORS (2093 2097)) (GOVERNS (2095 2093 2096 2092 2097 2130))
          (COINDEXED-WITH (2081)) (PROPERLY-GOVERNED? T))
(NP-TRACE
 (FEATURES (NODE-ID 2130) (HEAD-GOVERNORS #1#)
           (GOVERNORS (2093 2097 2094)) (PROPERLY-GOVERNED? T)))))))))))))))
```

D English Corpus

This appendix shows our current corpus used to test the English language coverage of our parser.

```
(setf *english-sentences*
  '(

;;; (1) WORKS
    (i saw abdul)

    ;; trace module (ECP)
      ; who[i] do you think t'[i] t[i] came ?  (Haegeman p. 456)
;;; (2) WORKS
    (who do you think came)
      ; * who[i] do you think t'[i] that t[i] came ?  (Haegeman p.
      ; 456)
;;; (3) DOES NOT WORK (produces a parse) -- eventually need to
;;;     implement government's "minimality condition" (see Haegeman p.
;;;     404)
    (who do you think that came)    ; *

    ;; binding module
      ; Poirot[i] hurt himself[i,*j] (Haegeman p. 193)
;;; (4) WORKS
    (poirot hurt himself)
      ; Poirot[i] hurt him[*i,j] (Haegeman p. 193)
;;; (5) WORKS
    (poirot hurt him)
      ; Poirot[i] said that he[i,j] is ill (Haegeman p. 193)
;;; (6) DOES NOT WORK -- produces three parses, including an incorrect
;;;     parse (see problems file for details)
    (poirot said that he is ill)

    ;; bounding module
      ; Whom[i] did he see t[i]? (Haegeman p. 365)
;;; (7) WORKS
    (whom did he see)
      ; Whom[i] did Poirot claim that he saw t[i]? (Haegeman p. 365)
;;; (8) WORKS
    (whom did poirot claim that he saw)
      ; * Whom[i] did Poirot make the claim that he saw t[i]?
      ; (Haegeman p. 365)
;;; (9) WORKS
    (whom did poirot make the claim that he saw)   ; *

    ;; theta module
;;; (10) WORKS
    (i saw a bird)
;;; (11) WORKS
    (i told a bird)          ; *

    ;; case module
;;; (12) WORKS
```

```
    (i saw myself)
;;; (13) WORKS
    (i saw i)                ; *, ruled out by case module
;;; (14) WORKS
    (i accuse abdul)
;;; (15) WORKS
    (miriam accuse abdul)    ; *, ruled out by case module
;;; (16) WORKS
    (miriam accused abdul)
;;; (17) WORKS
    (who will leslie invite) ; *, 'who' has the wrong case

    ;; NP movement
      ; [This story][i] was believed e[i] (Haegeman p. 282)
      ; [IP [NP this story][i] [I' was [VP believed t[i]]]]
;;; (18) WORKS
    (this story was believed)

    ;; Wh-movement
      ; Whom will Leslie invite? (Haegeman p. 339)
;;; (20) WORKS
    (whom will leslie invite)
;;; (21) WORKS
    (what did mary want)
;;; (22) WORKS
    (whom did mary think that john saw)
    )
```

References

[1] Abney, Steven, Ed.: *The MIT Parsing Volume, 1987-1988*; Cambridge, MIT Center for Cognitive Science, 1988.

[2] Dorr, Bonnie: *UNITRAN: A Principle-Based Approach to Machine Translation*; Cambridge, MIT AI Technical Report No. 1000, 1987.

[3] Dorr, Bonnie: *Lexical Conceptual Structure and Machine Translation*; Cambridge, Mass., Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, 1990.

[4] Fong, Sandiway. *Computational Properties of Principle-Based Grammatical Theories*. Cambridge, Mass., Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, 1991.

[5] Haegeman, Liliane: *Introduction to Government & Binding Theory*. Oxford, Basil Blackwell, 1991.

[6] Kashket, Michael. *Parsing Warlpiri, a Free-Word Order Language* in The MIT Parsing Volume 1987-1988. Cambridge, Mass., MIT Center for Cognitive Science, 1988.

[7] Sells, Peter: *Lectures on Contemporary Syntactic Theories*; Stanford, Center for the Study of Language and Information, 1985.

[8] Tenny, Carol, Ed.: *The MIT Parsing Volume, 1988-1989*; Cambridge, MIT Center for Cognitive Science, January 1990.

[9] Tenny, Carol, Ed.: *The MIT Parsing Volume, 1989-1990*; Cambridge, MIT Center for Cognitive Science, July 1990.

[10] Tomita, Masaru: *Efficient Parsing for Natural Language*; Boston, Kluwer Academic Publishers, 1986.

[11] Wehrli, Eric: *A Government-Binding Parser for French*; Geneva, University of Geneva, 1984.