Structure Sharing and Parallelization in a GB Parser

Marwan Shaban
shaban@cs.bu.edu

March 22, 1994

BU-CS Tech Report # 94-005

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

Abstract

By utilizing structure sharing among its parse trees, a GB parser can increase its efficiency dramatically. Using a GB parser which has as its phrase structure recovery component an implementation of Tomita's algorithm (as described in [Tom86]), we investigate how a GB parser can preserve the structure sharing output by Tomita's algorithm. In this report, we discuss the implications of using Tomita's algorithm in GB parsing, and we give some details of the structure-sharing parser currently under construction. We also discuss a method of parallelizing a GB parser, and relate it to the existing literature on parallel GB parsing. Our approach to preserving sharing within a shared-packed forest is applicable not only to GB parsing, but anytime we want to preserve structure sharing in a parse forest in the presence of features.

Contents

1 Introduction

This paper discusses a GB parser (currently under construction) with two unique attributes:

- It uses structure sharing between its parse trees to increase the performance of GB parsing by a factor up to the product of branching factors of the parser's generator modules.
- Beyond the effective parallelism achieved by the above structure sharing, the parsing process is further parallelized by using a facility we call "Forest Deltas" to allow multiple modules to simultaneously operate on the same parse tree (forest), and at the same time, allow each module to operate on all nodes of a parse tree (forest) simultaneously.

A typical GB parser could start to analyze a sentence by using a context-free grammar to recover the possible skeletal (X-bar) structures of the sentence's s-structure representation. This phrase structure recovery process typically generates in addition to the correct phrase structure many incorrect phrase structures which will be weeded out by the "filter" principles applied to them later in the parse process. At any point in the parse process, there are likely to be more than one hypothesized parse tree for the input sentence, and much of these parse trees will look similar. By representing these parse trees in a compact form (using structure sharing), a lot of duplicated effort will be avoided. For example, if a compact parse forest allows two (virtual) parse trees to share the same noun phrase, then when passing this forest through the case filter, the case filter will have to process the noun phrase once instead of twice, as would be the case if we passed the two parse trees through the case filter separately.

In this paper , we will briefly review the shared-packed parse forest representation output by Tomita's algorithm, go over some issues related to using Tomita's algorithm in a GB parser, then give some details regarding our strategy for preserving structure sharing in the parse forest as it makes its way through the modules of the GB parser.

To preserve structure sharing in a GB parser, we use several ideas:

- Relative addressing is employed to eliminate having to split a node when it participates in more than one (virtual) parse tree and points to a separate node from each parse tree.

- Node packing is used to eliminate having to split trees due to a "localized" generator operation.

- We propose a unique scheme for representing chains and coindexations within a forest which eliminates having to split the forest into a separate tree for each virtual tree output from the chain formation and NP coindexation generators.

In addition, we encode each principle such that it does not operate directly on the parse forest, but rather outputs a description of the changes ("deltas") it wants applied to the forest. This allows us to code the system more cleanly and opens some interesting avenues for parallelization in the system.

Previous work in the area of structure sharing in natural language parsers include that of Karttunen and Kay ([KK85]) and Pereira ([Per85]). Karttunen's work, in particular, provides us with two key ideas utilized in the current proposal, relative addressing and lazy copying.

Section 2 briefly explains the representation of compact forests as output by Tomita's algorithm. Section 3 discusses some benefits of using Tomita's algorithm, aside from the obvious benefit of its giving us compact parse forests. Section 4 talks about the changes we needed to make to Tomita's algorithm in order for it to parse using ϵ-grammars. Section 5 gives the basic principle (which we call the "Non-Discrimination Principle") upon which much of the following discussion is based. Section 6 shows why we need to use relative addressing, and explains how we extend Karttunen's relative addressing notion to handle packed nodes. Section 7 explains how we handle multiple concurrent updates to a parse forest through a facility we call "forest deltas". Section 8 notes some properties of our compact forest representation and how a GB module's control flow can be modified to accommodate compact forests. Section 9 explains the branch splitting problem and how we solve it. Section 11 explains how structure sharing allows us to easily deal with a class of generators we label "localized." Section 12 gives our proposed solution to the problem of representing long-distance relationships such as movement chains and coindexed NPs while preserving structure sharing in a compact forest.

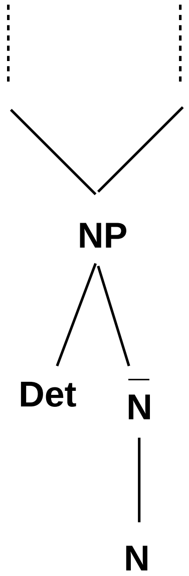


Figure 1: The NP node (and its subtree) is shared. This forest represents two trees.

2 Structure Sharing in Tomita's Algorithm

Tomita's algorithm ([Tom86]), which is used as the phrase structure recovery component of our GB parser, outputs a compact parse forest containing both node sharing and node packing. Node sharing is the sharing of a subtree between two or more parent nodes. Figure 1 shows an example of node sharing. Node packing is a compact representation of local ambiguity, where two or more nodes of the same category span the same substring of the input. Figure 2 shows an example of node packing. A packed node can also be shared. Figure 3 is an example of such a case.

One can see how this method of compactly representing a forest can lead to savings in terms of the space used to store the forest, as well as savings in the time spent processing the tree by each module of the GB parser.

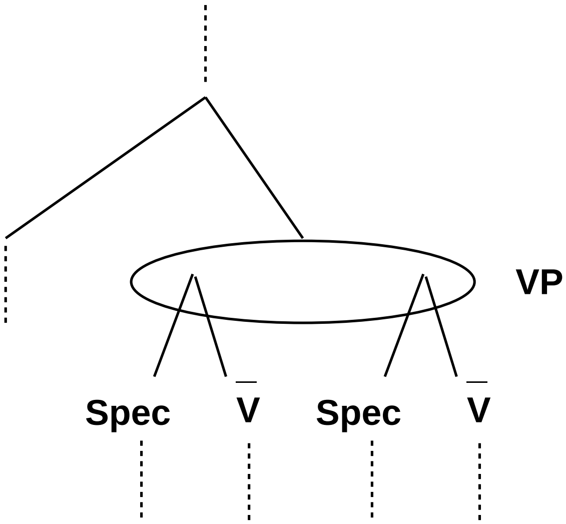


Figure 2: The VP node is packed. Everything above it is shared by the two subnodes of the packed VP node. This forest represents two trees.

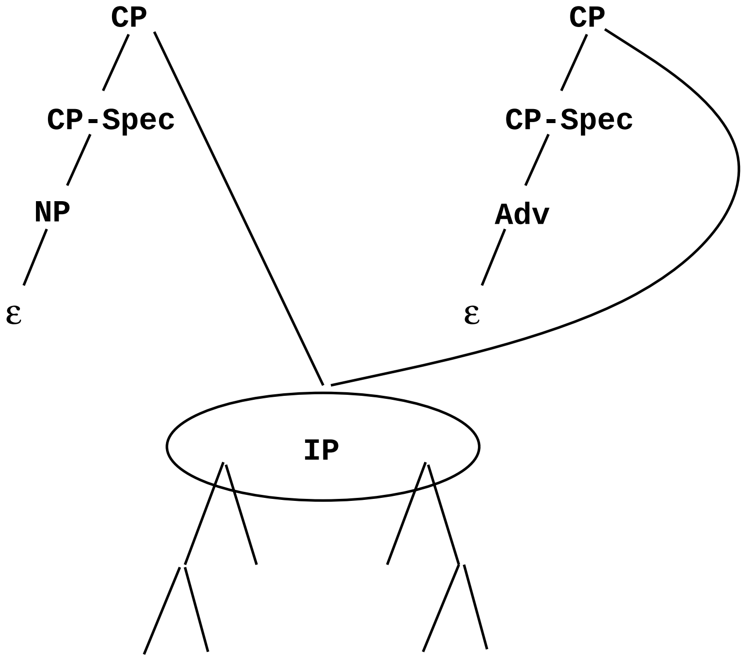


Figure 3: Sharing a packed node. This forest represents four parse trees.

3 Benefits of Using Tomita's Algorithm in a GB Parser

3.1 GB Parsing With and Without Tomita's Algorithm

As mentioned earlier, a typical GB parser could utilize a "core" context-free parser which recovers the core phrase structure of the input sentence (with respect to an s-structure context-free grammar derived from X-bar theory), and produces a set of under-specified parse trees. We say the parse trees are under-specified because the nodes of the parse trees which the context-free parser produces still need many features to be specified, such as case features, theta features, anaphoric/pronominal features, coindexation features, and so on. The point is that any context-free parser which can handle the context-free covering grammar for s-structure can be used to recover the phrase structure of an input sentence.

In contrast to the above scheme of using a generic context-free core parser, using Tomita's algorithm presents GB parsing with unique benefits which arise from the graph-structured stack it employs as well as its compact forest representation.

3.2 Comparing Tomita's and Fong's Tree-Structured Stacks

Fong's GB parser ([Fon91]) uses a PROLOG recursive descent parser which is similar to Tomita's parser in that it uses the same tables and traverses the same stack configuration search space to construct the same set of parse trees. However, the depth-first recursive descent control flow precludes the construction of shared-packed parse forests since the construction of these forests depends on the parallel exploration of the search space in a breadth-first manner. In effect, Fong's stack is a tree where each nondeterministic choice point (where the LR table entry has multiple actions) corresponds to a place in the tree-structured stack where a branch splits off. Fong's scheme achieves a limited degree of parse tree sharing by passing fragments constructed before a "fork" in the search space into each branch, so during the exploration of each of the branches, the same fragment (which was constructed before the fork was encountered) is used.

In [Tom86], Tomita presents a parser which maintains a tree-structured stack, and is similar to Fong's parser in that respect. However, the stack trees of the two parsers are rooted at different ends. Fong's stack trees are rooted at the start state, whereas Tomita's stack trees are rooted at the accepting state. It is interesting to contemplate how one would show that these two tree-structured stack mechanisms are equivalent in power, and whether some grammar/input-sentence combinations favor one mechanism over the other. However, the question seems rather moot because of the obvious superiority of the graph-structured stack mechanism over both tree-structured stack mechanisms.

3.3 The Benefits of Using a Graph-Structured Stack

Tomita [Tom86] showed that the use of a graph-structured stack is superior to using a tree-structured stack by virtue of reducing the amount of work used to traverse the search space for a parse.

In Fong's concluding remarks on principle interleaving (on page 207 of [Fon91]), he gives a rather bleak outlook for principle interleaving:

> "To make interleaving a considerably more attractive model, it seems that a large reduction in the amount of garden-pathing during phrase structure construction is necessary. The fact that the current system has employed a fairly efficient canonical LR(1)-based algorithm extended to include unbounded lookahead to further improve error detection suggests that the prospects for such an improvement are not good."

However, Fong was using a mechanism equivalent to Tomita's tree-structured stack (as discussed above). If we were to use principle interleaving with a graph-structured stack mechanism, the amount of work spent traversing the search space (and hence the garden-pathing that Fong refers to) could be significantly reduced. This is one question we are investigating.

Another basic result that we could derive is simply to verify that principle interleaving coupled with a graph-structured stack is better than a graph-structured stack without principle interleaving. Fong did this kind of analysis with his recursive LR parser. He achieved a speedup of up to 30% when using certain interleaved configurations, but most interleaved configurations slowed the parser down due to the overhead associated with interleaving.

3.4 The Benefits of Using Compact Forest Representations

As discussed in section 1 above, using a shared-packed forest can speed up the operation of a GB parser's modules. This is because applying a principle to a compact parse forest would have the effect of applying the principle to multiple parse trees (or multiple parse tree fragments) simultaneously, and the more structure sharing there is in the compact forest, the more benefit we will derive from applying the principle to the compact form of the forest.

Furthermore, the compact forest representation can present GB principles with benefits aside from the obvious structure-sharing that leads to reduced time and space requirements. For example, imagine a "generator" principle that takes an under-specified constituent, say an NP, and generates an arbitrary number of (more specified) NPs. If such a generator operates on the usual list-of-trees forest representation, it will take as input a tree, and produce a list of trees, each of which is a copy of the original tree except that the NP in question has been replaced by a more specified NP resulting from the generation process. If the same generator were applied to a shared-packed forest, it would produce the same shared-packed forest except that the NP node has been replaced by a "packed" node effectively containing many NP nodes. We call such generators "localized" because they do not cause forest nodes to "point" to other (potentially distant) nodes. The two main generators in our system, chain formation and NP coindexation, are not "localized."

Our parser contains just such a localized generator. The s-structure grammar does not distinguish between empty NP types. At some point during the parse, empty NPs' types are narrowed down to two of the four possible empty NP types, "pro", "PRO", NP-trace, and *wh*-trace. Without the ability to insert a packed node in place of the empty NP, we would have to duplicate a tree that contained an empty NP. If a tree contained two empty NPs, it would generate four possibilities, and so on. This is one way in which the shared-packed forest representation gives us a natural and convenient data structure on which principles can operate.

As mentioned above, using a graph-structured stack may be what principle interleaving needs to be cost effective. However, having the principles operate on compact forest representations would speed up the operation of the interleaved principles and further offset the overhead of interleaving. Of course, since we are both using a graph-structured stack and using a shared-packed forest representation, assuming that principle interleaving gives us better results than Fong obtained, it would be difficult to tell which of these features did the trick. Indeed, it is hard to separate these two features of Tomita's algorithm, since by interleaving the principles with the LR parse, we must operate on the data structure native to the parser, which is a shared-packed forest.

4 Handling ϵ-grammars

A problem with using Tomita's algorithm in a GB parser is that it cannot handle many epsilon-grammars (grammars containing erasing productions). Since any covering grammar for s-structure is likely to have many epsilon-productions (*e.g.*, you need an erasing rule for each constituent that can move), we are presented with a problem. While it is possible to convert an epsilon-grammar into one without epsilon-productions, it is unacceptable to do this both on grounds of faithfulness and clarity. Fong's solution, described on pages 142-145 of [Fon91], depends on an off-line analysis of the s-structure grammar to deduce which nonterminals may cause a problem, and how. Fong then uses a new "structure" stack which holds housekeeping information used by specially-coded hooks in the LR parser. The code figures out when epsilon-productions may be "licensed," and when they may not (*i.e.*, when they will cause a problem).

The problem with Fong's scheme is that it depends on an off-line analysis of the s-structure grammar, which must be carried out by a human. If the s-structure grammar changes, the off-line analysis must be redone to make sure that the LR parser will still handle the epsilon-production-containing grammar correctly. A better solution would be to modify Tomita's algorithm to handle epsilon-productions. Farshi [NF91] gives a recognizer based on Tomita's algorithm which handles a larger class of epsilon-grammars than Tomita's algorithm.

We converted Farshi's algorithm into a parsing algorithm, and using it, are able to avoid most of the trouble that Fong went to.[1] However, there remains a problem. Farshi's algorithm cannot handle "cyclic" grammars (a cyclic grammar is one where a grammar symbol can derive itself after a positive number of productions are applied, *i.e.*, $\alpha \overset{+}{\Rightarrow} \alpha$.) Our s-structure grammar is cyclic since every terminal type can erase, and a VP can select a CP as an internal argument, so we can have CP $\overset{+}{\Rightarrow}$ CP. The preferable solution to this problem is to extend Farshi's algorithm to handle cyclic grammars. If that doesn't work, we will have to use a scheme such as Fong's (*i.e.*, to hard-code within the parser a limit on the level of CP nesting).

5 The Non-Discrimination Principle

To preserve structure sharing in a forest, we must avoid replicating nodes needlessly. A node is "shared" if it participates in more than one tree within the forest. A node can remain shared only as long as its children and features are identical in all trees in which the node participates. If one tree needs to change a particular feature of a shared node, and the other trees sharing the node don't need that feature changed, the node must be split, creating two copies of the node, one identical to the original node, and the other containing the changed feature.

In this paper, we use the term "discrimination" to refer to a node having different relationships with two or more trees sharing the node. If a shared node is "treated differently" (by assigning it different values for a particular feature) by two trees sharing the node, we say that the node discriminates between the two trees.

Node sharing can be preserved only as long as the shared node does not discriminate between the trees sharing it. If at any point in time a node discriminates between the trees sharing it, it must be split (replicated), and each copy assigned to some of the trees sharing the node such that each node copy no longer discriminates between the trees sharing it. Thus, the principle of non-discrimination can be stated as follows.

> **Principle of Non-Discrimination:**
>
> **Every shared node X must hold the same relationships with each tree sharing X.**

[1] See [Sha94] for the details of our extensions to Farshi's algorithm.

It is important to note that by “shared node”, we don’t mean only nodes that have multiple immediate ancestors. All nodes that descend from a node with multiple ancestors are also shared (by virtue of participating in a shared subtree). By the same token, packed nodes (that have multiple subnodes, each with its own descendants) are shared, as are the ancestors of a packed node (by virtue of participating in a shared “supertree”).

6 Relative Addressing

In a previous implementation of our GB parser, each tree node had a unique node ID. Nodes that referred to others contained (as a feature) the IDs of the nodes being referred to. For example, the feature PROPER-GOVERNORS contained the node IDs of nodes which were proper governors of the current node (which contained the feature). This way of having a node point directly at other nodes will not work in a system that seeks to preserve structure sharing in a forest.

Take for example the AP-Spec node in figure 4. This node is shared by the two AP subtrees, and is properly governed by the A nodes of each of the two AP subtrees. If this AP-Spec node is to encode the proper government feature as described above, it would have to contain two possible values for the PROPER-GOVERNORS feature, one with the node ID of the node A_1 (from the first AP phrase), and another with the node ID of the node A_2 (from the second AP phrase). But our forest representation doesn’t allow two possible values for one feature. The node would have to be split into two copies, one belonging to the first AP subtree and pointing to A_1 in its proper-governors feature, and another belonging to the second AP subtree and pointing to A_2 in its proper-governors feature. The result is the destruction of the node sharing due to the node discriminating between the two phrases previously sharing it.

Our solution to this problem is to have the node point to its proper governors through relative addresses. The relative address would amount to directions to reach the proper governor from the current node’s location. In the case of our above example, the relative address would be “Go up one node, then take the leftmost branch down (to reach the $\overline{A}$ node), then take the leftmost branch down (to reach the proper governor).” A similar relative addressing scheme was devised by Karttunen and Kay (see [KK85]), but

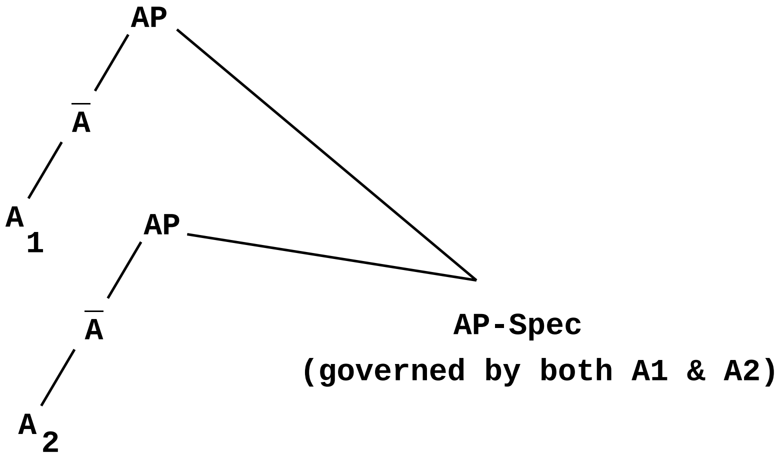


Figure 4: Relative addressing preserves structure sharing.

wasn't used for this exact purpose in their system. We adopt their relative addressing idea to preserve our structure sharing and extend it to allow for node packing.

In our current example, the result is that the shared node contains the relative address of the proper governor, and this relative address applies equally well to each of the two proper governors (one from each tree sharing the AP-Spec node), thus the principle of non-discrimination is not violated and we have preserved the sharing of the AP-Spec node.

The interpretation of a relative address in the proper-governors feature slot of a node is that *all* nodes having this structural relationship to the current node are proper governors. Operationally, we can find all proper governors of a node by following the relative address from the current node. If, while following the relative address upward, we encounter a branch (*i.e.*, there exist more than one ancestor for a node), we follow all branches. In the same way, if while following the relative address downward, we encounter a packed node, we go downward through all subnodes of the packed node.

7 Processing Forest Deltas

In our parser, GB modules do not directly modify the compact forest. Instead, they output a description of their intended changes. The output of a module is a list of “deltas”, each of which describes an individual change to a node of the forest. After a GB module has been applied to a forest, we take the delta list that it output, and apply its changes to the forest, producing a new compact forest. We use this delta mechanism for three reasons:

- To simplify the implementation of structure sharing.
- To speed up the operation of some principles (in a non-parallel parser implementation).
- To allow us to parallelize the parser.

This delta list facility has the potential of speeding up non-parallel (sequential) applications of GB modules to a forest. The reason for this is that if one change results in replication of some structures within a forest, applying another module *after* the first module's changes have been applied may be slower since the second module will have to operate on a larger structure, possibly duplicating lots of effort. Of course, we would have to be careful not to simultaneously[2] apply two modules where one module's correct operation depends on the other module first finishing its job (our GB module dependency graph is shown in figure 5).

Here is the format of a forest “delta”:

1. Pointer to the node to be changed.
2. Path between the node to be changed, and the node affecting the change. This is needed in case we need to split the node (and every node between the split node and the split's junction).
3. Delta type:

   Reject-tree: The node (or its subtree) is deemed ungrammatical, so the node and every tree in which it participates are deleted.

[2] Here, by “simultaneously,” we mean giving the two modules the same forest to look at, without applying the first module's changes to the forest before calling the second module.

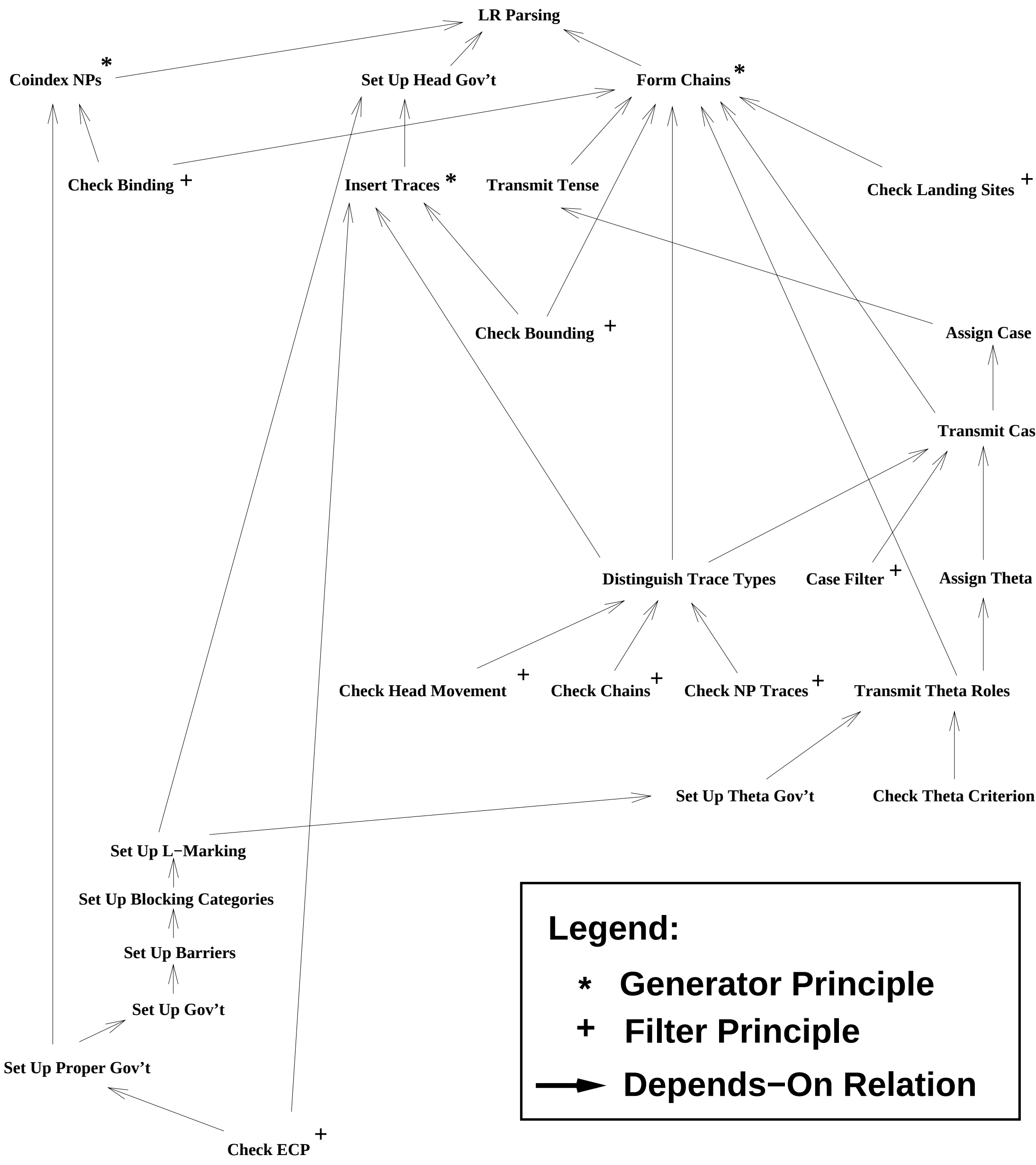


Figure 5: Module dependencies within our GB parser.

Insert-child: A new child node is added to the children of this node. Currently, this is only used to specify the type of an empty NP (*i.e.*, to add a "pro", "PRO", or trace child to an NP node which previously had no children), so we don't worry about the order of the new child within the existing child list.

Set-feature: A new feature value replaces an existing feature in the node. If the feature doesn't exist in the node, the feature is created.

Add-to-feature: A new element is added to the set of values currently occupying a feature slot. *E.g.*, adding a new node to the list of nodes governing the current node.

Since only independent GB modules will be applied to a forest before affecting their combined changes, we can apply their changes to the forest in any order. If a node is split by one delta, and another delta is to be applied to the split node, the second delta is applied to each copy of the original node. If a node is deleted by one delta, and other deltas were to be applied to the deleted node, they are discarded.

8 Re-coding GB Modules To Operate Non-Deterministically

There is a kind of duality between packed nodes looking down at their descendants and shared nodes looking up at their ancestors. In traversing a forest upward, if we encounter a branch (a shared node), each branch represents a distinct tree (where all trees are sharing the part below the shared node). In traversing a forest downward, if we encounter a branch (a packed node), each subnode of the packed node represents a distinct tree (where all trees are sharing the part above the packed node).

The GB modules in our parser must be able to handle (and preserve) the structure sharing that exists in the parse forest that they process. In the general case of a module doing some kind of traversal task and possibly changing nodes' attributes along the way, the control flow of the module will have to be altered to account for the nondeterminism encoded in the parse forest. This nondeterminism takes the form of branching that would not be seen in a normal parse tree. In the case of a module traveling upward in a tree and encountering a shared node (a node with more than one ancestor), the module must (simultaneously) follow all branches, *i.e.*, the procedure performs the next step on each of the ancestors of the shared node. In the case of a module traveling downward in a tree and encountering a packed node (with more than one subnode, each having its own descendant list), the module must (simultaneously) follow all branches, *i.e.*, the procedure performs the next step on each of the subnodes of the packed node. Operationally, we can re-code each GB module which previously operated on unshared trees as a state machine in which each state recursively calls the next state on the neighboring tree node (to traverse a tree arc). If a split is encountered where each branch represents a distinct tree, then the next state is called for each branch, and the results are merged appropriately.

9 Splitting Shared Nodes

During the processing of a compact forest, two major operations that we will have occasion to perform are joining subtrees that become identical (to maintain optimal sharing of structure in the forest) and splitting any nodes that become in violation the non-discrimination principle. The former operation will be needed infrequently, and we will ignore it for now. A much more common occurrence will be the need to split a node in order to obey the non-discrimination requirement. This section discusses the mechanism of splitting a node and gives some illustrative examples.

When a node is to be split, it is because it discriminates between two or more branches of the parse forest, which could be in the immediate vicinity of the node to be split or far away from it. If the node to be split is not the actual parting point of the branches discriminated between by the node, then every node along the path from the actual node to be split to the node at which the branches branch apart must be split. We call the node where the branches branch apart the split's "**junction**." The junction may be a packed node (if the branches point downward and share a common upper portion), or it may be a shared node (if the branches point upward and share a common lower portion).

Karttunen ([KK85]) talks about "lazy copying" which basically amounts to copying a node only when the copying is necessary to maintain the correctness of the shared tree structure. We adopt the same philosophy here, and note that the splitting of nodes between the node being split and the junction of the branches being discriminated between is necessary to maintain the correctness of the forest structure.

As noted above, when applying "deltas" to a node that is being split, each copy of the split node gets a copy of the original node's (unapplied) deltas.

The splitting of a node can be instigated by the node being split or by external nodes that need to distinguish between two instances of the split node. An example of the first case is a shared node that needs to discriminate between two external nodes by governing one and not the other (*e.g.*, in figure 4 if the $\overline{A}$ dominating A_1 is a barrier to government but the $\overline{A}$ dominating A_2 is not, the Spec node would govern A_2 but not A_1[3]). An example of the second case is a shared node N that is assigned structural case by an external node *node1* but not another external node *node2* that lies in the same position relative to N as *node1* does. Of course, a third occasion to split a node is when it lies on the path between a node being split and the split's junction (*e.g.*, in figure 6, CP-Spec is split for this reason).

We identify the following cases of splitting:

- Splitting a y-junction: occurs when we split a node shared from above by the branches discriminated between. We split every node from the node to be split to the branch's junction (*e.g.*, in figure 6 the node to be split is NP, and the branches discriminated between are the arcs from CP-Spec to CP_1 and from CP-Spec to CP_2).

- Splitting a λ-junction: occurs when we split a node shared from below by the branches discriminated between. Again, we split every node between the node to be split and the branch's junction (*e.g.*, in figure 7 the node to be split is CP-Spec and the branches discriminated between are the arcs from IP to $\overline{I}_1$ and from IP to $\overline{I}_2$). We try to preserve node packing by bubbling it upward, so the highest split node becomes a packed node (in figure 7 the CP node becomes packed).

[3] Assuming, for the example's sake, that the linguistic theory allowed an adjective to be governed by its specifier.

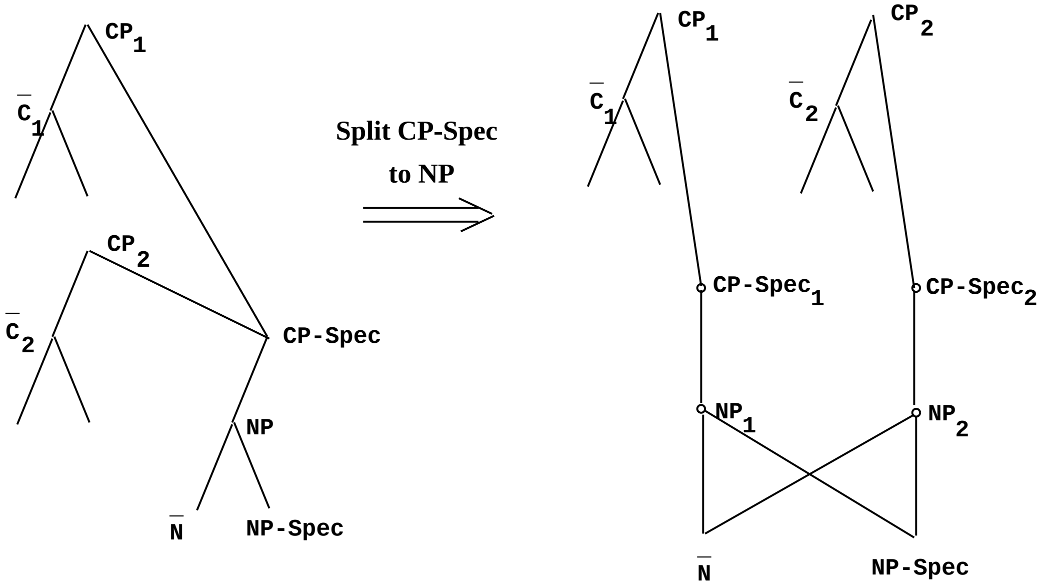


Figure 6: Splitting a y-junction. NP is discriminating between $\overline{C}_1$ and $\overline{C}_2$. NP & CP-Spec are no longer shared, but the $\overline{N}$ and NP-Spec nodes remain shared.

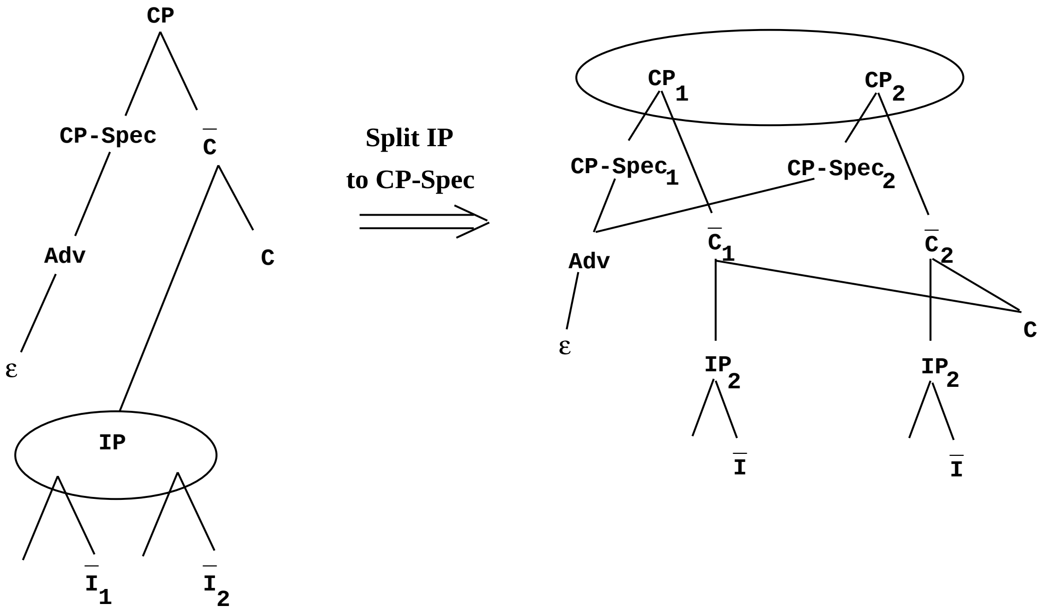


Figure 7: Splitting a λ-junction. CP-Spec is discriminating between $\overline{I}_1$ and $\overline{I}_2$. IP, $\overline{C}$, and CP-Spec are split. CP is now packed.

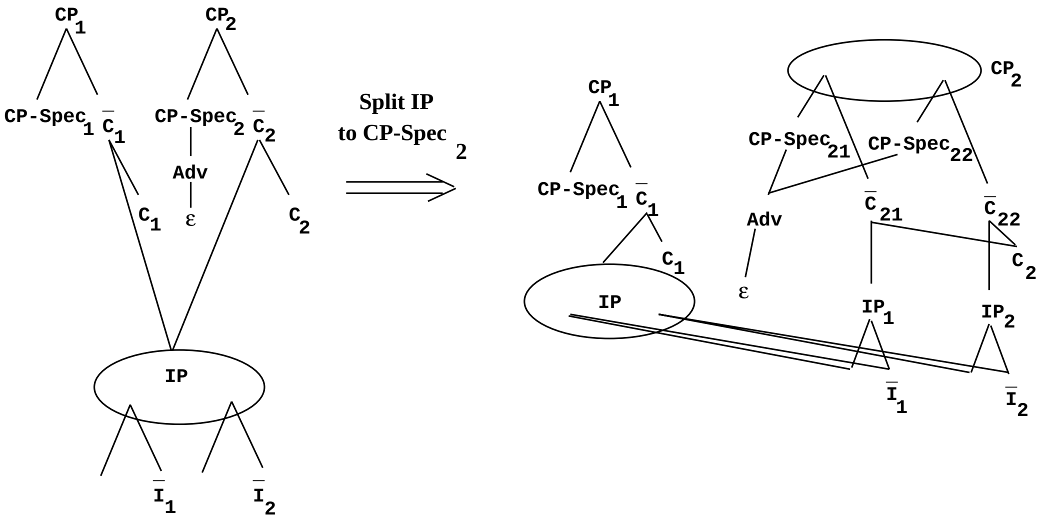


Figure 8: CP-Spec$_2$ is discriminating between $\overline{I}_1$ and $\overline{I}_2$. CP-Spec$_2$, CP$_2$, $\overline{C}_2$, and IP are split.

When encountering a shared node while splitting a λ-junction, we copy it, preserving the other arcs pointing to the shared node from above (*e.g.*, in figure 8, the IP node is both shared and packed, so we make a copy to preserve the arc pointing to IP from $\overline{C}_1$).

Observe in figure 9 how splitting a packed node ($\overline{C}$ in the figure) breaks up the packing since there were only two subnodes in the packed node. The same reasoning applies to splitting a shared node (such as the NP node in figure 10).

As a rule, when splitting nodes, we try to preserve structure sharing whenever we are permitted to do so (following the principle of lazy copying). For example, we try to preserve sharing below split shared nodes as much as possible (see figures 10 and 11 where the NP nodes' copies point to the same children). The same applies to preserving the sharing above packed nodes (see figures 8 and 9 where the packing is bubbled up to the CP nodes).

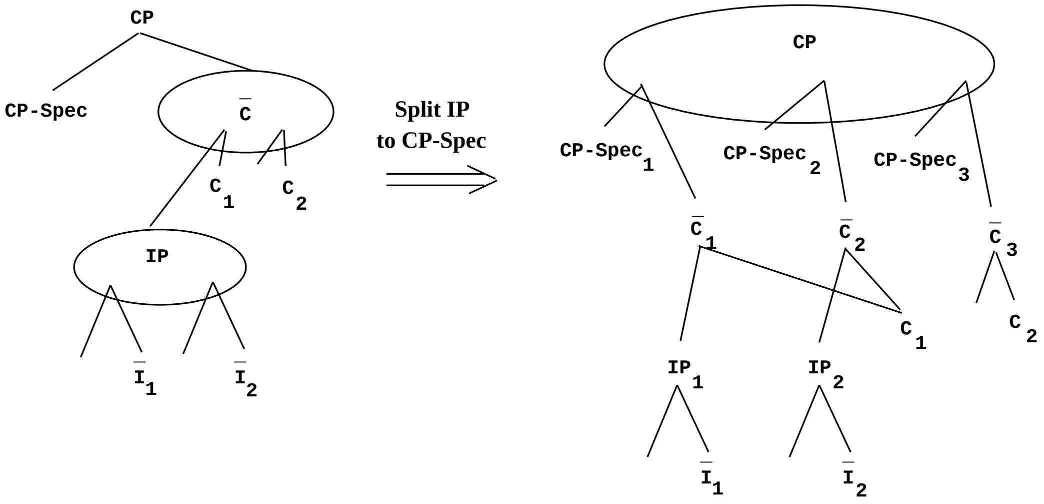


Figure 9: CP-Spec is discriminating between IP_1 and IP_2, the two subnodes of the packed IP node.

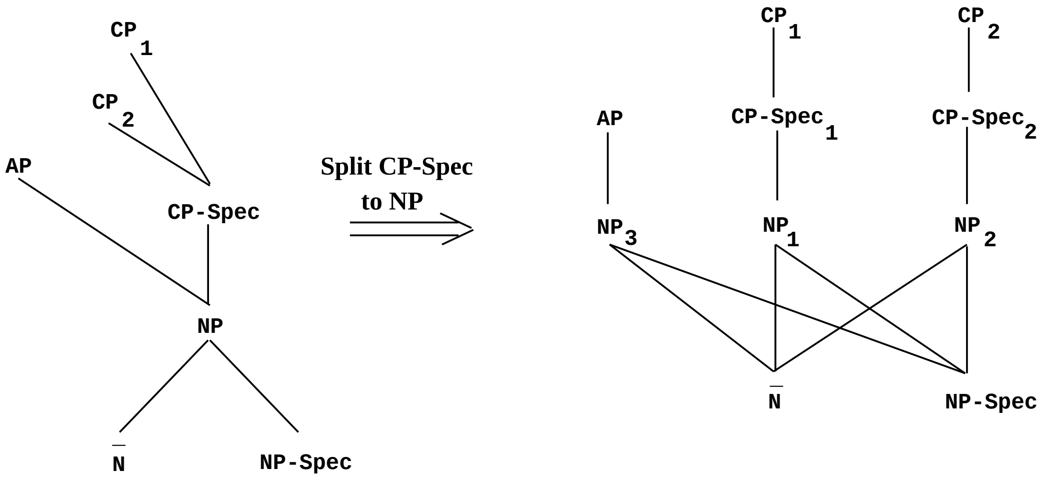


Figure 10: NP is discriminating between CP_1 and CP_2, so NP and CP-Spec are split. NP is split into three copies since it is also shared by AP.

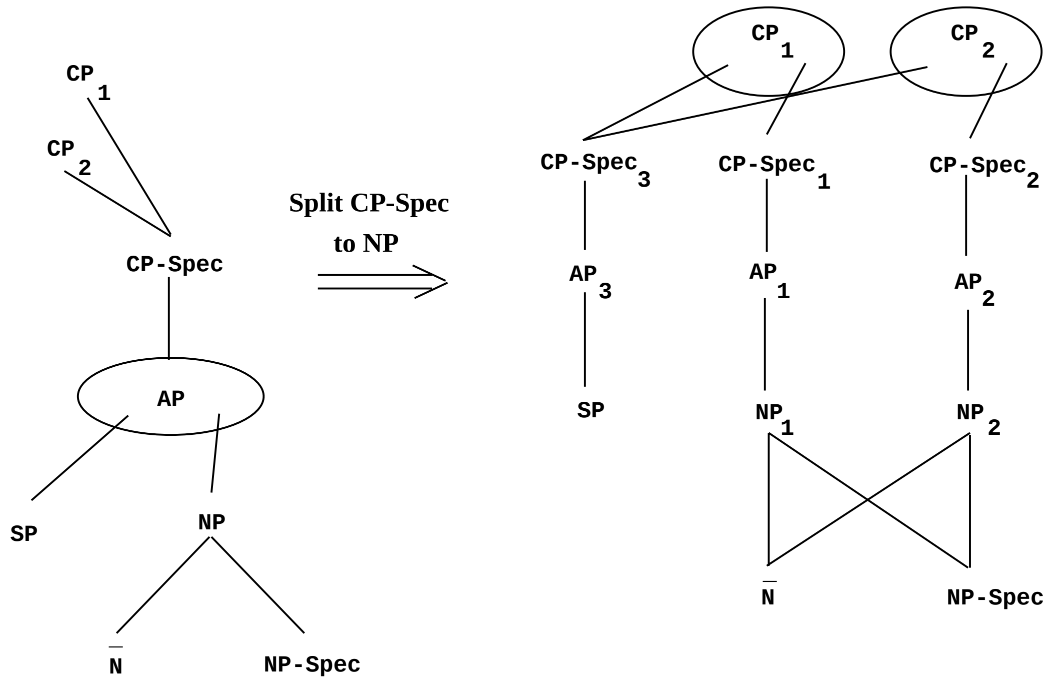


Figure 11: NP is discriminating between CP_1 and CP_2. The packed node is bubbled up.

Delta:

Node: The node for which the delta is scheduled.

Source Node: Address of the node which caused the delta to be scheduled (relative to **node**).

Action: The action to be performed on **node**.

Figure 12: Data structures used by the algorithms.

10 Algorithms

This section gives a possible formulation of the algorithms for processing deltas and performing the splitting of forest nodes. Figure 12 shows the main data structures used by the algorithms, and figure 13 shows the main functions used by the algorithms.

APPLY-DELTAS(F, D)

```
• for all $F_i \in F$ do
    – $D_i \Leftarrow$ set of deltas scheduled for $F_i$ $(D_i \subseteq D)$
• while there is a node $F_i$ where $D_i \neq \emptyset$ do
    – while there is an element $D_k$ in $D_i$ do
        * $D_j \Leftarrow \{D : D \in D_i \wedge$ Action$(D) =$ Action$(D_k) \wedge$ Source$(D) =$ Source$(D_k)$ $\}$ $(D_k \in D_j,\ D_j \subseteq D_i)$
        * $N_k \Leftarrow \{\ N : N \in F \wedge$ Address$(N,\ F_i) =$ Source$(D_k)$ $\}$ $(N_k \subset F)$
        * $N_j \Leftarrow \{N : N =$ Source$(D)$ for some $D \in D_j\ \}$
        * if $N_j \neq N_k$ then
            · $F_i \Leftarrow$ Split-set$(N_j, N_k, F, D, F_i)$
        * Apply any $D \in D_j$ to $F_i$
        * $D_i \Leftarrow D_i - D_j$
```

APPLY-DELTAS (F, D)**:** Applies a set of deltas D to a forest F, producing a modified forest.

NODE (D)**:** Takes a delta and returns the node for which the delta is scheduled.

SOURCE (D)**:** Takes a delta and returns the address of the node which caused the delta to be scheduled (relative to **Node**(D)).

ACTION (D)**:** Takes a delta and returns the action to be performed on **Node**(D).

SPLIT-SET (N_j, N_k, F, D, F_i)**:** Takes a node F_i, a set N_k of nodes which have the same address A relative to F_i, a subset N_j of N_k which are discriminated from $\{N_k - N_j\}$ by F_i, the set of nodes F in the forest, and the set of deltas D scheduled for F. **Split-set** splits the set of nodes N_j from $\{N_k - N_j\}$, and returns a copy of F_i with only (and all) nodes in N_j having the address A relative to F_i. F and D are modified appropriately and the modifications are visible to the caller.

FIRST (N)**:** Returns the first node in the set of nodes N.

COPY (N, D, F, *Direction*, A_1)**:** Returns a copy of the node N. The node copy gets a copy of every delta scheduled for node N. F and D are updated appropriately. If *Direction* = *Up*, we don't pack N and its copy. If *Direction* = *Down*, we pack N and its copy. If *Direction* = *Down*, the copy does not (yet) get a child pointer to its child at location A_1 (relative to N). Instead, a new (empty) packed node can be pointed to.

ADDRESS (N, S) **:** Returns the address of node N relative to the node S.

Figure 13: Functions defined by the algorithm.

```
SPLIT-SET($N_j, N_k, F, D, F_i$)

• $A \Leftarrow$ Address(First($N_j$), $F_i$)
• $A_1 \Leftarrow$ first step of address $A$.
• $A_{\text{rest}} \Leftarrow$ address $A$, minus its first step, $A_1$
• if $A_1$ is a step up then
    – $\textit{Direction} \Leftarrow \textit{Up}$
  else
    – $\textit{Direction} \Leftarrow \textit{Down}$
• $F_{\text{new}} \Leftarrow$ Copy($F_i, F, D$, $\textit{Direction}$, $A_1$)
• $N \Leftarrow \{n : n \in F \wedge$ Address$(n, F_i) = A_1$ $\}$ ($N$ is the set of nodes in $F$
  with address $A_1$ relative to $F_i$)
• for all $N_p \in N$ do
    – $N_{p_j} \Leftarrow \{N_m : N_m \in N_j \wedge$ Address$(N_m, N_p) = A_{\text{rest}}$ $\}$
    – $N_{p_k} \Leftarrow \{N_m : N_m \in N_k \wedge$ Address$(N_m, N_p) = A_{\text{rest}}$ $\}$
    – if $N_{p_j} = \emptyset$ then (base case)
        ∗ Do nothing (keep arc from $F_i$ to $N_p$)
    – else if $N_{p_k} - N_{p_j} = \emptyset$ then (base case)
        ∗ Replace arc from $F_i$ to $N_p$ with arc from $F_{\text{new}}$ to $N_p$ (Note:
          breaking an arc between a subnode of a packed node and a
          parent of the packed node is done by removing the subnode
          from the packed node)
    – else (recursion case)
        ∗ $N_{p_{\text{new}}} \Leftarrow$ Split-set($N_{p_j}, N_{p_k}, F, D, N_p$)
        ∗ Replace arc from $F_i$ to $N_{p_{\text{new}}}$ with arc from $F_{\text{new}}$ to $N_{p_{\text{new}}}$
• Return $F_{\text{new}}$
```

11 Handling Localized Generation Phenomena

Before proceeding to discuss NP coindexation, it is worth noting that node packing provides us with a convenient tool to handle localized generation phenomena. A "localized" generation operation can be defined as a parser operation that takes a parse tree and returns one or more parse trees whose only differences are in a small subtree. Trace type determination is such a localized generation operation. When the LR parser has generated the initial phrase structure trees, empty noun phrases have no children. Later in the parse process, each empty NP is further specified as an NP-trace, a wh-trace, "pro", or "PRO." Our current implementation of this trace type determination can usually narrow the possibilities for the type of empty NP down to two of the above four. Then, parse trees are generated with the hypothesized empty NP types, the incorrect trees to be weeded out later. This type of generator process is deemed "localized" since it doesn't try to make disparate constituents point to one another, as is the case with the NP coindexation and chain formation generators.

Such a localized generator could make use of the node packing mechanism by creating a packed node containing each of its newly-created alternatives for the subtree being operated upon. For example, in trace type determination, if we hypothesize an empty NP to be either an NP-trace or "pro," we would replace the empty NP node by a packed node of two NPs, one having the single child "NP-trace," and the other having the single child "pro." The generator thus changes only a small part of the forest, and doesn't replicate the entire input tree as it would have to if we were not operating on packed forests.

12 Representing Chains and Coindexation in a Shared-Packed Forest

The structure sharing scheme discussed so far works well in conjunction with most principles. There is a problem, however, with regard to the two main generators in our system, chain formation and NP coindexation. This section explains the problem and the proposed solution.

12.1 The Problem

In a shared-packed forest, long-distance relationships between nodes (*i.e.*, faraway nodes pointing to one another) tend to be destructive to node sharing and packing. This is because in the case that a node must discriminate between multiple nodes with which it holds a long-distance relationship, a potentially long path between the node and the split's junction must be replicated. So, structure sharing works well with short-distance relationships between nodes, but long-distance relationships between nodes tend to break down the structure sharing in a forest. The coindexation of NPs is just such a long-distance relationship. If the coindexation relationship between nodes is encoded directly into the coindexed nodes (as a feature of the nodes), we will be forced to keep separate copies of each NP node which has different indexes in different trees. Furthermore, since such NP nodes must be kept as separate copies, all of the forest structure between these NPs must be replicated as well. This negates the advantage of structure sharing, and in most cases would amount to destroying most of the sharing in the forest. We need a scheme with which to represent the different coindexations of NPs within a forest while preserving the structure sharing present in the forest. The same applies to representing chains within a forest.

12.2 A New Representation For Forests

Fong gives a compositional definition of NP coindexation from which an efficient bottom-up method of obtaining the possible coindexations of NPs within a tree can be constructed. Figure 14 shows a forest with four NPs, and the possible coindexations for these NPs. This forest represents the set of parse trees belonging to the terminal string "*noun noun noun noun*", given the grammar G_1:

G_1:
$X \rightarrow X\ X$
$X \rightarrow NP$
$NP \rightarrow noun.$

In figure 14, beside each node we have enumerated the possible NP coindexations of NPs in the node's subtree. The bottom-up coindexation algorithm would allow us to store these coindexations in the corresponding nodes

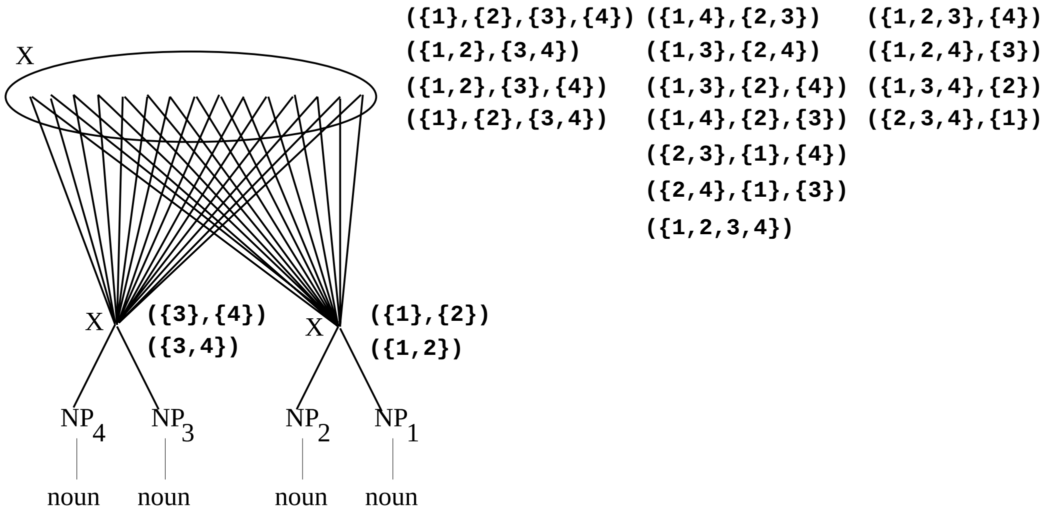


Figure 14: Naive representation of a shared forest with coindexation information.

as we traverse (or build) the tree in a bottom-up fashion. Fong's method was devised to allow the incremental construction of coindexation information so that NP coindexation can be effectively interleaved with the LR parsing. It turns out that this incremental process also helps us to preserve a forest's structure sharing.

In figure 14, there are fifteen possible coindexations of the four NPs in the tree. Therefore, the tree in figure 14 could be thought of as a forest of fifteen trees. The root node contains the fifteen possible coindexations of the four NPs, and the tree's phrase structure gives the remaining details of the forest. Note that while the intermediate nodes contain an account of their subtrees' possible NP coindexations, this information stored in the intermediate nodes cannot be said to contribute to the forest's representation in any meaningful way. It is only the root node that tells us the possible NP coindexations of the NPs in the forest.

The forest representation in figure 14 is unsatisfactory for two reasons. First, since only the root node of the forest gives us NP coindexation information, this creates an uneasy asymmetry between the root node of a forest and the rest of the forest's nodes, which will complicate any traversal of the forest. The second reason is that while constructing the parse tree (in a

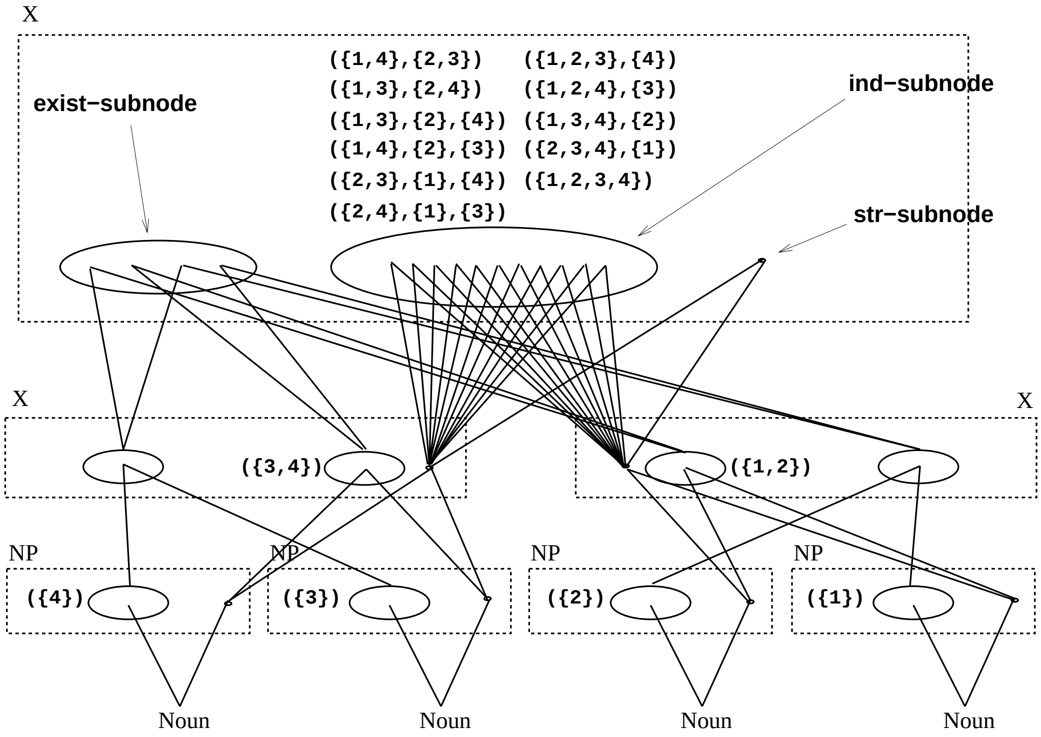


Figure 15: Representation of a shared forest with coindexation information.

bottom-up fashion), if we reach the point where we are to perform a reduction, each child node of the current reduction (previously the packed root of a subtree) must be unpacked, and a new packed root node must be constructed containing the information previously held in the packed children. A complicating factor is discriminating between packed children which were packed for NP coindexation reasons and packed children which were packed for other reasons.

A better representation of the same forest is shown in Figure 15. In this representation, each node contains only NP coindexations where NPs from different children are coindexed. The node packing in a forest's intermediate nodes is preserved, and the packing of these intermediate nodes provides a natural way of expressing the possible NP coindexations of the overall forest in terms of the coindexations of the NPs in each subtree.

The structure of this new forest representation is as follows. Each node of the forest is now replaced by a "supernode" which consists of two packed nodes and one unpacked node:

Structure Subnode (*str-subnode*): An unpacked subnode of the supernode, whose only function is to point to the structural children of the supernode. For example, in figure 15, the top X supernode's *str-subnode* points to the *str-subnodes* of the two child X supernodes.

New Coindexations Node (*ind-subnode*): A packed subnode of the supernode, which in turn will contain subnodes representing the different possible coindexings of the supernode's children where noun phrases under different children are coindexed. For example, in figure 15, the top X supernode's *ind-subnode* contains eleven subnodes corresponding to the eleven possible coindexations with one of the NPs from the right subtree being coindexed with one of the NPs from the left subtree.

Existing Coindexations Node (*exist-subnode*): A packed subnode of the supernode, which will represent all possible coindexations of the NPs under this supernode where no NP from one subtree is coindexed with an NP from another subtree. For example, in figure 15, the top X supernode's *exist-subnode* represents the four coindexations ({1}, {2}, {3}, {4}), ({1, 2}, {3}, {4}), ({1}, {2}, {3, 4}), and ({1, 2}, {3, 4}). The coidexation combinations represented by this *exist-subnode* are not explicitly kept within the node. They are implied by the packed structure of this node and its children.

Some things to note about the use of supernodes are:

1. It is important to note that the *str-subnode* of a supernode in a forest is not processed when traversing the forest. For example, to enumerate all trees represented by the forest, we traverse the subforest of the *ind-subnode* of the forest's root supernode, and the subforest of the *exist-subnode* of the forest's root supernode, but *not* the subtree of the *str-subnode* of the forest's root supernode, then add the resulting (virtual) trees. In other words, the *str-subnode* of a supernode is useful only when constructing nodes dominating the supernode.

2. We need only use "supernodes" for nodes in our forest that contain coindexation information. *I.e.*, NP nodes or nodes that dominate an NP node.

3. Features not related to coindexation are common to all elements of a supernode, so we place them outside the supernode's subnodes. Features related to coindexation (*i.e.*, the new coindexations that the supernode introduces) are placed in the *ind-subnode*).

4. Apart from accessing the features of a packed node residing in a supernode, which are global to the supernode, a GB module unrelated to coindexation need not pay much attention to the presence of a supernode within a forest when traversing it. The forest can be traversed by traversing the subforests of the *ind-subnode* and *exist-subnode* of the forest's root supernode. The rest of the sharing and packing of nodes works in the normal way.

The following is a more precise definition of the structure of a supernode:

Leaf Node: A "leaf" supernode (*e.g.*, any of the NP nodes in figure 15) consists of:

1. A *str-subnode* dominating each simple node below the supernode (as in a normal parse tree).
2. An *ind-subnode* with one subnode, dominating the children of the supernode. The subnode contains the unique index of this leaf supernode.

Interior Node: A supernode that dominates another supernode (*e.g.*, any of the X supernodes in figure 15) consists of:

1. A *str-subnode* dominating the *str-subnode* of each child supernode. If a child of the supernode is not itself a supernode, the *str-subnode* points at the simple child.
2. An *ind-subnode* with one child for each possible coindexation involving the indexes contained in the supernode's children where an index from one child is coreferenced with an index from another child. The coindexation associated with each child is explicitly kept with the child as a feature.
3. An *exist-subnode* with one child for each possible combination of the supernode's childrens' *ind-subnodes* and *exist-subnodes*. For

the case of binary branching, if both children have *exist-subnodes*, this *exist-subnode* has four subnodes. If only one of the children has an *exist-subnode*, this *exist-subnode* has two subnodes. If neither of the children has an *exist-subnode*, this *exist-subnode* has one subnode. See figure 15 for examples of this. The coidexation combinations represented by this *exist-subnode* are not explicitly kept within the node. They are implied by the packed structure of this node and its children.

This new representation of the forest is advantageous for these reasons:

1. While containing the NP coindexation information, the forest still contains a large amount of structure sharing.

2. There is no asymmetry between the forest's root node and the forest's other nodes.

3. This forest representation is amenable to principle interleaving. As each node is constructed, the node's subtree's NP coindexations can be computed and binding conditions can be checked for the new coindexations the node introduces. No binding condition checks are repeated because all NP coindexations introduced by a new node are guaranteed not to be present in the subtrees of the node.

4. We have kept the NP coindexation information as close as possible to the relevant nodes.

5. The most commonly used operations related to NP coindexation are still efficient. Namely,

    - To figure out in which indexations an NP node can participate, we start at the node and travel upward. All indexations in which the NP participates are located in the nodes dominating the NP.
    - To figure out whether two NPs are coindexed, we start at each of the NPs and move upward. The wanted information will be located at the lowest node dominating both NPs.
    - It is still a simple matter to traverse the forest enumerating the component trees and collecting the NP coindexations for each tree. A simple depth-first traversal is all that is needed for this.

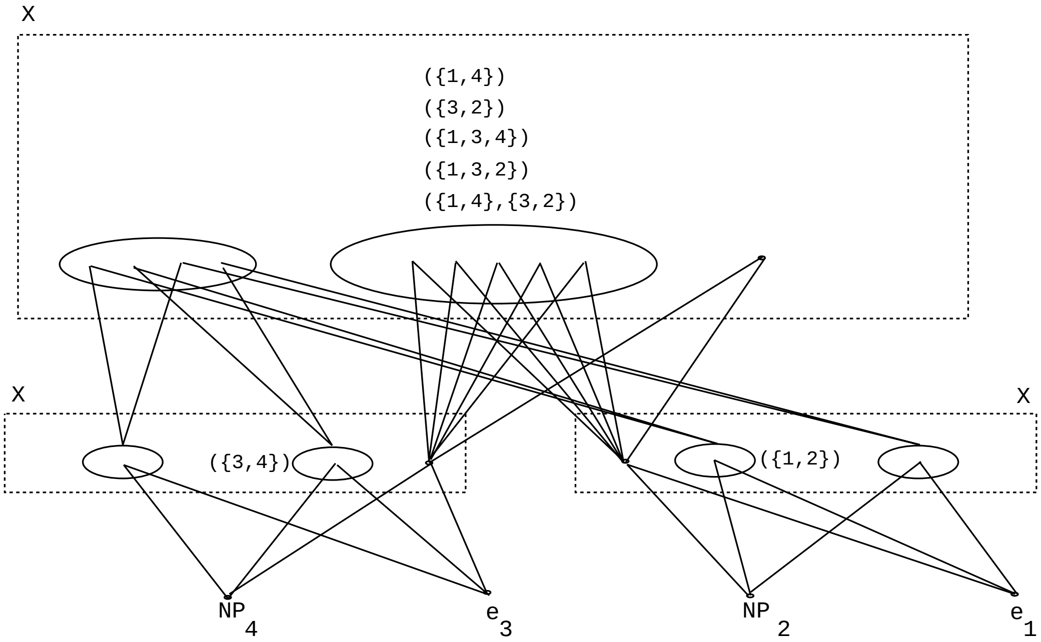


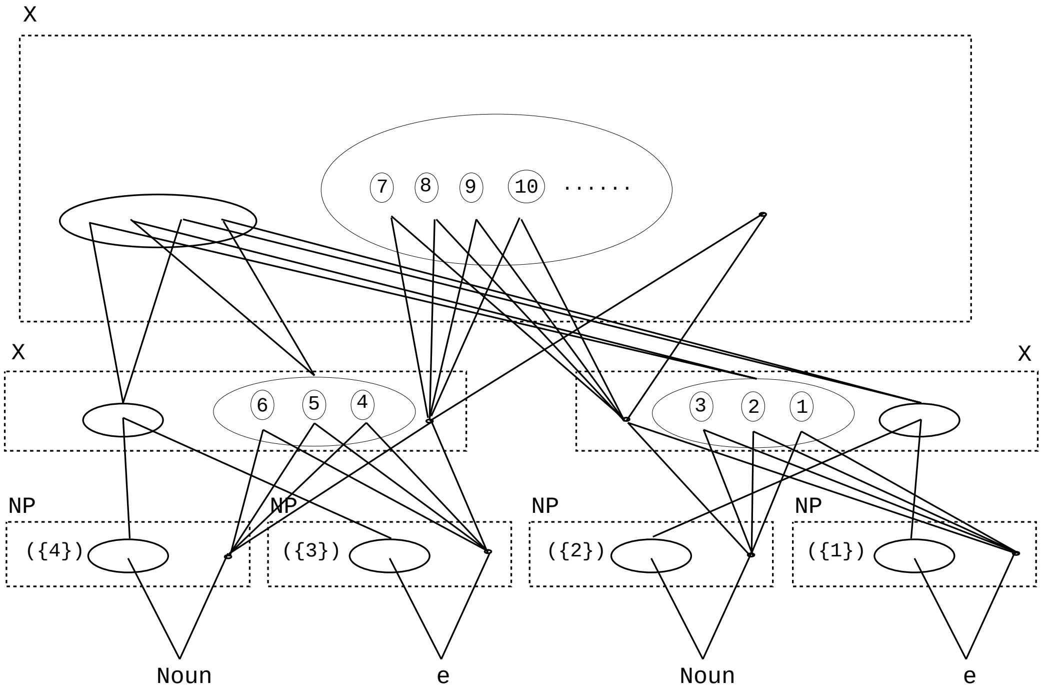
Figure 16: Representation of a shared forest with chain information.

The forest representation illustrated in figure 15 is more difficult to construct (while parsing) than that in figure 14. Since each node is now a supernode consisting of two packed nodes and an unpacked node, which must be constructed (during a reduce operation) according to the structure of a supernode given above.

It is easy to represent chains using this new forest representation. A sample representation for chains in a forest is shown in figure 16. It is also easy to combine coindexation and chain information in the same forest, as shown in figure 17.

13 Performance Gain When Using A Compact Forest

We have shown how to use compact forests within a GB parser. When applying a generator to the compact forest representation as outlined above, the number of forest nodes remains about equal to the number of nodes in



Features: (Chains, Coindexations)

1 (({1,2}),({1},{2},{3},{4}))
2 ((),({1,2},{3},{4}))
3 (({1,2}),({1,2},{3},{4}))
4 ((),({3,4},{1},{2}))
5 (({3,4}),({3,4},{1},{2}))
6 (({3,4}),({1},{2},{3},{4}))

7 ((),({1,3},{2},{4}))
8 ((),({1,3},{2,4}))
9 (({1,4}),({1},{2},{3},{4}))
10 (({1,4}),({1,2},{3,4}))

Figure 17: Representation of a shared forest with both coindexation and chain information.

the forest before the generator was applied. Applying a module unrelated to the generator will require about the same time whether we apply it before and after the generator has been applied. For example, take a forest with coindexation information. Applying the case filter operation on such a forest will require the examination of about the same number of nodes as would be required had we applied the case filter operation on the same forest without coindexation information. In contrast, if we were not using the compact forest representation, applying the case filter after a generator was applied will require going through all trees produced by the generator. If the generator produced 10 trees for each tree it took in, the case filter would have to do 10 times as much work as it would have to do if it were applied before the generator. Thus, in the best case, the performance gain of using compact forests is equal to the product of the branching factors of the parser's generators, minus the overhead of using compact forests.

Similar speedups were obtained by de Marcken [dM94] by applying memoization techniques to a GB parser's modules. de Marcken's optimization was generalized to handle any generate-and-test search problem, not just GB parsing. de Marcken's solution and the solution presented by this paper, in effect, solve the same problem but approach it in different ways, and have different overhead costs.

14 Parallelizing by Using Forest Deltas

The delta list facility presented in section 7 allows us to apply modules to a forest in parallel, collect their intended changes, reconcile any conflicting changes, then apply the changes to the forest in one step. The parallelization of GB modules can be performed on three separate levels:

- Separate forests can be operated on simultaneously if they do not share any structure (ideally, if we achieve our goal of preserving structure sharing throughout the parser's modules, all our parse trees would be combined into one forest).

- A single forest can be operated on by multiple (independent) modules simultaneously.

- Most modules can operate on a forest's nodes simultaneously. *I.e.*, if the module performs a task that is applied to each individual node of the forest (this is the case with most modules), then this task can be applied to all a forest's nodes simultaneously. For example, the case filter needs to check each overt NP node to make sure it possesses case. Another example would be setting up government relations. This is a procedure that is applied to each minimal projection (or maximal projection with an index) to find possible constituents which it governs, and mark them appropriately.

15 Performance Gain When Parallelizing Using Forest Deltas

Beyond the above-mentioned speedup related to structure sharing, further performance gains can be realized by parallelizing the parse process using the forest delta mechanism, as outlined above in section 7. It is difficult to predict beforehand the speedup we will obtain from this parallelization because the deltas which are constructed in parallel must be applied serially, so only part of the parsing process can be parallelized. In addition, we must deal with issues of process synchronization and interprocess communication, which will impose additional overhead.

Past work on parallelization of GB parsing was done by de Marcken [dM94], Kuhns [Kuh90], and [Mil92].

Kuhns' approach was to use the parallel programming language PARLOG to coroutine certain GB operations. His research as reported in [Kuh90] was not yet fully implemented, and it was not clear to what extent his system successfully parallelized the execution of GB modules. His parser was deterministic in the sense that it did not pursue different parse paths simultaneously. So, Kuhns' use of parallelism was less demanding (of processor power) than a parallel architecture that pursued the different parse paths simultaneously, but the determinism requirement may have placed undue restrictions on the coding of the GB modules.

De Marcken treats parsing as a search problem, where the parser searches through the search space determined by the GB modules (generators create forks in the search space, and filters cause some paths to terminate unsuccessfully). His approach to parallelizing the parsing process is to process separate search paths in parallel, *i.e.*, to simultaneously follow each hypothesized parse tree through the search space. He also provides the mechanism to apply several GB modules to a parse tree in parallel. His parallelization worked on the Lucid Common Lisp interleaved process environment, but was not extended to work on an actual parallel processing machine.

Millies used the Prolog-II *freeze* facility to achieve, in effect, the application of more than one GB principle to a parse tree, although his work, like Kuhns' and de Marcken's was not ported to an actual parallel machine, and no statistics related to the parallelization of GB modules was reported on.

16 Conclusion

We have discussed the benefits of using Tomita's algorithm in a GB parser, and talked about issues arising from the use of Tomita's algorithm, such as its possible beneficial impact on the performance of principle interleaving, and using ϵ-grammars. We gave an overview of how structure sharing within a parse forest can be preserved by the modules of a GB parser, including the use of relative addressing, using forest deltas, when and how a node must be split, how local generation phenomena can easily be handled when using the packed forest representation, and the use of new ways to represent chains and coindexation within a parse forest.

Finally, we touched on a method of parallelizing a GB parser using the forest delta mechanism.

Acknowledgements

I would like to thank Joyce Friedman and Carl de Marcken for helpful comments on earlier versions of this report.

References

[dM94] Carl de Marcken. Methods for parallelizing search paths in parsing. AI Memo 1453, A.I. Laboratory, Massachusetts Institute of Technology, 1994.

[Fon91] Sandiway Fong. *Computational Properties of Principle-Based Grammatical Theories.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., 1991.

[KK85] Lauri Karttunen and Martin Kay. Structure Sharing with Binary Trees. In *Proceedings of the 23rd Annual Meeting of the ACL.* Association for Computational Linguistics, 1985.

[Kuh90] Robert J. Kuhns. A PARLOG Implementation of Government-Binding Theory. In *Proceedings of COLING '90.* Association for Computational Linguistics, 1990.

[Mil92] Sebastian Millies. Modularity, Parallelism, and Licensing in a Principle-Based Parser for German. In *Proceedings of COLING '92.* Association for Computational Linguistics, 1992.

[NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing.* Kluwer Academic Publishers, Dordrecht, 1991.

[Per85] Fernando Pereira. A Structure-Sharing Representation for Unification-Based Grammar Formalisms. In *Proceedings of the 23rd Annual Meeting of the ACL.* Association for Computational Linguistics, 1985.

[Sha93] Marwan Shaban. A Minimal GB Parser. Technical Report 93-013, Computer Science Department, Boston University, Boston, Mass., 1993.

[Sha94] Marwan Shaban. A Hybrid GLR Algorithm for Parsing with Epsilon Grammars. Technical Report 94-004, Computer Science Department, Boston University, Boston, Mass., 1994.

[Tom86] Masaru Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Dordrecht, 1986.

[Tom91a] Masaru Tomita, editor. *Generalized LR Parsing.* Kluwer Academic Publishers, Dordrecht, 1991.

[Tom91b] Masaru Tomita. The generalized LR parsing algorithm. In Masaru Tomita, editor, *Generalized LR Parsing*. Kluwer Academic Publishers, Dordrecht, 1991.