SRI International

MORE NOTES FROM THE UNIFICATION UNDERGROUND:

A SECOND COMPILATION OF PAPERS ON UNIFICATION-BASED GRAMMAR FORMALISMS

Technical Note 361

August 1985

By: Stuart M. Shieber,
Lauri Karttunen, and
Fernando C. N. Pereira

Artificial Intelligence Center
SRI International
and
Center for the Study of Language and Information
Stanford University

Martin Kay

Xerox Palo Alto Research Center
and
Center for the Study of Language and Information
Stanford University

This research has been made possible in part by a gift from the System Development Foundation, and was also supported by the Defense Advanced Research Projects Agency under Contract N00039-84-K-0078 with the Naval Electronic Systems Command.
The views and conclusions contained in this document are those of the authors and should not be interpreted as representative of the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, or the United States government.

SRI International®
333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

More Notes from the Unification Underground:
A Second Compilation of Papers on Unification-Based Formalisms

Stuart M. Shieber, Lauri Karttunen, and Fernando Pereira

Artificial Intelligence Center
SRI International
and
Center for the Study of Language and Information
Stanford University

Martin Kay

Xerox Palo Alto Research Center
and
Center for the Study of Language and Information
Stanford University

August 20, 1985

Contents

Preface

This report is the second compilation of papers by members of the PATR group at SRI International and collaborators reporting on ongoing research on both practical and theoretical issues concerning grammar formalisms. The current formalism being simultaneously designed, implemented, and used by the group, PATR-II, is based on unification of directed-graph structures. The papers presented in this compilation describe techniques for efficiently implementing formalisms that make use of such a concept of unification. The first two chapters are devoted to the problem of representing directed graphs as data structures such that unification is efficiently implementable. The final chapter describes a general technique for extending context-free parsing methods to unification-based formalisms. The techniques described in these papers have all been implemented and tested. All three chapters are versions of papers presented at the Twenty-Third Annual Meeting of the Association for Computational Linguistics, held at the University of Chicago, Chicago, Illinois, during July 8 through 12, 1985, and appear in the proceedings of that conference.

Research on PATR-II was begun as part of the KLAUS (Knowledge Learning And Using System) project at SRI, and was set up with the intention of experimenting with mathematically well-defined alternatives to the DIALOGIC natural-language processing system. The more theoretical research was made possible in part by a gift from the System Development Foundation and was conducted as part of a coordinated research effort with the Situated Language program at the Center for the Study of Language and Information, Stanford University.

The PATR group at SRI is a rather liquid group of researchers which has included, at various times, John Bear, Lauri Karttunen, Paul Martin,

Fernando Pereira, Jane Robinson, Stan Rosenschein, Stuart Shieber, Susan Stucky, Mabry Tyson, and Hans Uszkoreit. In addition, the group has benefitted from interaction with many researchers at CSLI, Xerox PARC, and elsewhere: one of them, Martin Kay, appears in this compilation as a coauthor; many others are represented in spirit. The research reported here is a direct result of the aid and interaction of all of these researchers. However, they should not be held accountable for any errors in the present work, nor should the opinions expressed herein be construed as indicative of their personal predilections.

Chapter 1

Structure Sharing with Binary Trees

This chapter was written by Lauri Karttunen of the Artificial Intelligence Center, SRI International and the Center for the Study of Language and Information, Stanford University, and Martin Kay of the Xerox Palo Alto Research Center, and the Center for the Study of Language and Information, Stanford University.[1]

Many current interfaces for natural language represent syntactic and semantic information in the form of directed graphs where attributes correspond to vectors and values to nodes. There is a simple correspondence between such graphs and the matrix notation linguists traditionally use for

[1] This research, made possible in part by a gift from the System Development Foundation, was also supported by the Defense Advanced Research Projects Agency under Contracts N00039-80-C-0575 and N00039-84-C-0524 with the Naval Electronic Systems Command. The views and conclusions contained in this document are those of the author and should not be interpreted as representative of the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, or the United States government.

Thanks are due to Fernando Pereira and Stuart Shieber for their comments on earlier presentations of this material.

feature sets.

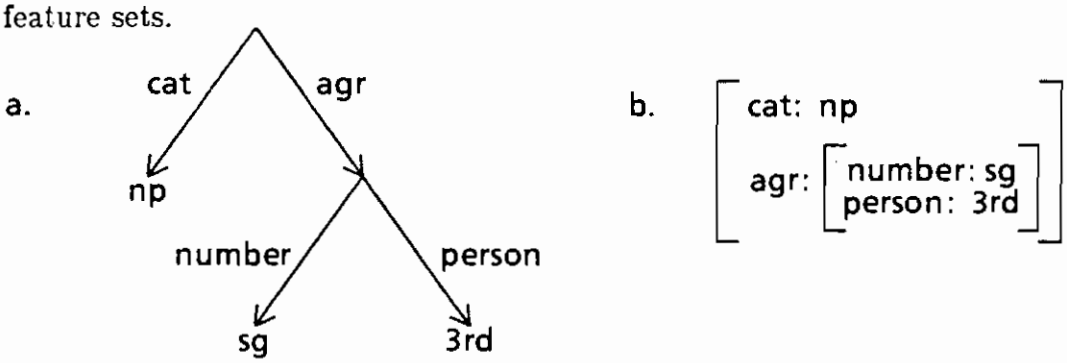

Figure 1

The standard operation for working with such graphs is unification. The unification operation succedes only on a pair of compatible graphs, and its result is a graph containing the information in both contributors. When a parser applies a syntactic rule, it unifies selected features of input constituents to check constraints and to build a representation for the output constituent.

1.1 Problem: Proliferation of Copies

When words are combined to form phrases, unification is not applied to lexical representations directly because it would result in the lexicon being changed. When a word is encountered in a text, a copy is made of its entry, and unification is applied to the copied graph, not the original one. In fact, unification in a typical parser is always preceded by a copying operation. Because of nondeterminism in parsing, it is, in general, necessary to preserve every representation that gets built. The same graph may be needed again when the parser comes back to pursue some yet unexplored option. Our experience suggests that the amount of computational effort that goes into producing these copies is much greater than the cost of unification itself. It accounts for a significant amount of the total parsing time. In a sense, most of the copying effort is wasted. Unifications that fail

typically fail for a simple reason. If it were known in advance what aspects of structures are relevant in a particular case, some effort could be saved by first considering only the crucial features of the input.

1.2 Solution: Structure Sharing

This paper lays out one strategy that has turned out to be very useful in eliminating much of the wasted effort. Our version of the basic idea is due to Martin Kay. It has been implemented in slightly different ways by Kay in Interlisp-D and by Lauri Karttunen in Zeta Lisp. The basic idea is to minimize copying by allowing graphs share common parts of their structure. This version of structure sharing is based on four related ideas:

- Binary trees as a storage device for feature graphs
- "Lazy" copying
- Relative indexing of nodes in the tree
- Strategy for keeping storage trees as balanced as possible

1.3 Binary Trees

Our structure-sharing scheme depends on represented feature sets as binary trees. A tree consists of cells that have a content field and two pointers which, if not empty, point to a left and a right cell respectively. For example, the content of the feature set and the corresponding directed graph in Figure 1 can be distributed over the cells of a binary tree in the following

way.

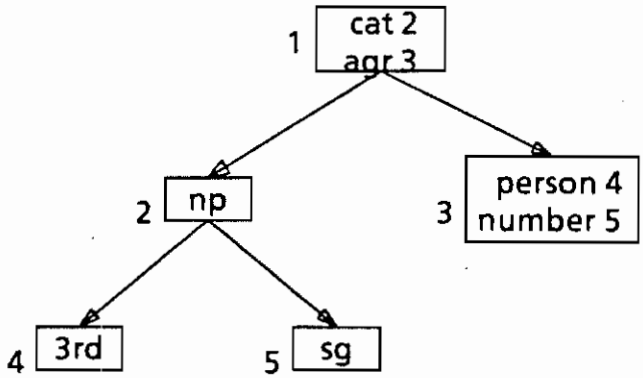

Figure 2

The index of the top node is 1; the two cells below have indices 2 and 3. In general, a node whose index is n may be the parent of cells indexed $2n$ and $2n + 1$. Each cell contains either an atomic value or a set of pairs that associate attribute names with indices of cells where their value is stored. The assignment of values to storage cells is arbitrary; it doesn't matter which cell stores which value. Here, cell 1 contains the information that the value of the attribute cat is found in cell 2 and that of agr in cell 3. This is a slight simplification. As we shall shortly see, when the value in a cell involves a reference to another cell, that reference is encoded as a relative index. The method of locating the cell that corresponds to a given index takes advantage of the fact that the tree branches in a binary fashion. The path to a node can be read off from the binary representation of its index by starting after the first 1 in this number and taking 0 to be a signal for a left turn and 1 as a mark for a right turn. For example, starting at node 1, node 5 is reached by first going down a left branch and then a right branch. This sequence of turns corresponds to the digits 01. Prefixed with 1, this is the same as the binary representation of 5, namely 101. The same holds for all indices. Thus the path to node 9 (binary 1001) would be LEFT-LEFT-RIGHT as signalled by the last three digits following the initial 1 in the binary numeral (see Figure 6).

1.4 Lazy Copying

The most important advantage is that the scheme minimizes the amount of copying that has to be done. In general, when a graph is copied, we duplicate only The operation that replaces copying in this scheme starts by duplicating the topmost node of the tree that contains it. The rest of the structure remains the same. Other nodes are modified only if and when destructive changes are about to happen. For example, assume that we need another copy of the graph stored in the tree in Figure 2. This can be obtained by producing a tree which has a different root node, but shares the rest of the structure with its original. In order to keep track of which tree actually owns a given node, each node carries a numeral tag that indicates its parentage. The relationship between the original tree (generation 0) and its copy (generation 1) is illustrated in Figure 3 where the generation is separated from the index of a node by a colon.

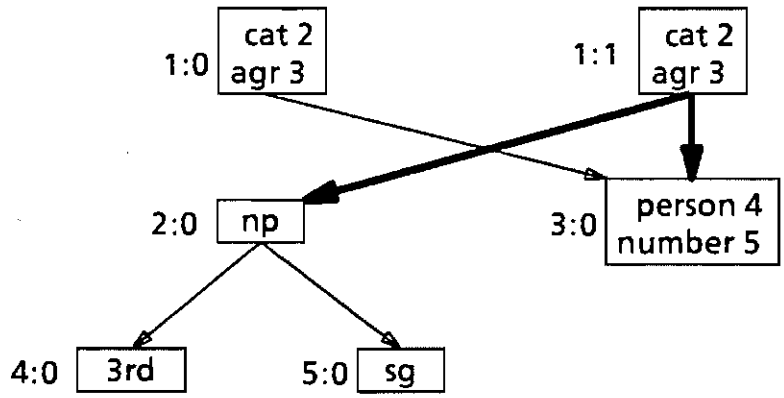

Figure 3

If the node that we want to copy is not the topmost node of a tree, we need to duplicate the nodes along the branch leading to it.

When a tree headed by the copied node has to be changed, we use the generation tags to minimize the creation of new structure. In general, all and only the nodes on the branch that lead to the site of a destructive

change or addition need to belong to the same generation as the top node of the tree. The rest of the structure can consist of old nodes. For example, suppose we add a new feature, say [gender: fem] to the value of agr in Figure 3 to yield the feature set in Figure 4.

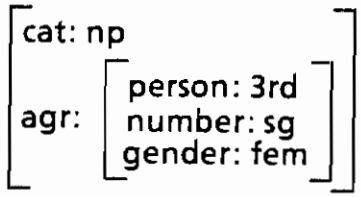

Figure 4

Furthermore, suppose that we want the change to affect only the copy but not the original feature set. In terms of the trees that we have constructed for the example in Figure 3, this involves adding one new cell to the copied structure to hold the value fem, and changing the content of cell 3 by adding the new feature to it.

The modified copy and its relation to the original is shown in Figure 5 . Note that one half of the structure is shared. The copy contains only three

new nodes.

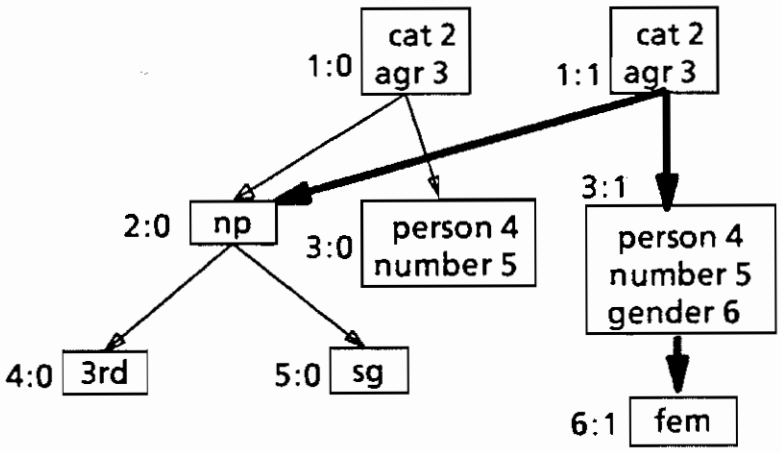

Figure 5

From the point of view of a process that only needs to find or print out the value of particular features, it makes no difference that the nodes containing the values belong to several trees as long as there is no confusion about the structure.

1.5 Relative Addressing

Accessing an arbitrary cell in a binary tree consumes time in proportion to the logarithm of the size of the structure, assuming that cells are reached by starting at the top node and using the index of the target node as an address: Another method is to use relative addressing. Relative addresses encode the shortest path between two nodes in the tree regardless of where they are are. For example, if we are at node 9 in Figure 6.a below and need to reach node 11, it is easy to see that it is not necessary to go all the way up to node 1 and then partially retrace the same path in looking up node 11. Instead, one can stop going upward at the lowest common ancestor,

node 2, of nodes 9 and 11 and go down from there.

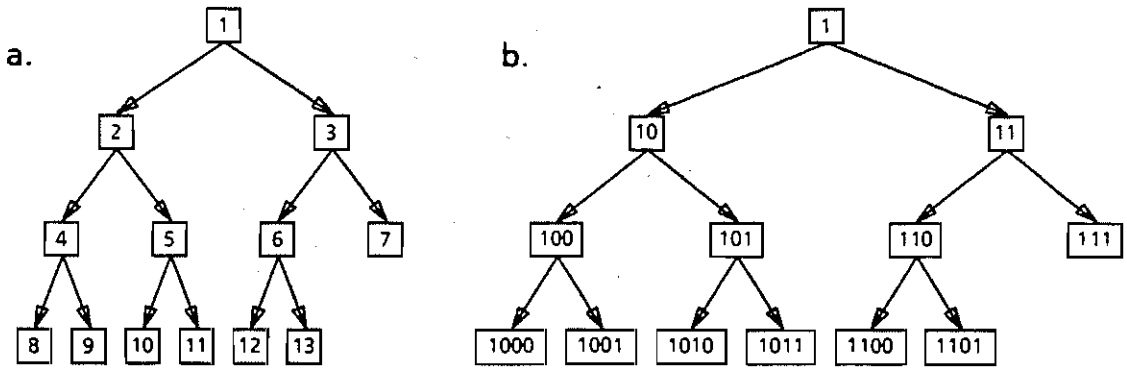

Figure 6

With respect to node 2, node 11 is in the same position as 7 is with respect 1. Thus the relative address of cell 11 counted from 9 is 2,7—"two nodes up, then down as if going to node 7". In general, relative addresses are of the form (up,down) where (up) is the number of links to the lowest common ancestor of the origin and (down) is the relative index of the target node with respect to it. Sometimes we can just go up or down on the same branch; for example, the relative address of cell 10 seen from node 2 is simply 0,6; the path from 8 or 9 to 4 is 1,1. As one might expect, it is easy to see these relationships if we think of node indices in their binary representation (see Figure 6.b).. The lowest common ancestor 2 (binary 10) is designated by the longest common initial substring of 9 (binary 1001) and 11 (binary 1011). The relative index of 11, with respect to, 7 (binary 111), is the rest of its index with 1 prefixed to the front.

In terms of number of links traversed, relative addresses have no statistical advantage over the simpler method of always starting from the top. However, they have one important property that is essential for our purposes: relative addresses remain valid even when trees are embedded in other trees; absolute indices would have to be recalculated. Figure 7 is a

recoding of Figure 5 using relative addresses.

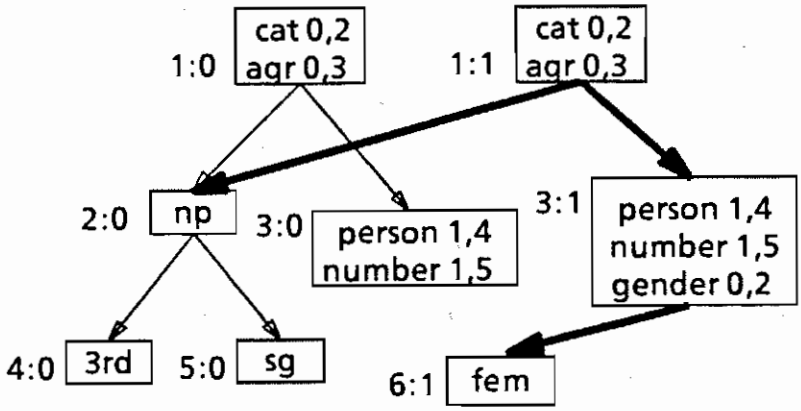

Figure 7

1.6 Keeping Trees Balanced

When two feature matrices are unified, the binary trees corresponding to them have to be combined to form a single tree. New attributes are added to some of the nodes; other nodes become "pointer nodes," i.e., their only content is the relative address of some other node where the real content is stored. As long as we keep adding nodes to one tree, it is a simple matter to keep the tree maximally balanced. At any given time, only the growing fringe of the tree can be incompletely filled. When two trees need to be combined, it would, of course, be possible to add all the cells from one tree in a balanced fashion to the other one but that would defeat the very purpose of using binary trees because it would mean having to copy almost all of the structure. The only alternative is to embed one of the trees in the other one. The resulting tree will not be a balanced one; some of the branches are much longer than others. Consequently, the average time needed to look up a value is bound to be worse than in a balanced tree. For example, suppose that we want to unify a copy of the feature set in Figure 1b, represented as in Figure 2 but with relative addressing, with

a copy of the feature set in Figure 8.

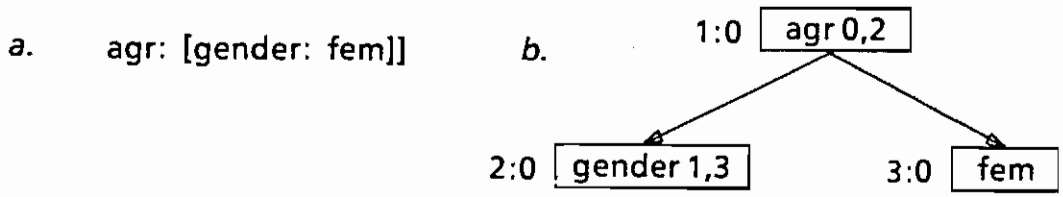

Figure 8

The resulting feature set and structure are shown in Figure 9.

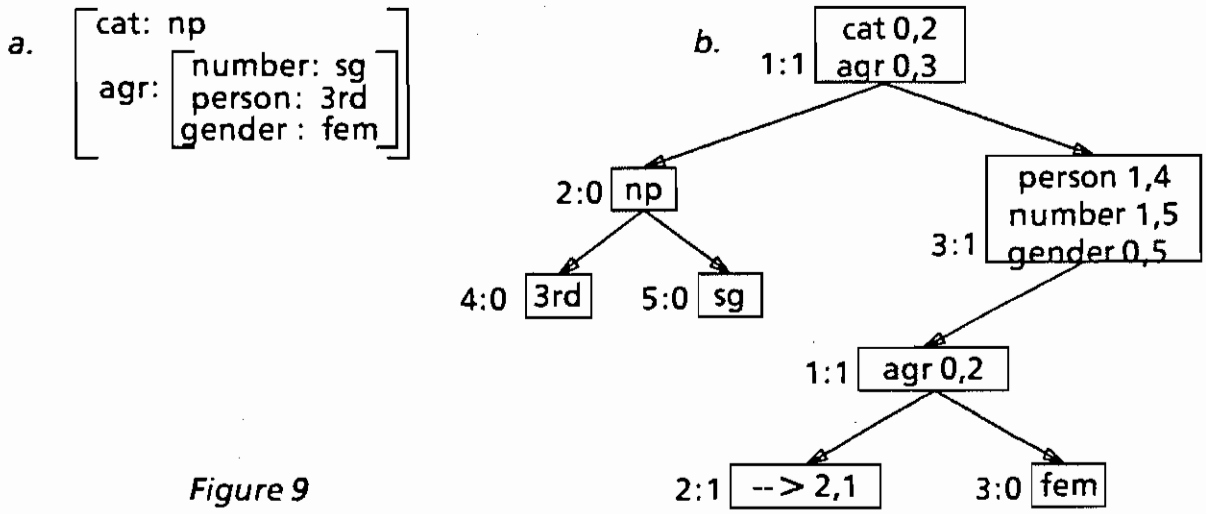

Figure 9

Although the feature set in Figure 9.a is the same as the one represented by the right half of Figure 7, the structure in Figure 9.b is more complicated because it is derived by unifying copies of two separate trees, not by simply adding more features to a tree, as in Figure 7. In 9.b, a copy of 8.b has been embedded as node 6 of the host tree. The original indices of both trees remain unchanged. Because all the addresses are relative; no harm comes from the fact that indices in the embedded tree no longer correspond to the true location of the nodes. Absolute indices are not used as addresses because they change when a tree is embedded. The symbol --> in node 2

of the lower tree indicates that the original content of this node—gender 1,3—has been replaced by the address of the cell that it was unified with, namely cell 3 in the host tree. In the case at hand, it matters very little which of the two trees becomes the host for the other. The resulting tree is about as much out of balance either way. However, when a sequence of unifications is performed, differences can be very significant. For example, if A, B, and C are unified with one another, it can make a great deal of difference, which of the two alternative shapes in Figure 10 is produced as the final result.

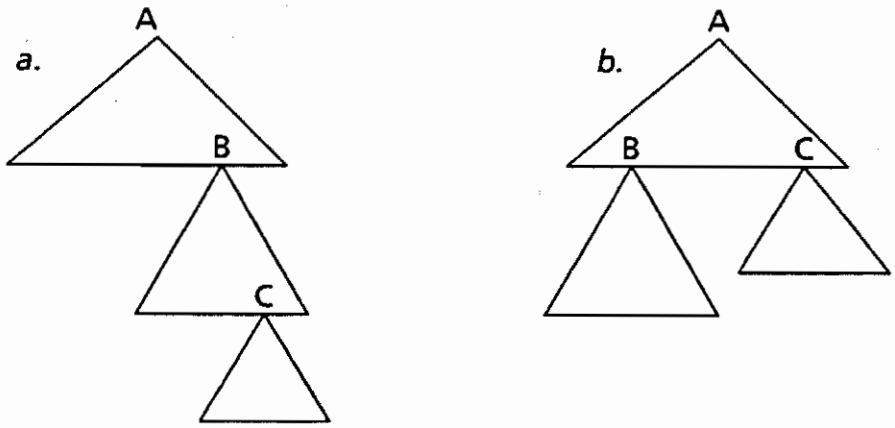

Figure 10

When a choice has to be made as to which of the two trees to embed in the other, it is important to minimize the length of the longest path in the resulting tree. To do this at all efficiently requires addtitional infornation to be stored with each node. According to one simple scheme, this is simply the length of the shortest path from the node down to a node with a free left or right pointer. Using this, it is a simple matter to find the shallowest place in a tree at which to embed another one. If the length of the longet path is also stored, it is also easy to determine which choice of host will give rise to the shallowest combined tree. Another problem which needs careful

attention concerns generation markers. If a pair of trees to be unified have independent histories, their generation markers will presumably be incommensurable and those of an embedded tree will therfore not be valide in the host. Various solutions are possible for this problem. The most straightforward is relate the histories of all trees at least to the extent of drawing generation markers from a global pool. In Lisp, for example, the simplest thing is to let them be CONS cells.

1.7 Conclusion

We will conclude by comparing our method of structure sharing with two others that we know of: R. Cohen's immutable arrays and the idea discussed in Fernando Pereira's paper at this meeting. The three alternatives involve different trade-offs along the space/time continuum. The choice between them will depend on the particular application they are intended for. No statistics on parsing are available yet but we hope to have some in the final version.

Chapter 2

A Structure-Sharing Representation for Unification-Based Grammar Formalisms

This chapter was written by Fernando Pereira of the Artificial Intelligence Center, SRI International and the Center for the Study of Language and Information, Stanford University.[1]

[1]This research, made possible in part by a gift from the Systems Development Foundation, was also supported by the Defense Advanced Research Projects Agency under Contracts N00039-80-C-0575 and N00039-84-C-0524 with the Naval Electronic Systems Command. The views and conclusions contained in this document are those of the author and should not be interpreted as representative of the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, or the United States government.

Thanks are due to Stuart Shieber, Lauri Karttunen, and Ray Perrault for their comments on earlier presentations of this material.

Abstract

This paper describes a structure-sharing method for the representation of complex phrase types in a parser for PATR-II, a unification-based grammar formalism.

In parsers for unification-based grammar formalisms, complex phrase types are derived by incremental refinement of the phrase types defined in grammar rules and lexical entries. In a naïve implementation, a new phrase type is built by copying older ones and then combining the copies according to the constraints stated in a grammar rule. The structure-sharing method was designed to eliminate most such copying; indeed, practical tests suggest that the use of this technique reduces parsing time by as much as 60%.

The present work is inspired by the structure-sharing method for theorem proving introduced by Boyer and Moore and on the variant of it that is used in some Prolog implementations.

2.1 Overview

In this paper I describe a method, ***structure sharing***, for the representation of complex phrase types in a parser for PATR-II, a unification-based grammar formalism.

In parsers for unification-based grammar formalisms, complex phrase types are derived by incremental refinement of the phrase types defined in grammar rules and lexical entries. In a naïve implementation, a new phrase type is built by copying older ones and then combining the copies according to the constraints stated in a grammar rule. The structure-sharing method eliminates most such copying by representing updates to objects (phrase types) separately from the objects themselves.

The present work is inspired by the structure-sharing method for theorem proving introduced by Boyer and Moore [2] and on the variant of it that is used in some Prolog implementations [20].

2.2 Grammars with Unification

The data representation discussed in this paper is applicable, with but minor changes, to a variety of grammar formalisms based on unification, such as definite-clause grammars [14], functional-unification grammar [9], lexical-functional grammar [6] and PATR-II [17]. For the sake of concreteness, however, our discussion will be in terms of the PATR-II formalism.

The basic idea of unification-based grammar formalisms is very simple. As with context-free grammars, grammar rules state how phrase types combine to yield other phrase types. But whereas a context-free grammar allows only a finite number of predefined atomic phrase types or *nonterminals*, a unification-based grammar will in general define implicitly an infinity of phrase types.

A phrase type is defined by a set of constraints. A grammar rule is a set of constraints between the type X_0 of a phrase and the types $X_1, \ldots, X_n$ of its constituents. The rule may be applied to the analysis of a string s_0 as the concatenation of constituents $s_1, \ldots, s_n$ if and only if the types of the s_i are compatible with the types X_i and the constraints in the rule.

Unification is the operation that determines whether two types are compatible by building the most general type compatible with both.

If the constraints are equations between attributes of phrase types, as is the case in PATR-II, two phrase types can be unified whenever they do not assign distinct values to the same attribute. The unification is then just the conjunction (set union) of the corresponding sets of constraints [13].

Here is a sample rule, in a simplified version of the PATR-II notation:

$$\begin{array}{lll} X_0 \rightarrow X_1\ X_2: & \langle X_0\ cat\rangle & = S \\ & \langle X_1\ cat\rangle & = NP \\ & \langle X_2\ cat\rangle & = VP \\ & \langle X_1\ agr\rangle & = \langle X_2\ agr\rangle \\ & \langle X_0\ trans\rangle & = \langle X_2\ trans\rangle \\ & \langle X_0\ trans\ arg_1\rangle & = \langle X_1\ trans\rangle \end{array} \tag{2.1}$$

This rule may be read as stating that a phrase of type X_0 can be the concatenation of a phrase of type X_1 and a phrase of type X_2, provided

that the attribute equations of the rule are satisfied if the phrases are substituted for their types. The equations state that phrases of types X_0, X_1, and X_2 have categories S, NP, and VP, respectively, that types X_1 and X_2 have the same agreement value, that types X_0 and X_2 have the same translation, and that the first argument of X_0's translation is the translation of X_1.

Formally, the expressions of the form $\langle l_1 \cdots l_m \rangle$ used in attribute equations are *paths* and each l_i is a *label*.

When all the phrase types in a rule are given constant *cat* (category) values by the rule, we can use an abbreviated notation in which the phrase type variables X_i are replaced by their category values and the category-setting equations are omitted. For example, rule (2.1) may be written as

$$\begin{array}{lll} S \rightarrow NP\ VP: & \langle NP\ agr \rangle & = \langle VP\ agr \rangle \\ & \langle S\ trans \rangle & = \langle VP\ trans \rangle \\ & \langle S\ trans\ arg_1 \rangle & = \langle NP\ trans \rangle \end{array} \qquad (2.2)$$

In existing PATR-II implementations, phrase types are not actually represented by their sets of defining equations. Instead, they are represented by symbolic solutions of the equations in the form of directed acyclic graphs (*dags*) with arcs labeled by the attributes used in the equations. Dag nodes represent the values of attributes and an arc labeled by l goes from node m to node n if and only if, according to the equations, the value represented by m has n as the value of its l attribute [13].

A dag node (and by extension a dag) is said to be *atomic* if it represents a constant value; *complex* if it has some outgoing arcs; and a *leaf* if is is neither atomic or complex, that is, if it represents an as yet completely undetermined value. The *domain* $\mathrm{dom}(d)$ of a complex dag d is the set of labels on arcs leaving the top node of d. Given a dag d and a label $l \in \mathrm{dom}(d)$ we denote by d/l the subdag of d at the end of the arc labeled l from the top node of d. By extension, for any path p whose labels are in the domains of the appropriate subdags, d/p represents the subdag of d at the end of path p from the root of d.

For uniformity, lexical entries and grammar rules are also represented by appropriate dags. For example, the dag for rule (2.1) is shown in Figure 2.1.

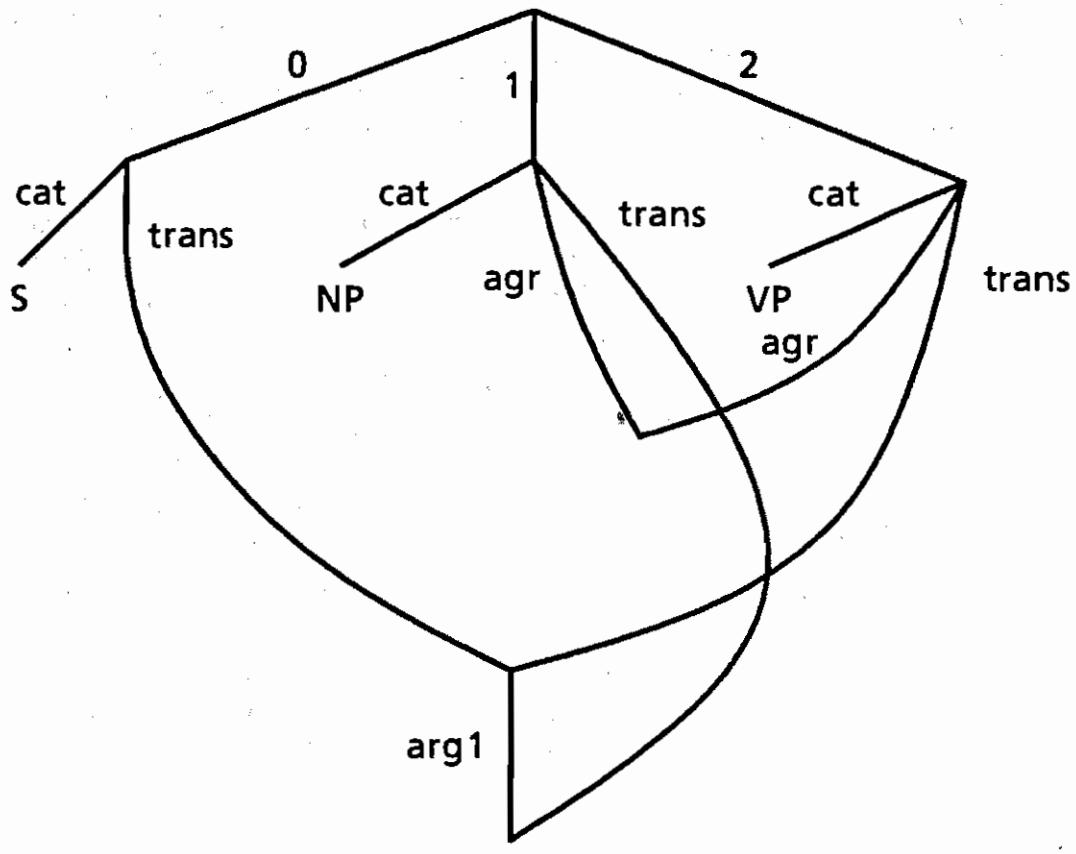

Figure 2.1: Dag Representation of a Rule

2.3 The Problem

In a chart parser [8] all the intermediate stages of derivations are encoded in *edges*, representing either incomplete (*active*) or complete (*passive*) phrases. For PATR-II, each edge contains a dag instance that represents the phrase type of that edge. The problem we address here is how to encode multiple dag instances efficiently.

In a chart parser for context-free grammars, the solution is trivial: instances can be represented by the unique internal names (that is, addresses) of their objects because the information contained in an instance is exactly the same as that in the original object.

In a parser for PATR-II or any other unification-based formalism, however, distinct instances of an object will in general specify different values for attributes left unspecified in the original object. Clearly, the attribute values specified for one instance are independent of those for another instance of the same object.

One obvious solution is to build new instances by copying the original object and then updating the copy with the new attribute values. This was the solution adopted in the first PATR-II parser [17]. The high cost of this solution both in time spent copying and in space required for the copies themselves constitutes the principal justification for employing the method described here.

2.4 Structure Sharing

Structure sharing is based on the observation that an initial object, together with a list of update records, contains the same information as the object that results from applying the updates to the initial object. In this way, we can trade the cost of actually applying the updates (with possible copying to avoid the destruction of the source object) against the cost of having to compute the effects of updates when examining the derived object. This reasoning applies in particular to dag instances that are the result of adding attribute values to other instances.

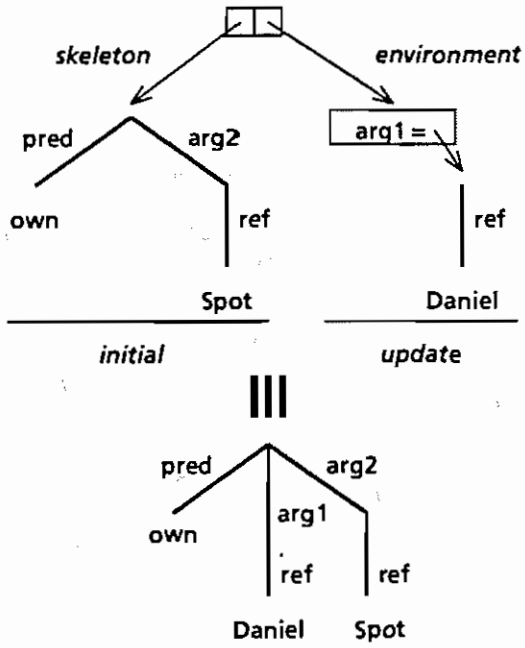

Figure 2.2: Molecule

As in the variant of Boyer and Moore's method [2] used in Prolog [20], I shall represent a dag instance by a *molecule* (see Figure 2.2) consisting of

1. [A pointer to] the initial dag, the instance's *skeleton*
2. [A pointer to] a table of updates of the skeleton, the instance's *environment.*

Environments may contain two kinds of updates: ***reroutings*** that replace a dag node with another dag; ***arc bindings*** that add to a node a new outgoing arc pointing to a dag. Figure 2.3 shows the unification of the dags

$$\begin{aligned} I_1 &= [a : x, b : y] \\ I_2 &= [c : [d : e]] \end{aligned}$$

After unification, the top node of I_2 is rerouted to I_1 and the top node of I_1 gets an arc binding with label c and a value that is the subdag $[d : e]$ of I_2. As we shall see later, any update of a dag represented by a molecule is either an update of the molecule's skeleton or an update of a dag (to which the same reasoning applies) appearing in the molecule's enviroment. Therefore, the updates in a molecule's environment are always shown in figures tagged by a boxed number identifying the affected node in the molecule's skeleton.

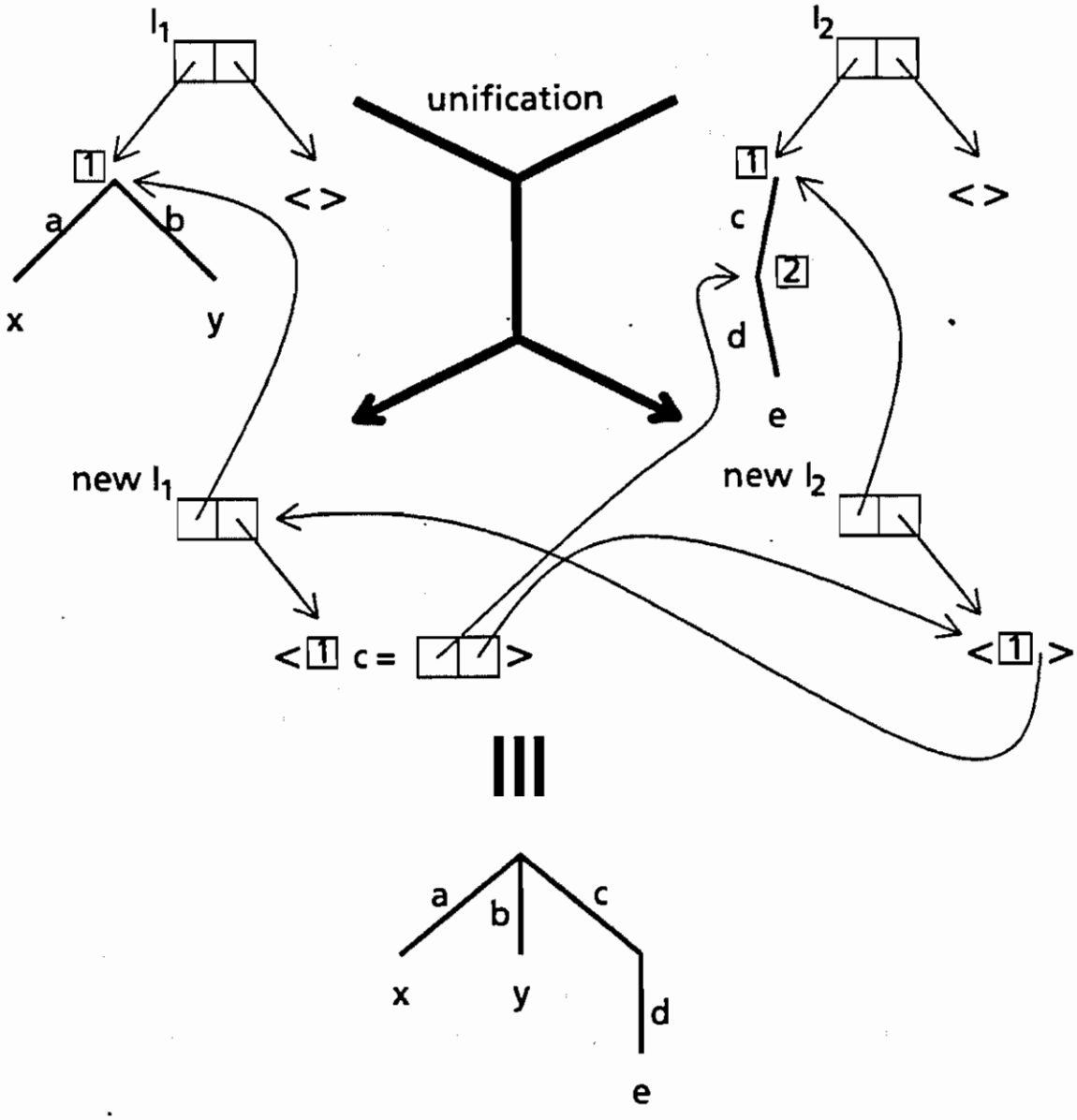

Figure 2.3: Unification of Two Molecules

The choice of which dag is rerouted and which one gets arc bindings is arbitrary.

For reasons discussed later, the cost of looking up instance node updates in Boyer and Moore's environment representation is $O(|d|)$, where $|d|$ is the length of the derivation (a sequence of resolutions) of the instance. In the present representation, however, this cost is only $O(\log |d|)$. This better performance is achieved by particularizing the environment representation and by splitting the representational scheme into two components: a *memory organization* and a *dag representation.*

A dag representation is a way of mapping the *mathematical entity* dag onto a memory. A memory organization is a way of putting together a memory that has certain properties with respect to lookup, updating and copying. One can think of the memory organization as the hardware and the dag representation as the data structure.

2.5 Memory Organization

In practice, random-access memory can be accessed and updated in constant time. However, updates destroy old values, which is obviously unacceptable when dealing with alternative updates of the same data structure. If we want to keep the old version, we need to copy it first into a separate part of memory and change the copy instead. For the normal kind of memory, copying time is proportional to the size of the object copied.

The present scheme uses another type of memory organization — *virtual-copy arrays* — which requires $O(\log n)$ time to access or update an array with highest used index of n, but in which the old contents are not destroyed by updating. Virtual-copy arrays were developed by David H. D. Warren [21] as an implementation of extensible arrays for Prolog.

Virtual-copy arrays provide a fully general memory structure: anything that can be stored in random-access memory can be stored in virtual-copy arrays, although pointers in machine memory correspond to indexes in a virtual-copy array. An updating operation takes a virtual-copy array, an index, and a new value and returns a new virtual-copy array with the new

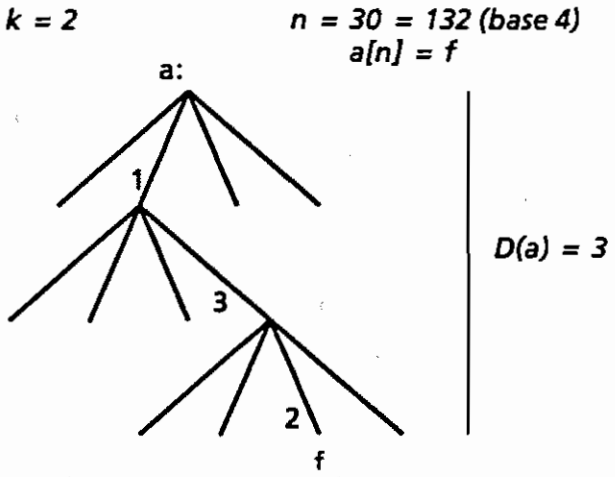

Figure 2.4: Virtual-Copy Array

value stored at the given index. An access operation takes an array and an index, and returns the value at that index.

Basically, virtual-copy arrays are 2^k-ary trees for some fixed $k > 0$. Define the *depth* $d(n)$ of a tree node n to be 0 for the root and $d(p)+1$ if p is the parent of n. Each virtual-copy array a has also a positive *depth* $D(a) \geq \max\{d(n) : n \text{ is a node of } a\}$. A tree node at depth $D(a)$ (necessarily a leaf) can be either an array element or the special marker $\perp$ for unassigned elements. All leaf nodes at depths lower than $D(a)$ are also $\perp$, indicating that no elements have yet been stored in the subarray below the node. With this arrangement, the array can store at most $2^{kD(a)}$ elements, numbered 0 through $2^{kD(a)} - 1$, but unused subarrays need not be allocated.

By numbering the 2^k daughters of a nonleaf node from 0 to 2^k-1, a path from a's root to an array element (a leaf at depth $D(a)$) can be represented by a sequence $n_0 \cdots n_{D(a)-1}$ in which n_d is the number of the branch taken at depth d. This sequence is just the base 2^k representation of the index n of the array element, with n_0 the most significant digit and $n_{D(a)}$ the least significant (Figure 2.4).

When a virtual-copy array a is updated, one of two things may happen. If the index for the updated element exceeds the maximum for the current depth (as in the $a[8] := g$ update in Figure 2.5), a new root node is created

for the updated array and the old array becomes the leftmost daughter of the new root. Other nodes are also created, as appropriate, to reach the position of the new element. If, on the other hand, the index for the update is within the range for the current depth, the path from the root to the element being updated is copied and the old element is replaced in the new tree by the new element (as in the $a[2] := h$ update in Figure 2.5). This description assumes that the element being updated has already been set. If not, the branch to the element may terminate prematurely in a $\perp$ leaf, in which case new nodes are created to the required depth and attached to the appropriate position at the end of the new path from the root.

2.6 Dag Representation

Any dag representation can be implemented with virtual-copy memory instead of random-access memory. If that were done for the original PATR-II copying implementation, a certain measure of structure sharing would be achieved.

The present scheme, however, goes well beyond that by using the method of structure sharing introduced in Section 2.4. As we saw there, an instance object is represented by a molecule, a pair consisting of a skeleton dag (from a rule or lexical entry) and an update environment. We shall now examine the structure of environments.

In a chart parser for PATR-II, dag instances in the chart fall into two classes.

Base instances are those associated with edges that are created directly from lexical entries or rules.

Derived instances occur in edges that result from the combination of a *left* and a *right parent edge* containing the *left* and *right parent instances* of the derived instance. The *left ancestors* of an instance (edge) are its left parent and that parent's ancestors, and similarly for *right ancestors*. I will assume, for ease of exposition, that a derived instance is always a subdag of the unification of its right parent with a subdag of its left parent. This is the case for most common parsing algorithms, although more general schemes are possible [15].

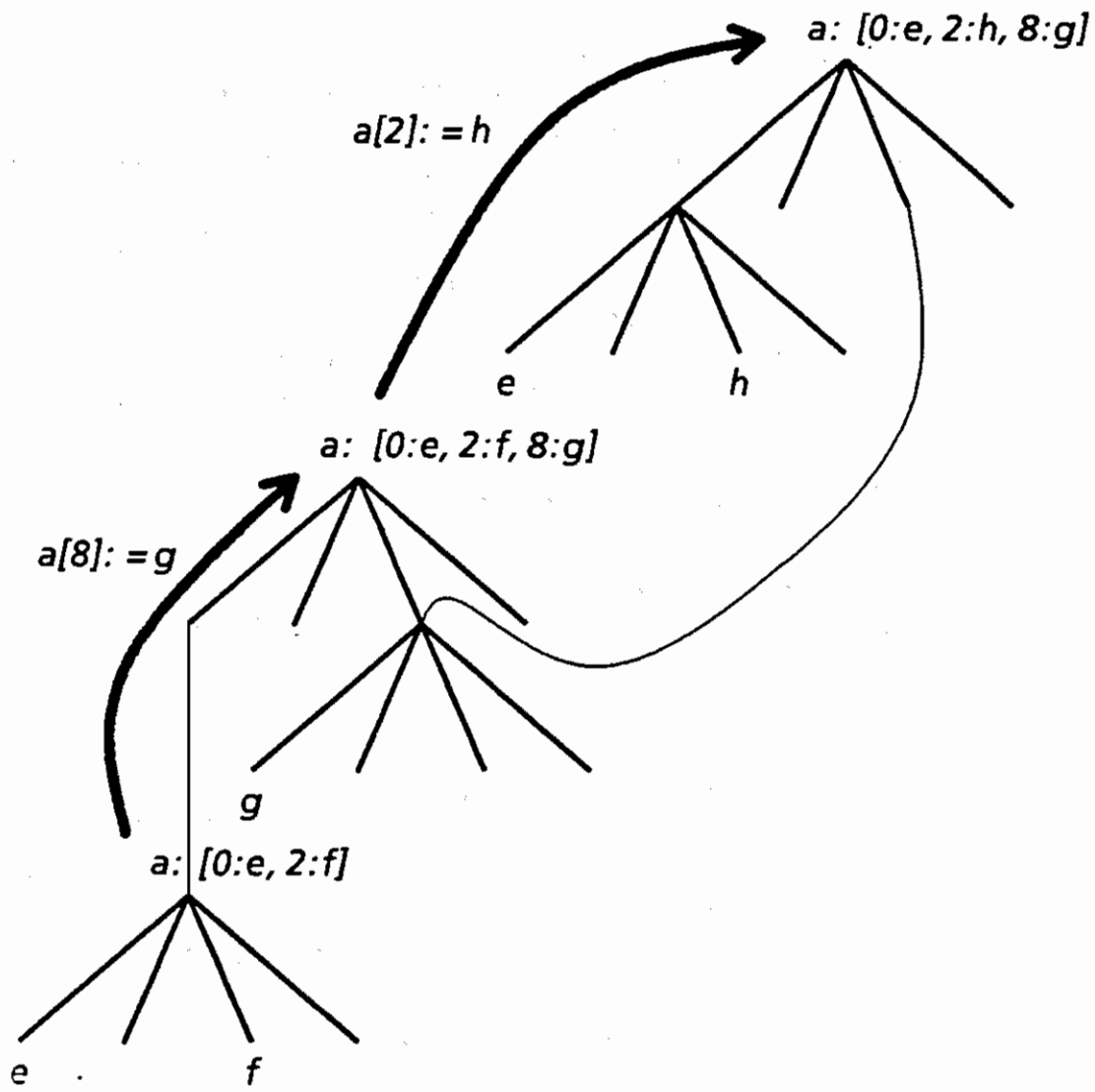

Figure 2.5: Updating Virtual-Copy Arrays

If the original Boyer-Moore scheme were used directly, the environment for a derived instance would consist of pointers to *left* and *right* parent instances, as well as a list of the updates needed to build the current instance from its parents. As noted before, this method requires a worst-case $O(|d|)$ search to find the updates that result in the current instance.

The present scheme relies on the fact that in the great majority of cases no instance is both the left and the right ancestor of another instance. I shall assume for the moment that this is always the case. In Section 2.9 this restriction will be removed.

It is a simple observation about unification that an update of a node of an instance I is either an update of I's skeleton or of the value (a subdag of another instance) of another update of I. If we iterate this reasoning, it becomes clear that every update is ultimately an update of the skeleton of a base instance ancestor of I. Since we assumed above that no instance could occur more than once in I's derivation, we can therefore conclude that I's environment consists only of updates of nodes in the skeletons of its base instance ancestors. By numbering the base instances of a derivation consecutively, we can then represent an environment by an array of *frames*, each containing all the updates of the skeleton of a given base instance.

Actually, the environment of an instance I will be a *branch environment* containing not only those updates directly relevant to I, but also all those that are relevant to the instances of I's particular branch through the parsing search space.

In the context of a given branch environment, it is then possible to represent a molecule by a pair consisting of a skeleton and the index of a frame in the environment. In particular, this representation can be used for all the values (dags) in updates.

More specifically, the frame of a base instance is an array of *update records* indexed by small integers representing the nodes of the instance's skeleton. An update record is either a list of arc bindings for distinct arc labels or a rerouting update. An arc binding is a pair consisting of a label and a molecule (the value of the arc binding). This represents an addition of an arc with that label and that value at the given node. A rerouting update is just a pointer to another molecule; it says that the subdag at

that node in the updated dag is given by that molecule (rather than by whatever was in the initial skeleton).

To see how skeletons and bindings work together to represent a dag, consider the operation of finding the subdag $d/\langle l_1 \cdots l_m \rangle$ of dag d. For this purpose, we use a *current skeleton* s and a current frame f, given initially by the skeleton and frame of the molecule representing d. Now assume that the current skeleton s and current frame f correspond to the subdag $d' = d/\langle l_1 \cdots l_{i-1} \rangle$. To find $d/\langle l_1 \cdots l_i \rangle = d'/l_i$, we use the following method:

1. If the top node of s has been rerouted in f to a dag v, *dereference* d' by setting s and f from v and repeating this step; otherwise
2. If the top node of s has an arc labeled by l_i with value s', the subdag at l_i is given by the moledule (s', f); otherwise
3. If f contains an arc binding labeled l_i for the top node of s, the subdag at l_i is the value of the binding

If none of these steps can be applied, $\langle l_1 \cdots l_i \rangle$ is not a path from the root in d.

The details of the representation are illustrated by the example in Figure 2.6, which shows the passive edges for the chart analysis of the string *ab* according to the sample grammar

$$\begin{array}{lll} S \rightarrow A\ B: & \langle S\ a \rangle & = \langle A \rangle \\ & \langle S\ b \rangle & = \langle B \rangle \\ & \langle S\ a\ x \rangle & = \langle S\ b\ y \rangle \\ A \rightarrow a: & \langle A\ u\ v \rangle & = a \\ B \rightarrow b: & \langle B\ u\ v \rangle & = b \end{array} \tag{2.3}$$

For the sake of simplicity, only the subdags corresponding to the explicit equations in these rules are shown (ie., the *cat* dag arcs and the rule arcs 0, 1,... are omitted). In the figure, the three nonterminal edges (for phrase types S, A and B) are labeled by molecules representing the corresponding dags. The skeleton of each of the three molecules comes from the rule used

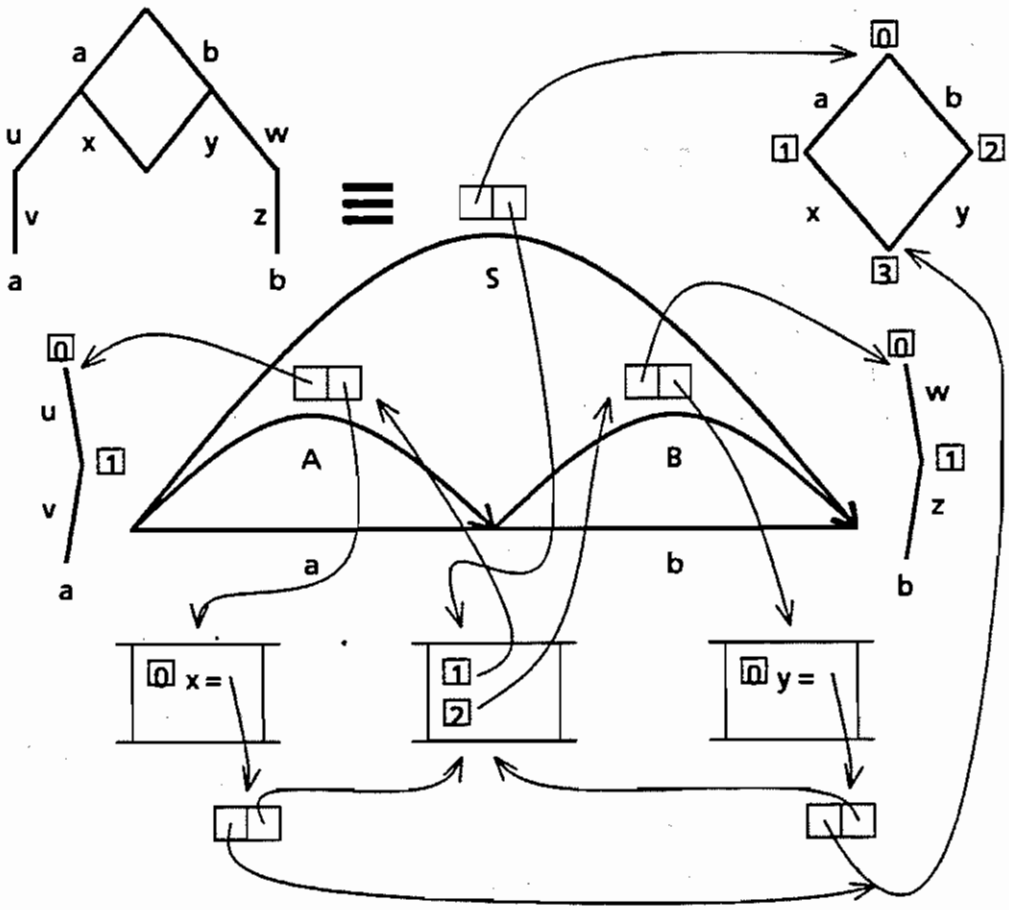

Figure 2.6: Structure-Sharing Chart

to build the nonterminal. Each molecule points (via a frame index not shown in the figure) to a frame in the branch environment. The frames for the A and B edges contain arc bindings for the top nodes of the respective skeletons whereas the frame for the S edge reroute nodes 1 and 2 of the S rule skeleton to the A and B molecules respectively.

2.7 The Unification Algorithm

I shall now give the unification algorithm for two molecules (dags) in the same branch environment.

We can treat a complex dag d as a partial function from labels to dags that maps the label on each arc leaving the top node of the dag to the dag at the end of that arc. This allows us to define the following two operations between dags:

$$\begin{aligned} d_1 \setminus d_2 &= \{(l,d) \in d_1 \mid l \notin \mathrm{dom}(d_2)\} \\ d_1 \lhd d_2 &= \{(l,d) \in d_1 \mid l \in \mathrm{dom}(d_2)\} \end{aligned}$$

It is clear that $\mathrm{dom}(d_1 \lhd d_2) = \mathrm{dom}(d_2 \lhd d_1)$.

We also need the notion of dag dereferencing introduced in the last section. As a side effect of successive unifications, the top node of a dag may be rerouted to another dag whose top node will also end up being rerouted. Dereferencing is the process of following such chains of rerouting pointers to reach a dag that has not been rerouted.

The unification of dags d_1 and d_2 in environment e consists of the following steps:

1. Dereference d_1 and d_2
2. If d_1 and d_2 are identical, the unification is immediately successful
3. If d_1 is a leaf, add to e a rerouting from the top node of d_1 to d_2; otherwise
4. If d_2 is a leaf, add to e a rerouting from the top node of d_2 to d_1; otherwise

5. If d_1 and d_2 are complex dags, for each arc $(l, d) \in d_1 \lhd d_2$ unify the dag d with the dag d' of the corresponding arc $(l, d') \in d_2 \lhd d_1$. Each of those unifications may add new bindings to e. If this unification of subdags is successful, all the arcs in $d_1 \setminus d_2$ are are entered in e as arc bindings for the top node of d_2 and finally the top node of d_1 is rerouted to d_2.

6. If none of the conditions above applies, the unification fails.

To determine whether a dag node is a leaf or complex, both the skeleton and the frame of the corresponding molecule must be examined. For a dereferenced molecule, the set of arcs leaving a node is just the union of the skeleton arcs and the arc bindings for the node. For this to make sense, the skeleton arcs and arc bindings for any molecule node must be disjoint. The interested reader will have no difficulty in proving that this property is preserved by the unification algorithm and therefore all molecules built from skeletons and empty frames by unification will satisfy it.

2.8 Mapping Dags Onto Virtual-Copy Memory

As we saw above, any dag or set of dags constructed by the parser is built from just two kinds of material: (1) frames; (2) pieces of the initial skeletons from rules and lexical entries. The initial skeletons can be represented trivially by host language data structures, as they never change. Frames, though, are always being updated. A new frame is born with the creation of an instance of a rule or lexical entry when the rule or entry is used in some parsing step (uses of the same rule or entry in other steps beget their own frames). A frame is updated when the instance it belongs to participates in a unification.

During parsing, there are in general several possible ways of continuing a derivation. These correspond to alternative ways of updating a branch environment. In abstract terms, on coming to a choice point in the derivation with n possible continuations, $n - 1$ copies of the environment are made, giving n environments — namely, one for each alternative. In fact,

the use of virtual-copy arrays for environments and frames renders this copying unnecessary, so each continuation path performs its own updating of its version of the environment without interfering with the other paths. Thus, all unchanged portions of the environment are shared.

In fact, derivations as such are not explicit in a chart parser. Instead, the instance in each edge has its own branch environment, as described previously. Therefore, when two edges are combined, it is necessary to merge their environments. The cost of this merge operation is at most the same as the worst case cost for unification proper ($O(|d| \log |d|)$). However, in the very common case in which the ranges of frame indices of the two environments do not overlap, the merge cost is only $O(\log |d|)$.

To summarize, we have sharing at two levels: the Boyer-Moore style dag representation allows derived dag instances to share input data structures (skeletons), and the virtual-copy array environment representation allows different branches of the search space to share update records.

2.9 The Renaming Problem

In the foregoing discussion of the structure-sharing method, I assumed that the left and right ancestors of a derived instance were disjoint. In fact, it is easy to show that the condition holds whenever the grammar does not allow empty derived edges.

In contrast, it is possible to construct a grammar in which an empty derived edge with dag D is both a left and a right ancestor of another edge with dag E. Clearly, the two uses of D as an ancestor of E are mutually independent and the corresponding updates have to be segregated. In other words, we need two copies of the instance D. By analogy with theorem proving, I call this the *renaming* problem.

The current solution is to use *real* copying to turn the empty edge into a skeleton, which is then added to the chart. The new skeleton is then used in the normal fashion to produce multiple instances that are free of mutual interference.

2.10 Implementation

The representation described here has been used in a PATR-II parser implemented in Prolog. Two versions of the parser exist — one using an Earley-style algorithm related to Earley deduction [15], the other using a left-corner algorithm.

Preliminary tests of the left-corner algorithm with structure sharing on various grammars and input have shown parsing times as much as 60% faster (never less, in fact, than 40% faster) than those achieved by the same parsing algorithm with structure copying.

Chapter 3

Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms

This chapter was written by Stuart Shieber of the Artificial Intelligence Center, SRI International and the Center for the Study of Language and Information, Stanford University.[1]

[1] This research has been made possible in part by a gift from the Systems Development Foundation, and was also supported by the Defense Advanced Research Projects Agency under Contract N00039-84-K-0078 with the Naval Electronics Systems Command. The views and conclusions contained in this document should not be interpreted as representative of the official policies, either expressed or implied, of the Defense Research Projects Agency or the United States government.

The author is indebted to Fernando Pereira and Ray Perrault for their comments on earlier drafts of this paper.

Abstract

Grammar formalisms based on the encoding of grammatical information in complex-valued feature systems enjoy some currency both in linguistics and natural-language-processing research. Such formalisms can be thought of by analogy to context-free grammars as generalizing the notion of nonterminal symbol from a finite domain of atomic elements to a possibly infinite domain of directed graph structures of a certain sort. Unfortunately, in moving to an infinite nonterminal domain, standard methods of parsing may no longer be applicable to the formalism. Typically, the problem manifests itself as gross inefficiency or even nontermination of the algorithms. In this paper, we discuss a solution to the problem of extending parsing algorithms to formalisms with possibly infinite nonterminal domains, a solution based on a general technique we call restriction. As a particular example of such an extension, we present a complete, correct, terminating extension of Earley's algorithm that uses restriction to perform top-down filtering. Our implementation of this algorithm demonstrates the drastic elimination of chart edges that can be achieved by this technique. Finally, we describe further uses for the technique—including parsing other grammar formalisms, including definite-clause grammars; extending other parsing algorithms, including LR methods and syntactic preference modeling algorithms; and efficient indexing.

3.1 Introduction

Grammar formalisms based on the encoding of grammatical information in complex-valued feature systems enjoy some currency both in linguistics and natural-language-processing research. Such formalisms can be thought of by analogy to context-free grammars as generalizing the notion of nonterminal symbol from a finite domain of atomic elements to a possibly infinite domain of directed graph structures of a certain sort. Many of the surface-based grammatical formalisms explicitly defined or presupposed in linguistics can be characterized in this way—e.g., lexical-functional grammar (LFG) [6], generalized phrase structure grammar (GPSG) [5], even categorial systems such as Montague grammar [11] and Ades/Steedman

grammar [1]—as can several of the grammar formalisms being used in natural-language processing research—e.g., definite clause grammar (DCG) [12], and PATR-II [17].

Unfortunately, in moving to an infinite nonterminal domain, standard methods of parsing may no longer be applicable to the formalism. For instance, the application of techniques for preprocessing of grammars in order to gain efficiency may fail to terminate, as in left-corner and LR algorithms. Algorithms performing top-down prediction (e.g. top-down backtrack parsing, Earley's algorithm) may not terminate at parse time. Implementing backtracking regimens—useful for instance for generating parses in some particular order, say, in order of syntactic preference—is in general difficult when LR-style and top-down backtrack techniques are eliminated.

In this paper, we discuss a solution to the problem of extending parsing algorithms to formalisms with possibly infinite nonterminal domains, a solution based on an operation we call restriction. In Section 3.2, we summarize traditional proposals for solutions and problems inherent in them and propose an alternative approach to a solution using restriction. In Section 3.3, we present some technical background including a brief description of the PATR-II formalism—which is used as the formalism interpreted by the parsing algorithms—and a formal definition of restriction for PATR-II's nonterminal domain. In Section 3.4, we develop a correct, complete and terminating extension of Earley's algorithm for the PATR-II formalism using the restriction notion. Readers uninterested in the technical details of the extensions may want to skip these latter two sections, referring instead to Section 3.4.1 for an informal overview of the algorithms. Finally, in Section 3.5, we discuss applications of the particular algorithm and the restriction technique in general.

3.2 Traditional Solutions and an Alternative Approach

Problems with efficiently parsing formalisms based on potentially infinite nonterminal domains have manifested themselves in many different ways.

Traditional solutions have involved limiting in some way the class of grammars that can be parsed.

3.2.1 Limiting the Formalism

The limitations can be applied to the formalism by, for instance, adding a context-free "backbone." If we require that a context-free subgrammar be implicit in every grammar, the subgrammar can be used for parsing and the rest of the grammar used as a filter during or after parsing. This solution has been recommended for functional unification grammars (FUG) by Martin Kay [7]; its legacy can be seen in the context-free skeleton of LFG, and the Hewlett-Packard GPSG system [4], and in the *cat* feature requirement in PATR-II that is described below.

However, several problems inhere in this solution of mandating a context-free backbone. First, the move from context-free to complex-feature-based formalisms was motivated by the desire to structure the notion of nonterminal. Many analyses take advantage of this by eliminating mention of major category information from particular rules[2] or by structuring the major category itself (say into binary N and V features plus a bar level-feature as in $\overline{X}$-based theories). Forcing the primacy and atomicity of major category defeats part of the purpose of structured category systems.

Second, and perhaps more critically, because only certain of the information in a rule is used to guide the parse, say major category information, only such information can be used to filter spurious hypotheses by top-down filtering. Note that this problem occurs even if filtering by the rule information is used to eliminate at the earliest possible time constituents and partial constituents proposed during parsing (as is the case in the PATR-II implementation and the Earley algorithm given below; cf. the Xerox LFG system). Thus, if information about subcategorization is left out of the category information in the context-free skeleton, it cannot be used to eliminate prediction edges. For example, if we find a verb that subcategorizes for a noun phrase, but the grammar rules allow postverbal NPs, PPs,

[2] See, for instance, the coordination and copular "be" analyses from GPSG [5], the nested VP analysis used in some PATR-II grammars [19], or almost all categorial analyses, in which general rules of combination play the role of specific phrase-structure rules.

$\bar{S}$s, VPs, and so forth, the parser will have no way to eliminate the building of edges corresponding to these categories. Only when such edges attempt to join with the V will the inconsistency be found. Similarly, if information about filler-gap dependencies is kept extrinsic to the category information, as in a slash category in GPSG or an LFG annotation concerning a matching constituent for a $\Uparrow$ specification, there will be no way to keep from hypothesizing gaps at any given vertex. This "gap-proliferation" problem has plagued many attempts at building parsers for grammar formalisms in this style.

In fact, by making these stringent requirements on what information is used to guide parsing, we have to a certain extent thrown the baby out with the bathwater. These formalisms were intended to free us from the tyranny of atomic nonterminal symbols, but for good performance, we are forced toward analyses putting more and more information in an atomic category feature. An example of this phenomenon can be seen in the author's paper on LR syntactic preference parsing [18]. Because the LALR table building algorithm does not in general terminate for complex-feature-based grammar formalisms, the grammar used in that paper was a simple context-free grammar with subcategorization and gap information placed in the atomic nonterminal symbol.

3.2.2 Limiting Grammars and Parsers

On the other hand, the grammar formalism can be left unchanged, but particular grammars developed that happen not to succumb to the problems inherent in the general parsing problem for the formalism. The solution mentioned above of placing more information in the category symbol falls into this class. Unpublished work by Kent Wittenburg and by Robin Cooper has attempted to solve the gap proliferation problem using special grammars.

In building a general tool for grammar testing and debugging, however, we would like to commit as little as possible to a particular grammar or style of grammar.[3] Furthermore, the grammar designer should not be held down

[3] See [16] for further discussion of this matter.

in building an analysis by limitations of the algorithms. Thus a solution requiring careful crafting of grammars is inadequate.

Finally, specialized parsing algorithms can be designed that make use of information about the particular grammar being parsed to eliminate spurious edges or hypotheses. Rather than using a general parsing algorithm on a limited formalism, Ford, Bresnan, and Kaplan [3] chose a specialized algorithm working on grammars in the full LFG formalism to model syntactic preferences. Current work at Hewlett-Packard on parsing recent variants of GPSG seems to take this line as well.

Again, we feel that the separation of burden is inappropriate in such an attack, especially in a grammar-development context. Coupling the grammar design and parser design problems in this way leads to the linguistic and technological problems becoming inherently mixed, magnifying the difficulty of writing an adequate grammar/parser system.

3.2.3 An Alternative: Using Restriction

Instead, we would like a parsing algorithm that placed no restraints on the grammars it could handle as long as they could be expressed within the intended formalism. Still, the algorithm should take advantage of that part of the arbitrarily large amount of information in the complex-feature structures that is *significant* for guiding parsing with the particular grammar. One of the aforementioned solutions is to require the grammar writer to put all such significant information in a special atomic symbol—i.e., mandate a context-free backbone. Another is to use *all* of the feature structure information—but this method, as we shall see, inevitably leads to nonterminating algorithms.

A compromise is to parameterize the parsing algorithm by a small amount of grammar-dependent information that tells the algorithm *which* of the information in the feature structures is significant for guiding the parse. That is, the parameter determines how to split up the infinite nonterminal domain into a finite set of equivalence classes that can be used for parsing. By doing so, we have an optimal compromise: Whatever part of the feature structure is significant we distinguish in the equivalence classes

by setting the parameter appropriately, so the information is used in parsing. But because there are only a finite number of equivalence classes, parsing algorithms guided in this way will terminate.

The technique we use to form equivalence classes is *restriction*, which involves taking a quotient of the domain with respect to a *restrictor*. The restrictor thus serves as the sole repository of grammar-dependent information in the algorithm. By tuning the restrictor, the set of equivalence classes engendered can be changed, making the algorithm more or less efficient at guiding the parse. But independent of the restrictor, the algorithm will be correct, since it is still doing parsing over a finite domain of "nonterminals," namely, the elements of the restricted domain.

This idea can be applied to solve many of the problems engendered by infinite nonterminal domains, allowing preprocessing of grammars as required by LR and LC algorithms, allowing top-down filtering or prediction as in Earley and top-down backtrack parsing, guaranteeing termination, etc.

3.3 Technical Preliminaries

Before discussing the use of restriction in parsing algorithms, we present some technical details, including a brief introduction to the PATR-II grammar formalism, which will serve as the grammatical formalism that the presented algorithms will interpret. PATR-II is a simple grammar formalism that can serve as the least common denominator of many of the complex-feature-based and unification-based formalisms prevalent in linguistics and computational linguistics. As such it provides a good testbed for describing algorithms for complex-feature-based formalisms.

3.3.1 The PATR-II Nonterminal Domain

The PATR-II nonterminal domain is a lattice of directed, acyclic, graph structures (dags).[4] Dags can be thought of as similar to the reentrant f-structures of LFG or functional structures of FUG, and we will use the bracketed notation associated with these formalisms for them. For example, the following is a dag (D_0) in this notation, with reentrancy indicated with coindexing boxes:

$$\left[\begin{array}{ll} a: & \left[\, b: \;\; c \,\right] \\ d: & \left[\begin{array}{ll} e: & \boxed{1}\left[\, f: \;\left[\, g: \;\; h \,\right]\,\right] \\ i: & \left[\, j: \;\boxed{1}\,\right] \\ k: & l \end{array}\right] \end{array}\right]$$

Dags come in two varieties, *complex* (like the one above) and *atomic* (like the dags h and c in the example). Complex dags can be viewed as partial functions from labels to dag values, and the notation $D(l)$ will therefore denote the value associated with the label l in the dag D. In the same spirit, we can refer to the domain of a dag ($dom(D)$). A dag with an empty domain is often called an *empty* dag or *variable*. A *path* in a dag is a sequence of label names (notated, e.g., $(d\ e\ f)$), which can be used to pick out a particular subpart of the dag by repeated application (in this case, the dag $[g : h]$). We will extend the notation $D(p)$ in the obvious way to include the subdag of D picked out by a path p. We will also occasionally use the square brackets as the dag constructor function, so that $[f : D]$ where D is an expression denoting a dag will denote the dag whose f feature has value D.

3.3.2 Subsumption and Unification

There is a natural lattice structure for dags based on *subsumption*—an ordering on dags that roughly corresponds to the compatibility and relative

[4] The reader is referred to earlier works [19,13] for more detailed discussions of dag structures.

specificity of information contained in the dags. Intuitively viewed, a dag D subsumes a dag D' (notated $D \sqsubseteq D'$) if D contains a subset of the information in (i.e., is more general than) D'.

Thus variables subsume all other dags, atomic or complex, because as the trivial case, they contain no information at all. A complex dag D subsumes a complex dag D' if and only if $D(l) \sqsubseteq D'(l)$ for all $l \in dom(D)$ and $D'(P) = D'(q)$ for all paths p and q such that $D(p) = D(q)$. An atomic dag neither subsumes nor is subsumed by any different atomic dag.

For instance, the following subsumption relations hold:

$$[\ ] \sqsubseteq [d : e] \sqsubseteq \begin{bmatrix} a: & [b:c] \\ e: & f \end{bmatrix} \sqsubseteq \begin{bmatrix} a: & \boxed{1}[b:c] \\ d: & \boxed{1} \\ e: & f \end{bmatrix} \quad .$$

Finally, given two dags D' and D'', the *unification* of the dags is the most general dag D such that $D' \sqsubseteq D$ and $D'' \sqsubseteq D$. We notate this $D = D' \sqcup D''$.

The following examples illustrate the notion of unification:

$$\begin{bmatrix} a: & [b:c] \end{bmatrix} \sqcup \begin{bmatrix} d: & e \end{bmatrix} = \begin{bmatrix} a: & [b:c] \\ d: & e \end{bmatrix}$$

$$\begin{bmatrix} a: & [b:c] \end{bmatrix} \sqcup \begin{bmatrix} a: & \boxed{1}[\,] \\ d: & \boxed{1} \end{bmatrix} = \begin{bmatrix} a: & \boxed{1}[b:c] \\ d: & \boxed{1} \end{bmatrix} \quad .$$

The unification of two dags is not always well-defined. In the cases where no unification exists, the unification is said to *fail*. For example the following pair of dags fail to unify with each other:

$$\begin{bmatrix} a: & \boxed{1}[\,] \\ d: & \boxed{1} \end{bmatrix} \sqcup \begin{bmatrix} a: & [b:c] \\ d: & [b:d] \end{bmatrix} = \mathit{fail} \quad .$$

3.3.3 Restriction in the PATR-II Nonterminal Domain

Now, consider the notion of *restriction* of a dag, using the term almost in its technical sense of restricting the domain of a function. By viewing dags as partial functions from labels to dag values, we can envision a process of restricting the domain of this function to a given set of labels. Extending this process recursively to every level of the dag, we have the concept of restriction used below. Given a finite specification Φ (called a restrictor) of what the allowable domain at each node of a dag is, we can define a functional, $\restriction$, that yields the dag restricted by the given restrictor.

Formally, we define restriction as follows. Given a relation Φ between paths and labels, and a dag D, we define $D\restriction\Phi$ to be the most specific dag $D' \sqsubseteq D$ such that for every path p either $D'(p)$ is undefined, or $D'(p)$ is atomic, or for every $l \in dom(D'(p))$, $p\Phi l$. That is, every path in the restricted dag is either undefined, atomic, or ***specifically allowed*** by the restrictor.

The restriction process can be viewed as putting dags into equivalence classes, each equivalence class being the largest set of dags that all are restricted to the same dag (which we will call its *canonical member*). It follows from the definition that in general $D\restriction\Phi \sqsubseteq D$. Finally, if we disallow infinite relations as restrictors (i.e., restrictors must not allow values for an infinite number of distinct paths) as we will do for the remainder of the discussion, we are guaranteed to have only a finite number of equivalence classes.

Actually, in the sequel we will use a particularly simple subclass of restrictors that are generable from sets of paths. Given a set of paths s, we can define Φ such that $p\Phi l$ if and only if p is a prefix of some $p' \in s$. Such restrictors can be understood as "throwing away" all values not lying on one of the given paths. This subclass of restrictors is sufficient for most applications. However, the algorithms that we will present apply to the general class as well.

Using our previous example, consider a restrictor Φ_0 generated from the set of paths $\{\langle a\ b\rangle, \langle d\ e\ f\rangle, \langle d\ i\ j\ f\rangle\}$. That is, $p\Phi_0 l$ for all p in the listed paths and all their prefixes. Then given the previous dag D_0, $D_0\restriction\Phi_0$ is

$$\left[\begin{array}{ll} a: & \left[\begin{array}{ll} b: & c \end{array}\right] \\ d: & \left[\begin{array}{ll} e: & \boxed{1}\left[\begin{array}{ll} f: & [] \end{array}\right] \\ i: & \left[\begin{array}{ll} j: & \boxed{1} \end{array}\right] \end{array}\right] \end{array}\right] .$$

Restriction has thrown away all the information except the direct values of $\langle a\ b\rangle$, $\langle d\ e\ f\rangle$, and $\langle d\ i\ j\ f\rangle$. (Note however that because the values for paths such as $\langle d\ e\ f\ g\rangle$ were thrown away, $(D_0 \restriction \Phi_0)(\langle d\ e\ f\rangle)$ is a variable.)

3.3.4 PATR-II Grammar Rules

PATR-II rules describe how to combine a sequence of constituents, $X_1, \ldots, X_n$ to form a constituent X_0, stating mutual constraints on the dags associated with the $n+1$ constituents as unifications of various parts of the dags. For instance, we might have the following rule:

$X_0 \rightarrow X_1 X_2$:
$\langle X_0\ cat\rangle = S$
$\langle X_1\ cat\rangle = NP$
$\langle X_2\ cat\rangle = VP$
$\langle X_1\ agreement\rangle = \langle X_2\ agreement\rangle$.

By notational convention, we can eliminate unifications for the special feature *cat* (the atomic major category feature) recording this information implicitly by using it in the "name" of the constituent, e.g.,

$S \rightarrow NP\ VP$:
$\langle NP\ agreement\rangle = \langle VP\ agreement\rangle$.

If we *require* that this notational convention always be used (in so doing, guaranteeing that each constituent have an atomic major category associated with it), we have thereby mandated a context-free backbone to the grammar, and can then use standard context-free parsing algorithms to parse sentences relative to grammars in this formalism. Limiting to a context-free-based PATR-II is the solution that previous implementations have incorporated.

Before proceeding to describe parsing such a context-free-based PATR-II, we make one more purely notational change. Rather than associating with each grammar rule a set of unifications, we instead associate a dag that incorporates all of those unifications implicitly, i.e., a rule is associated with a dag D_r such that for all unifications of the form $p = q$ in the rule, $D_r(p) = D_r(q)$. Similarly, unifications of the form $p = a$ where a is atomic would require that $D_r(p) = a$. For the rule mentioned above, such a dag would be

$$\left[\begin{array}{lll} X_0: & \left[\begin{array}{ll} cat: & S \end{array}\right] & \\ X_1: & \left[\begin{array}{ll} cat: & NP \\ agreement: & \boxed{1}[] \end{array}\right] & \\ X_2: & \left[\begin{array}{ll} cat: & VP \\ agreement: & \boxed{1} \end{array}\right] & \end{array}\right]$$

Thus a rule can be thought of as an ordered pair $\langle P, D \rangle$ where P is a production of the form $X_0 \rightarrow X_1 \cdots X_n$ and D is a dag with top-level features $X_0, \ldots, X_n$ and with atomic values for the *cat* feature of each of the top-level subdags. The two notational conventions—using sets of unifications instead of dags, and putting the *cat* feature information implicitly in the names of the constituents—allow us to write rules in the more compact and familiar format above, rather than this final cumbersome way presupposed by the algorithm.

3.4 Using Restriction to Extend Earley's Algorithm for PATR-II

We now develop a concrete example of the use of restriction in parsing by extending Earley's algorithm to parse grammars in the PATR-II formalism just presented.

3.4.1 An Overview of the Algorithms

Earley's algorithm is a bottom-up parsing algorithm that uses top-down prediction to hypothesize the starting points of possible constituents. Typically, the prediction step determines which *categories* of constituent can start at a given point in a sentence. But when most of the information is not in an atomic category symbol, such prediction is relatively useless and many types of constituents are predicted that could never be involved in a completed parse. This standard Earley's algorithm is presented in Section 3.4.2.

By extending the algorithm so that the prediction step determines which *dags* can start at a given point, we can use the information in the features to be more precise in the predictions and eliminate many hypotheses. However, because there are a potentially infinite number of such feature structures, the prediction step may never terminate. This extended Earley's algorithm is presented in Section 3.4.3.

We compromise by having the prediction step determine which *restricted dags* can start at a given point. If the restrictor is chosen appropriately, this can be as constraining as predicting on the basis of the whole feature structure, yet prediction is guaranteed to terminate because the domain of restricted feature structures is finite. This final extension of Earley's algorithm is presented in Section 3.4.4.

3.4.2 Parsing a Context-Free-Based PATR-II

We start with the Earley algorithm for context-free-based PATR-II on which the other algorithms are based. The algorithm is described in a

chart-parsing incarnation, vertices numbered from 0 to n for an n-word sentence $w_1 \cdots w_n$. An item of the form $[h, i, A \rightarrow \alpha.\beta, D]$ designates an edge in the chart from vertex h to i with dotted rule $A \rightarrow \alpha.\beta$ and dag D.

The chart is initialized with an edge $[0, 0, X_0 \rightarrow .\alpha, D]$ for each rule $\langle X_0 \rightarrow \alpha, D\rangle$ where $D(\langle X_0\ cat\rangle) = S$.

For each vertex i do the following steps until no more items can be added:

Predictor step: For each item ending at i of the form $[h, i, X_0 \rightarrow \alpha.X_j\beta, D]$ and each rule of the form $\langle X_0 \rightarrow \gamma, E\rangle$ such that $E(\langle X_0\ cat\rangle) = D(\langle X_j\ cat\rangle)$, add an edge of the form $[i, i, X_0 \rightarrow .\gamma, E]$ if this edge is not subsumed by another edge.

Informally, this involves *predicting top-down all rules whose left-hand-side category matches the category of some constituent being looked for.*

Completer step: For each item of the form $[h, i, X_0 \rightarrow \alpha., D]$ and each item of the form $[g, h, X_0 \rightarrow \beta.X_j\gamma, E]$ add the item $[g, i, X_0 \rightarrow \beta X_j.\gamma, E \sqcup [X_j : D(X_0)]]$ if the unification succeeds[5] and this edge is not subsumed by another edge.[6]

Informally, this involves *forming a new partial phrase whenever the category of a constituent needed by one partial phrase matches the category of a completed phrase and the dag associated with the completed phrase can be unified in appropriately.*

Scanner step: If $i \neq 0$ and $w_i = a$, then for all items $[h, i-1, X_0 \rightarrow \alpha.a\beta, D]$ add the item $[h, i, X_0 \rightarrow \alpha a.\beta, D]$.

Informally, this involves *allowing lexical items to be inserted into partial phrases.*

---

[5]Note that this unification will fail if $D(\langle X_0\ cat\rangle) \neq E(\langle X_j\ cat\rangle)$ and no edge will be added, i.e., if the subphrase is not of the appropriate category for insertion into the phrase being built.

[6]One edge subsumes another edge if and only if the first three elements of the edges are identical and the fourth element of the first edge subsumes that of the second edge.

Notice that the Predictor Step in particular assumes the availability of the *cat* feature for top-down prediction. Consequently, this algorithm applies only to PATR-II with a context-free base.

3.4.3 Removing the Context-Free Base: An Inadequate Extension

A first attempt at extending the algorithm to make use of more than just a single atomic-valued *cat* feature (or less if no such feature is mandated) is to change the Predictor Step so that instead of checking the predicted rule for a left-hand side that matches its *cat* feature with the predicting subphrase, we require that the whole left-hand-side subdag unifies with the subphrase being predicted from. Formally, we have

Predictor step: For each item ending at i of the form $[h, i, X_0 \rightarrow \alpha.X_j\beta, D]$ and each rule of the form $\langle X_0 \rightarrow \gamma, E\rangle$, add an edge of the form $[i, i, X_0 \rightarrow .\gamma, E \sqcup [X_0 : D(X_j)]]$ if the unification succeeds and this edge is not subsumed by another edge.

This step *predicts top-down all rules whose left-hand side matches the* dag *of some constituent being looked for.*

Completer step: As before.

Scanner step: As before.

However, this extension does not preserve termination. Consider a "counting" grammar that records in the dag the number of terminals in the string.[7]

$S \rightarrow T$:
$\quad \langle Sf\rangle = a.$
$T_1 \rightarrow T_2\ A$:
$\quad \langle T_1 f\rangle = \langle T_2 f f\rangle.$

[7]Similar problems occur in natural language grammars when keeping *lists* of, say, subcategorized constituents or gaps to be found.

$S \rightarrow A.$
$A \rightarrow a.$

Initially, the $S \rightarrow T$ rule will yield the edge

$$[0, 0, X_0 \rightarrow .X_1, \begin{bmatrix} X_0: & [\ cat: & S\ ] \\ X_1: & \begin{bmatrix} cat: & T \\ f: & a \end{bmatrix} \end{bmatrix}]$$

which in turn causes the Prediction step to give

$$[0, 0, X_0 \rightarrow .X_1 X_2, \begin{bmatrix} X_0: & \begin{bmatrix} cat: & T \\ f: & \boxed{1}a \end{bmatrix} \\ X_1: & \begin{bmatrix} cat: & T \\ f: & [\ f: & \boxed{1}\ ] \end{bmatrix} \\ X_2: & [\ cat: & A\ ] \end{bmatrix}]$$

yielding in turn

$$.[0, 0, X_0 \rightarrow .X_1 X_2, \begin{bmatrix} X_0: & \begin{bmatrix} cat: & T \\ f: & \boxed{1}[\ f: & a\ ] \end{bmatrix} \\ X_1: & \begin{bmatrix} cat: & T \\ f: & [\ f: & [\ f: & \boxed{1}\ ]\ ] \end{bmatrix} \\ X_2: & [\ cat: & A\ ] \end{bmatrix}]$$

and so forth ad infinitum.

3.4.4 Removing the Context-Free Base: An Adequate Extension

What is needed is a way of "forgetting" some of the structure we are using for top-down prediction. But this is just what restriction gives us, since a restricted dag always subsumes the original, i.e., it has strictly less information. Taking advantage of this property, we can change the Prediction Step to restrict the top-down information before unifying it into the rule's dag.

Predictor step: For each item ending at i of the form $[h, i, X_0 \rightarrow \alpha.X_j\beta, D]$ and each rule of the form $\langle X_0 \rightarrow \gamma, E\rangle$, add an edge of the form $[i, i, X_0 \rightarrow .\gamma, E \sqcup (D(X_j)\restriction\Phi)]$ if the unification succeeds and this edge is not subsumed by another edge.

This step *predicts top-down all rules whose left-hand side matches the* restricted dag *of some constituent being looked for.*

Completer step: As before.

Scanner step: As before.

This algorithm on the previous grammar, using a restrictor that allows through only the *cat* feature of a dag, operates as before, but predicts the first time around the more general edge:

$$[0, 0, X_0 \rightarrow .X_1X_2, \begin{bmatrix} X_0: & \begin{bmatrix} cat: & T \\ f: & \boxed{1}[] \end{bmatrix} \\ X_1: & \begin{bmatrix} cat: & T \\ f: & \begin{bmatrix} f: & \boxed{1} \end{bmatrix} \end{bmatrix} \\ X_2: & \begin{bmatrix} cat: & A \end{bmatrix} \end{bmatrix}] \ .$$

Another round of prediction yields *this same edge* so the process terminates immediately. duck Because the predicted edge is more general than (i.e., subsumes) all the infinite number of edges it replaced that were predicted under the nonterminating extension, it preserves *completeness.* On

the other hand, because the predicted edge is not more general than the rule itself, it permits no constituents that violate the constraints of the rule; therefore, it preserves *correctness*. Finally, because restriction has a finite range, the prediction step can only occur a finite number of times before building an edge identical to one already built; therefore, it preserves *termination*.

3.5 Applications

3.5.1 Some Examples of the Use of the Algorithm

The algorithm just described has been implemented and incorporated into the PATR-II Experimental System at SRI International, a grammar development and testing environment for PATR-II grammars written in Zetalisp for the Symbolics 3600.

The following table gives some data suggestive of the effect of the restrictor on parsing efficiency. It shows the total number of active and passive edges added to the chart for five sentences of up to eleven words using four different restrictors. The first allowed only category information to be used in prediction, thus generating the same behavior as the unextended Earley's algorithm. The second added subcategorization information in addition to the category. The third added filler-gap dependency information as well so that the gap proliferation problem was removed. The final restrictor added verb form information. The last column shows the percentage of edges that were eliminated by using this final restrictor.

	Prediction				%
Sentence	*cat*	+ *subcat*	+ *gap*	+ *form*	elim.
1	33	33	20	16	52
2	85	50	29	21	75
3	219	124	72	45	79
4	319	319	98	71	78
5	812	516	157	100	88

Several facts should be kept in mind about the data above. First, for sentences with no Wh-movement or relative clauses, no gaps were ever predicted. In other words, the top-down filtering is in some sense maximal with respect to gap hypothesis. Second, the subcategorization information used in top-down filtering removed all hypotheses of constituents except for those directly subcategorized for. Finally, the grammar used contained constructs that would cause nontermination in the unrestricted extension of Earley's algorithm.

3.5.2 Other Applications of Restriction

This technique of restriction of complex-feature structures into a finite set of equivalence classes can be used for a wide variety of purposes.

First, parsing algorithms such as the above can be modified for use by grammar formalisms other than PATR-II. In particular, definite-clause grammars are amenable to this technique, and it can be used to extend the Earley deduction of Pereira and Warren [15]. Pereira has used a similar technique to improve the efficiency of the BUP (bottom-up left-corner) parser [10] for DCG. LFG and GPSG parsers can make use of the top-down filtering device as well. FUG parsers might be built that do not require a context-free backbone.

Second, restriction can be used to enhance other parsing algorithms. For example, the ancillary function to compute LR closure—which, like the Earley algorithm, either does not use feature information, or fails to terminate—can be modified in the same way as the Earley predictor step to terminate while still using significant feature information. LR parsing techniques can thereby be used for efficient parsing of complex-feature-based formalisms. More speculatively, schemes for scheduling LR parsers to yield parses in preference order might be modified for complex-feature-based formalisms, and even tuned by means of the restrictor.

Finally, restriction can be used in areas of parsing other than top-down prediction and filtering. For instance, in many parsing schemes, edges are indexed by a category symbol for efficient retrieval. In the case of Earley's algorithm, active edges can be indexed by the category of the constituent following the dot in the dotted rule. However, this again forces

the primacy and atomicity of major category information. Once again, restriction can be used to solve the problem. Indexing by the restriction of the dag associated with the need permits efficient retrieval that can be tuned to the particular grammar, yet does not affect the completeness or correctness of the algorithm. The indexing can be done by discrimination nets, or specialized hashing functions akin to the partial-match retrieval techniques designed for use in Prolog implementations [22].

3.6 Conclusion

We have presented a general technique of restriction with many applications in the area of manipulating complex-feature-based grammar formalisms. As a particular example, we presented a complete, correct, terminating extension of Earley's algorithm that uses restriction to perform top-down filtering. Our implementation demonstrates the drastic elimination of chart edges that can be achieved by this technique. Finally, we described further uses for the technique—including parsing other grammar formalisms, including definite-clause grammars; extending other parsing algorithms, including LR methods and syntactic preference modeling algorithms; and efficient indexing.

We feel that the restriction technique has great potential to make increasingly powerful grammar formalisms computationally feasible.

Bibliography

[1] Ades, A. E. and M. J. Steedman. On the order of words. *Linguistics and Philosophy*, 4(4):517–558, 1982.

[2] Boyer, R. S. and J. S. Moore. The sharing of structure in theorem-proving programs. In *Machine Intelligence 7*, pp. 101–116, John Wiley and Sons, New York, New York, 1972.

[3] Ford, M., J. Bresnan, and R. Kaplan. A competence-based theory of syntactic closure. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, Massachusetts, 1982.

[4] Gawron, J. M., J. King, J. Lamping, E. Loebner, E. A. Paulson, G. K. Pullum, I. A. Sag, and T. Wasow. Processing English with a generalized phrase structure grammar. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, pp. 74–81, University of Toronto, Toronto, Ontario, Canada, 16-18 June 1982.

[5] Gazdar, G., E. Klein, G. K. Pullum, and I. A. Sag. *Generalized Phrase Structure Grammar.* Blackwell Publishing, Oxford, England, and Harvard University Press, Cambridge, Massachusetts, 1985.

[6] Kaplan, R. and J. Bresnan. Lexical-functional grammar: a formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, Massachusetts, 1983.

[7] Kay, M. An algorithm for compiling parsing tables from a grammar. 1980. Xerox Palo Alto Research Center, Palo Alto, California.

[8] Kay, M. *Algorithm Schemata and Data Structures in Syntactic Processing.* Technical Report, Xerox Palo Alto Research Center, Palo Alto, California, 1980. A version will appear in the proceedings of the Nobel Symposium on Text Processing, Gothenburg, 1980.

[9] Kay, M. *Unification Grammar.* Technical Report, Xerox Palo Alto Research Center, Palo Alto, California, 1983.

[10] Matsumoto, Y., H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing,* 1:145–158, 1983.

[11] Montague, R. The proper treatment of quantification in ordinary English. In R. H. Thomason, editor, *Formal Philosophy,* pp. 188–221, Yale University Press, New Haven, Connecticut, 1974.

[12] Pereira, F. C. N. Logic for natural language analysis. Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[13] Pereira, F. C. N. and S. M. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the Tenth International Conference on Computational Linguistics,* Stanford University, Stanford, California, 2-7 July 1984.

[14] Pereira, F. C. N. and D. H. D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence,* 13:231–278, 1980.

[15] Pereira, F. C. N. and D. H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* pp. 137–144, Massachusetts Institute of Technology, Cambridge, Massachusetts, 15-17 June 1983.

[16] Shieber, S. M. Criteria for designing computer facilities for linguistic analysis. To appear in *Linguistics.*

[17] Shieber, S. M. The design of a computer language for linguistic information. In *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California, 2-7 July 1984.

[18] Shieber, S. M. Sentence disambiguation by a shift-reduce parsing technique. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pp. 113–118, Massachusetts Institute of Technology, Cambridge, Massachusetts, 15-17 June 1983.

[19] Shieber, S. M., H. Uszkoreit, F. C. N. Pereira, J. J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, SRI International, Menlo Park, California, 1983.

[20] Warren, D. H. D. *Applied Logic – its use and implementation as programming tool.* Ph.D. thesis, University of Edinburgh, Scotland, 1977. Reprinted as Technical Note 290, Artificial Intelligence Center, SRI, International, Menlo Park, California.

[21] Warren, D. H. D. Logarithmic access arrays for Prolog. Unpublished program, 1983.

[22] Wise, M. J. and D. M. W. Powers. Indexing Prolog clauses via superimposed code words and field encoded words. In *Proceedings of the 1984 International Symposium on Logic Programming*, pp. 203–210, IEEE Computer Society Press, Atlantic City, New Jersey, 6-9 February 1984.