CONJUNCTION IN
META-RESTRICTION GRAMMAR

LYNETTE HIRSCHMAN

This paper describes Meta-Restriction Grammar for parsing coordinate conjunction in English. Meta-Restriction Grammar consists of Restriction Grammar, a logic grammar implementation of Sager's String Grammar, plus a metagrammatical component that automatically rewrites "base" grammar rules into more complex rules to handle coordinate conjunction. The approach resembles Sedogbo's approach of "empty elements" or "holes." This avoids the combinatorial explosion due to backtracking in the treatment of Woods, Sager, and Dahl and McCord. Restriction Grammar is well suited to metagrammar extensions, because the absence of parameters in grammar rules facilitates the statement of metarules. The metagrammatical component generates grammar rules specifying allowable conjoinings at limited types of nodes, to reduce redundancy. Meta-Restriction Grammar represents both the surface structure and a regularized structure (via pointers to elided elements) for efficient computation of selectional restrictions. This approach is sufficiently powerful to handle a number of complex phenomena, such as conjunction with comma (as distinguished from the appositive construction), paired conjunctions such as both...and, either...or, and scoping of left noun modifiers under conjunction. One of the great attractions of the metagrammar approach is that the grammar can be translated and compiled, resulting in an efficient treatment of conjunction (parse times of 1 to 3 seconds per parse). This contrasts with the interrupt-driven approach, where an interpreter generates rules for conjoining structures on demand, making it impossible to compile the complete grammar.

Address correspondence to Lynette Hirschman, System Development Corporation, P.O. Box 517, Paoli, Pennsylvania 19301.
Received 1 April 1985; accepted 28 January 1986.

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Publishing Co., Inc., 1986
52 Vanderbilt Ave., New York, NY 10017

0743-1066/86/$03.50
1. INTRODUCTION

Conjunction has long been a major problem for natural-language processing systems. Conjunction introduces scoping problems of adjuncts relative to the conjoined elements, problems with null (elided) elements, and generally a potential combinatorial explosion of parses. Because of these difficulties, there have been few attempts to treat the problem in its full generality.

Among the earliest (and most linguistically complete) efforts was the conjunction mechanism of the Linguistic String Parser (LSP) [10, 9]. Another early treatment was the SYSCONJ approach of Woods [14]. In both of these systems, the goal was a general treatment of conjunction that did not require an enormous proliferation or duplication of rules in the grammar. This was done by a general interrupt mechanism activated by recognition of conjunction words, such as and, but, or the conjunction comma, as in apples, oranges and bananas. Once such a word was recognized, normal parsing was suspended; a portion of the definition under construction was dynamically copied to accommodate the conjoined structure, at which point normal parsing was resumed. The resulting structure (parse tree) reflected the scoping of the conjunction with respect to other elements in the sentence, in particular, left and right adjuncts.

The advantage of these treatments was that one general "metarule" was sufficient to generate definitions for all conjoined structures. Also, in the LSP system, the handling of conjunction was made largely transparent to the grammar writer by invoking a special set of routines that automatically detected conjoined elements and then iterated restrictions over all conjoined elements.

The interrupt-driven approach has difficulty in controlling redundancy, as well as in controlling backtracking. Due to the generality of the conjunction mechanism, many redundant parses can be generated unless types of conjoining are severely restricted. The LSP solution to controlling redundancy has been to restrict conjoining to several types of nodes and to insert various restrictions to remove other redundancies. The top-down, backtracking strategy also results in inefficient parsing, especially when parsing multiple conjunctions and conjunctions within prepositional phrases.

More recently, there have been several treatments of conjunction within the framework of logic programming [2, 13]. Both of these works are based to some degree on the earlier approaches of Sager, Raze, and Woods. The work of Sedogbo requires the grammar writer to modify each relevant grammar rule in two ways: first, a predicate CONJ is added to generate a conjunction option for each rule where conjunction can occur; second, each element that may be omitted under conjunction has an empty option, hole, added to it. The Dahl-McCord approach uses an interrupt-driven interpreter, triggered by recognition of conjunction words, to generate appropriate conjunction options.

The present paper describes a metalogic grammar influenced by the metagrammatical approach of Generalized Phrase Structure Grammar (GPSG) [4]. Meta-Restriction Grammar provides an explicit metagrammatical component that automatically generates BNF definitions to parse conjunction. It is based on the Restriction Grammar framework [6, 7], which, in turn, is a logic implementation of Sager's String Grammar [5, 12]. Meta-Restriction Grammar uses a compact metagrammatical component to rewrite certain "base" rules of the grammar into more complex rules for handling conjunction. Because Restriction Grammar does
not allow parameters within BNF definitions, it is extremely well suited to a metagrammar approach. In this respect, it extends the treatment of conjunction proposed by Sedogbo [13]: it builds conjunctions only at certain types of nodes, and it uses "null elements" ("holes") to avoid the combinatorially explosive backtracking approach of Woods, Sager, and Dahl and McCord.

An important feature of Meta-Restriction Grammar is its ability to represent both the surface structure and a regularized structure. It does this by using unification to set a pointer from a gap to the filler of the gap; this preserves the scoping relationships, but also provides fast access to the filler, so that syntactic and selectional restrictions can be computed easily. Another important advantage of the metagrammar approach is that the grammar can be translated and compiled [3]. This contrasts with the interrupt-driven approach, where an interpreter generates rules for conjoining structures on demand, making it impossible to compile the complete grammar.

The remaining sections of the paper will describe the implementation of Meta-Restriction Grammar and its solution to the problem of coordinate conjunction. Section 2 presents a brief overview of the Restriction Grammar implementation, followed by an introduction to the String Grammar formalism. Section 3 outlines a wide range of conjunction problems and how they are treated in Meta-Restriction Grammar, including paired conjunction such as both...and, comma conjunction, elision of elements under conjunction, and distribution of left and right modifiers. This is followed by a brief section on implementation and a conclusion. Four appendices show various facets of the Meta-Restriction Grammar system: Appendix A contains a listing of the BNF definitions for a medium-coverage grammar of English; Appendix B contains a description of some of the basic data structures, restriction operators, and routines underlying Restriction Grammar; Appendix C contains the metagrammar for generating conjunctions, including all of the conjunction restrictions; finally Appendix D shows some sample sentences with conjunction which were parsed with the grammar in Appendices A–C.

2. RESTRICTION GRAMMAR

Restriction Grammar is a grammar-writing framework in PROLOG. It is derived from Sager's String Grammar [5, 12], which uses context-free BNF definitions, augmented by restrictions or constraints on the shape of the parse tree. Aside from any theoretical considerations about logic grammars, Restriction Grammar is useful simply because a very comprehensive English grammar exists in this framework [12]. As a member of the class of logic grammars, Restriction Grammar has several characteristics that distinguish it from definite-clause grammars (DCGs) [8]. First, the parse tree is automatically constructed by the interpreter, to reflect the context-free definitions successfully applied during sentence analysis. This contrasts with DCGs, where the grammar writer is responsible for specifying a parse tree by use of parameters. (Other logic grammar formalisms, such as Modifier Structure Grammar [2], also provide automatic generation of analysis trees.) As in DCGs, the non-context-free portion of the grammar is provided by restrictions or constraints. However, the restrictions in Restriction Grammar obtain the contextual information by examining the partially built parse tree or by inspecting the input word stream, rather than from additional parameters to the BNF definitions, as in DCGs.
There are several important advantages to eliminating explicit parameters from the BNF definitions. One advantage is the increased compactness and readability of the BNF definitions and their associated restrictions, since the BNF definitions do not become cluttered with numerous parameters (see Appendix A for the BNF portion of a running grammar covering a moderate subset of English). Although restrictions, when used in the BNF definitions, have no explicit parameters, all restrictions do, in fact, have parameters in Restriction Grammar, namely their starting point in the tree and the current word list; but the interpreter hides these from the grammar writer during the formulation of the BNF definitions. The second advantage of hiding parameters is that it simplifies the metagrammar enormously. The metagrammar which generates the new rule for conjunction is extremely compact, consisting of two rules: one for strings, and the other for head-plus-modifier structures (see Appendix B).

One drawback of Restriction Grammar is that it requires extra machinery for its execution, whereas DCGs require only a minimal interpreter supported directly in the PROLOG implementation. However, this extra machinery need not lead to a loss of performance. We have recently completed the implementation of a flexible translator for Restriction Grammar [3]. The translator converts each grammar rule into a PROLOG clause very similar to a DCG clause: the head consists of a parameterized BNF definition, while the body consists of conjunction and/or disjunction of further parameterized definitions and constraints. The resulting translated code is then compiled by the PROLOG compiler. Our flexible translator provides additional efficiency by supporting dynamic rule pruning based on the input word stream. This is done via mutual recursion between the translated code (where no dynamic interaction is required) and interpreted code (for dynamic rule pruning). The flexible translator coupled with dynamic rule pruning produces a six fold speedup over the interpreted version; parse times for most sentences are in the range of 1–3 seconds, including sentences with conjunction (see Appendix D).

A limitation on Restriction Grammar is that the parse tree reflects only the surface structure analysis. However, the Restriction Grammar execution mechanism keeps a pair of parameters for the construction of a separate semantic representation during parsing. Restriction Grammar itself makes no constraint on the type of representation constructed by the semantic component. Our present semantic representation (not discussed here) is assembled by a special set of restrictions; however, we are in the process of implementing a compositional semantics based on lambda conversion.

It is actually an advantage to decouple the semantic representation from the syntactic representation of the input. First, the syntactic structure is available for those phenomena that are influenced by surface structure (e.g., analysis of focus, or the interaction of conjunction and wh-constructions). Second, separation of semantic representation from syntactic analysis provides a more modular system and facilitates experimentation with alternative styles of semantic representation (e.g., predicate-logic expressions, or lambda notation).

The Restriction Language

An execution mechanism controls the application of BNF definitions and associated restrictions. Restrictions are applied at the point at which they appear in a BNF
definition, that is, after the node to the left has been constructed, and before the node to the right has been created. Each restriction imposes some constraint on the parse tree or on the incoming word stream. For restrictions which examine the parse tree, the starting point is the parent node (left-hand side of the BNF definition). No modifications to the basic mechanism have been required (for either interpreter or translator) in order to support conjunction.

Restrictions follow the layered implementation strategy of the Linguistic String Project system. *Primitive tree relations* support *restriction language operators*, which are used to build syntactically motivated *routines* such as *head*, *adjunct*, *main verb*, etc. The routines, together with the lower-level operators, are used to implement the actual *restrictions*.

The lowest level primitive tree relations are supported by a data structure of the form $\text{link(TreeTerm, Path)}$, where $\text{TreeTerm}$ represents the current location (a node) in the parse tree, and $\text{Path}$ is a set of directions from $\text{TreeTerm}$ back to the root node. $\text{TreeTerm}$ is a recursive data structure containing four fields:

$$\text{tt(Label, Child, RightSib, Word)}.$$  

Here, $\text{Label}$ is the name of the node as given by the left-hand side of the BNF definition; $\text{Child}$ and $\text{RightSib}$ are themselves $\text{TreeTerms}$ which represent the first child and immediate right sibling of the current node in the tree; and $\text{Word}$ is the word field containing the lexical item and its attributes. The $\text{tt}$ (tree term) structure provides the ability to find daughter and (right) sibling nodes in a tree.

The $\text{Path}$ data structure consists of the functor $\text{up}$ or $\text{left}$, with two arguments, namely the node reached by going one node up or left, and the remainder of the path as the second argument:

$$\text{link(TreeTerm, up(Parent,ParentPath))}.$$  

$$\text{link(TreeTerm, left(LeftSib,LeftSibPath))}.$$  

The four primitive tree relations are shown in Figure 1. These relations assume that the first argument (the current location) is instantiated; during execution, the second argument is instantiated to the new location (child/right sib/parent/left sib). The *daughter* (down) or *right sib* (right) node of the current node is given by the appropriate field of the current node’s $\text{TreeTerm}$; the $\text{NewPath}$ from a daughter (or right sib) is returned as $\text{up(TreeTerm, Path)}$ (or $\text{left(TreeTerm, Path)}$). The

FIGURE 1. Basic tree relations.

down(link(TreeTerm,Path), link(NewTreeTerm,up(TreeTerm,Path))) :-
  TreeTerm = tt(_,NewTreeTerm,_,_), nonvar(NewTreeTerm).

got_left(link(TreeTerm,Path), link(NewTreeTerm,left(TreeTerm,Path))) :-
  TreeTerm = tt(_,_,NewTreeTerm,NewPath), nonvar(NewTreeTerm).

up(link(_,up(NewTreeTerm,NewPath)), link(NewTreeTerm,NewPath)) :-
  nonvar(NewTreeTerm), !.
up(link(_,left(NewTreeTerm,NewPath)), Parent) :-
  nonvar(NewTreeTerm), up(link(NewTreeTerm,NewPath), Parent).

left(link(_,left(NewTreeTerm,NewPath)), link(NewTreeTerm,NewPath)) :-
  nonvar(NewTreeTerm).
parent (up) or left sib (left) node is found from the Path of the current node: in the case of parent, it may be necessary to move through all siblings until the parent is reached.

These relations have a procedural flavor, inherited from the implementation of Restriction Language in the Linguistic String Project system [11]. They have been implemented in their present form in order to permit easy reuse of LSP restrictions by preserving the functionality of Restriction Language. The higher levels of Restriction Language are more syntactic in motivation and correspondingly less procedural. As our system diverges from the LSP system, we plan to replace these low-level procedural operators with their nonprocedural counterparts (e.g., child/parent, sibling).

In addition to the tree-examination primitives, there are the label(Node, Name) relation between a node and its name (given by the label field in the tree term), and the word(Node, Word) relation between a node and the word(s) it subsumes (given by the word field of the tree term). Layered on top of the basic tree relations are the restriction operators [11]. Some of these operators are described in Figure 2. Routines are built up from the elementary restriction operators and other routines; a few are described in Figure 3. The elementary restriction operators and the routines provide a modular framework and allow the grammar writer to capture important linguistic generalizations within the string grammar framework.

**FIGURE 2.** Primitive restriction operators.

- **lookahead:** scans the word stream for a particular word;
- **empty:** checks a node to test if it is empty;
- **ascend:** ascends to the node (type) given in argument 1, passing through nodes (or node types) listed in argument 2, not passing through nodes (or node types) listed in argument 3, starting at the node in argument 4, terminating at the node in argument 5;
- **descend:** does a breadth-first descent through the parse tree, with the same five arguments as **ascend**;
- **wordl:** examines the current word in the word stream;
- **nextl:** examines the next word in the word stream.

**FIGURE 3.** Syntactic routines.

- **core(Start, Head):** finds the linguistic Head of the construction dominated by the node Start.
- **left_adjunct(Start, LeftAdjunct)/right_adjunct(Start, RightAdjunct):** finds the left/right adjunct of the construction in which Start occurs.
- **head(Start, Head):** given that Start is within an adjunct, finds the Head which the adjunct modifies.
- **get_verb(Start, Verb):** starts from the assertion and locates the main verb under assertion.
- **get_obj(Start, Object):** starts from the assertion and locates the object of the main verb.
String Grammar Concepts

String Grammar distinguishes two classes of structures: *exocentric* ("headless") constructions, and *endocentric* constructions (constructions with a head or center). The distinction between endocentric and exocentric constructions is taken from Harris’s early work on string analysis [5], but has its root in classical structural linguistics, e.g. in Bloomfield [1].

The constituents of an *endocentric* construction form a phrase of the same category as the *head* of the construction; for example, *the old disks* is a noun phrase, whose head is the noun *disks*. The constituents of an *exocentric* construction form a phrase of a *different* category than the categories of its constituents. An assertion, for example, consists of a noun phrase plus a verb phrase, but is not of the same category as either. Similarly, a prepositional phrase is made up of a preposition and a noun phrase; it is considered to be neither a noun phrase nor a preposition but rather a distinct entity whose distribution differs from nouns and prepositions.

In string analysis, exocentric constructions are called *strings* (it is this notion of string that gives its name to the Linguistic String Project). A string consists of two or more obligatory elements plus optional string adjunct (*sa*) elements. For example, an assertion can be defined as having obligatory elements subject + verb + object, with interspersed *sa*’s; a prepositional phrase (*pn*) is also a string, with obligatory preposition *p* plus noun string object (*nstgo*):

```
assertion ::= sa,subject,sa,verb,sa,object,sa.
pn ::= *p,nstgo. (* indicates a terminal symbol )
```

In Restriction Grammar, the endocentric constructions are called *lxr* constructions, derived from LSP terminology for left-adjunct + x (head) + right-adjunct. An *lxr* construction consists of a (possibly empty) left adjunct, a head, and a (possibly empty) right adjunct. This is used to define noun phrases, adjective phrases, and various forms of verb plus associated modifiers (tensed verb *tv* in the *ltvr* construction, present participle *ving* in the *lvingr*, past participle *ven* in *lvenr*, and infinitive *u* in *lvr*). These endocentric constructions are shown in Figure 4.

Restriction Grammar, following the LSP implementation, provides a mechanism for grouping nodes together into syntactically motivated *types*. Some of the major types are shown in Figure 5. These are atomic (terminal) elements, strings, *lxr* constructions, adjuncts (*adjset*), and left/right adjuncts (*ladjset/radjset*).

Using the type definitions and the related notions of *lxr* and *string* structures, we can define a number of additional routines which capture basic linguistic relationships. For example, the routine *core* finds the head of an *lxr* construction; *get_verb* finds the main verb under an assertion; and *get_obj* finds the complement of the main verb under an assertion. These last two operations are nontrivial because the

---

**FIGURE 4. lxr constructions.**

```
lnr ::= ln,nvar,rm.
lar ::= la,*adj,ra.
ltvr ::= lv,*tv,rv.
lvingr ::= lv,*ving,rv.
lvenr ::= lv,*ven,rv.
ltvr ::= lv,*v,rv.
```
grammar handles all verbs, including auxiliaries, uniformly in terms of strict subcategorization for possible complement types. Thus a verb such as have has as its complement types the past-participle construction (veno = past participle + object), as well as the direct-object (nstgo) construction. Similarly, the verb be has the progressive construction (vingo = present participle + object) as a complement type, and also passive (venpass = past participle + passive object), as well as objectbe, which contains predicate adjectives, predicate nominals and predicate adverbials. The subcategorization is applied to allow only appropriate objects for each verb. The result is a very uniform handling of verb complements, but also a “nested” complement structure, where an object node may well contain a participial form of a verb (the main verb of the sentence) and its complement. Appendix A shows the BNF definitions for such objects containing participial forms of the verb.

The distinction between endocentric and exocentric constructions is central to String Grammar and is reflected in the specifics of the grammar described here. However, our general approach to conjunction is largely independent of String Grammar theory: if a particular theory categorizes certain constituents differently (e.g., prepositional phrases as endocentric, with the preposition seen as a type of case marker), this could be accommodated with minor changes in BNF definitions and specific constraints; it would not affect the general conjunction mechanism.

3. TREATMENT OF CONJUNCTION

The goal of our treatment of coordinate conjunction is to provide a compact, efficient, linguistically motivated treatment of conjunction. Ideally, this treatment should be transparent to the grammar writer, so that treatment of conjunction can be separated from a statement of general language rules.

The overall approach of Meta-Restriction Grammar resembles closely that taken by Sedogbo [13]. In Sedogbo's treatment, a conjoined structure is accommodated by duplicating the entire preconjunction structure following the conjunction. To account for the elision or reduction that may take place under conjunction, certain elements are designated as null elements ("holes" in Sedogbo's terminology). This eliminates the problem, for example, of separately generating three distinct rules to account for reduced subject, reduced verb, or reduced object, in the cases shown in Figure 6. (Parses for these sentences are shown in Appendix D.)

Meta-Restriction Grammar uses a variant on this approach. Rather than copying the conjoined structure, the metagrammar copies the definition. This means that all options and restrictions of the original definition are available for the conjunct. A “gap” created by reduction under conjunction appears as a special null element, nulc, in the parse. Each gap keeps a record of the corresponding explicit element by
The field engineer replaced the board and adjusted the disk drive.

missing subject in second clause

The field engineer installed a board and the supervisor a disk drive.

missing verb in second clause

The field engineer has installed and the supervisor has adjusted the disk drive.

missing object in first clause

FIGURE 6. String conjunction examples.

setting a pointer to this element;\(^1\) the pointer is kept in the word field of the tree term associated with the gap. This makes the implicit "fillers" of the gaps available for subsequent semantic and syntactic restrictions, but distinguishes these implicit elements from elements explicitly present in the input word stream.

Preservation of scoping information is critical to a correct treatment of conjunction. For example, in the sentence Everyone takes the bus or drives a car, the subject of takes the bus and drives a car is everyone, for purposes of agreement, selection, etc. However, it is important that there be only one copy of everyone, with logical scope over the conjoined assertions: for all persons X, X takes the bus or X drives a car. The responsibility of the syntactic component is to preserve this scope information while applying all syntactic constraints. The syntax uses the pointer from filler to gap, together with a special restriction, to generate (via backtracking) all syntactically consistent permutations of these scoping relations; it is left to semantics to determine which is the correct scoping of adjuncts.

Proper integration of restrictions and conjunction is important. The LSP system employed an elegant reexecution mechanism [9] to handle the interaction of restrictions with conjunction. A special routine detected the existence of conjuncts and placed them on a reexecution stack; when one conjunct had been examined, the stack was popped to yield the next conjunct, and the restriction was reapplied. The Meta-Restriction Grammar approach does not provide automatic reexecution for conjunction at this time. However, by setting pointers to the implicit information, it does provide a more regularized structure for subsequent restrictions. For example, by modifying the routine core to look in the word field of a null node for implicit information, all restrictions involving the head of a construction work equally well on explicit or implicit information. This illustrates the modularity of the approach and the advantage of using general routines (such as core) to access information in appropriate structures.

As discussed in the previous section, Restriction Grammar distinguishes two classes of structures: exocentric constructions or strings, and endocentric constructions or \(lxr\) constructions. Since these raise somewhat different problems, we will discuss them separately.

Conjoining of Strings

A string consists of two or more obligatory elements and optional string adjunct (\(sa\)) elements. Conjunction within a string is handled by a metarule (shown in Figure 7) that allows the optional addition of a conjunct at the end of the string.

\(^1\)Of course, in a logic program, this merely involves unifying an uninstantiated variable in the node representation of the gap to the tree term representing the explicit element.
generate_conj_str(STRING) :-
retract((STRING :: = Rule)),
assert((STRING :: = [either], Rule, [or], sa, STRING)),
assert((STRING :: = Rule, ([conj_wd, sa, STRING,
                     {wconj3}, {wconj4}, {wnullcObj});
null])), !.

FIGURE 7. Metarule to generate conjunction in strings.

followed by a new string and some additional constraints specific to conjunction:

assertion ::= [either], sa, subject, sa, verb, sa, object, sa, [or], sa, assertion.
assertion ::= sa, subject, sa, verb, sa, object, sa, ((conj_wd, sa, assertion, {wconj3}, {wconj4}, {wnullcObj});
null).

The rule generate_conj_str shown in Figure 7 is called by a rule which instantiates the variable STRING for each definition of type string; generate_conj_str removes the original version of assertion and replaces it by the two rules, for two different cases of conjunction. The metarule also adds the conjunction-specific restrictions {wconj3}, {wconj4}, and {wnullcObj}, which are explained in Figure 9.

In addition to replacing nonconjunction rules with rules containing conjunction, there is an additional set of definitions required for handling conjunction. These are shown in Figure 8; the restrictions in these definitions are explained in Figure 9. These include a definition for the conjunction itself (conj_wd), which can be either a conjunction or a comma followed by a conjunction. The term spword ("special word") used as the conjunction word class is terminology inherited from the LSP's original interrupt-driven mechanism, which recognized "special words" to trigger the interrupt. The spwords include and, or, but, as well as, etc. The remaining definitions in Figure 8 provide for elision of elements under string conjunction via the nullc option. These definitions also contain restrictions to control elision.

There are several issues peculiar to the conjunction of string structures. One is that the required elements of a (nonconjoined) string are nonempty. However, under conjunction, one of these required elements (subject, verb, object) may be reduced, as illustrated by the sentences of Figure 6. This general approach applies equally to conjunction under the complex objects, accounting for sentences such as:

*The field engineer hopes to install the drive and to replace the board.*

*The field engineer plans to install but not to adjust the head.*

*The supervisor has installed and the field engineer will adjust the disk.*

FIGURE 8. Additional BNF definitions to handle conjunction.

subject ::= (dnullsubj), nullc, (wnullsubj).
verb ::= (dnullverb), nullc, (wnullverb).
object ::= (dnullobj), nullc.
nullec ::= .
conj_wd ::= ['.'], *spword.
conj_wd ::= (dconj2), *spword.
dconj2: if present word is comma, allows conjunction only if there is a “real” conjunction ahead, skipping over the next word (to avoid taking comma as conjunction in “, and”).

wconj3: checks that if conj_wd is comma, then there is a noncomma conjunct ahead.

wconj4: if verb is nullc, then both subject is not nullc, and object is not empty, and it is within a conjunction

dnullsubj: checks to make sure subject is under a conjunction

wnullsubj: allows nullc in subject if in a conjunction; also sets pointer to subject tree from previous conjunct.

dnullverb: checks to make sure verb is under a conjunction

wnullverb: allows verb to be nullc if in a conjunction; also sets pointer to verb tree from previous conjunct

dnullobj: checks to make sure that the next word is a conjunction

wnulicobj: if object is nullc, then locates the main verb & checks that it is compatible with the object; sets pointer to explicit object tree in following conjunct

FIGURE 9. String conjunction restrictions.

Restrictions are generally divided into two classes: disqualify restrictions (whose names begin with a d); and well-formedness restrictions (whose names begin with a w). Disqualify restrictions apply before a node is built, to determine whether the appropriate environment exists for applying a particular rule. Well-formedness restrictions apply after a node and all its children have been completed. These restrictions check consistency, for example, between a verb and its object, or subject verb agreement. In the case of conjunctions, they have another important function, namely setting the pointer of the nullc (gap) node to the corresponding explicit (filler) information. The set of restrictions for string conjunction is shown in Figure 9.

Conjoining lxr Structures

Conjunction of lxr nodes presents a different set of problems. The lxr nodes are characterized by having one essential element (the head) and left and right adjuncts which may be empty or filled. The principal problem in handling conjoined lxr elements is to indicate the proper distribution of the adjuncts over the conjoined elements. The phrases in Figure 10 illustrate this problem.

Because adjuncts may be empty in the normal course of parsing, we chose not to generate special nullc elements for them, but simply to let them take on null values when reduced under conjunction. Thus nullc is an option reserved for essential elements in a string. Adjuncts take on the value null, which can be updated later to include a pointer to the elided information, if it turns out that they are reduced
the last replacement and adjustment
  = the last replacement and the last adjustment
distribution of adjuncts over both conjuncts

she has had the measles and mumps
  = she has had the measles and she has had mumps
  no distribution of article over second conjunct

develop quickly replaced and adjusted the controller.
  = they quickly replaced the controller and
distribution of adverb over both conjoined verbs

due to a bad circuit board or wire in the disk drive
  = due to a bad circuit board or due to a bad wire ?? in the drive
  ambiguous: in the drive may distribute over one or both conjuncts.

FIGURE 10. Examples of conjoined \textit{lxr} structures.

under conjunction. This avoids having to decide prematurely whether an element is
empty because it is really empty, or empty because it has been reduced under
conjunction.

The examples of Figure 10 illustrate that conjunction over \textit{lxr} constructions can
often be ambiguous from a strictly syntactic point of view. The constructions can
only be disambiguated by the use of semantic information. Therefore the job of the
syntactic component is to generate \textit{all} possible readings, so that the semantic
component can select the correct one. Ideally, of course, the syntactic and semantic
components are interleaved, so that the choice can be made as soon as possible, to
avoid a combinatorial explosion of parses. However, the current treatment of \textit{lxr}
conjunctions avoids some of the combinatorial explosion of the LSP or \textit{SYS\textsc{CONJ}}
treatments. The LSP and \textit{SYS\textsc{CONJ}} treatments produced distinct surface structures
corresponding to each distinct distribution of adjuncts. This led to inefficiencies due
to large amounts of backtracking when parsing conjunctions. The Meta-Restriction
Grammar approach is to confine the problem of adjunct distribution to several final
restrictions within the \textit{lxr} node. Figure 11 illustrates the metagrammar rules used to
generate \textit{lxr} constructions. Note in particular that the rules governing adjunct
distribution, \textit{wconj \_lx} and \textit{wconj \_rx} (described in Figure 12), are applied only when
the final conjunct has been reached; at this point, they are applied once, to the
entire conjoined structure. Thus if it turns out later that it is incorrect, backtracking
occurs only into one of these restrictions, not into the \textit{lxr} structure itself.

As indicated by the metagrammar rules, there are three restrictions involved in
parsing the \textit{lxr} constructions; one of them, \textit{wconj}\textsubscript{3}, allows comma as a conjunction
word only if there is also a "real" conjunction word connecting the final conjunct.

FIGURE 11. Metagrammar rules for \textit{lxr} constructions.

\begin{verbatim}
generate conj_lxr(LXR) :-
  retract((LXR :: = Rule)),
  assert((LXR :: = [both],Rule,
    [and],sa,LXR,{wconj\_lx},{wconj\_rx})),
  assert((LXR :: = Rule,
    ((conj\_wd,sa,LXR,{wconj3}));
    {wconj\_lx},{wconj\_rx}))),!.
\end{verbatim}
computes distributed elements of $lx$ under conjunction; This procedure should compute only distinct readings, via backtracking; initial reading is distributed reading; backs up into local reading. Algorithm climbs to top of conjoined LXR pile, and computes distribution pairwise, traversing down the chain.

computes the distribution of the right adjunct under conjunction; it starts from the lowest (last) pair of conjuncts, assigns the higher of the two either a distributed reading, or a local reading; then finds the next right-adjunct above the current pair, and calls itself recursively.

**FIGURE 12.** Restrictions for parsing $lxr$ constructions.

This rule is general to both the $lxr$ and string constructions and has been discussed above. The other two have to do specifically with the distribution of left and right adjuncts.

Both $wconj_{lx}$ and $wconj_{rx}$ operate from the final element in a series of conjoined elements. Both operate recursively, computing distribution of adjuncts pairwise, before progressing to the next higher $lx/rx$. However, they traverse the tree in opposite directions. Since the first (higher) $lx$ is explicit and the second may be implicit, the algorithm for $lx$ climbs to the top of the $lxr$ nest, and then traverses the structure recursively downwards. For the $rx$, it is reversed: the lower $rx$ is explicit, and the preceding one may be omitted. Therefore, traversal is done from the bottom up.

Both constructions draw on the same routines: if the explicit element is empty, then there is no difference between a distributed reading and a local reading, since there is nothing to distribute, as in *cats and dogs*. In this case, nothing happens and the restriction moves on to the next pair. If the explicit element is *not* empty, then it looks at the other adjunct. If it is filled, then again, there is no distribution of adjuncts possible, as in *the controller and the head*. If, however, the other adjunct is empty, as in *the controller and head*, then there is a possibility of distributing the adjunct; this is done by setting a pointer from the word field of the reduced adjunct slot to the explicitly filled adjunct, as described for string conjoinings. The distributed reading is the first reading generated. On failure, there is a backtrack point, allowing a *local* reading to be generated; this local reading is explicitly marked by inserting a flag *local* into the word field of the empty adjunct. The local reading, however, may not always be allowed, for example, if the lower conjoined noun must share the determiner of the first noun.

The above description applies to the general case of left and right adjuncts, but not to a very important exception, namely the left noun adjunct or $ln$. This is because the $ln$ is itself a string, consisting of a series of slots for various types of modifiers:

$$ln ::= tpos, qpos, apos, npos.$$ 

These slots are for positions for articles ($tpos$), numerical quantifiers ($qpos$), adjectives ($apos$), and compound noun modifiers ($npos$), as in *the dozen crisp doughnut holes*. Distribution of adjuncts within the $ln$ must preserve proper bracket-
ing. That is, in the phrase *the dozen crisp doughnut holes and six oatmeal cookies*, the adjective *crisp* cannot be construed to modify *cookies*, because there is a quantifier *six* "blocking" this reading. Without the second quantifier, *crisp* could modify *oatmeal cookies: the dozen crisp doughnut holes and oatmeal cookies*. To capture this "proper bracketing" effect, the code for distribution of *in* elements is substantially more complex, as is described below: The general restriction *wconj lx* determines whether it is dealing with a left noun modifier, in which case it invokes the specialized code in Figure 13; otherwise, it uses the general algorithm described above. Figures 14 and 15 trace the assignment of adjuncts for the phrase *the last*

**FIGURE 13.** Algorithm for computing distribution of left noun modifiers.

```plaintext
conj ln:
  computes distributed elements of ln under conjunction, based
  on the following observation:
  if an element of ln (e.g., apos) is "local", then to preserve
  proper scoping, everything to its right must be local.

Algorithm to compute whether parts of lower ln are in scope of upper ln:
given an upper conjoined ln and a lower ln,
1. move through lower ln and if an element is filled,
   mark everything to its right as "local";
2. locate last element of upper ln and of lower ln;
3. if lower element neither filled nor local,
   then if upper element is filled,
      then either set pointer to it in lower element
      and mark it and all elements to its left as distributed
      or mark it as local and
      go left in lower & upper Ins and repeat step 3
     until can't go left (have assigned scope to all elements).
This procedure should compute only distinct readings, via backtracking.
```

**FIGURE 14.** Initial assignment of distributed adjuncts.

*Trace showing distribution of adjuncts in lnr for phrase:*
*the last replacement and adjustment of the disk.*

*Reading: the last replacement of the disk and the last adjustment of the disk.*
** doing wconj lx in LN
  > > marked npos local
  apos is nonnull
  > > marked apos distributed
  (will mark all elements to its left as "copied")
  (NOTE: "copied" means setting a pointer from gap to filler)
  > > Tree from upper LN after marking elements

```
ln
  tpos
    t = = the
  apos
    adj = = last
nvar
  n = = replacement
rm = = copied pn
conj wd
  spword = = and
lnr
  ln
```
FIGURE 14.  (Continued)

FIGURE 15. Assignment of adjuncts on backtracking into \texttt{lnr}.

Trace showing backtracking into local reading of \texttt{apos in lnr} for phrase: the last replacement and adjustment of the disk.

Reading: the last replacement of the disk and the adjustment of the disk

\> \> marked \texttt{apos local}
\> \> marked \texttt{apos local}
\texttt{tpos is non-null}
\> \> marked \texttt{tpos distributed}
\> \> Tree from upper \texttt{LN after marking elements}

\begin{verbatim}
ln
  tpos
    t = = the
    apos
    adj = = last
nvar
  n = = replacement
rn = = copied \texttt{pn}
conj._wd
  spword = = and
lnr
  tpos = = copied the
    qpos = = tagged local
    apos = = tagged local
    npos = = tagged local
nvar
  n = = adjustment
rn
  p = = of
nstgo
  nstg
lnr
  tpos
    t = = the
nvar
  n = = disk
\end{verbatim}
replacement and adjustment of the disk. Figure 14 shows the initial distributed
assignment; Figure 15 shows the parser backtracking into the in and assigning a
local reading to the adjective slot.

The distinction between conjunction at the string level and conjunction at the lxr
level allows the grammar to avoid the generation of spurious ambiguities. For
example, Sedogbo comments that his grammar, unless constrained by restrictions,
generates two readings for the sentence John and Peter sleep, one with conjunction
at the noun-noun level, and one with conjunction at the level of conjoined subjects:

subject (Inr and Inr) verb nullobj.
subject (nullc verb + nullobj) and subject verb nullobj.

We avoid this in Meta-Restriction Grammar by limiting string-level conjunction to
a single “hole,” which may be the object in the upper conjunct or the subject or
verb in the lower conjunct. Thus we get a single parse for this, with no additional
restrictions.

Coverage

The grammar described here covers a wide range of conjunction phenomena, as
listed below.

- Paired conjuncts such as both . . . and and either . . . or are handled by the
  appropriate alternative in the metarules.
- Comma conjunction, such as I ate apples, oranges, and pears, is handled by
  the same mechanism as other conjunction types, except that the comma is
  allowed as a conjunct only if it occurs in a sequence of conjunctions ending
  with a “real” conjunction, such as and.
- Distribution of left and right modifiers over conjoined head-plus-modifier
  (lxr) constructions is accounted for in full generality.
- Conjunction of lxr constructions is accounted for by a single meta-rule: e.g.,
  black and white
  can and should
  heart and kidneys
  installed and adjusted
  are all instances of conjoining of lxr constructions.
- Conjunction at the string level, including prepositional phrases, assertions,
  and complex objects, is handled by a single metarule, e.g.: in the water and on
  land, and they ate well and slept well, and they like to work and to play.
- Spurious ambiguities are avoided by constraining string conjuncts to allow
  only a single hole per string.

At the present time, Marcia Linebarger of the SDC Natural Language group is
extending the grammar to cover relative clauses and questions. The integration of
conjunction with the relative clause and question constructions appears to be
straightforward; the grammar currently handles the constructions shown in Figure
16. We will describe this work in detail in a separate report.
1. Conjoined assertions with reduced object in relative clause:
   *The disk which he installed and she repaired has failed.*
2. Conjoined assertions with reduced subject in relative clause:
   *The disk which was installed properly but was not repaired regularly has been removed.*
3. Conjoined ltr in relative clause:
   *The disk which he repaired and installed has failed.*
4. Conjoined veno in relative clause:
   *The disk she has repaired but not installed is old.*
5. Conjoined wh question with reduced subject:
   *What has been installed but has not been repaired?*
6. Conjoined wh question with reduced subject:
   *Who has installed or will soon install the disk?*
7. Conjoined yesnoq with reduced object:
   *Has he installed or does he intend to install the disk?*

**FIGURE 16.** Coverage of conjunction interacting with relative clauses and questions.

There are two limitations to the current coverage of conjunction: first, we have not yet investigated distribution of sentence adjuncts at the string level; and second, conjunction of subparts of certain complex objects is not yet handled (e.g., *the car drove through and completely demolished the window*). Implementations for both of these have been sketched out and do not appear to impose any major problems.

### 4. IMPLEMENTATION

The current system is implemented in Quintus PROLOG running on a VAX 11/785 under Berkeley 4.2 Unix. The system also runs in PROLOG on a Symbolics 3640. The parsing times shown in Appendix D are for the translated, compiled grammar plus metagrammar listed in Appendices A and C, running in Quintus PROLOG on a moderately loaded VAX 11/785. This grammar includes some 60 BNF definitions and some 30 restrictions.

### 5. CONCLUSION

The previous sections have described a broad-coverage treatment of coordinate conjunction within a metagrammatical framework. The emphasis has been on parsing strategies to generate all possible readings. Clearly, the parsing strategy must be coupled with semantic strategies to choose the correct reading, in case of ambiguities. However, the parsing strategy has been carefully designed to eliminate spurious redundancies, as well as to handle some difficult constructions rarely mentioned in the literature (comma as a conjunction, scoping of left noun modifiers, parsing of appositives, treatment of paired conjunctions such as *both ... and, either ... or*). The metagrammar treatment offers an unusually compact and efficient method of capturing a wide range of conjunction phenomena.
APPENDIX A. LISTING OF BNF DEFINITIONS AND TYPES

/* root node */
rootnode(sentence).

/* bnf definitions */
sentence ::= center, (['.'];['?']).
center ::= assertion.
tense ::= (lv,*w,rv);null.
assertion ::= sa,subject, {wsl1},sa,verb, {wagree},sa,object,sa.
  sa ::= *d,*int;pn;null.
null ::= .
  pn ::= *p,nstgo,commaopt.
commaopt ::= [','];null.
subject ::= nstg;{(dquest4),nullwh};[there].
nullwh ::= .
nstg ::= (lnr,{ wcount});nrep.
lnr ::= ln, {wln},nvar, {noun_agree},rn.
nvar ::= *n;*pro;{dn2},nulln,(wn1)).
nulln ::= .
ln ::= tpos,qpos,apos,npos,{np_agree}.
tpos ::= *t;wln;null.
whln ::= {[whose];[which];[what];howqastg}.
howqastg ::= {[how],[much],[many]},{(of)*t};null).
qpos ::= *q;null.
apos ::= *adj;*ven;null.
npos ::= nnn;null.
nnn ::= {dn1},(*n;(*n,nnn)).
nr ::= pn;vingo;venpass;null;appos.
appos ::= [',],nstg,[''].
nrep ::= [what],subject,sa,verb, {wwhl},sa.
verb ::= ltvr;lvr.
lvr ::= lv,*tv,rv.
lvingr ::= lv,*ving,rv.
lvenr ::= lv,*ven,rv.
lvr ::= lv,*v,rv.
lv ::= *d;null.
rn ::= pn,*d;null.
object ::= {dverbobj},{nstgo;pn;npon;{dsel6},objectbe};veno;nullobj;tovo,
  {wverbobj}.
nstgo ::= nstg;{(dwh1),nullwh}.
npn ::= nstgo;pn.
objectbe ::= = astg;nstg;pn;vingo;venpass;{(dwh2),nullwh}.
astg ::= lar.
lar ::= la,*adj,ra.
la ::= *d;null.
ra ::= pn;null.
vingo ::= {dsel5},lvingr,sa,object,sa.
venpass ::= {dsel4},lvenr,{wpassobj1},sa,passobj,sa.
veno ::= {dsel4},lvenr,sa,object,sa.
nullobj :: =~
thats :: = [that],assertion.
tovo :: = [to],lvr,sa,object,sa.
passobj :: = (nullobj;pn),{wpassobj2}.

/* lists */
type(atom,[d,n,int,null,t,adj,ven,pro,rv,nullobj,v,ving,q,nnln,nnlc,nnllwh,nnlw]).
type(string,[assertion,pn,npn,nrep,nlts,tono,veno,vingo,venpass]).
type(stgseg,[assertion,tovo,vingo]).
type(adjset,[sa,ln,lv,rv,la,ra]).
type(ladjset,[ln,lv,la]).
type(radjset,[rn,rv,ra]).
type(lx,[ln,lv,lnr,ltvr,lvr,lvingr,lvenr,lrv,lar]).
type(verb,[lvingr,lvenr,lvr,ltvr,verb]).
type(conj_word,[conj wd, 'and', 'or']).

APPENDIX B. ELEMENTARY DATA STRUCTURES, OPERATORS, AND ROUTINES

% DATA STRUCTURES

\[ \text{tt(Label, Child, RightSib, Word)} \]

\[ \text{link(TreeTerm, up(TreeTerm,Path))} \]

\[ \text{link(TreeTerm, left(TreeTerm,Path))} \]

% MOVEMENT OPERATORS

\[ \text{down(link(TreeTerm,Path),link(NewTreeTerm,up(TreeTerm,Path)))} \]
\[ \text{TreeTerm = tt(\_\_\_\_\_\_, \_\_\_),nonvar(NewTreeTerm).} \]

\[ \text{right(link(TreeTerm,Path),link(NewTreeTerm,left(TreeTerm,Path)))} \]
\[ \text{TreeTerm = tt(\_\_\_\_\_\_, \_\_\_),nonvar(NewTreeTerm).} \]

\[ \text{up(link(\_,up(NewTreeTerm,NewPath)),link(NewTreeTerm,NewPath))} \]
\[ \text{nonvar(NewTreeTerm),!} \]

\[ \text{up(link(\_,left(NewTreeTerm,NewPath)),Parent)} \]
\[ \text{nonvar(NewTreeTerm),up(link(NewTreeTerm,NewPath),Parent).} \]

\[ \text{left(link(\_,left(NewTreeTerm,NewPath)),link(NewTreeTerm,New Path))} \]
\[ \text{nonvar(NewTreeTerm).} \]

% RESTRICTION OPERATORS

lookahead: scans the word stream for a particular word;
empty: checks a node to test if it is empty;
ascend: ascends to the node (type) given in argument 1,
    passing through nodes (or node types) listed in argument 2,
    not passing through nodes (or node types) listed in argument 3,
    starting at the node in argument 4,
    terminating at the node in argument 5;
descend: does a breadth-first descent through the parse tree, with the same five arguments as ascend;
wordl: examines the current word in the word stream;
nextl: examines the next word in the word stream.

% ELEMENTARY ROUTINES

\[\text{element(NodeNames, Start, End)}:\]
searches for a node of type NodeNames (a list of node names or the name of a type of node) among the children of Start, storing the node in End;

\[\text{l(NodeNames, Start, End)}/r(NodeNames, Start, End):\]
searches for a node of type NodeNames to the left/right of Start;

\[\text{last_element(Start, End)}:\]
searches for last child under Start and stores it in End;

\[\text{last(Start, End)}:\]
searches for last sibling (or coelement) under Start, storing it in End.

% SYNTACTIC ROUTINES

\[\text{core(Start, Head)}:\]
finds the linguistic Head of the construction dominated by the node Start.

\[\text{left_adjunct(Start, LeftAdjunct)}/\text{right_adjunct(Start, RightAdjunct)}:\]
finds the left/right adjunct of the construction in which Start occurs.

\[\text{head(Start, Head)}:\]
given that Start is within an adjunct, finds the Head which the adjunct modifies.

\[\text{get_verb(Start, Verb)}:\]
starts from the assertion and locates the main verb under assertion.

\[\text{get_obj(Start, Object)}:\]
starts from the assertion and locates the object of the main verb.

% CONJUNCTION ROUTINES

\[\text{checkNotEmpty(Node)}:\]
checks that a node contains neither an actual word, nor a copy of a word implicit under conjunction;

\[\text{copyNullc(EmptyNode, FilledNode)}:\]
1. if FilledNode has an entry in its word field that is a tree term (it has already been restored under conjunction—see 2), then it is stored in the word field of EmptyNode;
2. If the word field of FilledNode is a word (the normal case), then the tree term associated with FilledNode is copied into the word field of EmptyNode.

APPENDIX C. METAGRAMMAR FOR CONJUNCTION

% META-RULES FOR GENERATING CONJUNCTION STRINGS

\[-op(1200,xfx,:: =).\]
\[-op(950,xfx,\text{.}).\]
\[-op(500,fx,\text{*}).\]
CONJUNCTION IN META-RESTRICTION GRAMMAR

gen Conj :- of_type(lxr,LXR), generate Conj_lxr(LXR), fail.
gen Conj :- of_type(string,STRING), generate Conj_str(STRING), fail.
gen Conj.

generate Conj_lxr(LXR) :-
  retract((LXR :: = Rule)),
  assert((LXR :: = [both],Rule,[and],sa,LXR,(wconj_lx),(wconj_rx))),
  assert((LXR :: = Rule,
    ((conj wd,sa,LXR,(wconj3));
    {wconj_lx},{wconj_rx}))),!.

generate Conj_str(STRING) :-
  retract((STRING :: = Rule)),
  assert((STRING :: = [either],Rule,[or],sa,STRING)),
  assert((STRING :: = Rule,((conj wd,sa,STRING, 
    {wconj3},{wconj4},{wnullcObj});
    null))),!.

of_type(Type,Member) :-
  type(Type,List),!.,isin(Member,List).

% CONJUNCTION DEFINITIONS

subject :: = {dnullsubj},nullc,{wnullsubj}.
verb :: = {dnulverb},nullc,{wnulverb}.
object :: = {dnullobj},nullc.
nulc :: = _.
conj wd :: = [','], *spword.
conj wd :: = {dconj2}, *spword.

% CONJUNCTION RESTRICTIONS

/* dnullsubj
checks if in scope of conjunction by looking for conjunction
   to the left of the parent (assertion) node
*/
dnullsubj(S_W) :- up(S,Assert),l(conj_word,Assert,CW).
/* dnullobj
checks if in scope of conjunction by checking that next word
is conjunction word (has attribute “spword”)
*/
dnullobj(_S_W) :-
  wordl(W, X:[_Root,spword:_Y]).
/* dnulverb
checks that verb is within conjunction scope by ascending to nearest
assertion and checking that it is preceded by a conjunction
*/
dnullverb(S_W) :-
  ascend(assertion,_adjset,S,Assert,l(conj_word,Assert,CW).
/* wnullsubj
  allows subject to be null if in a conjunction;
  also fills in word field with subject tree.
*/
wnullsubj(S,_) :-
  % test that immediate node is conj_word
  up(S,Assert),l(conj_word,Assert,Conj),
  % locate value of subject,
  l(subject,Conj,Subj), down(Subject,SubjectValue),
  % locate nullc and set pointer from nullc to subject into word field
  down(S,Nullsubj),copyNullc(Nullsubj,SubjectValue).
/* wnullverb
  allows verb to be null if in a conjunction;
  also sets pointer in word field to verb tree.
*/
wnullverb(S,_) :-
  % test that immediate node is conj_word
  up(S,Assert),l(conj_word,Assert,Conj),
  % check that subject is not nullsubj
  ((element(subject,Assert,Subj),!,not(element(nullc,Subj,Nullc)));true),
  % locate core of verb, to set pointer from nullverb
  get_verb(Conj,V),down(V,VValue),
  % locate nullc and set pointer to verb
  down(S,Nullverb),copyNullc(Nullverb,VValue).
/* wnullcObj
  if object is nullc, then locate the main verb;
  make sure that it is compatible with the object;
  set pointer to explicit object in word field of nullc object.
*/
wnullcObj(S,W) :-
  % if object is nullc
  element([object,passobj],S,Obj),down(Obj,Nullc),
  test(nullc,Nullc,Nullc),!,
  % then locate the verb
  get_verb(S,V),
  % find the conjoined explicit object
  last_coelement(Obj,LowS),
  get_obj(LowS,LowObj),down(LowObj,ObjType),
  % make sure that the explicit object goes with the verb
  checkVerbObj(ObjType,V),
  % set pointer to it from the word field of the nullc object
  copyNullc(Nullc,ObjType).
wnullcObj(_,_).
/* dconj2
  if present word is comma, allows conjunction only if, skipping
  next word (to avoid taking comma as conjunction in "., and"),
  there is a "real" conjunction ahead.
*/
CONJUNCTION IN META-RESTRICTION GRAMMAR

dconj2(_S,[',':_X|MoreWords]) :-!,
    nextl(MoreWords,Next),
    lookahead(spword,[andstg,orstg],Next,_Out).

dconj2(_S,_W).

/*
  checks that if conjstg is comma, then there is a real conjunction ahead.
*
wconj3(S,W) :-
    (element(conj_word,S,C),last_element(C,CWD),not(word(CWD,'')),!);
    (element(lxr,S,LXR),!,wconj3(LXR,W));
    (element(string,S,String),wconj3(String,W)).

/*
  if verb is nullc, then both subject is not nullc, and object not empty
  and it is within a conjunction
*/
wconj4(_,_) :-
    l(conj_word,S,C),!,
    element(verb,S,V),((element(nullc,V,Nullc),!,element(subject,S,Subj),
    not(empty(Subj)), element(object,S,Obj), not(empty(Obj)));true).

wconj4(_,_).

/*
  computes distributed elements of lx under conjunction;
  this procedure should compute only distinct readings, via backtracking;
  initial reading is distributed reading; backs up into local reading.
  Algorithm climbs to top of conjoined LXR pile,
  and computes distribution pairwise, traversing down the chain.
*/
wconj_lx(LXR,_):-
    topLX(LXR,TopLX),
    last_coelement(TopLX,NextLXR),
    ((test(lxr,NextLXR,NextLXR),!
      nl,print('# found next lx?),
      element(ladjset,NextLXR,LowLX),conj_lx(TopLX,LowLX));
    true).

topLX(LXR,TopLX) :-
    l(ladjset,LXR,UpLX),!,up(UpLX,NewLXR),topLX(NewLXR,TopLX).

topLX(LXR,TopLX) :-
    element(ladjset,LXR,TopLX).

conj_lx(UpLX,LowLX) :-
    test(ln,UpLX,UpLX),!,
    conj_ln(UpLX,LowLX).

conj_lx(UpLX,LowLX) :-
    ((checkNullValue(UpLX),!);distrib_adj(UpLX,LowLX)),
    (next_pair(LowLX,NewLowLX),!,conj_lx(LowLX,NewLowLX));true).

*/
% conj_in
% computes distributed elements of ln under conjunction, based
% on the following observation:
% if an element of ln (e.g., apos) is “local”, then to preserve
% proper scoping, everything to its right must be local.
% Algorithm to compute whether parts of lower ln are in scope of upper ln:
% given an upper conjoined ln and a lower ln,
% 1. move through lower ln and if an element is filled,
% mark everything to its right as “local”;
% 2. locate last element of upper ln and of lower ln;
% 3. if lower element neither filled nor local,
% then if upper element is filled,
% then either set pointer to it in lower element
% and mark it and all elements to its left as distributed
% or mark it as local and
% go left in lower & upper Ins and repeat step 3
% until can’t go left (have assigned scope to all
% elements).
% This procedure should compute only distinct readings, via backtracking.
%
conj_in(UpLN, LN) :-
    down(LN, LN_element), mark_in(LN_element),
% get last elements of upper and lower Ins
    last_element(UpLN, ULast),
    last_element(LN, Last),
    mark_distrib(Last, ULast),
do_next_conj(LN).
do_next_conj(LN) :-
    next_pair(LN, LowLN), !, conj_in(LN, LowLN).
    do_next_conj(_LN).
next_pair(LN, LowLN) :- !,
    r(lxr, LN, LowLXR),
    element(ladjset, LowLXR, LowLN).
/* if copy filled, then don’t do anything */
/* otherwise get next elements of Main and Copy and repeat, */
/* or succeed if can’t go left */
mark_distrib(Copy, Main) :-
    not(empty(Copy)), !,
    ((left(Copy, NewCopy), !, left(Main, NewMain),
        mark_distrib(NewCopy, NewMain));
true.)
% if Copy is empty and unmarked, try to get from upper ln...
mark_distrib(Copy, Main) :-
    not_filled(Copy), down(Copy, LV),
    not(checkNullValue(Main)),
down(Main, UV),
copyNullc(LV, UV),
mark_left_distr(Copy, Main).

% else mark local.
mark_distrib(Copy,Main) :-
    mark_local(Copy),
    ((left(Copy,NextCopy),!,left(Main,NextMain),
      mark_distrib(NextCopy,NextMain));
     true).

mark_left_distr(Copy,Main) :-
    left(Copy,Next),!,down(Next,LV),left(Main,UNext),down(UNext,UV),
    copyNullc(LV,UV),
    mark_left_distr(Next,UNext).
mark_left_distr(_,Main).

not_filled(E) :-
    empty(E), not(checkWdField(E,local)).
mark_ln(E) :-
    not(empty(E)),!,mark_right(E).
mark_ln(E) :-
    right(E,Next),!,mark_ln(Next).
mark_ln(_E).

mark_right(E) :-
    right(E,Next),!,mark_local(Next),mark_right(Next).
mark_right(_E).

mark_local(E) :-
    down(E,Value), getWdField(Value,local),!.
mark_local( ). % if there is already a word, no need to mark local

copyNullc(Node,Tree) :-
    makeNullc(Node,Tree).

/* wconj_rx
computes the distribution of the right adjunct under conjunction;
it starts from the lowest (last) pair of conjuncts,
assigns the higher of the two either a local reading, or a
distributed reading; then finds the next right-adjunct above
the current pair, and calls itself recursively.
*/

wconj_rx(LXR,_) :-
% get upper and lower RX
    l(radjset,LXR,UpRX),!, element(radjset,LXR,RX),
% either value is null, or choose distrib or local reading (distrib_adj)
% in either case, go to next LXR and reexecute
    (checkNullValue(RX),!); distrib_adj(RX,UpRX)),
    up(UpRX,UpLXR), wconj_rx(UpLXR, _).
wconj_rx(_, _).

% 1. succeed in case Copy is not empty
% 2. mark distributed—will fail if Copy is not empty
% 3. otherwise, mark local—this will fail if Copy not empty
distrib_adj(_Main,Copy):-
    not(checkNullValue(Copy)),!.
distrib_adj(Main,Copy):-
    mark_distrib_adj(Main,Copy).
distrib_adj(_Main,Copy):-
    mark_local(Copy),!.
mark_distrib_adj(Main,Copy):-
    down(Copy,Null),
    down(Main,Value),copyNullc(Null,Value).

APPENDIX D. COVERAGE OF SENTENCES WITH CONJUNCTION

Parse times are for the translated, compiled grammar in Quintus PROLOG on a VAX 11/785 running first (second) parse and to termination.

Sentence: The field engineer replaced the board and adjusted the disk drive.

Lexicon lookup completed

(The resulting diagram is shown in Figure 17.)

remaining words: []
runtime: 2.266 sec.
more? y

no more parses
runtime: 4.91599 sec.

Sentence: The last replacement and adjustment of the drive took an hour.

Parse 1
Left adjunct fully distributed; right adjunct distributed:
Paraphrase: The last replacement of the drive and the last adjustment of the drive took an hour.
Runtime: 1.9 sec.

(The resulting diagram is shown in Figure 18.)

Parse 2
Right adjunct is local, left adjunct is fully distributed:
Paraphrase: The last adjustment of the drive and the last replacement took an hour.
Cumulative runtime: 3.314 sec.

Parse 3
Right adjunct distributed, the distributed, but apos = last local:
Paraphrase: The last replacement of the drive and the adjustment of the drive took an hour.
Cumulative runtime: 5.666 sec.
Parse 4
Right adjunct local, the distributed, but apos = last local:
Paraphrase: *The adjustment of the drive and the last replacement took an hour.*
Cumulative runtime: 6.64 sec.

The remaining two parses are not shown; they place the prepositional phrase *of the drive* at the sentence level (with last distributed or local).

**Sentences Handled by the Conjunction Mechanism**

1. Conjunction in assertion with reduced verb (2.0/4.4 sec):
   *The field engineer installed a board and the supervisor a drive.*

2. Conjunction in assertion with reduced object (2.3/4.8 sec):
   *The field engineer has installed and the supervisor has adjusted the drive.*

3. Comma conjunction in the noun phrase (1.6/6.9 sec):
   *The field engineer installed a disk, the board and a controller.*

4. Comma conjunction or appositive in noun phrase (2 parses: 2.1/6.5/10.8 sec):
   *The field engineer replaced the disk, an old model, and a board.*

5. Both/and conjunction in the noun phrase (1.5/3.4 sec):
   *Both the field engineer and the supervisor adjusted the drives.*

6. Either/or conjunction in assertion (2.0/4.2 sec):
   *Either the board was replaced or they have adjusted the head.*

7. Comma conjunction in assertion with reduced subject (2.7/6.3 sec):
   *The field engineer replaced the board, adjusted the controller, and installed a new drive.*

8. Both/and conjunction in noun phrase, and conjunction in assertion (2.3/4.5 sec):
   *Both the disk and the head were replaced and a new motor was installed.*

9. Conjunction of participles in prenominal adjective position (1.5/4.2 sec):
   *The repaired and adjusted drive is working.*

10. Conjoined prepositional phrase (1.9/6.9 sec):
    *The boards of the controller and of the cpu were replaced.*

11. Conjoined noun phrase under preposition (2 parses: 1.5/3.6/7.6 sec):
    *The boards of the controller and the cpu were installed.*

12. Conjoined adjective in predicate position (0.9/2.0 sec):
    *The drive is old and worn.*

13. Conjoined quantifier (1.3/3.0 sec):
    *The field engineer repaired two or three disks.*

14. Conjoined complex object (2.0/4.3 sec):
    *She attempted to adjust the disk and to replace the motor.*
(15) Conjoined participle or tensed verb (2 parses: 1.9/3.9/5.5 sec):
The field engineer has repaired and adjusted the drive.

This paper has profited greatly from comments by Deborah Dahl, John Dowding, Marcia Linebarger, Donald McKay, Martha Palmer, Naomi Sager, and Rebecca Shiffman, as well as Michael McCord and several anonymous reviewers. Much of the original implementation of Restriction Grammar was done by Karl Puder, currently at Digital Equipment Corporation. John Dowding has been responsible for subsequent maintenance of the system, including much of the recent work on the dynamic translator, as well as numerous enhancements to the system implementation and interfaces. Marcia Linebarger is primarily responsible for the treatment of relative clauses and questions, and their integration with the conjunction mechanism. Margaret Heineman is responsible for the trees in Appendix D.

REFERENCES