New Approaches to Parsing Conjunctions Using Prolog
Sandiway Fong
Robert C. Berwick
Artificial Intelligence Laboratory
M.I.T.
545 Technology Square
Cambridge MA 02139, U.S.A.

Abstract
Conjunctions are particularly difficult to parse in traditional, phrase-based grammars. This paper shows how a different representation, not based on tree structures, markedly improves the parsing problem for conjunctions. It modifies the union of phrase marker model proposed by Goodall [1981], where conjunction is considered as the linearization of a three-dimensional union of a non-tree based phrase marker representation. A PROLOG grammar for conjunctions using this new approach is given. It is far simpler and more transparent than a recent phrase-based extraposition parser conjunctions by Dahl and McCord [1984]. Unlike the Dahl and McCord or ATN SYSCONJ approach, no special trail machinery is needed for conjunction, beyond that required for analyzing simple sentences. While of comparable efficiency, the new approach unifies under a single analysis a host of related constructions: respectively sentences, right node raising, or gapping. Another advantage is that it is also completely reversible (without cuts), and therefore can be used to generate sentences.

Introduction
The problem addressed in this paper is to construct a grammatical device for handling coordination in natural language that is well founded in linguistic theory and yet computationally attractive. The linguistic theory should be powerful enough to describe all of the phenomenon in coordination, but also constrained enough to reject all ungrammatical examples without undue complications. It is difficult to achieve such a fine balance - especially since the term grammatical itself is highly subjective. Some examples of the kinds of phenomenon that must be handled are shown in fig. 1
John and Mary went to the pictures
Simple constituent coordination
The fox and the hound lived in the fox hole and
kennel respectively
Constituent coordination with the 'respectively' reading
John and I like to program in Prolog and Hope
Simple constituent coordination but can have a collective or respectively reading
John likes but I hate bananas
Non-constituent coordination
Bill designs cars and Jack aeroplanes
Gapping with 'respectively' reading
The fox, the hound and the horse all went to market
Multiple conjuncts
*John sang loudly and a carol
Violation of coordination of likes
*Who did Peter see and the car?
Violation of coordinate structure constraint
*I will catch Peter and John might the car
Gapping, but component sentences contain unlike auxiliary verbs
?The president left before noon and at 2, Gorbachev

Fig 1: Example Sentences

The goal of the computer implementation is to produce a device that can both generate surface sentences given a phrase marker representation and derive a phrase marker representation given a surface sentences. The implementation should be as efficient as possible whilst preserving the essential properties of the linguistic theory. We will present an implementation which is transparent to the grammar and perhaps cleaner & more modular than other systems such as the interpreter for the Modifier Structure Grammars (MSGs) of Dahl & McCord [1983].

The MSG system will be compared with a simplified implementation of the proposed device. A table showing the execution time of both systems for some sample sen-
tences will be presented. Furthermore, the advantages and
disadvantages of our device will be discussed in relation to
the MSG implementation.

Finally we can show how the simplified device can
be extended to deal with the issues of extending the sys-
tem to handle multiple conjuncts and strengthening the
constraints of the system.

The RPM Representation

The phrase marker representation used by the theory
described in the next section is essentially that of the
Reduced Phrase Marker (RPM) of Lasnik & Kupin [1977]. A
reduced phrase marker can be thought of as a set consist-
ing of monostings and a terminal string satisfying certain
predicates. More formally, we have (fig. 2):

Let \( \Sigma \) and \( N \) denote the set of terminals and
non-terminals respectively.

Let \( \varphi, \psi, \chi \in (\Sigma \cup N)^* \).

Let \( x, y, z \in \Sigma^* \).

Let \( A \) be a single non-terminal.

Let \( \mathcal{P} \) be an arbitrary set.

Then \( \varphi \) is a monosting w.r.t. \( \Sigma \& N \) if \( \varphi \in \Sigma^* N \Sigma^* \).

Suppose \( \varphi = xAz \) and that \( \varphi, \psi \in \mathcal{P} \) where \( \mathcal{P} \)
is a some set of strings. We can also define the
following predicates:

- \( y \) isa* \( \varphi \) in \( \mathcal{P} \) if \( xyz \in \mathcal{P} \)
- \( \varphi \) dominates \( \psi \) in \( \mathcal{P} \) if \( \psi = zxy, x \neq \emptyset \) and
  \( x \neq A \).
- \( \varphi \) precedes \( \psi \) in \( \mathcal{P} \) if \( \exists y \) s.t. \( y \) isa* \( \varphi \) in \( \mathcal{P} \)
  \( \psi = zyx \) and \( x \neq z \).

Then:

- \( \mathcal{P} \) is an RPM if \( \exists A, x \) s.t. \( A, x \in \mathcal{P} \) and
  \( \forall \{\psi, \varphi\} \subseteq \mathcal{P} \) then
- \( \psi \) dominates \( \varphi \) in \( \mathcal{P} \) or \( \varphi \) dominates \( \psi \) in \( \mathcal{P} \)
or \( \psi \) precedes \( \varphi \) in \( \mathcal{P} \) or \( \varphi \) precedes \( \psi \) in \( \mathcal{P} \).

This representation of a phrase marker is equiva-
lent to a proper subset of the more common syntactic tree
representation. This means that some trees may not be
representable by an RPM and all RPMs may be re-cast as
trees. (For example, trees with shared nodes representing
overlapping constituents are not allowed.) An example of
a valid RPM is given in fig. 3:

Sentence: Alice saw Bill

RPM representation:

\{ S, Alice.saw.Bill, NP saw.Bill, Alice.V.Bill, Alice.VP, Alice.saw.NP \}

Fig 3: An example of RPM representation

This RPM representation forms the basis of the
linguistic theory described in the next section. The set
representation has some desirable advantages over a tree
representation in terms of both simplicity of description
and implementation of the operations.

Goodall's Theory of Coordination

Goodall's idea in his draft thesis [Goodall??] was to
extend the definition of Lasnik and Kupin's RPM to cover
coordination. The main idea behind this theory is to ap-
ply the notion that \( \text{coordination results from the union of phrase markers} \) to the reduced phrase marker. Since RPMs
are sets, this has the desirable property that the union of
RPMs would just be the familiar set union operation. For
a computer implementation, the set union operation can be
realized inexpensively. In contrast, the corresponding op-
eration for trees would necessitate a much less simple and
efficient union operation than set union.

However, the original definition of the RPM did
not envisage the union operation necessary for coordina-
tion. The RPM was used to represent 2-dimensional struc-
ture only. But under set union the RPM becomes a rep-
resentation of 3-dimensional structure. The admissibility
predicates dominates and precedes defined on a set of
monostings with a single non-terminal string were inade-
quate to describe 3-dimensional structure.

Basically, Goodall's original idea was to extend the
domines and precedes predicates to handle RPMs un-
der the set union operation. This resulted in the relations
de-dominates and e-precedes as shown in fig. 4:

119
Assuming the definitions of fig. 2 and in addition let \( \omega, \Omega, \Theta \in (\Sigma \cup N)^* \) and \( q, r, s, t, v \in \Sigma^* \). then :

- \( \varphi \) e-dominates \( \psi \) in \( P \) if \( \varphi \) dominates \( \psi' \) in \( P \). \( \chi \omega = \psi' \). \( \Theta \omega = \psi \) and \( z \equiv y \) in \( P \).
- \( \varphi \) e-precedes \( \psi \) in \( P \) if \( y \) isa \( \varphi \) in \( P \). \( v \) isa \( \psi \)

where the relation \( \equiv \) (terminal equivalence) is defined as :

\[ z \equiv y \text{ in } P \text{ if } x \omega \in P \text{ and } \chi \omega \in P \]

**Figure 4: Extended definitions**

The above example indicates that the extended RPM definition of Goodall allows some ungrammatical sentences to slip through. Although the device presented in the next section doesn't make direct use of the extended definitions, the notion of equivalence is central to the implementation. The basic system described in the next section does have this deficiency but a less simplistic version described later is more constrained - at the cost of some computational efficiency.

**Linearization and Equivalence**

Although a theory of coordination has been described in the previous sections - in order for the theory to be put into practice, there remain two important questions to be answered :-

- How to produce surface strings from a set of sentences to be conjoined?
- How to produce a set of simple sentences (i.e. sentences without conjunctions) from a conjoined surface string?

This section will show that the processes of linearization and finding equivalences provide an answer to both questions. For simplicity in the following discussion, we assume that the number of simple sentences to be conjoined is two only.

The processes of linearization and finding equivalences for generation can be defined as :-

Given a set of sentences and a set of candidates which represent the set of conjoinable pairs for those sentences. linearization will output one or more surface strings according to a fixed procedure.

Given a set of sentences. finding equivalences will produce a set of conjoinable pairs according to the definition of equivalence of the linguistic theory.

For generation the second process (finding equivalences) is called first to generate a set of candidates which is then used in the first process (linearization) to generate the surface strings. For parsing, the definitions still hold - but the processes are applied in reverse order.

To illustrate the procedure for linearisation, consider the following example of a set of simple sentences (fig. 6) :-
\{ John liked ice-cream, Mary liked chocolate \}
\textit{set of simple sentences}

\{ \{ John, Mary \}, \{ ice-cream, chocolate \} \}
\textit{set of conjoinable pairs}

\textit{Fig 6: Example of a set of simple sentences}

Consider the plan view of the 3-dimensional representation of the union of the two simple sentences shown in fig. 7:

\[ \begin{align*}
\text{John} & \quad \text{liked} \quad \text{ice-cream} \\
\text{Mary} & \quad \text{liked} \quad \text{chocolate}
\end{align*} \]
\textit{Fig 7: Example of 3-dimensional structure}

The procedure of linearization would take the following path shown by the arrows in fig. 8:

\[ \begin{align*}
\text{John} & \quad \text{liked} \quad \text{ice-cream} \\
\text{Mary} & \quad \text{liked} \quad \text{chocolate}
\end{align*} \]
\textit{Fig 8: Example of linearization}

Following the path shown we obtain the surface string "John and Mary liked ice-cream and chocolate".

The set of conjoinable pairs is produced by the process of finding equivalences. The definition of equivalence as given in the description of the extended RPM requires the generation of the combined RPM of the constituent sentences. However it can be shown [by comparing the constraints imposed by the definitions of equivalence and linearization, that the same set of equivalent terminal strings can be produced just by using the terminal strings of the RPM alone. There are considerable savings of computational resources in not having to compare every element of the set with every other element to generate all possible equivalent strings - which would take $O(n^2)$ time - where $n$ is the cardinality of the set. The corresponding term for the modified definition (given in the next section) is $O(1)$.

\section*{The Implementation in Prolog}

This section describes a runnable specification written in Prolog. The specification described also forms the basis for comparison with the MSG interpreter of Dahl and McCord. The syntax of the clauses to be presented is similar to the Dec-10 Prolog [Bowen et al. 1982] version. The main differences are:

- The symbols `\text{"-"} and `\text{","} have been replaced by the more meaningful reserved words `\text{"if"} and `\text{"and"} respectively.
- The symbol `\text{"."} is used as the list constructor and `\text{"nil"} is used to represent the empty list.
- An an example, a Prolog clause may have the form:

\[ a(X \ Y \ ... \ Z) \text{ if } b(U \ V \ ... \ W) \text{ and } c(R \ S \ ... \ T) \]

where $a, b, \text{ and } c$ are predicate names and $R, S, ..., T$ may represent variables, constants or terms. (Variables are distinguished by capitalization of the first character in the variable name.) The intended logical reading of the clause is:

"$a$ holds if $b$ and $c$ both hold for consistent bindings of the arguments $X, Y, ..., Z, U, V, ..., W, R, S, ..., T$"

- Comments (shown in italics) may be interspersed between the arguments in a clause.

\section*{Parse and Generate}

In the previous section the processes of linearization and finding equivalences are described as the two components necessary for parsing and generating conjoined sentences. We will show how these processes can be combined to produce a parser and a generator. The device used for comparison with Dahl & McCord scheme is a simplified version of the device presented in this section.

First, difference lists are used to represent strings in the following sections. For example, the pair (fig. 9):
is a difference list representation of the sentence "John liked ice-cream".

We can now introduce two predicates `linearize` and `equivalentpairs` which correspond to the processes of linearization and finding equivalences respectively (fig. 10):

```prolog
linearize( pairs S1 E1 and S2 E2 candidates Set gives Sentence)
```

`Linearize` holds when a pair of difference lists `([S1. E1] & [S2. E2])` and a set of candidates `Set` are consistent with the string `Sentence` as defined by the procedure given in the previous section.

```prolog
equivalentpairs( X Y from S1 S2)
```

`Equivalentpairs` holds when a substring `X` of `S1` is equivalent to a substring `Y` of `S2` according to the definition of equivalence in the linguistic theory.

Additionally, let the meta-logical predicate `setof` as in "setof(Element Goal Set)" hold when `Set` is composed of elements of the form `Element` and that `Set` contains all instances of `Element` that satisfy the goal `Goal`. The predicates `generate` can now be defined in terms of these two processes as follows (fig. 11):

```prolog
generate(Sentence from S1 S2)
if setof(X.Y.nil in equivalentpairs(X Y from S1 S2) is Set)
and linearize( pairs S1 nil and S2 nil candidates Set gives Sentence)
```

The definitions for parsing and generating are almost logically equivalent. However the sub-goals for parsing are in reverse order to the sub-goals for generating - since the Prolog interpreter would attempt to solve the sub-goals in a left to right manner. Furthermore, the subset relation rather than set equality is used in the definition for parsing. We can interpret the two definitions as follows (fig. 12):

```prolog
Generate holds when Sentence is the conjoined sentence resulting from the linearization of the pair of difference lists `([S1. nil] & [S2. nil])` using as candidate pairs for conjoining, the set of non-redundant pairs of equivalent terminal strings `Set`.
```

```prolog
Parse holds when Sentence is the conjoined sentence resulting from the linearization of the pair of difference lists `([S1. E1] & [S2. E2])` provided that the set of candidate pairs for conjoining `Subset` is a subset of the set of pairs of equivalent terminal strings `Set`.
```

The subset relation is needed for the above definition of parsing because it can be shown [Fong??] that the process of linearization is more constrained (in terms of the permissible conjoinable pairs) than the process of finding equivalences.

**Linearize**

We can also fashion a logic specification for the process of linearization in the same manner. In this section we will describe the cases corresponding to each Prolog clause necessary in the specification of linearization. However, for simplicity the actual Prolog code is not shown here. (See Appendix A for the definition of predicate `linearize`.)

In the following discussion we assume that the template for predicate `linearize` has the form "linearize( pairs S1 E1 and S2 E2 candidates Set gives Sentence)" shown previously in fig. 10. There are three independent cases to consider during linearization:

1. **The Base Case.**

   If the two difference lists `([S1. E1] & [S2. E2])` are both empty then the conjoined string `Sentence` is also empty. This simply states that if two empty strings are conjoined then the result is also an empty string.
2. Identical Leading Substrings.
The second case occurs when the two (non-empty) difference lists have identical leading non-empty substrings. Then the conjoined string is identical to the concatenation of that leading substring with the linearization of the rest of the two difference lists. For example, consider the linearization of the two fragments “likes Mary” and “likes Jill” as shown in fig. 13:

\{likes Mary, likes Jill\}

which can be linearized as:

\{likes X\}

where X is the linearization of strings \{Mary, Jill\}

Fig. 13: Example of identical leading substrings

3. Conjoining.
The last case occurs when the two pairs of (non-empty) difference lists have no common leading substring. Here, the conjoined string will be the concatenation of the conjunction of one of the pairs from the candidate set, with the conjoined string resulting from the linearization of the two strings with their respective candidate substrings deleted. For example, consider the linearization of the two sentences “John likes Mary” and “Bill likes Jill” as shown in fig. 14:

\{John likes Mary, Bill likes Jill\}

given that the selected candidate pair is \{John, Bill\},
the conjoined sentence would be:

\{John and Bill X\}

where X is the linearization of strings \{likes Mary, likes Jill\}

Fig. 14: Example of conjoining substrings

Finding Equivalences

Goodall's definition of equivalence was that two terminal strings were said to be equivalent if they had the same left and right contexts. Furthermore we had previously asserted that the equivalent pairs could be produced without searching the whole RI*P. For example consider the equivalent terminal strings in the two sentences “Alice saw Bill” and “Mary saw Bill” (fig. 16):

\{Alice saw Bill, Mary saw Bill\}

would produce the equivalent pairs:

\{Alice saw Bill, Mary saw Bill\}
\{Alice, Mary\}
\{Alice saw, Mary saw\}

Fig. 16: Example of equivalent pairs

We also make the following restrictions on Goodall’s definition:

what linearizations the system would produce for an example sentence. Consider the sentence “John and Bill liked Mary” (fig. 15):

\{John and Bill liked Mary\}

would produce the strings:

\{John and Bill liked Mary, John and Bill liked Mary\}
with candidate set {};
\{John liked Mary, Bill liked Mary\}
with candidate set \{(John, Bill)\};
\{John Mary, Bill liked Mary\}
with candidate set \{(John, Bill liked)\};
\{John, Bill liked Mary\}
with candidate set \{(John, Bill liked Mary)\}

Fig. 15: Example of linearizations
• If there exists two terminal strings $X \& Y$ such that $X=x\Omega$ & $Y=y\Omega$, then $x$ & $\Omega$ should be the strongest possible left & right contexts respectively - provided $x$ & $y$ are both nonempty. In the above example, $x=nil$ and $\Omega="saw Bill"$, so the first and the third pairs produced are redundant.

In general, a pair of terminal strings are redundant if they have the form $(uv, uw)$ or $(uv, zw)$, in which case - they may be replaced by the pairs $(v, w)$ and $(u, z)$ respectively.

• In Goodall's definition any two terminal strings themselves are also a pair of equivalent terminal strings (when $x$ & $\Omega$ are both null). We exclude this case it produces simple string concatenation of sentences.

The above restrictions imply that in fig. 16 the only remaining equivalent pair ({Alice, Mary})is the correct one for this example.

However, before finding equivalent pairs for two simple sentences, the process of finding equivalences must check that the two sentences are actually grammatical. We assume that a recognizer/parser (e.g. a predicate parse(S E)) already exists for determining the grammaticality of simple sentences. Since the process only requires a yes/no answer to grammaticality, any parsing or recognition system for simple sentences can be used.

We can now specify a predicate findcandidates($X \& Y$ $S1$ $S2$) that holds when ($X, Y$) is an equivalent pair from the two grammatical simple sentences ($S1, S2$) as follows (fig. 17):

\[
\text{findcandidates}(X \& Y \text{ in } S1 \text{ and } S2) \newline
\text{if parse}(S1 \text{ nil}) \\
\text{and parse}(S2 \text{ nil}) \\
\text{and equiv}(X \& Y \text{ in } S1 \& S2) \\
\]

where equiv is defined as :

\[
\text{equiv}(X \& Y \text{ in } S1 \& S2) \\
\text{if append3}(\text{Chi} \& X \Omega \text{ga} \text{Xi}) \\
\text{and terminals}(X) \\
\text{and append3}(\text{Chi} \& Y \Omega \text{ga} \text{Yi}) \\
\text{and terminals}(Y) \\
\]

where append3(L1 L2 L3 L4) holds when $L4$ is equal to the concatenation of $L1, L2, L3$. terminals(X) holds when $X$ is a list of terminal symbols only.

Fig.17: Logic definition of Findcandidates

Then the predicate findequivalences is simply defined as (fig. 18):

\[
\text{findequivalences}(X \& Y \text{ in } S1 \& S2) \newline
\text{if findcandidates}(X \& Y \text{ in } S1 \& S2) \newline
\text{and not redundant}(X \& Y) \\
\]

where redundant implements the two restrictions described.

Fig.18: Logic definition of Findequivalences

Comparison with MSGs

The following table (fig. 19) gives the execution times in milliseconds for the parsing of some sample sentences mostly taken from Dahl & McCord [1983]. Both systems were executed using Dec-20 Prolog. The times shown for the MSG interpreter is based on the time taken to parse and build the syntactic tree only - the time for the subsequent transformations was not included.

| Sample sentences                      | MSG system | RPM device |
|---------------------------------------|------------|------------|
| Each man ate an apple and a pear     | 662        | 292        |
| John ate an apple and a pear          | 613        | 233        |
| A man and a woman saw each train      | 319        | 506        |
| Each man and each woman ate an apple  | 320        | 503        |
| John saw and the woman heard a man that laughed | 788 | 834 |
| John drove the car through and completely demolished a window | 275 | 1032 |
| The woman who gave a book to John and drove a car through a window laughed | 1007 | 3375 |
| John saw the man that Mary saw and Bill gave a book to laughed | 439 | 311 |
| John saw the man that heard the woman that laughed and saw Bill | 636 | 323 |
| The man that Mary saw and heard gave an apple to each woman | 501 | 982 |

Fig.19: Timings for some sample sentences

From the timings we can conclude that the proposed device is comparable to the MSG system in terms of computational efficiency. However, there are some other advantages such as :

• Transparency of the grammar - There is no need for phrasal rules such as "S $\rightarrow$ S and S". The device also allows non-phrasal conjunction.

• Since no special grammar or particular phrase marker representation is required, any parser can be used - the device only requires an accept/reject answer.
• The specification is not biased with respect to lying or generation. The implementation is reversible allowing it to generate any sentence it can parse and vice versa.

• Modularity of the device. The grammaticality of sentences with conjunction is determined by the definition of equivalence. For instance, if needed we can filter the equivalent terminals using semantics.

A Note on SYSCONJ

It is worthwhile to compare the phrase marker approach to the ATN-based SYSCONJ mechanism. Like SYSCONJ, our analysis is extragrammatical: we do not tamper with the basic grammar, but add a new component that handles conjunction. Unlike SYSCONJ, our approach is based on a precise definition of "equivalent phrases" that attempts to unify under one analysis many different types of coordination phenomena. SYSCONJ relied on a rather complicated, interrupt-driven method that restarted sentence analysis in some previously recorded machine configuration, but with the input sequence following the conjunction. This captured part of the "multiple planes" analysis of the phrase marker approach, but without a precise notion of equivalent phrases. Perhaps as a result, SYSCONJ handled only ordinary conjunction, and not respectively or gapping readings. In our approach, a simple change to the linearization process allows us to handle gapping.

Extensions to the Basic Device

The device described in the previous section is a simplified version for rough elucidation with the MSG interpreter. However, the system can easily be generalized to handle multiple conjuncts. The only additional phase required is to generate templates for multiple readings. Also, gapping can be handled just by adding clauses to the definition of linearize - which allows a different path from that of fig. 8 to be taken.

The simplified device permits some examples of ungrammatical sentences to be parsed as if correct (fig. 5). The modularity of the system allows us to constrain the definition of equivalence still further. The extended definitions in Goodall's draft theory were not included in his thesis (Goodall84) presumably because it was not constrained enough. However, in his thesis he proposes another definition of grammaticality using RPMs. This definition can be used to constrain equivalence still further in our system at a loss of some efficiency and generality. For example, the required additional predicate will need to make explicit use of the combined RPM. Therefore, a parser will need to produce a RPM representation as its phrase marker. The modifications necessary to produce the representation is shown in appendix B.

Acknowledgements

This work describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. The first author is also funded by a scholarship from the Kennedy Memorial Trust.

References

Bowen et al: D.L. Bowen (ed.), L. Byrd, F.C.N. Pereira, L.M. Pereira, D.H.D. Warren. Decsystem-10 Prolog User’s Manual. University of Edinburgh. 1982.

Dahl & McCord: V. Dahl and M.C. McCord. Treating Coordination in Logic Grammars. American Journal of Computational Linguistics. Vol. 9, No. 2. (1983).

Fong?: Sandiway Fong. To appear in S.M. thesis - "Specifying Coordination in Logic" - 1986

Goodall??: Grant Todd Goodall. Draft - Chapter 2 (sections 2.1. to 2.7): Coordination.

Goodall84: Grant Todd Goodall. Parallel Structures in Syntax. Ph.D thesis, University of California, San Diego (1984).

Lasnik & Kuper: H. Lasnik and I. Kuper. A restrictive theory of transformational grammar. Theoretical Linguistics 4 (1977).

Appendix A: Linearization

The full Prolog specification for the predicate linearize is given below.

/ Linearize for generation /
/ terminating condition /
linearize(pairs S1 S1 and S2 S2 candidates last giving nil) if nonvar(List)
/ applicable when we have a common substring /
linearize(pairs S1 E1 and S2 E2 candidates List giving Sentence) if var(Sentence)
and not same(S1 as E1)
and not same(S2 as E2)
and similar(S1 to S2 common Similar)
and not same(S1 as S2)
and remove(Similar from S1 leaving NewS1)
and remove(Similar from S2 leaving NewS2)
and linearize(pairs NewS1 E1 and NewS2 E2
candidates List giving RestOfSentence)
and append(Similar RestOfSentence Sentence)

/ linearize for parsing /
linearize(pairs nil nil and nil nil
candidates List giving Sentence)
if var(List)
and same(List as nil)
/ Case for common substring /
linearize(pairs Common,NewS1 nil and Common,NewS2 nil
candidates List giving Sentence)
if not var(Sentence)
and same(Common,RestOfSentence as Sentence)
and linearize(pairs NewS1 nil and NewS2 nil
candidates NewList giving RestOfSentence)

/ Case for conjoin /
linearize(pairs S1 nil and S2 nil
candidates Element,Rest giving Sentence)
if not var(Sentence)
and append(Conjoined RestOfSentence giving Sentence)
and append(list Cand1,Common Element and Cand2 Conjoined)
and not same(Cand1 as nil)
and not same(Cand2 as nil)
and linearize(pairs NewS1 nil and NewS2 nil
candidates Rest giving RestOfSentence)
and append(Cand1 NewS1 S1)
and append(Cand2 NewS2 S2)

/ append is a special form of append such that
the first list must be non-empty
append(Head,Tail to Tail giving Head,Tail)
append(First,Second,Others to Tail giving First,Rest)
if append(Second,Others to Tail giving First,Rest)
similar(nil to nil common nil)
similar(Head, Tail to Head, Tail common Head, Rest)
if not same(Head1 as Head2)
similar(Head, Tail to Head, Tail common Head, Rest)
if similar(Tail to Tail common Rest)

/ conjoin is reversible /
conjoin(list First,Second nil using Conjoined giving Conjoined)
if not var(First)
and nonvar(Second)
and append(First Conjoined,Second Conjoined)
conjoin(list First,Second nil using Conjoined giving Conjoined)
if not var(Conjoined)
and append(First Conjoined,Second Conjoined)
remove(nil from List leaving List)
delete(Head from nil leaving nil)
delete(Head from Head, Tail leaving Tail)
delete(Head from First, Rest leaving First, Tail)
if not same(Head as First)
and delete(Head from Rest leaving Tail)

Appendix B: Building the RPM

A RPM representation can be built by adding three extra parameters to each grammar rule together with a call to a concatenation routine. For example, consider the simple sentence “liked Mary” from the form: “Mary” to “liked Mary”. The non-string corresponding to the non-terminal VP is constructed by taking the left and right contexts of “liked Mary” and placing the non-terminal symbol VP between them. In general, we have something of the form :-

phrase( from Point1 to Point2
using Start to End giving MS, RPM)
and buildmonostring(Start Point1 plus 'VP'
Point2 End MS)

where difference pairs {Start, Point1} and {Point2, End} represent the left context, the right context and the sentence string respectively. The concatenation routine buildmonostring is just :-

buildmonostring(Start Point1 plus NonTerminal
Point2 End MS)
if append(Start Left Start)
and append(Point2 Right End)
and append(Left NonTerminal, Right MS)