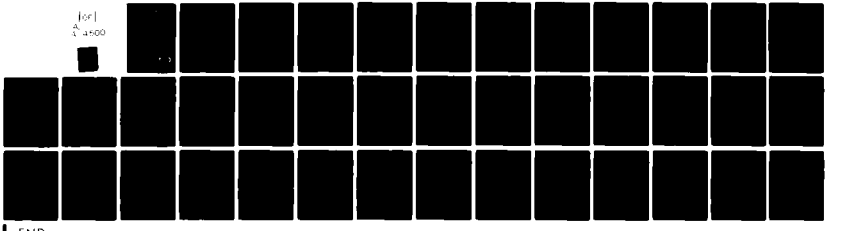
AD-A114 500 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 5/7
COPING WITH SYNTACTIC AMBIGUITY OR HOW TO PUT THE BLOCK IN THE --ETC(U)
APR 82 K CHURCH, R PATIL N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TM-216 NL



END
DATE FILMED
6-82
DTIC

(12)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-216

COPING WITH SYNTACTIC AMBIGUITY

OR

HOW TO PUT THE BLOCK IN THE BOX ON THE TABLE

Kenneth Church

Ramesh Patil

April 1982

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MIT/LCS/TM-216	2. GOVT ACCESSION NO.: AD-A114500
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): COPING WITH SYNTACTIC AMBIGUITY OR HOW TO PUT THE BLOCK IN THE BOX ON THE TABLE.	5. TYPE OF REPORT & PERIOD COVERED: Memorandum, April 1982
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TM-216
7. AUTHOR(s): Kenneth Church and Ramesh Patil	8. CONTRACT OR GRANT NUMBER(s): DARPA N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Laboratory for Computer Science M.I.T. 545 Technology Sq. Cambridge, Ma. 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA 1400 Wilson Blvd., Arlington, Va. 22217	12. REPORT DATE: April 1982 13. NUMBER OF PAGES: 36
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research Department of the Navy Information Systems Program Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

THIS DOCUMENT IS APPROVED FOR PUBLIC SALE AND RELEASE.
DISTRIBUTION IS UNLIMITED.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

natural language, parsing, ambiguity

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Sentences are far more ambiguous than one might have thought. There may be hundreds, perhaps thousands of syntactic parse trees for certain very natural sentences of English. This fact has been a major problem confronting natural language processing because it indicates that it may require a long time to construct a list of all the parse trees, and furthermore, it isn't clear what to do with the list once it has been constructed. This list may be so numerous that it is probably not the most convenient representation for communication

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

(n superscript 3)

with the semantic and pragmatic processing modules. In this paper we propose some methods for dealing with syntactic ambiguity in ways that take advantage of certain regularities among the alternative parse trees. These regularities will be expressed as linear combinations of ATN networks, and also as sums and products of formal power series.

We will suggest some ways that practical processor can take advantage of this modularity in order to deal more efficiently with combinatoric ambiguity. In particular, we will show how a processor can efficiently compute the ambiguity of an input sentence (or any portion thereof). Furthermore, we will show how to compile certain grammers into a form that can be processed more efficiently. In some cases, including the "every way ambiguous" grammer (e.g., conjunction, prepositional phrases, noun-noun modification), processing time will be reduced from 0 (n^3) to 0 (n). Finally, we will show how to uncompile certain highly optimized grammars into a form suitable for linguistic analysis.

Coping with Syntactic Ambiguity
or
How to Put the Block in the Box on the Table

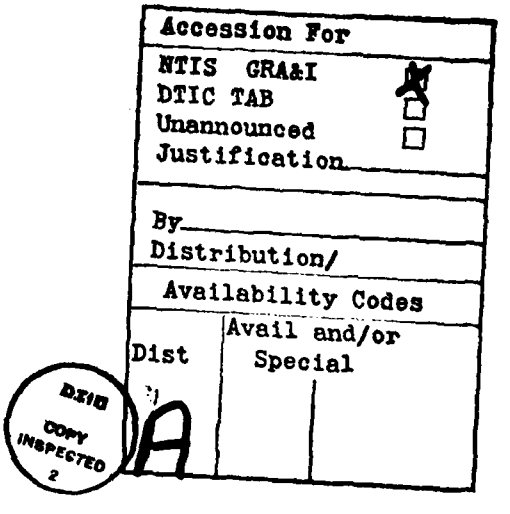
Kenneth Church

Ramesh Patil

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts 02139



This research was supported (in part) by the National Institutes of Health Grant No. 1 P01 LM 03374-02 from the National Library of Medicine, and by the Defense Advanced Research Projects Agency (DOD) monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

Abstract

Sentences are far more ambiguous than one might have thought. There may be hundreds, perhaps thousands of syntactic parse trees for certain very natural sentences of English. This fact has been a major problem confronting natural language processing because it indicates that it may require a long time to construct a list of all the parse trees, and furthermore, it isn't clear what to do with the list once it has been constructed. This list may be so numerous that it is probably not the most convenient representation for communication with the semantic and pragmatic processing modules. In this paper we propose some methods for dealing with syntactic ambiguity in ways that take advantage of certain regularities among the alternative parse trees. These regularities will be expressed as linear combinations of ATN networks, and also as sums and products of formal power series.

We will suggest some ways that practical processor can take advantage of this modularity in order to deal more efficiently with combinatoric ambiguity. In particular, we will show how a processor can efficiently compute the ambiguity of an input sentence (or any portion thereof). Furthermore, we will show how to compile certain grammars into a form that can be processed more efficiently. In some cases, including the "every way ambiguous" grammar (e.g., conjunction, prepositional phrases, noun-noun modification), processing time will be reduced from $O(n^3)$ to $O(n)$. Finally, we will show how to uncompile certain highly optimized grammars into a form suitable for linguistic analysis.

Keywords: natural language, parsing, ambiguity

CONTENTS

1. Ambiguity is a Practical Problem 5

2. Formal Power Series 8

3. Catalan Numbers 11

4. Table Lookup 12

5. Parallel Decomposition 14

6. Series Decomposition 16

    6.1 Auto-Convolution of Catalan Grammars 18
    6.2 Chart Parsing 20
    6.3 Auto-Convolution of Unit Step Grammars 21

7. Computing the Power Series Directly from the Grammar 24

8. Computing the Power Series from the ATN 25

9. An Example 27

10. Lexical Restrictions 30

11. Inverse Transforms 31

12. Conclusion 33

13. Acknowledgments 35

    References 35

Most parsers find the set of parse trees by starting with the empty set and adding to it each time they find a new possibility. We make the observation that in certain situations it would be much more efficient to work in the other direction, starting from the universal set (i.e., the set of all binary trees) and ruling trees out when the parser decides that they cannot be parses. Ruling-out is easier when the set of parse trees is closer to the universal set and ruling-in is easier when the set of parse trees is closer to the empty set. Ruling-out is particularly suited for "every way ambiguous" constructions such as prepositional phrases which have just as many parse trees as there are binary trees over the terminal elements. Since every tree is a parse, the parser doesn't have to rule any of them out.

In some sense, this is a formalization of an idea that has been in the literature for some time. That is, it has been noticed for a long time that these sorts of very ambiguous constructions are very difficult for most parsing algorithms, but (apparently) not for people. This observation has led some researchers to hypothesize additional parsing mechanisms, such as pseudo-attachment [1: pp. 65-71][1] and permanent predictable ambiguity [14: pp. 64-65], so that the parser could "attach all ways" in a single step. However, these mechanisms have always lacked a precise interpretation; we will present a much more formal way of coping with "every way ambiguous" grammars, defined in terms of *Catalan numbers* [8: pp. 388-389, pp. 531-533].

Certain constructions, including the "every way ambiguous" grammar, will be treated as primitive objects (modules) which can be combined in various ways to produce composite constructions such as lexical ambiguity which are also very ambiguous, but not quite "every way ambiguous". Composite constructions will be analyzed as linear combinations of primitive components, in a sense to be made precise in terms of formal power series. Equivalently, in ATN notation, composite networks can be analyzed as series and parallel combinations of primitive networks. This approach has been strongly influenced by linear systems theory, a classic engineering notion of modularity.

We will suggest some ways that practical processor can take advantage of this modularity in order to deal more efficiently with combinatoric ambiguity. In particular, we will show how a processor can efficiently compute the ambiguity of an input sentence (or any portion thereof). Furthermore, we will show how to compile certain grammars into a form that can be processed more efficiently. In some cases, including the "every way ambiguous grammar", processing time will be reduced from $O(n^3)$ to $O(n)$. Finally, we will show how to uncompile certain highly optimized grammars into a form suitable for linguistic analysis.

---

1. The idea of pseudo-attachment was first proposed by Marcus (private communication), though Marcus does not accept the formulation in [1].

1. Ambiguity is a Practical Problem

Sentences are far more ambiguous than one might have thought. There may be hundreds, perhaps thousands of syntactic parse trees for certain very natural sentences of English. For example, consider the following sentence with two prepositional phrases:

(1) Put the block in the box on the table.

which has two interpretations:

(2a) Put the block [in the box on the table].
(2b) Put [the block in the box] on the table.

These syntactic ambiguities grow "combinatorially" with the number of prepositional phrases. For example, when a third PP is added to the sentence above, there are five interpretations:

(3a) Put the block [[in the box on the table] in the kitchen].
(3b) Put the block [in the box [on the table in the kitchen]].
(3c) Put [[the block in the box] on the table] in the kitchen.
(3d) Put [the block [in the box on the table]] in the kitchen.
(3e) Put [the block in the box] [on the table in the kitchen].

When a fourth PP is added, there are fourteen trees, and so on. This sort of combinatoric ambiguity has been a major problem confronting natural language processing because it indicates that it may require a long time to construct a list of all the parse trees, and furthermore, it isn't clear what to do with the list once it has been constructed. This list may be so numerous that it is probably not the most convenient representation for communication with the semantic and pragmatic processing modules. In this paper we propose some methods for dealing with syntactic ambiguity in ways that take advantage of certain regularities among the alternative parse trees.

In particular, we observe that enumerating the parse trees as above misses the very important generalization that prepositional phrases are "every way ambiguous", or more precisely the set of parse trees over *i* PPs is the same as the set of binary trees that can be constructed over *i* terminal elements. Notice, for example, that there are two possible binary trees over three elements,

(4a) [... block ... [... box ... table ...]]
(4b) [[... block ... box ...] ... table ...]

corresponding to (2a) and (2b) respectively, and that there are five binary trees over four elements corresponding to (3a)-(3c) respectively (see figure 1).

These "worst case" scenarios occur very often in practice, as indicated by our experience with the EQSP parser [11] on the Malhotra Corpus [10].[2] Almost 2% of the Malhotra Corpus has 300 or more interpretations according to EQSP. The sentences are given below with the number of parse trees. Note that the first sentence is almost a thousand ways ambiguous.

958 In as much as allocating costs is a tough job I would like to have the total costs related to each product.

692 For each plant give the ratio of 1973 to 1972 figures for each type of production cost and overhead cost.

654 Do you have a model to maximize contribution to the company subject to production and other constraints?

556 Give actual and budgeted operating costs for all plants, and actual and budgeted management salaries and interest costs.

512 Give me a breakdown of difference between list and average quoted price for each product for 1972 and 1973.

510 The intent of my question is to find out if you know if your accounting methods can relate the changes in sales to changes in your expense structures.

322 Display the difference between list price and actual costs (direct + overhead) divided by list price for plant 2 for the past four years.

382 What was the number of units of product 2 produced at plant 2 in 1973 times the unit price of product 2?

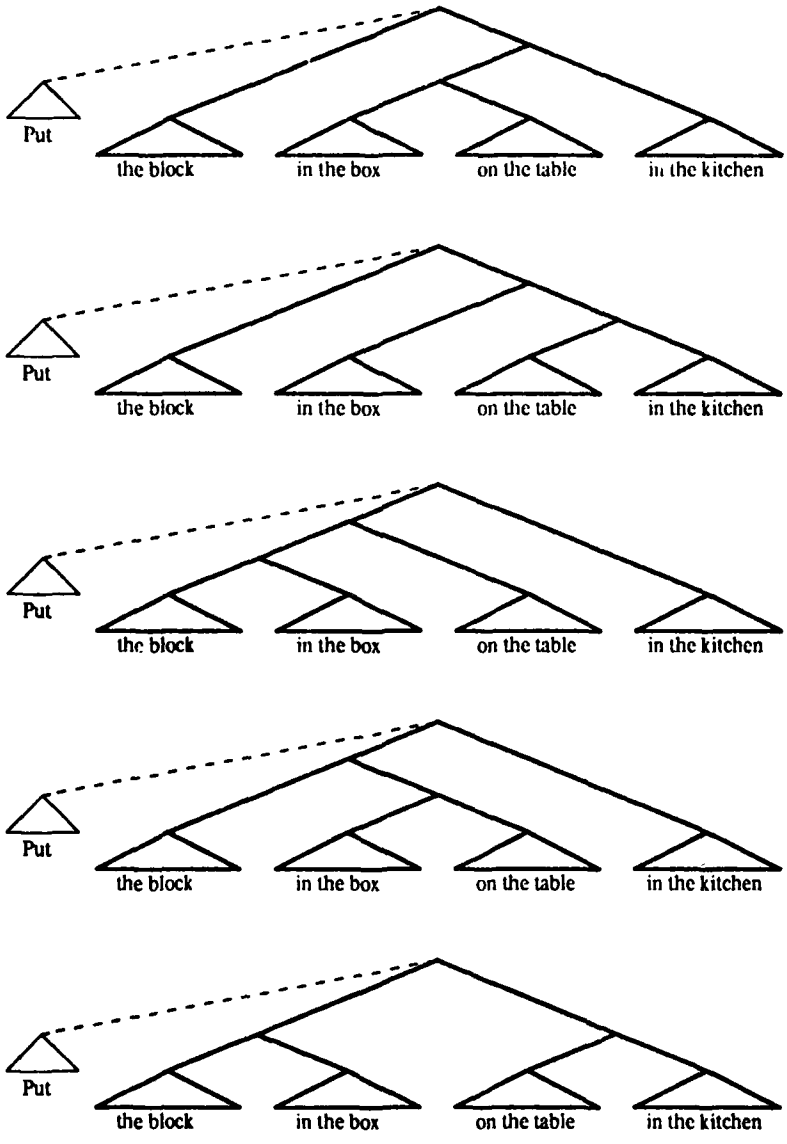
These sentences show that syntactic constraints are not always very restrictive. This fact has been a major problem confronting natural language processing because it indicates that it may require a long time to construct a list of all the parse trees, and furthermore, it isn't clear what to do with the list once it has been constructed. The list of parse trees can be so numerous that it is probably not the most efficient representation for communication with the semantic and pragmatic processing modules. A list representation fails to take advantage of certain generalizations among the alternative parse trees, especially the "every way ambiguous" generalization.

---

2. Malhotra gathered approximately 500 sentences in an experiment which fooled businessmen into believing that they were interacting with a computer when they were actually communicating with a person in an another room.

Fig. 1. Binary Trees Over Four Terminals

Over four terminals, it is possible to construct five binary trees. These five trees are illustrated below in solid lines.



The "every way ambiguous" generalization is missed by most parsing algorithms currently in practice including our own EQSP. These algorithms all construct the set of possible parse trees by starting from the empty set and adding to it each time they find a new set of analyses. We make the observation that there are certain situations where it would be much more efficient to work in the other direction, starting from the universal set and ruling trees out when the parser decides that they cannot be parses. Ruling-out is easier when the set of parse trees is closer to the universal set and ruling-in is easier when the set of parse trees is closer to the empty set. Ruling-out is particularly suited for "every way ambiguous" grammars like PPs because there are no trees to exclude. Similar comments hold for other "every way ambiguous" constructions such as adjuncts, conjuncts, noun-noun modification, and stacked relative clauses.

These constructions, which will be treated as primitive objects, can be combined in various ways to produce composite constructions such as lexical ambiguity which may also be very ambiguous, but not necessarily "every way ambiguous". Composite constructions can be analyzed as linear combinations of primitive components. Lexical ambiguity, for example, will be analyzed as the sum of its senses, or in flow graph terminology [13], as a parallel connection of its senses. Structural ambiguity, on the other hand, will be analyzed as the product of its components, or in flow graph terminology as a series connection. For example, the sentence

(5) Was the block in the box on the table?

is structurally ambiguous. The "box" can be associated with either the "block" or the "table". We will analyze this sentence as a product of two polynomials, the first corresponding to the subject noun phrase and the second corresponding to the complement noun phrase. The standard definition of polynomial multiplication correctly accounts for the two possible attachments of "box". We prefer this linear systems view to heuristic search strategies (e.g. [6]), because linear systems can capture generalizations that hold across alternative interpretations, whereas search strategies tend to probe only a single interpretation (context) at a time. At the very least, our approach is an improvement over enumerating each tree individually, which consumes exponential time in the worst case.

2. Formal Power Series

This section will make the linear systems analogy more precise by relating context-free grammars to formal power series (polynomials). Formal power series are a well-known device in the formal language literature (e.g. [15]) for developing the algebraic properties of context-free grammars. We introduce them here to establish a formal basis for our upcoming discussion of processing issues.

The power series for grammar (6a) is (6b).

(6a) NP → John | NP and NP

(6b) NP = John + John and John + 2 John and John and John
+ 5 John and John and John and John
+ 14 John and John and John and John and John + ...

Each term consists of a sentence generated by the grammar and an ambiguity coefficient[3] which counts how many ways the sentence can be generated. For example, the sentence "John" has one parse tree

(7a) [John] *1 tree*

because the zero-th coefficient of the power series is one. Similarly, the sentence "John and John" also has one tree because its coefficient is also one,

(7b) [John and John] *1 tree*

and "John and John and John" has two because its coefficient is two,

(7c) [[John and John] and John], [John and [John and John]] *2 trees*

and "John and John and John and John" has five,

(7d) [John and [[John and John] and John]], [John and [John and [John and John]]], *5 trees*
[[[John and John] and John] and John], [[John and [John and John]] and John],
[[John and John] and [John and John]]

and so on. The reader can verify for himself that "John and John and John and John and John" has fourteen trees.

Note that the power series encapsulates the ambiguity response of the system (grammar) to all possible input sentences. In this way, the power series is analogous to the impulse response in electrical engineering, which encapsulates the response of the system (circuit) to all possible input frequencies. (Ambiguity coefficients bear a strong resemblance to frequency coefficients in Fourier analysis.) All of these transformed representation systems (e.g., power series, impulse response, and Fourier series) provide a complete

3. The formal language literature [5, 15] uses the term support instead of ambiguity coefficient.

description of the system with no loss of information[4] (and no heuristic approximations (e.g., search strategies [6])). Transforms are often very useful because they provide a different point of view. Certain observations are more easily seen in the transform space than in the original space, and vice versa.

This paper will discuss several ways to generate the power series. Initially let us consider successive approximation. Of all the techniques to be presented here, successive approximation most closely resembles the approach taken by most current chart parsers including EQSP. The alternative approaches take advantage of certain regularities in the power series in order to produce the same results more efficiently.

Successive approximation works as follows. First we translate grammar (6a) into the equation

(8) $NP = John + NP \cdot and \cdot NP$

where "+" connects two ways of generating an NP and "·" concatenates two parts of an NP. In some sense, we want to "solve" this equation for NP. This can be accomplished by refining successive approximations. An initial approximation NP_0 is formed by taking NP to be the empty language.

(9a) $NP_0 = 0$

Then we form the next approximation by substituting the previous approximation into equation (8), and simplifying according to the usual rules of algebra (e.g. assuming distributivity, associativity,[5] identity element, and zero element).

(9b) $NP_1 = John + NP_0 \cdot and \cdot NP_0 = John + 0 \cdot and \cdot 0 = John$

We continue refining the approximation in this way.

(9c) $NP_2 = John + NP_1 \cdot and \cdot NP_1 = John + John\ and\ John$

---

4. This needs a qualification. It is true that the power series provides a complete description of the ambiguity response to any input sentence. However, the power series representation may be losing some information that would be useful for parsing. In particular, there might be some cases where it is impossible to recover the parse trees exactly as we will see, though this may not be too serious a problem for many practical applications. That is, it is often possible to recover most (if not all) of the structure, which may be adequate for many applications.

5. The careful reader may correctly object to this assumption. We include it here for expository convenience, as it greatly simplifies the derivations though it should be noted that many of the results could be derived without the assumption. Furthermore, this assumption is valid for counting ambiguity. That is, $|A \cdot B| * |C| = |A| * |B \cdot C|$, where A, B and C are sets of trees and $|A|$ denotes the number of members of A, and * is integer multiplication.

(9d) NP_3 = John + NP_2 and NP_2
= John + (John + John and John) · and · (John + John and John)
= John + John and John + John and John and John + John and John and John
+ John and John and John and John
= John + John and John + 2 John and John and John
+ John and John and John and John
⋮

Eventually, we have NP expressed as an infinitely long polynomial (6b) above. This expression can be simplified by introducing a notation for exponentiation. Let x^i be an abbreviation for multiplying x · x · ... · x, i times.

(10) NP = John + John and John + 2 John (and John)^2
+ 5 John (and John)^3 + 14 John (and John)^4 + ...

Note that parentheses are interpreted differently in algebraic equations than in context-free rules. In context-free rules, parentheses denote optionality, whereas in equations they denote precedence relations among algebraic operations.

3. Catalan Numbers

Ambiguity coefficients take on an important practical significance when we can model them directly without resorting to successive approximation as above. This can result in substantial time and space savings in certain special cases where there are much more efficient ways to compute the coefficients than successive approximation (chart parsing). Equation (10) is such a special case; the coefficients follow a well-known combinatoric series called the *Catalan Numbers* [8: pp. 388-389, pp. 531-533].[6] This section will describe Catalan numbers and their relation to parsing.

The first few Catalan numbers are: 1, 1, 2, 5, 14, 42, 132, 469, 1430, 4862, ... They are generated by the closed form expression:[7]

(11) $$Cat_n = \binom{2n}{n} - \binom{2n}{n-1}$$

---

6. This fact was first pointed out to us by V. Pratt. We suspect that it is a generally well-known result in the formal language community, though its origin is unclear.

7. $\binom{a}{b}$ is known as a *binomial coefficient*. It is equivalent to $\frac{a!}{b!(a-b)!}$, where a! is equal to the product of all integers between 1 and a. Binomial coefficients are very common in combinatorics where they are interpreted as the number of ways to pick b objects out of a set of a objects.

This formula can be explained in terms of parenthesized expressions, which are equivalent to trees. Cat_n is the number of ways to parenthesize a formula of length n. There are two conditions on parenthesization: (a) there must the same number of open and close parentheses, and (b) they must be properly nested so that an open parenthesis precedes its matching close parenthesis. The first term counts the number of sequences of $2n$ parentheses, such that there are the same number of opens and closes. The second term subtracts out cases violating condition (b). This explanation is elaborated in [8: p. 531].

It is very useful to know that the ambiguity coefficients are Catalan numbers because this observation enables us to replace equation (10) with (12), where Cat_i denotes the i^{th} Catalan number. (All summations range from 0 to ∞ unless noted otherwise.)

$$(12) \quad \text{NP} = \sum_i Cat_i \text{ John (and John)}^i$$

The i^{th} Catalan number is the number of binary trees that can be constructed over i phrases. This model correctly predicts EQSP's behavior with prepositional phrases. That is, the EQSP parser [11] found exactly the Catalan number of parse trees for each sentence in the following sequence:

1	It was the number.
1	It was the number of products.
2	It was the number of products of products.
5	It was the number of products of products of products.
14	It was the number of products of products of products of products.
	⋮

These predictions continue to hold with as many as nine prepositional phrases (4862 parse trees).

4. Table Lookup

We could improve EQSP's performance on PPs if we could find a more efficient way to compute Catalan numbers than chart parsing, the method currently employed by EQSP. Let us propose two alternatives: table lookup and evaluating expression (11) directly. Both are very efficient over practical ranges of n, say no more than 20 phrases or so.[8] In both cases, the ambiguity of a sentence in grammar (6a) can be determined by counting the number of occurrences of "and John" and then retrieving the Catalan of that number. These

8. The table lookup scheme ought to have a way to handle the theoretical possibility that there are an unlimited number of prepositional phrases. The table lookup routine will employ a more traditional parsing algorithm (e.g., Earley's Algorithm) when the number of phrases in the input sentence is not stored in the table.

approaches both take linear time (over practical ranges of n),[9] whereas chart parsing requires cubic time to parse sentences in these grammars, a significant improvement.

So far we have shown how to compute in linear time the number of ambiguous interpretations of a sentences in an "every way ambiguous" grammar. However, we are really interested in finding parse trees, not just the number of ambiguous interpretations. We could extend the table lookup algorithm to find trees rather than ambiguity coefficients, by modifying the table to store trees instead of numbers. For parsing purposes, Cat_i can be thought of as a pointer to the i^{th} entry of the table. So, for a sentence in grammar (6a) for example, the machine could count the number of occurrences of "and John" and then retrieve the table entry for that number.

index	trees
0	{[John]}
1	{[John and John]}
2	{[[John and John] and John], [John and [John and John]]}
	⋮

The table would be more general if it did not specify the lexical items at the leaves. Let us replace the table above with

index	trees
0	{[x]}
1	{[x x]}
2	{[[x x] x], [x [x x]]}
	⋮

and assume the machine can bind the x's to the appropriate lexical items.

There is a real problem with this table lookup machine. The parse trees may not be exactly correct because the power series computation assumed that multiplication was associative, which is an appropriate assumption for counting ambiguity, but inappropriate for constructing trees. For example, we observed that prepositional phrases and conjunction are both "every way ambiguous" grammars because their ambiguity coefficients are Catalan numbers. However, it is not the case that they generate exactly the same parse trees.

---

9. The linear time result depends on the assumption that table lookup (or closed form computation) can be performed in constant time. This may be a fair assumption over practical ranges of n, but it is not true in general.

Nevertheless we present the table lookup pseudo-parser here because it seems to be a speculative new approach with considerable promise. It is often more efficient than a real parser, and the trees that it finds may be just as useful as the correct one for many practical purposes. For example, many speech recognition projects employ a parser to filter out syntactically inappropriate hypotheses. However, a full parser is not really necessary for this task; a recognizer such this table lookup pseudo-parser may be perfectly adequate for this task.

Furthermore, it is often possible to recover the correct trees from the output of the pseudo-parser. In particular, the difference between prepositional phrases and conjunction could be accounted for by modifying the interpretation of the PP category label, so that the trees would be interpreted correctly even though they are not exactly correct. In short, the table lookup pseudo-parser is worth exploring even though the results are not always correct. The results are close enough for many applications (e.g., speech recognition) and the mistakes can often be corrected.

The table lookup approach works for primitive grammars. The next two sections will show how to decompose composite grammars into series and parallel combinations of primitive grammars.

(13a) $G = G_1 \cdot G_2$ *series*

(13b) $G = G_1 + G_2$ *parallel*

5. Parallel Decomposition

Parallel decomposition can be very useful for dealing with lexical ambiguity, as in

(14) ... to total with products near profits ...

where "total" can be taken as a noun or as a verb, as in:

(15a) The accountant brought the daily sales to total with products near profits organized according to the new law. *noun*

(15b) The daily sales were ready for the accountant to total with products near profits organized according to the new law. *verb*

The analysis of these sentences will make use of the additivity property of linear systems. That is, each case, (15a) and (15b), will be treated separately, and then the results will be added together. Assuming "total" is a noun, there are three prepositional phrases contributing Cat_3 bracketings, and assuming it is a verb, there are two prepositional phrases for Cat_2 ambiguities. Combining the two cases produces $Cat_3 + Cat_2 = 5 + 2 = 7$ parses. Adding another prepositional phrase yields $Cat_4 + Cat_3 = 14 + 5 = 19$ ambiguities. (EQSP

behaved as predicted in both cases.)

This behavior is generalized by the following power series:

$$(16) \quad \left\{ \begin{matrix} P\ N \\ \text{to}\ V \end{matrix} \right\} \sum_i (Cat_{i+1} + Cat_i)\,(P\ N)^i$$

which is the sum of the two cases:

$$(17a) \quad \sum_{i>0} Cat_i\,(P\ N)^i = P\ N \sum_i Cat_{i+i}\,(P\ N)^i \qquad \textit{noun}$$

$$(17b) \quad \text{to}\ V \sum_i Cat_i\,(P\ N)^i \qquad \textit{verb}$$

This observation can be incorporated into the table lookup pseudo-parser outlined above. Recall that Cat_i is interpreted as the i^{th} index in a table containing all binary trees dominating i leaves. Similarly, $Cat_i + Cat_{i+1}$ will be interpreted as an instruction to "append" the i^{th} entry and $i+1^{st}$ entry of the table.

(18) (ADD-TREES (CAT-TABLE i) (CAT-TABLE (+ i 1)))

(This can be implemented efficiently, given an appropriate representation of sets of trees.)

Now suppose there were an oracle that disambiguated the word "total". How could we incorporate this information once we have already parsed the input sentence and found that it was the sum of two Catalans? The parser can simply subtract out the inappropriate interpretations. If the oracle says that "total" is a verb, then (17a) would be subtracted from the combined sum, and if the oracle says that "total" is a noun, then (17b) would be subtracted.

Furthermore, suppose that we wanted to evaluate the usefulness of a particular oracle. For example, suppose that there was a semantic routine that could disambiguate "total", but this semantic routine is very expensive to execute so that we don't want to run it unless we are very sure that it has a desirable cost/benefit ratio. We need a way to estimate the usefulness of the semantic routine so that we don't waste time working on semantic constraints when they won't help very much. This analysis provides a very simple way to estimate the benefit of disambiguating "total". If it turns out to be a verb, then (17a) trees have been ruled out, and it it turns out to be a noun, then (17b) trees have been ruled out.

6. Series Decomposition

Suppose we have a non-terminal S which is a series combination of two other non-terminals, NP and VP. By inspection, the power series of S is:

(19) $S = NP \cdot VP$

This result is easily verified when there is an unmistakable dividing point between the subject and the predicate. For example, the verb "is" separates the PPs in the subject from those in the predicate in (20a), but not in (20b).

(20a) The number of products over sales of ... *is* near the number of sales under ... *clearly divided*

(20b) *Is* the number of products over sales of ... near the number of sales under ...? *not clearly divided*

In (20a), the total number of parse trees is the product of the number of ways of parsing the subject times the number of ways of parsing the predicate. Both the subject and the predicate produce a Catalan number of parses, and hence the result is the product of two Catalan numbers, which was verified by EQSP [11: p. 53]. This result can be formalized in terms of the power series:

(21) $\left(N \sum_i Cat_i (P\ N)^i \right) \left(is \sum_j Cat_j (P\ N)^j \right)$

which is formed by taking the product of the two subcases:

(22a) $N \sum_i Cat_i (P\ N)^i$ *subject*

(22b) $is \sum_j Cat_j (P\ N)^j$ *predicate*

The power series says that the ambiguity of a particular sentence is the product of Cat_i and Cat_j, where i is the number of PPs before "is" and j is the number after "is". This could be incorporated in the table lookup parser as an instruction to "multiply" the i^{th} entry in the table times the j^{th} entry. Multiplication is a cross-product operation; $L \times R$ generates the set of binary trees whose left sub-tree l is from L and whose right sub-tree r is from R.

(23) $L \times R = \{(l, r) \mid l \in L \,\&\, r \in R\}$

This is a formal definition. For practical purposes, it may be more useful for the parser to output the list in the factored form:

(24) (MULTIPLY-TREES (CAT-TABLE i) (CAT-TABLE j))

which is much more concise than a list of trees. It is possible, for example, that semantic processing can take advantage of factoring, capturing a semantic generalization that holds across all subjects or all predicates. Imagine, for example, that there is a semantic agreement constraint between predicates and arguments. For example, subjects and predicates might have to agree on the feature ±human. Suppose that we were given sentences where this constraint was violated by all ambiguous interpretations of the sentence. In this case, it would be more efficient to employ a feature vector scheme [3] which propagates the features in factored form. That is, it computes a feature vector for the union of all possible subjects, and a vector for the union of all possible VPs, and then compares (intersects) these vectors to check if there are any interpretations which meet the constraint. A system such as this, which keeps the parses in factored form, is much more efficient than one that multiplies them out. Even if semantics cannot take advantage of the factoring, there is no harm in keeping the representation in factored form, because it is straightforward to expand (24) into a list of trees (though it may be somewhat slow).

This example is relatively simple because "is" helps the parser determine the value of *i* and *j*. Now let us return to example (20b) where "is" does not separate the two strings of PPs. Again, we determine the power series by multiplying the two subcases:

$$(25)\quad \text{is}\left(\text{N}\sum_i \text{Cat}_i\,(\text{P N})^i\right)\left(\sum_j \text{Cat}_j\,(\text{P N})^j\right) = \text{is N}\sum_i\sum_j \text{Cat}_i\,\text{Cat}_j\,(\text{P N})^{i+j}$$

However this form is not so useful for parsing because the parser cannot easily determine *i* and *j*, the number of prepositional phrases in the subject and the number in the predicate. It appears the parser will have to compute the product of two Catalans for each way of picking *i* and *j*, which is somewhat expensive.[10] Fortunately the Catalan function has some special properties so that it is possible algebraically to remove the references to *i* and *j*. In the next section we will show how this expression can be reformulated in terms of *n*, the total number of PPs.

10. Earley's algorithm and most other context-free parsing algorithms actually work this way.

6.1 Auto-Convolution of Catalan Grammars

Some readers may have noticed that expression (25) is in *convolution* form. We will make use of this in the reformulation. Notice that the Catalan series is a fixed point under *auto-convolution* (except for a shift); that is, multiplying a Catalan power series (i.e., $1 + x + 2x^2 + 5x^3 + 14x^4 + \ldots \mathrm{Cat}_i x^i \ldots$) with itself produces another polynomial with Catalan coefficients.[11] The multiplication is worked out below for the first few terms.

$$\begin{array}{rccccccc}
 & 1 + & x + & 2x^2 + & 5x^3 + & 14x^4 + & \ldots \\
\times & 1 + & x + & 2x^2 + & 5x^3 + & 14x^4 + & \ldots \\
\hline
 & 1 + & x + & 2x^2 + & 5x^3 + & 14x^4 + & \ldots \\
 & & x + & x^2 + & 2x^3 + & 5x^4 + & \ldots \\
 & & & 2x^2 + & 2x^3 + & 4x^4 + & \ldots \\
 & & & & 5x^3 + & 5x^4 + & \ldots \\
+ & & & & & 14x^4 + & \ldots \\
\hline
 & 1 + & 2x + & 5x^2 + & 14x^3 + & 42x^4 + & \ldots
\end{array}$$

This property can be summarized as:

$$(26) \quad \sum_i \mathrm{Cat}_i x^i \sum_j \mathrm{Cat}_j x^j = \sum_n \mathrm{Cat}_{n+1} x^n$$

where n equals i + j.

Intuitively, this equation says that if we have two "every way ambiguous" (Catalan) constructions, and we combine them in every possible way (convolution), the result is an "every way ambiguous" (Catalan) construction. With this observation, equation (25) reduces to:

$$(27) \quad \mathrm{is}\left(\mathrm{N} \sum_i \mathrm{Cat}_i (\mathrm{P\ N})^i\right)\left(\sum_j \mathrm{Cat}_j (\mathrm{P\ N})^j\right) = \mathrm{is\ N} \sum_n \mathrm{Cat}_{n+1} (\mathrm{P\ N})^n$$

Hence the number of parses in the auxiliary-inverted case is the Catalan of one more than in the non-inverted cases. As predicted, EQSP found the following inverted sentences to be more ambiguous than their non-inverted counter-parts (previously discussed on page 12) by one Catalan number.

---

11. The proof immediately follows from the z-transform of the Catalan series [8: p. 388]: $zB(z)^2 = B(z) - 1$.

1 Was the number?
2 Was the number of products?
5 Was the number of products of products?
14 Was the number of products of products of products?
42 Was the number of products of products of products of products?

1 It was the number.
1 It was the number of products.
2 It was the number of products of products.
5 It was the number of products of products of products.
14 It was the number of products of products of products of products.

How could this result be incorporated into the table lookup pseudo-parser? Recall that the pseudo-parser implements Catalan grammars by returning an index into the Catalan table. For example, if there were i PPs, the parser would return: (CAT-TABLE i). We now extend the indexing scheme so that the parser implements a series connection of two Catalan grammars by returning one higher index than it would for a simple Catalan grammar. That is, if there were n PPs, the parser would return: (CAT-TABLE (+ n 1)).

Series connections of Catalan grammars are very common in every day natural language, as illustrated by the following two sentences which have received considerable attention in the literature because the parser cannot separate the direct object from the prepositional complement.

(28a) I saw the man on the hill with a telescope ...
(28b) Put the block in the box on the table in the kitchen ...

Both examples have a Catalan number of ambiguities because the auto-convolution of a Catalan series yields another Catalan series.[12] This result can improve parsing performance because it suggests ways to re-organize (compile) the grammar so that there will be fewer references to quantities that are not readily available. This re-organization will reap benefits that chart parsers (e.g. Earley's algorithm) do not currently achieve because the re-organization is taking advantage of a number of combinatoric regularities, especially *convolution*, that are not easily encoded into a chart. Section 9 will present an example of the re-organization.

12. There is a difference between these two sentences because "put" subcategorizes for two objects unlike "see". Suppose we analyze "see" as lexically ambiguous between two senses, one which selects for exactly two objects like "put" and one which selects for exactly one object as in "I saw it." The first sense contributes the same number of parses as "put" and the second sense contributes an additional Catalan factor.

6.2 Chart Parsing

Perhaps it is worthwhile to reformulate chart parsing in our terms in order to show which of the above results can be captured by such an approach and which cannot. Traditionally chart parsers maintain a chart (or matrix) M, whose entries M_{ij} contain the set of category labels which span from position i to position j in the input sentence. This is accomplished by finding a position k in between i and j such that there is a phrase from i to k which can combine with another phrase from k to j. An implementation of the inner loop looks something like:

(29) $M_{ij} := \{\}$
loop for k from i to j do
$M_{ij} := M_{ij} \cup M_{ik} * M_{kj}$

Essentially, then, a chart parser is maintaining the invariant

$$(30) \quad M_{ij} = \sum_{k} M_{ik} \cdot M_{kj}$$

Recall that addition and multiplication were previously defined over polynomials. We can preserve these definitions if we modify the contents of the chart. Let us replace the set of category labels in M_{ij} with a set of factored polynomials. That is, let M_{ij}^{x} denote the polynomial describing the ways to parse a phrase of category x from position i to position j. For example, the notation

$$(31) \quad M_{04}^{S} = M_{01}^{NP} \cdot M_{14}^{VP} + M_{02}^{NP} \cdot M_{24}^{VP}$$

indicates that there are two ways to combine an NP and a VP to form an S from position 0 to position 4.

This formulation of the chart can be compared with serial and parallel decomposition. Note that $M_{01}^{NP} \cdot M_{14}^{VP}$ is essentially the same as (MULTIPLY-TREES M_{01}^{NP} M_{14}^{VP}). Similarly, adding matrix elements corresponds to ADD-TREES. Hence, chart parsing is more similar to serial and parallel combinations than one might have suspected. When the grammar is factored appropriately, chart parsers will be able to take advantage of serial and parallel decompositions discussed above.

However, the examples above illustrate cases where chart parsers are inefficient. In particular, chart parsers cannot take advantage of convolution and the "every way ambiguous" generalization. That is, Earley's algorithm performs convolution the "long way", by picking each possible dividing point k, and parsing from i to k and from k to j. It is incapable of reducing the convolution of two Catalan as we did above. Similarly, Earley's algorithm is incapable of using the "every way ambiguous" generalization. That is, it requires $O(n^3)$ time to parse Catalan grammars because there are no constraints on the choice of i, j and k.

The algorithm will eventually enumerate all possible values of i, j and k. We suggest that a processor ought to be able to notice the lack of constraints, and thus avoid enumerating the space as Earley's algorithm does.

Finally, in passing, we have one constructive suggestion for chart parsers. We observe that it is possible to count the number of ambiguous interpretations in $O(n^3)$ time. This is an improvement over the obvious algorithm which multiplies out all the trees just as if they were being printed. (Such an exponential algorithm was actually implemented in EQSP.) We suggest keeping a second matrix Λ, where Λ_{ij}^{x} holds the number of ways of deriving a phrase of category x between i and j. The two matrices, Λ and M, are almost identical, except that Λ holds integers and M holds polynomials. Accordingly, addition and multiplication are defined slightly differently on the two matrices. In Λ, they map integers into integers in the obvious way; in M, they map polynomials into polynomials as discussed above. Note that both matrices, Λ and M, can be computed with exactly the same sequences of multiplications and additions. Hence, it is possible to compute the number of ambiguous interpretations in cubic time.

6.3 Auto-Convolution of Unit Step Grammars

Let us return to the discussion of convolution. This section will illustrate a second practical example of convolution. Consider the following grammar ("Λ" denotes the empty string):[13]

(32) $A \rightarrow a\,\Lambda \mid \Lambda$

We call this grammar a unit step grammar because all of its ambiguity coefficients are 1.

(33) $\Lambda = 1 + a + a^2 + a^3 + a^4 + a^5 + \ldots = \sum_n a^n$

In other words, the grammar is unambiguous.[14] Embedded sentences are a typical example of (32) in English.

(34) I believe you said he thought you were ...

Sunpose for the sake of discussion that we choose to analyze adjuncts with a right branching grammar. (By convention, terminal symbols appear in lower case.)

(35) ADJS $\rightarrow$ adj ADJS $\mid \Lambda$

---

13. Note that the empty language { } is distinct from the language of the empty string {Λ}. In particular, {Λ} is the identity element under series connection and { } is the identity element under parallel connection. Thus, {Λ} is modeled as 1 in the power series representation, whereas { } is modeled as 0.

14. Unit step grammars are not exactly the same as unambiguous grammars. The ambiguity coefficients of a unit step grammar are all 1, whereas the ambiguity coefficients of an unambiguous grammar are either 1 or 0.

so that

(36) Will you go to the store tomorrow in the morning about 10:00 after ...?

has one parse, independent of the number of adjuncts. A similar analysis of adjuncts is adopted in [7]. This analysis can also be defended on performance grounds as an efficiency approximation. (This approximation is in the spirit of pseudo-attachment [1].)

The power series is

$$(37) \quad \text{ADJS} = \sum_i \text{adj}^i$$

Now, how many ambiguities will there be if we add a second clause to (36) as in:

(38) I will *ask* if you will *go* to the store tomorrow in the morning about 10:00 after ...?

Some of the adjuncts will attach to "go" and the rest will attach to "ask". The number of parses is determined by multiplying the two subgrammars.

$$(39) \quad \text{ADJS} \cdot \text{ADJS} = \sum_i \text{adj}^i \sum_j \text{adj}^j = \sum_i \sum_j \text{adj}^{i+j}$$

This equation has the same problem as equation (25); because there is no clear dividing line between the adjuncts that attach to "go" and the ones that attach to "ask", it is not very easy for the parser to determine *i* and *j*. Again, it might appear that the parser will have to try all possible values of *i* and *j*, a moderately expensive process. However, there are some special properties of the step function that enable us to remove the references to *i* and *j* in equation (39). In engineering jargon, the convolution of two steps is a ramp. That is, the product of two polynomials with step coefficients is a polynomial with increasing coefficients [8: pp. 89, equation 16]. We have multiplied out the first few terms below.

$$\begin{array}{rccccccc}
 & 1 + & x + & x^2 + & x^3 + & x^4 + & \ldots \\
\times & 1 + & x + & x^2 + & x^3 + & x^4 + & \ldots \\
\hline
 & 1 + & x + & x^2 + & x^3 + & x^4 + & \ldots \\
 & & x + & x^2 + & x^3 + & x^4 + & \ldots \\
 & & & x^2 + & x^3 + & x^4 + & \ldots \\
 & & & & x^3 + & x^4 + & \ldots \\
+ & & & & & x^4 + & \ldots \\
\hline
 & 1 + & 2x + & 3x^2 + & 4x^3 + & 5x^4 + & \ldots
\end{array}$$

The general result is:

$$(40) \quad \sum_i x^i \sum_j x^j = \sum_n (n+1)\, x^n$$

Now equation (39) can be simplified so that the references to *i* and *j* are replaced with *n*, the total number of adjuncts. This is much easier for the parser to deal with because for a given input sentence there is a single value for *n*, whereas there are multiple values for *i* and *j*.

$$(41) \quad \sum_i adj^i \sum_j adj^j = \sum_n (n+1)\, adj^n$$

This says that a string of *n* adjuncts induces *n+1* parse trees, because there are *n+1* ways to cut the string into two substrings.[15] Now suppose there were three matrix clauses instead of just two.

(42) I will *ask* if he will *persuade* you to *go* to the store tomorrow in the morning about 10:00 after ...?

The number of parses in this case is the convolution of three steps.

$$(43) \quad \sum_i adj^i \sum_j adj^j \sum_k adj^k$$

Again this form is ill-suited for parsing because there is no easy way to determine *i*, *j* and *k*. However, it is possible to remove the references to the offending variables by taking advantage of some special properties of the step function. In particular, there is a closed form for the convolution of *d+1* step functions [8: p. 90, equation 20]:

$$(44) \quad \left(\sum_i x^i \right)^{d+1} = \sum_n \binom{n+d}{d} x^n$$

Now we can remove the references to *i*, *j* and *k*:

$$(45) \quad \left(\sum_i adj^i \right)^3 = \sum_n \binom{n+2}{2} adj^n = \sum_n \tfrac{1}{2}(n+1)(n+2)\, adj^n$$

---

15. The string can be cut between any two words (n – 1 places) or at either end (2 places).

These examples show that standard well-known combinatorics can be used to determine the number of ambiguities in many common cases.

7. Computing the Power Series Directly from the Grammar

In fact, the result derived in the previous section can be computed directly from the grammar itself. First we translate the grammar into an equation in the usual way. That is, ADJS is modeled as a parallel combination of two subgrammars, adj ADJS and Λ. (Recall that Λ is modeled as 1 because it is the identity element under series combination.)

(46a) $\text{ADJS} \rightarrow \text{adj ADJS} \mid \Lambda$

(46b) $\text{ADJS} = \text{adj} \cdot \text{ADJS} + 1$

We can simplify (46b) so the right hand side is expressed in terminal symbols alone, with no references to non-terminals. This is very useful for processing because it is much easier for the parser to determine the presence or absence of terminals, than of non-terminals. That is, it is easier for the parser to determine, for example, whether a *word* is an adj, than it is to decide whether a *substring* is an ADJS phrase. The simplification moves all references to ADJS to the left hand side, by subtracting from both sides,

(46c) $\text{ADJS} - \text{adj} \cdot \text{ADJS} = 1$

factoring the left hand side,

(46d) $(1 - \text{adj})\,\text{ADJS} = 1$

and dividing from both sides.

(46e) $\text{ADJS} = (1 - \text{adj})^{-1}$

This result is equivalent to the step formulation (37), as can been seen by performing the long division:

(46f) $$\frac{1}{1 - \text{adj}} = 1 + \frac{\text{adj}}{1 - \text{adj}} = 1 + \text{adj} + \frac{\text{adj}^2}{1 - \text{adj}}$$

$$= 1 + \text{adj} + \text{adj}^2 + \frac{\text{adj}^3}{1 - \text{adj}} = \ldots = \sum_n \text{adj}^n$$

The purpose of this section was two folded. First, we presented a simpler derivation of the power series for a unit step grammar. Secondly, and more importantly, we have introduced the notion of division. We now have four combination rules:

(47a) series combination (multiplication)
(47b) parallel combination (addition)
(47c) inverse of series combination (division)
(47d) inverse of parallel combination (subtraction)

Series and parallel combinations are frequently found in many grammars formalisms currently employed in the literature (e.g. context-free grammars, ATNs), and consequently, they required very little motivation. Subtraction was introduced as a "ruling-out" operation. The next section will provide an intuition for division in terms of ATNs.

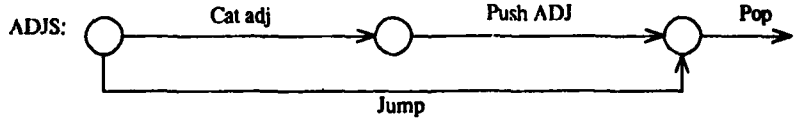
8. Computing the Power Series from the ATN

This section will re-derive the power series for the unit step grammar directly from the ATN representation by treating the networks as *flow graphs* [13]. The graph transformations presented here are directly analogous to the algebraic simplifications employed in the previous section.

First we translate the grammar into an ATN in the usual way [16].

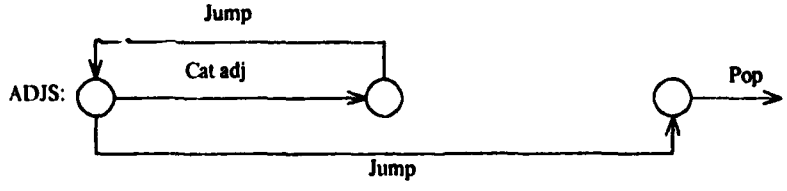
(48) ADJS → adj ADJS | Λ

(49)



This graph can be simplified by performing a compiler optimization called *tail recursion* ([2] and references therein). This transformation replaces the final push arc with a jump:

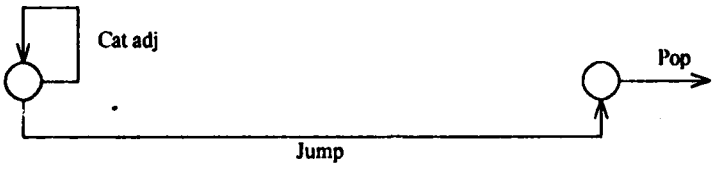
(50)



Tail recursion corresponds directly to the algebraic operations of moving the ADJS terms to left hand side, factoring out the ADJS, and dividing from both sides.

Then we remove the top jump arc by series reduction. This step corresponds to multiplying by 1 since a jump arc is the ATN representation for the identity element under series combination.

(51) ADJS:



The loop can be treated as an infinite series:

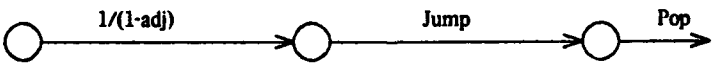
(52) $1 + adj + adj^2 + adj^3 + ...$

where the zero-th term corresponds to zero iterations around the loop, the first term corresponds to a single iteration, the second term to two iterations, and so on. Recall that (52) is equivalent to:

(53) $\frac{1}{1 - adj}$

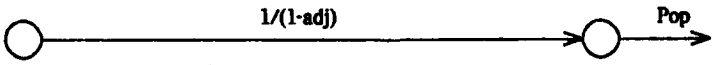
With this observation, it is possible to open the loop:

(54) ADJS:



After one final series reduction, the ATN is equivalent to expression (46c) above.

(54c) ADJS:



Now we can motivate division in intuitive terms. Division is a loop in an ATN.

How can division be implemented? We have two answers. First, division can be implemented as an ATN loop. Alternatively, we can employ the table lookup scheme discussed above. That is, we formulate division as an infinite sum:

(55) $\frac{1}{1 - adj} = \sum_i adj^i$

Then we construct a table such that the i^{th} entry contains the i^{th} ambiguity coefficient. In other words, the i^{th} location in the table tells the parser how to parse i occurrences of adj. The table lookup scheme is somewhat more general than the ATN loop, because the table allows the i^{th} coefficient to take on arbitrary values whereas the ATN loop restricts the coefficients to 1. For example, the Catalan grammar (56a) could be implemented with a table (56b), but not with an ATN loop.

(56a) $A \rightarrow A\ A \mid a$ *Catalan Grammar*

(56b) $\sum_i Cat_i\ a^i$ *table implementation*

However the table has the theoretical problem that it requires an infinite amount of memory. This is not a problem in practice since the regions of interest are not that large. It is unlikely, for example, that a sentence would contain more than twenty prepositional phrases.

So far we have discussed five primitive grammars: Catalan, Unit Step, 1, and 0, and terminals, and four composition rules: addition, subtraction, multiplication and division. Furthermore we have outlined three implementation strategies: successive approximation (chart parsing), table lookup, and ATNs. We have seen that it is often possible to employ these tools in order to re-organize the grammar so that these implementations will perform more efficiently. We have identified certain situations where the ambiguity is combinatoric, and have sketched a few modifications to the grammar that enables processing to proceed in a more efficient manner. In particular, we have observed it is important for the grammar to avoid referencing quantities that are not easily determined such as the dividing point between a noun phrase and a prepositional phrase.

9. An Example

Suppose for example that we were given the following grammar:

(57a) S → NP VP ADJS
(57b) S → V NP (PP) ADJS ADJS
(57c) VP → V NP (PP) ADJS
(57d) PP → P NP
(57e) NP → N (PP)
(57f) ADJS → adj ADJS | Λ

(In this example, we will assume no lexical ambiguity among N, V, P and adj.)

By inspection, we notice that NP and PP are Catalan grammars and that ADJS is a Step grammar.

$$(58a)\quad PP = \sum_{i>0} Cat_i\,(P\ N)^i$$

$$(58b)\quad NP = N \sum_{i} Cat_i\,(P\ N)^i$$

$$(58c)\quad ADJS = \sum_{i} adj^i$$

With these observations, the parser can process PPs, NPs and ADJS by counting the number of occurrences of terminal symbols and looking up those numbers in the appropriate tables. We now substitute (58a-c) into (57c)

$$(59)\quad VP = V\ NP\ (1+PP)\ ADJS = V \left(N \sum_{i} Cat_i\,(P\ N)^i \right) \left(\sum_{i} Cat_i\,(P\ N)^i \right) \left(\sum_{i} adj^i \right)$$

and simplify the convolution of the two Catalan functions

$$(60)\quad VP = V \left(N \sum_{i} Cat_{i+1}\,(P\ N)^i \right) \left(\sum_{i} adj^i \right)$$

so that the parser can also find VPs by just counting occurrences of terminals symbols. Now we simplify (57a-b) so that S phrases can also be parsed by just counting occurrences of terminal symbols. First, translate (57a-b) into the equation:

$$(61)\quad S = NP\ VP\ ADJS + V\ NP\ (1+PP)\ ADJS\ ADJS$$

and then expand VP

$$(62)\quad S = NP\ (V\ NP\ (1+PP)\ ADJS)\ ADJS + V\ NP\ (1+PP)\ ADJS\ ADJS$$

and factor

$$(63)\quad S = (NP + 1)\ V\ NP\ (1+PP)\ ADJS^2$$

This can be simplified considerably because

$$(64)\quad \mathrm{NP}\,(1+\mathrm{PP}) = \mathrm{N}\sum_i \mathrm{Cat}_i\,(\mathrm{P\ N})^i \sum_i \mathrm{Cat}_i\,(\mathrm{P\ N})^i = \mathrm{N}\sum_i \mathrm{Cat}_{i+1}\,(\mathrm{P\ N})^i$$

and

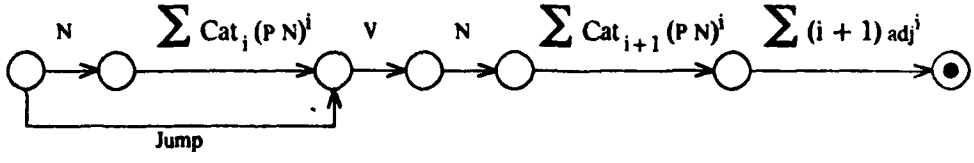
$$(65)\quad \mathrm{ADJS}^2 = \sum_i \mathrm{adj}^i \sum_i \mathrm{adj}^i = \sum_i (i+1)\,\mathrm{adj}^i$$

so that

$$(66)\quad \mathrm{S} = \left(\mathrm{N}\sum_i \mathrm{Cat}_i\,(\mathrm{P\ N})^i + 1\right)\mathrm{V\ N}\sum_i \mathrm{Cat}_{i+1}\,(\mathrm{P\ N})^i \sum_i (i+1)\,\mathrm{adj}^i$$

which has the following ATN realization:

(67)

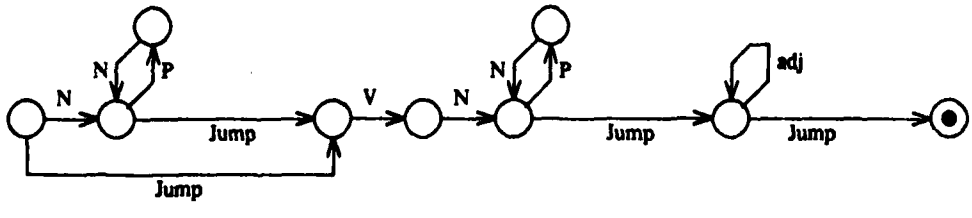


The entire example grammar has now been compiled into a form that is easier for parsing. This formula says that sentences are all of the form:

$$(68)\quad \mathrm{S} \xrightarrow{\bullet} (\mathrm{N}\,(\mathrm{P\ N})^{\bullet})\ \mathrm{V\ N}\,(\mathrm{P\ N})^{\bullet}\,\mathrm{adj}^{\bullet}$$

which could be recognized by the following finite state machine:

(69)



Furthermore, the number of parse trees for a given input sentence can be found by multiplying three numbers: (a) the Catalan of the number of P N's before the verb, (b) the Catalan of one more than the number of P N's after the verb, and (c) the ramp of the number of adj's. For example, the sentence

(70) The man on the hill saw the boy with a telescope yesterday in the morning.

has $Cat_1 * Cat_2 * 3 = 6$ parses. That is, there is one way to parse "the man on the hill", two ways to parse "saw the boy with a telescope" (either "telescope" is a complement of "see" as in (71a-c) or it is attached to "boy" as in (71d-f)), and three ways to parse the adjuncts (they could both attach to the S (71a,d), or they could both attach to the VP (71b,e), or they could split (71c,f)).

(71a) [The man on the hill [saw the boy with a telescope] [yesterday in the morning.]]
(71b) The man on the hill [[saw the boy with a telescope] [yesterday in the morning.]]
(71c) The man on the hill [[saw the boy with a telescope] yesterday] in the morning.
(71d) [The man on the hill saw [the boy with a telescope] [yesterday in the morning.]]
(71e) The man on the hill [saw [the boy with a telescope] [yesterday in the morning.]]
(71f) The man on the hill [saw [the boy with a telescope] yesterday] in the morning.

All and only these possibilities are permitted by the grammar.

10. Lexical Restrictions

Now suppose there were an oracle (e.g. lexical restrictions) that disambiguated some of these possibilities. How could we incorporate this information once we have already parsed the input sentence as above? For example, the verb "see" has two lexical forms, a predicate of two arguments as in "I saw it" and a predicate on three arguments as in "I saw it with a telescope". Now suppose we had an oracle which disambiguated these two possibilities. How could we take advantage of this information?

Consider the two argument case first. The previously assumed VP grammar (72a) simplifies to (72b) with the two argument restriction.

(72a) VP → V NP (PP) ADJS
(72b) VP → V NP ADJS

If we re-derive the power series for S, we obtain:

$$(73) \quad S = \left(N \sum_i Cat_i (P\,N)^i + 1 \right) V\,N \sum_i Cat_i (P\,N)^i \sum_i (i+1)\, adj^i$$

This equation is the same as (66) except that Cat_{i+1} in (66) has been replaced with Cat_i. The Cat_{i+1} resulted from convolving the PPs generated in object position with those generated in complement position. Under the two argument restriction, it is no longer possible to generate any PPs in complement position, and hence

all the PPs must be in object position. There are Cat_i ways to put them in object position as we have discussed.

With this formula, we see that three of the six parses given in (71) meet the two argument restriction. That is, there is still only one way to parse "the man on the hill" and three ways to parse the adjuncts, by the same the reasoning applied previously. However, there are now only Cat_1 ways to parse "saw the boy with a telescope" whereas there were Cat_2 ways before. The complement interpretations (71a-c) have been excluded by the two argument restriction.

Now suppose the oracle had selected the three argument form of "see". How could we take advantage of this information? In this case, the power series for S is the difference between (66) and (73).

$$(74) \quad S = \left(N \sum_i Cat_i (P\,N)^i + 1 \right) V\,N \sum_i (Cat_{i+1} - Cat_i)(P\,N)^i \sum_i (i+1)\,adj^i$$

We hope to generalize this approach to handle selectional restrictions and agreement facts.

11. Inverse Transforms

(Inverse transforms are a fairly self-contained topic which can be left for a second reading of this paper.)

The previous few sections have outlined how it might be possible to use formal power series to compile a grammar into a form for more efficient processing. This section will discuss the inverse process. That is, given a compiled representation of the grammar, how can we recover a form suitable for linguistic analysis? This section will present a partial solution which we found very useful for analyzing EQSP.

Let us consider an anecdotal example based on our experience with the EQSP conjunction mechanism. Deep inside the code, there was a function called *syntactically-parallelp* which decided whether or not to conjoin two constituents. Over the years, this function had acquired so many special case heuristics that it was no longer understandable. However, we were able to determine the ambiguity coefficients by running EQSP on the following sequence of conjunction sentences:

1 It was.

1 It was actual products.

2 It was actual products and actual products.

3 It was actual products and actual products and actual products.

5 It was actual products and actual products and actual products and actual products.

8 It was actual products and actual products and actual products and actual products and actual products.

13 It was actual products and actual products and actual products and actual products and actual products and actual products.

21 It was actual products and actual products and actual products and actual products and actual products and actual products.

To our surprise the ambiguity coefficients did not follow the Catalan sequence as predicted, but rather they followed another well-known sequence called the Fibonacci numbers [8]. The first few Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, ... The next value is formed by taking the sum of the two previous values, or more precisely:

$$(75)\quad \begin{aligned} \mathrm{Fib}_1 &= \mathrm{Fib}_0 = 1 \\ \mathrm{Fib}_n &= \mathrm{Fib}_{n-1} + \mathrm{Fib}_{n-2} \end{aligned}$$

We can model the sentences above with the following power series (ignoring the word "and" which complicates the analysis in ways that are irrelevant to the current discussion):

$$(76)\quad S = \text{It was} \sum_i \mathrm{Fib}_i\,(\text{actual products})^i$$

We were then able to recover the grammar from the power series because the Fibonacci series has a well-known inverse transform. That is, a power series with Fibonacci coefficients obeys the following identity.

$$(77)\quad \sum_i \mathrm{Fib}_i\, x^i = \frac{1}{1 - x - x^2}$$

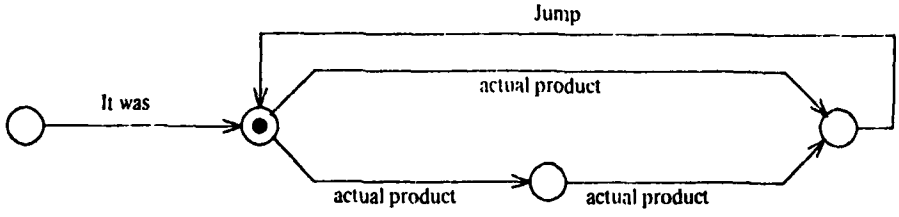
The reader can verify that this identity is correct by performing the long division. We were fortunate in this case that the inverse transform for the Fibonacci numbers has a well-known closed form. In general, such closed forms are very difficult to discover (if they exist at all), and for this reason, it can be very difficult or even impossible to find a linguistically attractive grammar for an arbitrary processor. Nevertheless, closed forms do exist for a large number of interesting cases. With some practice and a few educated guesses based on partial knowledge of what the machine is doing, one can successfully "crack" quite a number of constructions. At least, this has been our experience with BQSP.

Returning to the conjunction sentences, we now have a closed form of the power series:

$$(78)\quad S = \text{It was} \frac{1}{1 - (\text{actual products}) - (\text{actual products})^2}$$

This has the following ATN realization:

(79) S:



We observe that EQSP employs a heuristic which prevents conjuncts from attaching more than two phrases back. A full non-heuristic conjunction mechanism would permit conjuncts to "fold back" arbitrarily far. In which case the conjunction mechanism would be a Catalan grammar.

In this way, we were able to perform the inverse transform on the ambiguity coefficients in order to recover the underlying behavior of the EQSP conjunction mechanism. We are now in a position to rewrite syntactically-parallelp to be more comprehendable and more efficient, without disturbing the external behavior.

12. Conclusion

We began our discussion with the observation that certain grammars are "every way ambiguous" and suggested that this observation could lead to improved parsing performance. Catalan grammars were then introduced to remedy the situation so that the processor could delay attachment decisions until it discovers some more useful constraints. Until such time, the processor can do little more than note that the input sentence is "every way ambiguous". We suggested that a table lookup scheme might be an effective method to implement such a processor.

In some sense, this approach is a formalization of a very old idea. That is, it has been noticed for a long time that it might be advantageous to enrich a processor with the capability to attach certain ambiguous constituents to several places in a single step. Pseudo-attachment [1: pp. 65-71] and permanent predictable ambiguity [14: pp. 64-65] are two such proposals. However, these mechanisms have always lacked a precise interpretation; Catalan grammars provide a much more formal way of coping with "every way ambiguous" grammars.

We then introduced rules for combining primitive grammars, such as Catalan grammars, into composite grammars. This linear systems view "bundles up" all the parse trees into a single concise description which is capable of telling us everything we might want to know about the parses, (including how much it might cost to ask a particular question). This abstract view of ambiguity enables us to ask questions in the most

convenient order, and to to delay asking until it is clear that that the pay-off will exceed the cost. This abstraction was very strongly influenced by the notion of delayed binding.

We have presented combination rules in three different representation systems: power series, ATNs, and context-free grammars, each of which contributed its own insights. Power series are convenient for defining the algebraic operations, ATNs are most suited for discussing implementation issues, and context-free grammars enable the shortest derivations. Perhaps the following quotation best summaries our motivation for alternating among these three representation systems:

(80) "A thing or idea seems meaningful, only when we have *several* different ways to represent it — different perspectives and different associations. Then you can turn it around in your mind, so to speak: however it seems at the moment, you can see it another way; you never come to a full stop." [12: p. 19]

In each of these representation schemes, we have introduced five primitive grammars: Catalan, Unit Step, 1, and 0, and terminals, and four composition rules: addition, subtraction, multiplication and division. We have seen that it is often possible to employ these analytic tools in order to re-organize (compile) the grammar into a form more suitable for processing efficiently. We have identified certain situations where the ambiguity is combinatoric, and have sketched a few modifications to the grammar that enables processing to proceed in a more efficient manner. In particular, we have observed it is important for the grammar to avoid referencing quantities that are not easily determined such as the dividing point between a noun phrase and a prepositional phrase as in

(81) Put the block in the box on the table in the kitchen ...

We have seen that the desired re-organization can be achieved by taking advantage of the fact that the auto-convolution of a Catalan series produces another Catalan series. This reduced processing time from $O(n^3)$ to $O(n)$. Similar analyses have been discussed for a number of lexically and structurally ambiguous constructions, culminating with the example in section 9 where we transformed a grammar into a form that could be parsed by a single left-to-right pass over the terminal elements. Currently, these grammar re-formulations have to be performed by hand. It ought to be possible to automate this process so that the re-formulations could be performed by a grammar compiler. We leave this project open for future research.

13. Acknowledgments

We would like to thank Jon Allen, Lowell Hawkinson, Kris Halvorsen, Bill Long, Mitch Marcus, Rohit Parikh and Peter Szolovits for their very useful comments on earlier drafts. We would especially like to thank Bill Martin for initiating the project.

References

1. Church, K., *On Memory Limitations in Natural Language Processing*, MIT/LCS/TR-245, 1980.

2. Church, K. and Kaplan, R., *Removing Recursion from Natural Language Processors Based on Phrase-Structure Grammars*, paper presented at conference on Modeling Human Parsing Strategies, University of Texas at Austin, 1981.

3. Dostert, B. and Thompson, F., *How Features Resolve Syntactic Ambiguity*, in Proceedings of the Symposium on Information Storage and Retrieval, Minker, J. and Rosenfeld, S. (eds.), 1971.

4. Earley, J., *An Efficient Context-Free Parsing Algorithm*, Communications of the ACM, Volume 13, Number 2, February, 1970.

5. Harrison, M., *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

6. Kaplan, R., *Augmented Transition Networks as Psychological Models of Sentence Comprehension*, Artificial Intelligence, 3, 77-100, 1972.

7. Kaplan, R. and Bresnan, J., *Lexical-Functional Grammar: A Formal System for Grammatical Representation*, in Bresnan (ed.), *The Mental Representation of Grammatical Relations*, MIT Press, 1981.

8. Knuth, D., *Fundamental Algorithms*, Vol 1 in *The Art of Computer Programming*, Addison Wesley, 1975.

9. Liu, C. and Liu, J., *Linear Systems Analysis*, McGraw-Hill, 1975.

10. Malhotra, A., *Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis*, MIT/LCS/TR-146, 1975.

11. Martin, W., Church, K., and Patil, R., *Preliminary Analysis of a Breadth-First Parsing Algorithm: Theoretical and Experimental Results*, MIT/LCS/TR-261, 1981.

12. Minsky, M., *Music, Mind, and Meaning*, MIT A.I. Memo No. 616, 1981.

13. Oppenheim, A. and Schafer, R., *Digital Signal Processing*, Prentice-Hall, 1975.

14. Sager, N., *The String Parser for Scientific Literature*, in Rustin, R. (ed.), *Natural Language Processing*, Algorithmic Press, 1973.

15. Salomaa, A., *Formal Languages*, Academic Press, 1973.

16. Woods, W., *Transition Network Grammars for Natural Language Analysis*, CACM, 13:10, 1970.

OFFICIAL DISTRIBUTION LIST

Director — 2 copies
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

Attention: Program Management

Office of Naval Research — 3 copies
800 North Quincy Street
Arlington, Virginia 22217

Attention: Marvin Denicoff, Code 437

Office of Naval Research — 1 copy
Resident Representative
Massachusetts Institute of Technology
Building E19-628
Cambridge, Mass. 02139

Attention: A. Forrester

Director — 6 copies
Naval Research Laboratory
Washington, D.C. 20375

Attention: Code 2627

Defense Technical Information Center — 12 copies
Cameron Station
Arlington, Virginia 22314

Office of Naval Research — 1 copy
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, Mass. 02210

National Science Foundation — 2 copies
Office of Computing Activities
1800 G Street, N.W.
Washington, D.C. 20550

Attention: Thomas Keenan, Program Director


FILMED
8