American Journal of Computational Linguistics *Microfiche 47*

A SURVEY OF SYNTACTIC ANALYSIS PROCEDURES FOR NATURAL LANGUAGE

RALPH GRISHMAN

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York. 10012

This survey was prepared under contract No. N00014-67A-0467-0032 with the Office of Naval Research, and was originally issued as Report No. NSO-8 of the Courant Institute of Mathematical Sciences, New York University.

Copyright © 1976
Association for Computational Linguistics

A SURVEY OF SYNTACTIC ANALYSIS PROCEDURES FOR NATURAL LANGUAGE

RALPH GRISHMAN

Computer Science Department
Courant Institute of Mathematical Sciences
New York University

This survey was prepared under contract No. N00014-67A-0467-0032 with the Office of Naval Research, and was originally issued as Report No. NSO-8 of the Courant Institute of Mathematical Sciences, New York University.

SUMMARY

This report includes a brief discussion of the role of automatic syntactic analysis, a survey of parsing procedures used in the analysis of natural language, and a discussion of the approaches taken to a number of difficult linguistic problems, such as conjunction and graded acceptability. It also contains precise specifications in the programming language SETL of a number of parsing algorithms, including several context-free parsers, a unrestricted rewriting rule parser, and a transformational parser.

Table of Contents

		Page
1.	INTRODUCTION	4
1.1	The Role of Syntactic Analysis	6
1.2	Computational and Theoretical Linguistics	9
2.	GENERAL SURVEY OF PARSING PROCEDURES	11
2.1	Early Systems: Context-Free and Context-Sensitive Parsers	11
2.2	Transformational Analyzers: First Systems	14
2.3	Transformational Analyzers: Subsequent Developments	18
2.4	Other Syntactic Analysis Procedures	24
2.5	Parsing with Probability and Graded Acceptability	26
2.6	Conjunction and Adjunction	28
3.	ALGORITHM SPECIFICATIONS	31
3.1	Parsing Algorithms for Context-Free Grammars	31
3.2	A Parser for Unrestricted Rewriting Rule Grammars	54
3.3	Parsing Procedures for Transformational Grammars	60
	APPENDIX. A Very Short Introduction to SETL	84
	BIBLIOGRAPHY	92

1. INTRODUCTION

The Computer Science Department of New York University, under contract to the Office of Naval Research, has prepared a series of reports examining selected areas of artificial intelligence. We hope in these critical surveys to place in perspective the main lines of past research and thereby perhaps to suggest fruitful directions for future work. As part of these surveys we have prepared precise specifications, in the programming language SETL, of some of the basic algorithms in each area. These specifications are intended to provide clear points of reference for structuring our review of past work.

* * * * *

This first report is concerned with natural language processing systems, systems which are able to accept instructions or data in natural language. In very general terms, these systems process the text through several stages:

(1) syntactic: analyzes the structure of each sentence (or other text unit) and rearranges the elements of the sentence to simplify its structures; this stage should recognize paraphrases due to alternative arrangements of words in a sentence

(2) semantic: restructures the sentences into the form used for internal processing, such as inference or data retrieval; depending on the application, the output may be a command in an information retrieval language, a structure based on some set of semantic "primitives", or a tabular structure suitable as a data base; this stage should recognize some of the paraphrases due to alternative choices of words.

(3) pragmatic: interprets the text based on particular context (problem situation or data base); this stage should recognize sentences which are equivalent in effect (such as "Throw that switch." and "Turn on the light.").

The reader will note that these stages are very vaguely characterized. Current language processing systems differ very greatly

in their structure and not even these general divisions can be identified in all systems.

The pragmatic stage is the most heterogeneous and the common threads which do appear are based more on general problem-solving methods than on specifically linguistic techniques. Since the semantic stage maps into the notation required by the pragmatics, it is correspondingly varied. There is, however, a fair amount of current research on the selection of semantic primitives or semantic classes; some of this research is reviewed in the proceedings of a recent Courant Institute symposium on Directions in Artificial Intelligence (Courant Computer Science Report No. 7).

The syntactic stage is by far the best established and most clearly defined. There is a general (although far from total) agreement on the most basic underlying principles, and there are a number of widely-used procedures. For this stage, therefore, it seems possible to present a survey of current research in some organized fashion. In the report which follows, we have endeavored to show the relation between the various syntactic analyzers in terms of their historical development, linguistic assumptions, and analysis procedures. For a broader survey of automated language processing, readers are referred to [Walker 1973].

1.1 The Role of Syntactic Analysis

The systems we shall be describing are all motivated by particular applications requiring natural language input, rather than by purely linguistic considerations. Consequently, the parsing of a text (determining its structure) will be viewed as an essential step preliminary to processing the information in the text, rather than as an end in itself.

There are a wide variety of applications involving natural language input, such as machine translation, information retrieval, question answering, command systems, and data collection. It may therefore seem at first that there would be little text processing

which would be generally useful beyond the determination of a structural description (e.g. a parse tree) for each sentence. There are, however, a number of operations which can regularize sentence structure, and thereby simplify the subsequent application-specific processing. For example, some material in sentences (enclosed in brackets in the examples below) can be omitted or "zeroed":

John ate cake and Mary [ate] cookies.
. . . five or more [than five] radishes
He talks faster than John [talks].
. . . the man [whom] I met . . .

Sentence structure can be regularized by restoring such zeroed information. Other transformations can relate sentences with normal word order (I crushed those grapes. That I like wine is evident.) to passive (Those grapes were crushed by me.) and cleft (It is evident that I like wine.) constructions, and can relate nominal (the barbarians' destruction of Rome) and verbal (the barbarians destroyed Rome) constructions. Such transformations will permit further (e.g., semantic) processing to concern itself with a much smaller number of structures. In addition, if the structures are appropriately chosen, operator-operand relations should be clearly evident in the output of the syntactic stage.

Some lexical processes, such as nominalization and lexical decomposition, are considered syntactic by some and semantic by others. Whether a clear division between the syntactic and semantic stages is possible at all has been a major point of controversy in linguistics -- between interpretive and generative semanticists -- over the past decade. We may therefore expect that, while some transformations will clearly be the province of the syntactic stage and others the province of the semantic stage, there will be a considerable fuzzy area in between. This, however, should not disqualify automatic syntactic analysis as an area of separate research;

there is hardly a field of science or engineering which is clearly delineated from its neighbors.

The last few years have seen most work in language processing devoted to the development of integrated systems, combining syntactic, semantic, pragmatic, and generative components. This was a healthy and predictable reaction to the earlier research, which had largely approached syntactic processing in isolation from these other areas. It produced some systems whose modest successes dispelled the skepticism that natural language processors would ever be able to do anything. These systems indicated how syntactic, semantic, and pragmatic information must interact to select the correct sentence analysis.

It is now generally understood that syntactic processing by itself is inadequate to select the intended analysis of a sentence. We should not conclude from this, however, that it is impossible to study the processes of syntax analysis separately from the other components. Rather, it means that syntax analysis must be studied with an understanding of its role in a larger system and the information it should be able to call upon from other components (i.e., the processing which the subsequent components must do to select among the analyses produced by the syntactic component).

While recognizing the importance of total systems in insuring that none of the problems has fallen in the gaps between stages and been forgotten, it still seems that more specialized research projects are essential if the field of natural language processing is to mature. The development of another total system will not advance the field unless it endeavors to perform some particular processing task better than its predecessors; the problems are too vast for each research project to usefully attack the problems involved in all the phases of processing at once.

* * *

Some researchers have asserted recently that natural language processing can be done without syntax analysis. It seems to us that such claims are exaggerated, but they do arise out of some observations that are not without validity:

(1) For the relatively simple sentences whose semantics is within the scope of current artificial intelligence systems, sophisticated syntactic processing is unnecessary.

This was certainly true of some early question-answering systems, whose syntax was limited to a few fixed imperative structures, into which adjective and prepositional phrase modifiers could be inserted. It is questionable whether this is true of the most syntactically sophisticated of today's systems (such as Petrick's) In any case, it is hard to imagine how sentences of the complexity typical in technical writing could be understood without utilizing syntactic (as well as semantic) restrictions to select the correct analysis.

(2) Syntactic analysis may appear in guises other than the traditional parsing procedures; it can be interwoven with other components of the system and can be embedded into the analysis programs themselves. This will often increase the parsing speed considerably.

The "grammar in program" approach which characterized many of the early machine translation efforts is still employed in some of today's systems. Its primary justification seems to be parsing efficiency, but this should be a secondary consideration for research purposes at present, since most current systems are able to parse (or, as often, reject as unanalyzable) a sentence in under a minute. More important as research goals should be the ability to manage grammatical complexity and the ability to communicate successful methods to others. In both these regards, a syntactic analyzer using a unified, semiformal set of rules is bound to be more effective.

(3) Syntax analvsis can be driven by semantic analysis (instead of being a separate, earlier stage), and, in particular, can be done by looking for semantic patterns in the sentence.

Syntax analysis is done separately because there are rules of sentence formation and transformation which can be stated in terms of the relatively broad syntactic categories (tensed verb, count noun, etc.). If the semantic classes are subcategorizations of the syntactic ones, then clearly the transformations could be stated in terms of sequences of semantic classes. For those transformations which are properly syntactic, however, we would find that several transformations at the semantic stage would be required in place of one at the syntactic stage; certain useful generalizations would be lost.

The strongest argument of those advocating a semantics-driven syntax is the ability of people to interpret sentences from semantic clues in the face of syntactic errors or missing information ("I want to xx to the movies tonight."). This argument works both ways, however -- people can also use syntactic rules when semantics is lacking; for example, to understand the function of a word in a sentence without knowing its meaning ("Isn't that man wearing a very frimple coat?"). Ultimately, we want an analyzer which can work from partial information of either kind, and research in that direction is to be welcomed (some work on parsing in the face of uncertainty has been done by speech-understanding groups). At the same time, since successful processing of "perfect" sentences is presumably a prerequisite for processing imperfect sentences, it seems reasonable to continue devoting substantial effort to the considerable problems which remain in analyzing perfect sentences.

1.2 Computational and Theoretical Linguistics

Theoretical linguists and the sort of computational linguists we have been considering espouse quite different research objectives. A primary interest of transformational linguists is explaining grammatical competence -- how people come to accept some sentences as grammatical and reject others as ungrammatical. In particular, they are concerned with

language universals -- principles of grammar which apply to all natural languages.

Computational linguists, in contrast, are usually delighted if they can manage to handle one language (two, if they're translating). Their primary concern lies in transforming sentences -- often assumed to be grammatical -- into a form acceptable to some particular application system. They are concerned with the efficiency of such processing, whereas theoretical linguists generally don't worry about the recognition problem at all.

Nonetheless, the two specialties should have many common areas of interest. Questions of grammaticality are important, because experience has shown that a grammatical constraint which in one case determines if a sentence is or is not acceptable will in other cases be needed to choose between correct and incorrect analyses of a sentence. The relations between sets of sentences, which are a prime focus of transformational grammar, particularly in the Harrisian framework, are crucial to the success of syntactic analysis procedures, since they enable a large variety of sentences to be reduced to a relatively small number of structures.

More generally, both specialties seek to understand a particular mode of communication. Traditional linguists are interested in a mode which has evolved as an efficient means of communicating ideas between people; ultimately, we may hope that they will understand not only the principles of language structure, but also some of the reasons why language has developed in this way. Computational linguists, in studying how language can be used for man-machine communication, are really asking much the same questions. They want to develop a mode of communication for which people are naturally suited and they want to understand the principles for designing languages which are efficient for communicating ideas.

2. GENERAL SURVEY OF PARSING PROCEDURES

We can impose several rough groupings on the set of parsers in order to structure the following survey. To begin with, we may try to separate those systems developed with some reference to transformational theory from the nontransformational systems. This turns out also to be an approximate historical division, since most systems written since 1965 have made some connection with transformational theory, even though their methods of analysis may be only distantly related to transformational mechanisms.

The transformational systems may in turn be divided into those parsers which have been systematically derived from a specific transformational generative grammar and those which have "sacrificed" this direct connection with a generative grammar in order to obtain a more direct and efficient algorithm for recovering base structures. This division appears to be in part a result of our inadequate theoretical understanding of transformational grammars, and may be reduced by some recent theoretical work on transformational grammars.

2.1 Early Systems: Context-Free and Context-Sensitive Parsers

The pretransformational systems, developed mostly between 1959 and 1965, were, with a few exceptions, parsers for context-free languages, although cloaked in a number of different guises. These systems were based on immediate constituent analysis, dependency theory, linguistic string theory, or sometimes no theory at all.

The largest and probably the most important of these early projects was the Harvard Predictive Analyzer [Kuno 1962]. A predictive analyzer is a top-down parser for context-free grammars written in Greibach normal form; this formulation of the grammar was adopted from earlier work by Ida Rhodes for her Russian-English translation project. The size of the

grammar was staggering: a 1963 report [Kuno 1963] quotes figures of 133 word classes and about 2100 productions. Even with a grammar of this size, the system did not incorporate simple agreement restrictions of English syntax. Since the program was designed to produce parses for sentences which were presumed to be grammatical (and not to differentiate between grammatical and nongrammatical sentences), it was at first hoped that it could operate without these restrictions. It was soon discovered, however, that these restrictions were required to eliminate invalid analyses of grammatical sentences. Because the direct inclusion of, say, subject-verb number agreement would cause a large increase in an already very large grammar, the Harvard group chose instead to include a special mechanism in the parsing program to perform a rudimentary check on number agreement. Thus the Harvard Predictive Analyzer, though probably the most successful of the context-free analyzers, clearly indicated the inadequacy of a context-free formulation of natural language grammar.

The Harvard Predictive Analyzer parsing algorithm progressed through several stages. The first version of the predictive analyzer produced only one analysis of a sentence. The next version introduced an automatic backup mechanism in order to produce all analyses of a sentence. This is an exponential time algorithm, hence very slow for long sentences; a 1962 report gives typical times as 1 minute for an 18 word sentence and 12 minutes for a 35 word sentence. An improvement of more than an order of magnitude was obtained in the final version of the program by using a bit matrix for a path-elimination technique [Kuno 1965]. When an attempt was made to match a nonterminal symbol to the sentence beginning at a particular word and no match was found, the corresponding bit was turned on; if the same symbol came up again later in the parsing at the same point in the sentence, the program would not have to try to match it again.

Another important early parser was the immédiate constituent analyzer used at RAND. This system used a grammar in Chomsky normal form and a parsing algorithm designed by John Cocke, which produced all analyses bottom-up in a single left-to-right scan of the sentence [Hays 1967]. This was a fast algorithm but because all parses were developed simultaneously, it needed a lot of space for long sentences; the Rand system appears therefore to have been limited to sentences of about 30 words.

A different bottom-up analysis procedure was used in the first linguistic string analysis program developed at the University of Pennsylvania [Harris 1965]. This procedure, called a cycling cancelling automaton, makes a series of left-to-right passes through the sentence; in each pass one type of reduction was performed. The string parser recognized two classes of strings: first order, not containing verb-object, and second order, containing verb-object; the reduction of the sentences was correspondingly done in two stages. In addition to these reductions, which corresponded to context-free rules the parsing program also included some syntactic restrictions which were checked when second order strings were reduced. A system incorporating this cycling automaton scheme was later used by Bross at Rosewll Park for the analysis of medical reports [Bross 1968, Shapiro 1971].

As far as we know, only one major parsing system has been developed using a context-sensitive phrase structure grammar. This was DEACON, Direct English Access and Control, which was designed as a natural language interface to a command, control, and information retrieval system for the Army and was developed at General Electric [Craig 1966]. DEACON was one of the first systems to provide flexible, systematic interaction between the parser and the semantic component. Associated with each production in the grammar was a semantic rule. These rules operated on a ring-structured data base and had the functions

of locating, adding, and changing information in the data base. The parsing was done bottom-up, developing all analyses of the sentence in parallel. As each reduction was performed, the associated semantic rule was invoked. In the case of a query, the sequence of rules associated with the correct analysis was supposed to locate the desired answer in the data base. In some cases a rule could not be applied to the data base (e.g., a particular relation between two items did not exist); the rule then returned a failure signal to the parser, indicating that the analysis was semantically anomalous, and this analysis was aborted.

Woods has noted [Woods 1970a] that the parser used in the DEACON project may produce redundant parses, and has given a parsing algorithm for context-sensitive languages which remedies this deficiency.

2.2 Transformational Analyzers: First Systems

When the theory of transformational grammar was elaborated in the early 1960's there was considerable interest in finding a corresponding recognition procedure. Because the grammar is stated in a generative form, however, this is no simple matter. A (Chomsky) tree transformational grammar consists of a set of context-sensitive phrase structure rules, which generate a set of base trees, and a set of transformations, which act on the base trees to produce the surface trees. A (Harris) string transformational grammar consists of a finite set of sequences of word categories, called kernel sentences, and a set of transformations which combine and modify these kernel sentences to make the other sentences of the language. There are at least three basic problems in reversing the generative process:

(1) for a tree transformational grammar, assigning to a given sentence a set of parse trees which includes all the surface trees which would be assigned by the transformational grammar

(2) given a tree not in the base, determining which sequences of transformations might have applied to generate this tree
(3) having decided on a transformation whose result may be the present tree, undoing this transformation

If we attack each of these problems in the most straightforward manner, we are likely to try many false paths which will not lead to an analysis. For the first problem, we could use a context-free grammar which will give all the surface trees assigned by the transformational grammar, and probably lots more The superabundance of "false" surface trees is aggravated by the fact that most English words have more than one word category (play more than one syntactic role), although normally only one is used in any given sentence. For the second and third problems, we can construct a set of reverse transformations; however, since we are probably unable to determine uniquely in advance the transformations which produced a given tree, we will have to try many sequences of reverse transformations which will not yield a base tree.

Because of these problems, the earliest recognition procedure, suggested by Matthews, was based on the idea of synthesizing trees to match a given sentence. Although some checks were to have been made against the sentence during the generation procedure, it was still an inherently very inefficient procedure and was never implemented. Two major systems were developed in the mid-60's, however, which did have limited success: the system of Zwicky et al. at MITRE and that of Petrick.

The transformational generative grammar from which the MITRE group worked had a base component with about 275 rules and a set of 54 transformations [Zwicky 1965]. For the recognition procedure they developed manually a context-free "covering" grammar with about 550 productions to produce the surface trees and a set of 134 reverse transformational rules. Their recognition procedure had four phases:

(1) analysis of the sentence using the context-free covering grammar (with a bottom-up parser)

(2) application of the reverse transformational rules

(3) for each candidate base tree produced by steps (1) and (2), a check whether it can in fact be generated by the base component

(4) for each base tree and sequence of transformations which passes the test in step (3), the (forward) transformations are applied to verify that the original sentence can in fact be generated

(The final check in step (4) is required because the covering grammar may lead to spurious matches of a transformation to the sentence in the reverse transformational process and because the reverse transformations may not incorporate all the constraints included in the forward transformations.) The covering grammar produced a large number of spurious surface analyses which the parser must process. The 1965 report for example, cites a 12 word sentence which produced 48 parses with the covering grammar; each must be followed through steps (2) and (3) before most can be eliminated. The system was therefore very slow; 36 minutes were required to analyze one 11 word sentence.

Two measures were taken by the MITRE group to speed up the program: "super-trees" and rejection rules [Walker 1966]. "Super-trees" was the MITRE term for a nodal span representation, in which several parse trees were represented in a single structure. They intended to apply the reverse transformations to these super-trees, thus processing several possible surface trees simultaneously; it is not clear if they succeeded in implementing this idea. Rejection rules were tests which were applied to the tree during the reverse transformational process (step (2) above), in order to eliminate some trees as early as possible in the parsing. The rejection rules incorporated some constraints which previously were only in the forward transformational component, and so eliminated some trees in step (2) which before had survived to step (4). The rejection

rules had a significant effect on parsing times -- the 11 word sentence which took 36 minutes before now took only 6 $\frac{1}{2}$

The system developed by Petrick [Petrick 1965, 1966; Keyser 1967] is similar in outline: applying a series of reverse transformations, checking if the resulting tree can be generated by the base component, and then verifying the analysis by applying the forward transformations to the base tree. There are, however, several differences from the MITRE system, motivated by the desire to have a parser which could be produced automatically from the generative formulation of the grammar. Petrick devised a procedure to generate, from the base component and transformations, an enlarged context-free grammar sufficient to analyze the surface sentence structures. He also automatically converted a set of forward transformations meeting certain conditions into pseudo-inverse (reverse) transformations. His parsing procedure also differed from the MITRE algorithm in the way in which the reverse transformations are applied. In the MITRE program reverse transformations operated on a sentence tree, just like forward transformations in a Chomsky grammar. Petrick, on the other hand, did not construct a surface tree in the analysis phase; when a particular reverse transformation came up for consideration, he built just enough structure above the sentence (using the enlarged context-free grammar) to determine if the transformation was applicable. If it was, the transformation was applied and the structure above the sentence then torn down again; what was passed from one reverse transformation to the next was only the string of word categories. In the verifying phase, of course, Petrick had to follow the rules of Chomsky grammar and apply the forward transformations to a sentence tree.

The price for generality was paid in efficiency. Petrick's problems were more severe than MITRE's for two reasons. First, the absence of a sentence tree during the application of the reverse transformational rules meant that many sequences of

reverse transformations were tried which did not correspond to any sequence of tree transformations and hence would eventually be rejected. Second, if several reverse transformations could apply at some point in the analysis, the procedure could not tell in advance which would lead to a valid deep structure. Consequently, each one had to be tried and the resulting structure followed to a deep structure of a "dead end" (where no more transformations apply). This produces a growth in the number of analysis paths which is exponential in the number of reverse transformations applied. This explosion can be avoided only if the reverse transformations include tests of the current analysis tree to determine which transformations applied to generate this tree. Such tests were included in the manually prepared reverse transformations of the MITRE group, but it would have been far too complicated for Petrick to produce such tests automatically when inverting the transformations.

Petrick's system has been significantly revised over the past decade [Petrick 1973, Plath 1974a]. In the current system the covering grammar and reverse transformations are both prepared manually. The transformational decomposition process works on a tree (as did MITRE's), and considerably flexibility has been provided in stating the transformations and the conditions of applicability. The transformations and conditions may be stated either in the traditional form (used by linguists) or in terms of elementary operations combined in LISP procedures. The resulting system is fast enough to be used in an information retrieval system with a grammar of moderate size; most requests are processed in less than one minute.

2.3 Transformational Analyzers: Subsequent Developments

One result of the early transformational systems was a recognition of the importance of finding an efficient parsing procedure if transformational analysis was ever to be a useful

technique. As the systems indicated, there are two main obstacles to an efficient procedure. First, there is the problem of refining the surface analysis, so that each sentence produces fewer trees for which transformational decomposition must be attempted. This has generally been approached by using a more powerful mechanism than a context-free parser for the surface analysis. Second, there is the problem of determining the base structure (or kernel sentences) from the surface structure in a relatively direct fashion. This has generally been done by associating particular rules for building the deep structure with rules of the surface structure analysis. The approach here has generally been ad hoc, developing a reverse mapping without explicit reference to a corresponding set of forward transformations.

Several groups which have played a significant role in the development of current parsing systems have been tied together by their common use of recursive transition networks. Although their use of these transition networks is not central to their basic contribution, it is frequently referred to and so deserves a few words of explanation. A transition network is a set of nodes (including one initial and at least one terminal node) and a set of directed arcs between the nodes, labeled with symbols from the language; it is a standard representation for regular languages. A recursive transition network is a set of transition networks in which the arcs of one network may also be labeled with the names of other networks; it is a form of representation of context-free languages. In contrast to the usual context-free phrase structure grammars, this is equivalent to allowing regular expressions in place of finite sequences of elements in productions. This does not increase the weak generative capactity of the grammars, but allows nonrecursive formulations for otherwise recursive constructions.

The first system using such a network was developed by Thorne, Bratley, and Dewar at Edinburgh [Thorne 1968, Dewar 1969]. They started with a regular base grammar, i.e., a transition

network. The importance of using a regular base lies in their claim that some transformations are equivalent in effect to changing the base to a recursive transition network. Transformations which could not be handled in this fasion, such as conjunction, were incorporated into the parsing program. Parsing a sentence with this surface grammar should then also give some indication of the associated base and transformational structure. Their published papers do not describe, however, the process by which the surface grammar is constructed and so it is not clear just how the transformation and base structure is extracted from their parse.

The recursive transition network was developed into an augmented recursive transition network grammar in the system of Bobrow and Fraser [Bobrow 1969]. An augmented network is one in which an arbitrary predicate, written in some general purpose language (in this case, LISP) may be associated with each arc in the network. A transition in the network is not allowed if the predicate associated with the arc fails. These predicates perform two functions in the grammar. First, they are used to incorporate restrictions in the language which would be difficult or impossible to state within the context-free mechanisms of the recursive network, e.g., agreement restrictions. Second, they are used to construct the deep structure tree as the sentence is being parsed.

The augmented transition network was further developed by Woods at Bolt Beranek and Newman. In order to regularize the predicates, he introduced a standard set of operations for building and testing the deep structure [Woods 1970b]. He considerably enlarged the scope of the grammar and added a semantic component for translating the deep structure into information retrieval commands. With these additions, the system served as a moderately successful natural language input interface to a retrieval system for data about moon rocks [Woods 1972, 1973]. The augmented transition network, and in particular the formalism developed by Woods, has proven to be an

effective instrument for constructing natural language front-ends which is relatively simple to implement and use; it is probably the most widely used procedure today.

Like several of the systems described above, Proto-RELADES, developed } IBM Cambridge [Culicover 1969], tried to obtain an efficient transformational decomposition algorithm by linking the rules for building the deep structure to the productions of the surface grammar. Their surface grammar was also augmented by restrictions (in PL/I this time). However, their system differed from those mentioned earlier in several important respects: First, the surface grammar allowed context-sensitive as well as context-free rules. Second, the rules which built the deep structure during the parse were in the form of reverse transformations acting on an (incomplete) sentence tree (in contrast to the rules used by Woods, for example, which first put words into registers labeled "subject", "verb", and "object" and later build a tree out of them). Proto-RELADES was tested as a restricted English language preprocessor for a library card catalog retrieval system [Loveman 1971].

One drawback of these procedures was the relatively ad hoc methods, from a linguistic point of view, used to construct the surface grammars and to tie them in to the appropriate reverse transformations. A more principled approach to transformational decomposition was proposed by Joshi and Hiz [Joshi 1962, Hiz 1967]. In contrast to the systems described above, their procedure was based on Harris' string transformational grammar. One advantage of the Harrisian theory over that of Chomsky is the theoretical basis it provides for the segmentation of the sentence into "linguistic strings" (Chomsky's theory, in contrast, makes no general assertions about the surface structure of sentences.) The procedure of Joshi and Hiz was predicated on the claim that, from an analysis of the sentence into linguistic strings, one could directly determine the transformations which acted to produce the sentence, without having to try many sequences

of reverse transformations. Their proposed system therefore consisted of a procedure for linguistic string analysis (a context-free parsing problem at the level of simplification of their original proposal) and a set of rules which constructed from each string a corresponding kernel-like sentence.

Their original proposal was a simplified scheme which accounted for only a limited set of transformations. It has been followed by a good deal of theoretical work on adjunct grammars and trace conditions [Joshi 1973] which has laid a formal basis for their procedures. These studies indicate how it may be possible, starting from a transformational grammar not specifically oriented towards recognition, to determine the features of a sentence which indicate that a particular transformation applied in generating it, and hence to produce an efficient analysis procedure.

Another group which has used linguistic string analysis is the Linguistic String Project at New York University, led by Sager [Sager 1967, 1973; Grishman 1973a, 1973b]. Their system, which has gone through several versions since 1965, is based on a context-free grammar augmented with restrictions. Because they were concerned with processing scientific text, rather than commands or queries, they were led to develop a grammar of particularly broad coverage. The present grammar has about 250 context-free rules and about 200 restrictions; although not as swift as some of the smaller systems, the parser is able to analyze most sentences in less than one minute. Because of the large size of their grammar, this group has been particularly concerned with techniques for organizing and specifying the grammar which will facilitate further development. In particular, the most recent implementation of their system has added a special language designed for the economical and perspicuous statement of the restrictions [Sager 1975].

One of the earlier versions of this system, with a much more restricted grammar, was used as the front end for an information

retrieval system developed by Cautin at the University of Pennsylvania [Cautin 1969].

The Linguistic String Project system has recently been extended to include a transformational decomposition phase; this phase follows the linguistic string analysis [Hobbs 1975]. As in the case of the Joshi-Hiz parser, the strings identified in the sentence generally indicate which reverse transformations must be applied. The transformations are written in an extension of the language which was used for writing the restrictions.

The systems of Woods, Petrick, and Sager exhibit a range of approaches to the problem of transformational decomposition. Their parsing procedures are similar in many respects: they have a context-free grammar as the framework for their surface analysis, and they use procedures both to express grammatical constraints and to effect the reverse transformations. Petrick's system differs from the others in two primary respects: the restrictions on the context-free grammar are imposed by filtering transformations which act early in the transformational phase to reject ill-formed trees, rather than by procedures operating during the surface analysis. This would seem to be disadvantageous from the point of view of efficiency, since erroneous parses which might be aborted at the beginning of the surface analysis must be followed through the entire surface analysis and part of the transformational decomposition. Second, the transformations are not associated with particular productions of the surface grammar, but rather with particular patterns in the tree ("structural descriptions"), so pattern matching operations are required to determine which transformations to apply. These differences reflect Petrick's desire to remain as close as is practical to the formalism of transformational linguistics.

The primary distinction of the Woods system is that the deep structure tree is built during the surface analysis. Consequently, his "transformational" procedures consist of tree building rather than tree transforming operations. The tradeoffs between this approach and the two-stage analyzers of Petrick

and Sager are difficult to weigh at this time. They are part of the more general problem of parallel vs. serial processing; e.g. should semantic analysis be done concurrently with syntactic analysis. Parallel processing is preferred if the added time required by the deeper analysis is outweighed by the fraction of incorrect analyses which can be eliminated early in the parsing process. In the case of semantic analysis, it clearly depends on the relative complexity of the syntactic and semantic components. In the case of transformational analysis, it depends on the fraction of grammatical and selectional constraints which can be expressed at the surface level (if most of these can only be realized through transformational analysis, concurrent transformational analysis is probably more efficient). This may depend in turn on the type of surface analysis; for example, the relationships exhibited by linguistic string analysis are suitable for expressing many of these constraints, so there is less motivation in the Linguistic String Project system for concurrent transformational decomposition.

2.4. Other Syntactic Analysis Procedures

The system developed by Winograd at M.I.T. [Winograd 1971] for accepting English commands and questions about a "block world" also uses a context-free grammar augmented by restrictions. Winograd's context-free grammar was encoded as a set of procedures instead of a data structure to be interpreted, but this is not a material difference. His grammar is based on Halliday's "systemic grammar" to the extent that it extracts from a sentence the set of features described by Halliday; however, Halliday's grammar (at least in its present stage of development) is essentially descriptive rather than generative, so most of the detailed grammatical structure had to be supplied by Winograd. His parser does not construct a deep structure; rather, it builds semantic structures directly

during parsing. The primary distinctive feature of his system is the integration of the syntactic component with semantics and pragmatics (the manipulation of objects in the block world); his parser is thus able to use not only syntactic constraints but also semantic and pragmatic information in selecting a proper sentence analysis. With regard to the serial vs. parallel distinction drawn in the previous section, his system would be characterized as highly parallel.

A number of natural language systems have used grammars composed of unrestricted phrase-structure rewriting rules. Since unrestricted rewriting rules, like transformational grammars, can be used to define any recursively enumerable language, they may be sufficient for analyzing both surface and deep structure. As with transformational grammars, it will in practice be necessary to impose some constraint (such as ordering) on the rules, so that the language defined is recursive; otherwise a parser will never be able to determine whether some sentences are grammatical or not.

One parser for unrestricted rewriting rules was described by Kay [Kay 1967]. This parser included a number of mechanisms for restricting the application of rules, such as rule ordering, specifying part of the structure dominated by one element of the rule, or requiring the equality of the structures dominated by two elements. These mechanisms do not increase the generative power of the grammars, but are designed to make grammars easier to write. Kay described how his parser could be used to effect some reverse transformations.

Kay's parser was incorporated into a system called REL (Rapidly Extensible Language) developed by Thompson, Dostert, et al. at the California Institute of Technology [Thompson 1969, Dostert 1971]. Kay's original parser was augmented by allowing a set of binary features to be associated with each node, including feature tests as part of the rewrite rules, and permitting more general restrictions where the features were inadequate. The REL system was designed to support a number

of grammars, each interfaced to its own data base. One of these is REL English, which analyzes a subset of English into a set of subject-verb-object-time modifier deep structures; this grammar has 239 rules. In support of the use of general rewrite rules with features, they note that only 29 of the 239 rules required constraints which could not be conveniently stated in terms of feature tests. This is also a factor in efficiency, since binary feature tests can be performed very quickly.

Another system which uses unrestricted rewriting rules with optional conditions on the elements is the "Q" system developed by Colmerauer [Colmerauer 1970]. This system is presently being used in a machine translation project at the University of Montreal [Kittredge 1973].

Colmerauer and de Chastellier [de Chastellier 1969] have also investigated the possibility of using Wijngaarden grammars (as were developed for specifying ALGOL 68) for transformational decomposition and machine translation. Like unrestricted rewriting rules, W-grammars can define every recursively enumerable language, and so can perform the functions of the surface and reverse transformational components. They show how portions of transformational grammars of English and French may be rewritten as W-grammars, with the pseudo-rules in the W-grammar taking the place of the transformations.

2.5 Parsing with Probability and Graded Acceptability

In all the systems described above, a sharp line was drawn between correct and incorrect parses: a terminal node either did or did not match the next word in the sentence; an analysis of a phrase was either acceptable or unacceptable. There are circumstances under which we would want to relax these requirements. For one thing, in analyzing connected speech, the segmentation and identification of words can never be done with

complete certainty. At best, one can say that a certain sound has some probability of being one phoneme and some other probability of being another phoneme; some expected phonemes may be lost entirely in the sound received. Consequently, one will associate some number with each terminal node, indicating the probability or quality of match; nonterminal nodes will be assigned some value based on the values of the terminal nodes beneath. Another circumstance arises in natural language systems which are sophisticated enough to realize that syntactic and semantic restrictions are rarely all-or-nothing affairs, and that some restrictions are stronger than others. For example, the nominative-accusative distinction has become quite weak for relative pronouns (?The man who I met yesterday.) but remains strong for personal pronouns (*The man whom me met yesterday.). As a result, a parser which wants to get the best analysis even if every analysis violates some constraint must associate a measure of grammaticality or acceptability with the analyses of portions of the sentence, and ultimately with the analyses of the entire sentence.

In principle, one could generate every sentence analysis with a nonzero acceptability or probability of match, and then select the best analysis obtained. Hobbs [1974] has described a modification to the bottom-up nodal spans parsing algorithm which uses this approach. Wilks [1975] uses an essentially similar technique in his language analyzer based on "preference semantics"

A more efficient approach, called "best-first" parsing, has been developed by Paxton and Robinson of the Stanford Research Institute as part of a speech understanding system [Paxton 1973]. Their procedure involves a modification of the standard top-down serial parsing algorithm for context-free grammars. The standard algorithm generates one possible parse tree until it gets stuck (generates a terminal node which does not match the next sentence word); it then "backs up" to try another alternative.

The best-first procedure instead tries all alternatives in parallel. A measure is associated with each alternative path, indicating the likelihood that this analysis matches the sentence processed so far and that it can be extended to a complete sentence analysis. At each moment, the path with the highest likelihood is extended; if its measure falls below that of some other path, the parser shifts its attention to that other path.

2.6 Conjunction and Adjunction

There are certain pervasive natural language constructions which do not fit naturally into the standard syntax analysis procedures, such as augmented context-free parsers. Two of these are coordinate conjunctions and adjuncts. Special measures have been developed to handle these constructions; these measures deserve brief mention here.

The allowed patterns of occurrence of conjoinings in a sentence are quite regular. Loosely speaking, a sequence of elements in the sentence tree may be followed by a conjunction and by some or all of the elements immediately preceding the conjunction. For example, allowed patterns of conjoining include subject-verb-object-and-subject-verb-object (I drank milk and Mary ate cake.), subject-verb-object-and-verb-object (I drank milk and ate cake.), and subject-verb-object-and-object (I drank milk and seltzer.). There are certain exceptions, known as gapping phenomena, in which one of the elements following the conjunction may be omitted; for example, subject-verb-object-and-subject-object (I drank milk and Mary seltzer.).

The trouble with coordinate conjunctions is that they can occur almost anywhere in the structure of a sentence. Thus, although it would be possible to extend a context-free surface grammar to allow for all possible conjoinings, such an extension would increase the size of the grammar by perhaps an order of magnitude. The alternative scheme which has therefore been

developed involves the automatic generation of productions which allow for conjunction as required during the parsing process. When a conjunction is encountered in the sentence, the normal parsing procedure is interrupted and a special conjunction node is inserted in the parse tree. The alternative values of this node provide for the various conjoined element sequences allowed at this point.

An interrupt mechanism of this sort including provision for gapping, is part of the Linguistic String Project parser [Sager 1967]. A similar mechanism is included in Woods' augmented transition network parser [Woods 1973] and a number of other systems.

This solves the problem of correcting the context-free grammar for conjunctions, but the context-free grammar is generally only a small part of the total system. The task remains of modifying the routines which enforce grammatical constraints and the transformations to account for conjunctions. Since practically every routine which examines a parse tree is somehow affected by conjunction, this can be a large job, but fortunately the changes are very regular for most routines. The Linguistic String Project grammar, by performing all operations on the parse tree through a small number of low-level routines, was able to localize the changes to these routines and a small number of restrictions (such as number agreement) which are specially affected by conjunction [Raze 1974].

Certain classes of adjuncts or modifiers give rise to a different kind of problem: a high degree of syntactic ambiguity. For instance, in the sentence, "I fixed the pipe under the sink in the bathroom with a wrench." there is no syntactic basis for deciding whether the pipe had a wrench the sink had a wrench, the bathroom had a wrench, or the fixing was done with a wrench. If semantic and pragmatic restrictions are invoked during the syntactic analysis, the parser will have to generate

several analyses, all but one of which will (hopefully) be rejected by the restrictions; this is moderately inefficient. If syntactic analysis precedes semantic processing, the ambiguities of the various adjuncts will be multiplied producing dozens of analyses for a sentence of moderate size; this is hopelessly inefficient.

A more efficient solution has the parser identify the adjuncts and list for each adjunct the words it could be modifying, without generating a complete separate analysis for each possibility. The ambiguities associated with the adjuncts are thus factored out. The semantic and pragmatic components may then choose for each adjunct its most likely or acceptable host (modified word). This may be done either during the syntactic analysis [Woods 1973, Simmons 1975] or after the syntax phase is complete [Borgida 1975, Hobbs 1975].

3. ALGORITHM SPECIFICATIONS

We present below precise specifications for some of the parsing algorithms which have been discussed. These algorithms are presented in SETL, a programming language which is based on concepts from set theory and has been developed at New York University by a group led by Jack Schwartz. The large variety of data types, operators, and control structures in SETL makes it possible to specify the algorithms in a relatively compact and natural fashion. An implementation is available which includes most of the features of the specification language, so that algorithms can be tested in essentially the form in which they are published. A description of the subset of SETL which has been used in this report is given in the appendix.

3.1 Parsing Algorithms for Context-Free Grammars

Context-free grammars played a major role i the early stages of automatic natural language analysis. Although they have now generally been superceded by more complex and powerful grammars, many of these grammars are based on or have as one of their components a context-free grammar. The selection of an efficient context-free parser therefore remains an important consideration in natural language analysis.

Because so many different context-free parsers have been proposed, a comprehensive survey would be impracticable. We shall rather present a taxonomy according to which most context-free parsers can be classified, and illustrate this classification with five of the possible basic algorithms. At the end we shall mention which of these are being used in current natural language systems.

The first division we shall make is according to the amount of memory space required by the parser. Type 0 parsers store only the parse tree currently being built. The other parsers

gradually accumulate data from which all parses of a sentence can be extracted; types 1, 2 and 3 store this data in decreasingly compact representations. The four types are:

(0) Develops a single parse tree at a time; at any instant the store holds a set of nodes corresponding to the nodes of an incomplete potential parse tree

(1) The store holds a set of nodes, each of which represents the fact that some substring of the sentence, from word f to word ℓ, can be analyzed as some symbol N.

(2) The store holds a set of nodes, each of which represents an analysis of some substring of the sentence, from word f to word ℓ, as some symbol N (if there are several different analyses of words f to ℓ as some symbol N, there will be several nodes corresponding to a single node in a type 1 parser).

(3) The store holds a set of nodes, each of which corresponds to an analysis of some substring of the sentence, from word f to word ℓ, as some symbol N appearing as part of some incomplete potential parse tree (if symbol N, spanning words f to ℓ, appears in several of the incomplete potential parse trees, there will be several nodes corresponding to each node in a type 2 parser).

Type (0) parsers require only an amount of storage proportional to the length of the input sentence. The storage requirements of type (1) parsers grow as the cube of the length, while the requirements for types (2) and (3) grow exponentially.

A second division can be made between top-down and bottom-up parsers. A third criterion for classification is whether alternative parses of a sentence are all produced together (parallel parser) or are generated sequentially (serial parser); this division does not apply to type (0) parsers.

Finer divisions can be made of some of these categories. For example, among bottom-up parsers we can distinguish those which perform a reduction only when all required elements have been found from those which make a tentative reduction when the first element of a production is found (so-called "left-corner parsers"). Parallel parsers can be classified according to the

ordering strategy they use in building nodes: by leftmost or rightmost word subsumed (i.e., spanned) by the node or by level. In addition, we shall not consider a number of optimization strategies, such as selectivity matrices and shaper and generalized shaper tests for top-down parsers.

We shall now describe algorithms in five of the categories:

A	Type 0	Top-down serial
B	Type 2	Bottom-up parallel
C	Type 1	Bottom-up parallel
D	Type 2	Top-down serial
E	Type 1	Top-down serial

We have not included any type 3 parsers because, despite their profligate use of storage, they do not operate much faster than type 0 parsers. The only reported use of such a parser of which we are aware is the "Error-Correcting Parse Algorithm" of Irons (Comm. ACM 6, 669 (1963)). A top-down left-to-right parallel strategy was employed so that the parser could make a suitable modification to the sentence when it "got stuck" because of an error in the input.

SETL procedures are given for these five parsers. The input data structures are the same in all cases: The sentence, passed through parameter SENTENCE, is a tuple. The elements of the tuple, the words of the sentence, are to be matched by terminal symbols from the grammar. The context-free grammar, passed through parameter GRAMMAR, is a set each of whose elements corresponds to a production. The production

$$a_0 \rightarrow a_1 \; a_2 \; \ldots \; a_n$$

is transformed into the (n+1)-tuple

$$\langle a_0, a_1, a_2, \ldots, a_n \rangle \; .$$

The root symbol of the grammar is passed to the parser in parameter ROOT.

Algorithm A. Type 0 Top-down Serial

This procedure builds the parse trees for the input sentence sequentially in a two-dimensional array TREE. The first subscript of TREE specifies the number of the node, the second selects a component of the node as follows:

TREE(n,'NAME')	name of node n
TREE(n,'PARENT')	number of parent node of node n (= 0 for root node)
TREE(n,'DAUGHTERS')	tuple of numbers of daughter nodes of node n
TREE(n,'CURRENT OPTION')	tuple of current production used to expand this node
TREE(n,'ALTERNATIVE OPTIONS')	set of tuples representing productions not yet tried for this node
TREE(n,'FW')	number of first sentence word subsumed by node n
TREE(n,'LW+1')	(number of last sentence word subsumed by node n) + 1

As each analysis of the sentence is completed, it is added to the set PARSES. When parsing is finished, this set of trees is returned as the value of the function PARSE.

The variable NODES holds a count of the number of nodes in the parse tree; this is also the number of the node most recently added to the tree. WORD holds the number of the next word in the sentence to be matched.

The heart of the parser is the recursive procedure EXPAND. EXPAND is passed one argument, the number of a node in the parse tree. If EXPAND has not been called for this node before, it will try to expand the node, i.e., build a parse tree below the

node which matches part of the remainder of the sentence. If EXPAND has already been called once for this node -- so that a tree already exists below this node -- EXPAND tries to find an alternate tree below the node which will match up with part of the remainder of the sentence.

If EXPAND is successful -- an (alternate) tree below the node was found -- it returns the value true; if it is unsuccessful, it returns false. In the case where the node corresponds to a terminal symbol, EXPAND will return true on the first call only if the symbol matches the next word in the sentence; it will always return false on the second call.

```
definef PARSE(GRAMMAR,ROOT,SENTENCE);
local PARSES, TREE, NODES, WORD;
TREE = nℓ;
PARSES = nℓ;
WORD = 1;
NODES = 1;
   /* set up root node (node 1) */
TREE(1, 'NAME') = ROOT;
TREE(1, 'FW') = WORD;
   /* loop until all parse trees have been formed.*/
(while EXPAND(1))
   /* if tree spans entire sentence, add to set */
   if WORD eq ((#SENTENCE)+1) then
        PARSES = PARSES ∪ {TREE};
        end if WORD;
   end while EXPAND;
return PARSES;
end PARSE;
```

```
definef EXPAND(X);
local I, OPT;
if GRAMMAR{TREE(X,'NAME')} eq nℓ then
   /* terminal symbol */
   if TREE(X, 'ALTERNATE OPTIONS') eq Ω then
   /* first call -- test for match with sentence */
        TREE(X, 'ALTERNATE OPTIONS') = nℓ;
        if WORD le #SENTENCE then
             if SENTENCE(WORD) eq TREE(X, 'NAME') then
                  WORD = WORD + 1;
                  TREE(X, 'LW+1') = WORD;
                  return true;
                  end if SENTENCE;
             end if WORD;
        else /* second call */
        WORD = WORD-1;
        end if TREE;
   return false;
   end if GRAMMAR;

   /* nonterminal  symbol */
if TREE(X, 'ALTERNATE OPTIONS') eq Ω then
   /* first call, retrieve options from grammar */
   TREE(X, 'ALTERNATE OPTIONS') = GRAMMAR{TREE(X, 'NAME')};
   TREE(X, 'DAUGHTERS') = nult;
   OPT = Ω;
else /* second or subsequent call */
   OPT = TREE(X, 'CURRENT OPTION');
   I = #OPT;
   end if TREE;
```

```
   /* select next option to try */
GETOPT: if OPT eq Ω then
   OPT from TREE(X, 'ALTERNATE OPTIONS');
   TREE(X, 'CURRENT OPTION') = OPT;
   I = 1;
   end if OPT;
   /* expand node */
 while I ge 1)
   /* work on Ith element of current option */
   /* if corresponding node not in parse tree, add it */
   if TREE(X, 'DAUGHTERS')(I) eq Ω then
        NODES = NODES + 1;
        TREE(NODES,'NAME') = OPT(I);
        TREE(NODES,'FW') = WORD;
        TREE(NODES,'PARENT') = X;
        TREE(X, 'DAUGHTERS')(I) = NODES;
        end if TREE;
   /* try for an(other) expansion of this node */
   if EXPAND(TREE(X, 'DAUGHTERS')(I)) then
   /* expansion found... if this is last element, return
      successfully, else advance to next element */
        if I eq #OPT then
             TREE(X,'LW+1') = WORD;
             return true;
        else
             I = I + 1;
             end if I;
   else
       /* no expansion found ... erase this node and examine
          previous element */
        TREE(NODES) = Ω;
        NODES = NODES-1;
        TREE(X, 'DAUGHTERS')(I) = Ω;
        I = I-1;
        end if EXPAND;
   end while I;
```

```
   /* all expansions for this option have been generated;
      if more options, loop, else return false. */
OPT = Ω;
if TREE(X,'ALTERNATE OPTIONS') ne nℓ then go to GETOPT;;
return false;
end EXPAND;
```

One way of viewing this procedure is to consider each node as a separate process. Each process creates and invokes the processes corresponding to its daughter nodes. In SETL, the algorithm cannot be represented directly in this way, since there are no mechanisms for creating and suspending processes. Instead, the data which would correspond to the local variables of the process are stored as components of each node in the parse tree. In languages which provide for the suspension of processes, such as SIMULA, the algorithm can be represented even more succinctly (see, for example, a version of this algorithm in "Hierarchical Program Structures" by O.-J. Dahl and C. A. R. Hoare, in _Structured Programming_ by O.-J. Dahl et al., page 201).

Algorithm B. Type 2 Bottom-up Parallel

This algorithm is sometimes called the "Immediate Constituent Analysis" (ICA) algorithm, because it was used quite early in parsing natural language with ICA grammars. It constructs all nodes in a single left-to-right pass over the sentence. As each word is scanned, the parser builds all nodes which subsume a portion of the sentence ending at that word. The nodes ("spans") are accumualted in a two-dimensional array SPAN, whose first subscript specifies the number of the span and whose second subscript selects a component of the span, as follows:

```
SPAN (n, 'NAME') = name of span n
SPAN (n, 'FW') = number of first sentence word subsumed by span n
SPAN (n,'LW+1')= (number of last sentence word subsumed
                                by span n) + 1
SPAN (n, 'DAUGHTERS') = tuple of numbers of daughter spans
                                of span n.
```

At the end of the routine is some code to convert SPAN, a graph structure with each span potentially a part of many parses, into a set of parse trees. This code has two parts: a loop to find all root nodes created in the immediate constituent analysis, and a recursive routine EXPAND which makes copies of all descendants of the root node and puts them in TREE. Each node in the tree has the following components:

```
TREE(n, 'NAME')   =  name of node n
TREE(n, 'FW')     =  number of first sentence word subsumed
                                by node n
TREE(n, 'LW+1')   = (number of last sentence word subsumed
                                by node n) + 1
TREE(n, 'DAUGHTERS') = tuple of numbers of daughter nodes
                                of node n
TREE(n, 'PARENT')=  number of parent node of node n.
```

The set of parse trees is accumulated in PARSES and finally returned as the value of the function PARSE.

```
definef PARSE(GRAMMAR,ROOT,SENTENCE);
   local TODO, WORD, CURRENT, DEF, DEFNAME, DEFELIST, REM, SPAN, ,
        SPANS, TREE, NODES, PARSES, MS, I;
             /* initialization */
   SPAN = nl;
   SPANS = 0;
   TODO = nl;
             /* iterate over WORD=last word subsumed
                                 by spans being constructed */
   (1 <= ∀WORD <= #SENTENCE)
             /* add span whose name is sentence word */
        ADDSPAN(SENTENCE(WORD), WORD, WORD+1, nult);
             /* TODO contains the numbers of spans which were
                just created and for which we have not yet
                checked whether they can be used as the last
                daughter span in building some more spans */
        (while TODO ne nl)
             /* select a span from TODO */
        CURRENT from TODO;
             /* loop over all productions whose last element
                = name of current span */
        (∀DEF ∈ GRAMMAR|DEF(#DEF) eq SPAN(CURRENT,'NAME'))
             /* separate left and right sides of production */
                  DEFNAME = hd DEF;
                  DEFELIST = tl DEF;
             /* if elements preceding last element of production
                can be matched by spans, add a new span whose
                name = left-hand side of production for each match*/
             (∀REM ∈ MATCH(DEFELIST(1:(#DEFELIST)-1),
                                 SPAN(CURRENT,'FW')))
                ADDSPAN(DEFNAME, hd REM, SPAN(CURRENT,'LW+1'),'
                                 (tl REM) + <CURRENT>);
             end ∀REM;  end ∀DEF;
        end while TODO;
 end 1 <= ∀WORD;
```

```
            /* extract trees from set of spans */
PARSES = nl;
(1 <= ∀I <= SPANS | (SPAN(I,'NAME') eq ROOT) and
                  (SPAN(I,'FW') eq 1)       and
                  (SPAN(I,'LW+1') eq ((#SENTENCE)+1)))
   NODES = 1;
   TREE = nl;
   TREE{1} = SPAN{I};
   TREE(1, 'DAUGHTERS') = EXPAND(TREE(1,'DAUGHTERS'),1);
   end 1 <= ∀I;
return PARSES;
end PARSE;

define MATCH(ELIST, ENDWDP1);
   local I, NTUP;
            /* MATCH finds all n-tuples of spans whose names
               match the elements of the n-tuple ELIST and which
               span a portion of the sentence whose last word +1
               = ENDWDP1;  returns a set, each element of which
               is an (n+1)-tuple, whose tail is one of the n-tuples
               of spans and whose head is the number of the first
               word spanned by the n-tuple of spans */
   if ELIST eq nult
        then return {<ENDWDP1>};
        else return [∪: 1 <= I <= NODES,
             (if(SPAN(I,'NAME') eq ELIST(#ELIST))
                 and (SPAN(I,'LW+1') eq ENDWDP1)
               then {NTUP → <I>, NTUP ∈
                 MATCH(ELIST(1:(#ELIST)-1),SPAN(I,'FW'))}
               else nl);;
end MATCH;
```

```
define ADDSPAN(NAME, FW, LWP1, DAUGHTERS)
             /* ADDSPAN builds a span whose components
                are passed in the four parameters */
   SPANS = SPANS+1;
   SPAN(SPANS,'NAME') = NAME;
   SPAN(SPANS,'FW') = FW;
   SPAN(SPANS,'LW+1') = LWP1;
   SPAN(SPANS,'DAUGHTERS') = DAUGHTERS;
   SPANS in TODO;
   return;
end ADDSPAN;

definef EXPAND(DAW,PAR);
             /* creates a node for each span in DAW and each
                descendant thereof, and returns a tuple with
                the numbers of the nodes (in TREE) corresponding
                to the spans in DAW */
local S, D, N;
if DAW eq Ω then return Ω;;
D = nult;
(∀S ∈ DAW)
   NODES = NODES + 1;
   N = NODES;
   TREE{N} = SPAN{S};
   TREE(N,'PARENT') = PAR;
   TREE(N,'DAUGHTERS') = EXPAND(TREE(N,'DAUGHTERS'),N);
   D = D + <N>;
   end ∀S;
return D;
end EXPAND;
```

Algorithm C   Type 1   Bottom-up Parallel

Algorithm C is the basic "nodal spans" parsing algorithm [Cocke 1970]. The sequencing logic is identical to that for Algorithm B. The only difference in the tree representation is that all spans in Algorithm B with common values in the NAME, FW, and LW+1 components are joined into a single span in Algorithm C. The DAUGHTERS component now becomes a set, each of whose elements corresponds to the value of the DAUGHTERS component of one of the spans in Algorithm B (this set is called the "division list" in the nodal spans algorithm):

SPAN(n,'DAUGHTERS') = a set each of whose elements is a tuple of numbers of daughter nodes of span n

In order to effect this change in the tree, it is necessary only to modify the procedure ADDSPAN to check whether a span with the specified value of NAME, FW, and LW+1 already exists:

```
define ADDSPAN(NAME, FW, LWP1, DAUGHTERS);
    local S;
    if  1 <= ∃S <= SPANS | (SPAN(S,'NAME') eq NAME) and
                           (SPAN(S,'FW') eq FW) and
                           (SPAN(S,'LW+1') eq LWP1)
        then DAUGHTERS in SPAN(S,'DAUGHTERS');
        else SPANS = SPANS+1;
             SPAN(SPANS,'NAME') = NAME;
             SPAN(SPANS,'FW') = FW;
```

```
            SPAN(SPANS,'LW+1') = LWP1;
            SPAN(SPANS,'DAUGHTERS') = {DAUGHTERS};
            SPANS in TODO;
        end if;
        return;
end ADDNODE;
```

The procedure for converting the spans into a set of trees is now more complicated than for Algorithm B; see, for example, Owens [1975], Sec. 7.

Algorithm D. Type 2 Top-down Serial

We now seek to combine the advantages of algorithm A with those of algorithms B and C. Algorithms B and C would construct any given tree over a portion of the sentence only once, whereas algorithm A might construct some trees many times during the course of a parse. On the other hand, B and C would construct many trees which A would never try to build. More precisely, B and C would build trees while processing word n+1 which could not enter into any parse for any sentence whose first n words were those processed so far.

To combine these algorithms, we shall return to the basic framework provided by algorithm A. To this we add a mechanism for recording "well formed substrings." The first time the parser tries to analyze a portion of the sentence beginning at word f as an instance of symbol N, this mechanism records any and all trees constructed below node N. The next time the parser tries symbol N at word f, the saving mechanism retrieves this information so that the trees below N need not actually be rebuilt.

The previously-completed trees are stored in the two-dimensional array WFS, whose structure is identical to that of SPAN in algorithm B:

```
WFS(n,'NAME')      =  name of well-formed substring n
WFS(n,'FW')        =  first word of well-formed substring n
WFS(n,'LW+1')      =  (last word of well-formed substring n) + 1
WFS(n,'DAUGHTERS') = tuple of numbers of daughter substrings
                     of substring n
```

WFSS holds the number of substrings in WFS. When the parsing operation is complete, WFS will contain a subset of the elements which were in TREE at the end of algorithm B.

The tree used by the top-down parser must be augmented to allow for the possibility that the parser is not building a tree below a given node but rather consulting the table of well-formed substrings for that node. In that case the node will have, instead of a tuple of daughters and a set of alternative options, the number of the well-formed substring currently being used in the tree and the set of alternative well-formed substrings. The structure of a node is thus:

```
TREE(n,'NAME')   =  name of node n
TREE(n,'PARENT') = number of parent node of node n
                    (= 0 for root node)
TREE(n,'DAUGHTERS') = tuple of numbers of daughter nodes of node n
                    (= nult if node is matched by well-formed substring)
TREE(n,'WFS')       = number of well-formed substring matched to node n
                    (= Ω if not matched to a substring)
TREE(n,'CURRENT OPTION') = tuple of current production used to
                    expand node n
                    (= Ω if node is matched by a well-formed substring)
TREE(n,'ALTERNATE OPTIONS') =
                    set of tuples representing productions not
                    yet tried for node n
TREE(n,'ALTERNATE WFS') =
                    set of numbers of well-formed substrings not
                    yet tried for node n
TREE(n,'FW')        =  number of first word subsumed by node n
TREE(n,'LW+1')      = (number of last word subsumed by node n) + 1
```

Finally, we require a table which indicates, for each symbol N and sentence word f, whether all the well-formed substrings for N starting at f have been recorded in WFS. For this the parser uses the two-dimensional array EXPANDED: EXPANDED(N,f) = true if all substrings have been recorded, Ω if not.

The text of procedure D is given below; comments are included only for those statements added to procedure A.

```
definef PARSE(GRAMMAR, ROOT, SENTENCE);
local PARSES, TREE, NODES, WORD, WFS, WFSS, EXPANDED;
TREE = nℓ;
PARSES = nℓ;
WFS = nℓ;
WFSS = 0;
EXPANDED = nℓ;
WORD = 1;
NODES = 1;
TREE(1,'NAME') = ROOT;
TREE(1,'FW') = WORD;
(while EXPAND(1))
   if WORD eq (#SENTENCE + 1) THEN
        PARSES = PARSES ∪ {TREE};
        end if WORD;
   end while;
return <PARSES,WFS>;
end PARSE;

definef EXPAND(X);
local I, S, LAST, OPT;
if EXPANDED(TREE(X,'NAME'), TREE(X,'FW')) eq true then
   /* the expansions for this symbol have been computed before */
   /* if this is a new node, get its WFS entries */
   if TREE(X,'ALTERNATE WFS') eq Ω then
        TREE(X,'ALTERNATE WFS') = {S, 1 ≤ S ≤ WFSS |
             (WFS(S,'NAME') eq TREE(X,'NAME')) and
             (WFS(S,'FW') eq TREE(X,'FW'))}
        end if TREE;
   if TREE(X,'ALTERNATE WFS') eq nℓ then
        /* all WFSs tried for this node */
        WORD = TREE(X,'FW');
        return false;
   else
```

```
        * select next WFS for node */
       TREE(X,'WFS') from TREE(X,'ALTERNATE WFS');
       WORD = WFS(TREE(X,'WFS'), 'LW+1');
       TREE(X,'LW+1') = WORD;
       return true;
       end if TREE;
   end if EXPANDED;

if GRAMMAR{TREE(X,'NAME')} eq nℓ then
   if TREE(X,'ALTERNATE OPTIONS') eq Ω then
       TREE(X,'ALTERNATE OPTIONS') = nℓ;
       if WORD le #SENTENCE then
              if SENTENCE(WORD) eq TREE(X,'NAME') then
                 WORD = WORD+1;
                 TREE(X,'LW+1') = WORD;
        * add WFS recording match to terminal symbol */
                 ADDWFS(TREE{X});
                 TREE(X,'WFS') = WFSS;
                 return true;
                 end if SENTENCE;
              end if WORD;
   else WORD = WORD - 1;
       end if TREE;
   /* match to terminal symbol, if successful, has been recorded*/
   EXPANDED(TREE(X,'NAME'),WORD) = true;
   return false;
   end if GRAMMAR;

if TREE(X,'ALTERNATE OPTIONS') eq Ω then
   TREE(X,'ALTERNATE OPTIONS') = GRAMMAR{TREE(X,'NAME')};
   TREE(X,'DAUGHTERS') = nult;
   OPT = Ω;
else OPT = TREE(X,'CURRENT OPTION');
   I = #OPT;
   end if TREE;
```

```
GETOPT: if OPT eq Ω then
   OPT from TREE(X,'ALTERNATE OPTIONS');
   TREE(X,'CURRENT OPTION')= OPT;
   I = 1;
   end if OPT;
(while I ge 1)
   if TREE(X,'DAUGHTERS')(I) eq Ω then
        NODES = NODES + 1;
        TREE(NODES,'NAME') = OPT(I);
        TREE(NODES,'FW') = WORD;
        TREE(NODES,'PARENT') = X;
        TREE(X,'DAUGHTERS')(I) = NODES;
        end if TREE;
   if EXPAND(TREE(X,'DAUGHTERS')(I)) then
        if I eq #OPT then
           TREE(X,'LW+1') = WORD;
           /* record substring matched by node X */
           ADDWFS(TREE{X});
           TREE(X,'WFS') = WFSS;
           return true;
          se
           I = I+1
           end if I;
   else TREE(NODES) = Ω;
        NODES = NODES-1;
        TREE(X,'DAUGHTERS')(I) = Ω;
        I = I-1;
        end if EXPAND;
   end while I;
OPT = Ω;
if TREE(X,'ALTERNATE OPTIONS') ne nℓ then go to GETOPT;;
   /* all expansions tried */
EXPANDED(TREE(X,'NAME'),WORD) = true;
return false;
end EXPAND;
```

```
define ADDWFS(NODEX)
   /* add an entry to WFS */
local I;
WFSS = WFSS+1;
WFS(WFSS,'NAME') = NODEX('NAME');
WFS(WFSS,'FW') = NODEX('FW');
WFS(WFSS,'LW+1') = NODEX('LW+1');
if NODEX('DAUGHTERS') ne Ω then
   WFS(WFSS,'DAUGHTERS') = [+: I ∈ NODEX('DAUGHTERS')]
                              <TREE(I,'WFS')>;
   end if NODEX;
return;
end ADDWFS;
```

Note that this parser returns an ordered pair consisting of the set of trees and the set of well-formed substrings, since the trees alone do not contain complete information about the sentence analysis.

Algorithm E   Type 1   Top-Down Serial

To complete our set of algorithms, we shall apply to Algorithm D the same change we made to convert Algorithm B to Algorithm C. That is, where in Algorithm D we may have had several well formed substrings with the same values of NAME, FW, LW+1, we shall combine these into a single substring in Algorithm E. The component DAUGHTERS becomes a set, each of whose elements is a tuple corresponding to the value of DAUGHTERS of one of the substrings in Algorithm D. Just as we only had to change ADDNODE in Algorithm B, we only have to change ADDWFS in Algorithm D.

```
define ADDWFS(NODEX);
local W, I, DAUGHTERS;
   /* compute DAUGHTERS for substring */
```

```
DAUGHTERS = Ω;
if NODEX('DAUGHTERS') ne Ω then
   DAUGHTERS = [+: I ∈ NODEX('DAUGHTERS')]<TREE(I,'WFS')>;
   end if NODEX;
   /* search for well formed substring with identical NAME,
                                                    FW, LW+1 */
if  1 <= ∃W <= WFSS|((WFS(W,'NAME') eq NODEX('NAME'))and
                     (WFS(W,'FW') eq NODEX('FW'))and
                     (WFS(W,'LW+1') eq NODEX('LW+1')))
   /* found one, add daughter to set */
then if DAUGHTERS ne Ω then
   DAUGHTERS in WFS(W,'DAUGHTERS');
   end if DAUGHTERS;
else WFSS = WFSS+1;
   WFS(WFSS,'NAME') = NODEX('NAME');
   WFS(WFSS,'FW') = NODEX('FW');
   WFS(WFSS,'LW+1') = NODEX('LW+1');
   WFS(WFSS,'DAUGHTERS') =
        if DAUGHTERS eq Ω then nℓ else  DAUGHTERS ;
   end if ∃W;
return;
end ADDWFS;
```

Use of the Various Algorithms in Natural Language Systems

The type 0 top-down algorithm (algorithm A) is one of the simplest and most frequently used. For example, a special version of this algorithm (for Greibach normal form grammars) was used in the original Harvard Predictive Analyzer [Kuno 1962]. The later version of the Harvard system, incorporating a "path elimination" technique, was a type 1 top-down serial parser, a variant of Algorithm E; instead of saving all daughters in WFS during the parse, they were recomputed later for those nodes appearing in a parse tree [Kuno 1965].

Several current systems use augmented context-free grammars: grammars to which have been added restrictions on the parse tree typically in the form of LISP predicates, which must be true if the tree is to be accepted as a sentence analysis. The Winograd [1971] system uses an augmented context-free grammar with the context-free component encoded as a program rather than data. The parsing strategy is essentially that of a type 0 top-down algorithm, except that back-up is explicitly controlled rather than automatic. Woods' system [1970b] also uses a type 0 top-down algorithm, although somewhat different from the one presented here since his grammar is a recursive transition network. The Linguistic String Project system [Sager 1967] started out with a parser based on a type 0 top-down algorithm; for efficiency it later progressed to a type 2 top-down algorithm. A type 2 rather than a type 1 algorithm was used because the restrictions can reject one analysis of a portion of the sentence as a particular symbol while accepting another analysis of the same portion of the sentence as the same symbol. For a type 2 algorithm, this means simply eliminating some nodes in WFS; for a type 1 algorithm, where a single node may represent several trees, a complicated procedure which could create new nodes would have been required in general. The Linguistic String Project parser is considerably more complex than the type 1 top-down serial parser shown above (algorithm D), in part because of the restrictions which must be evaluated during the parse, in part because (for reasons of storage economy) the system makes it possible to save only selected nodes in WFS.

The type 2 bottom-up parallel algorithm also saw early use in natural language processing. The parser designed by Cocke for the Rand system was a special version of this algorithm for Chomsky normal form grammars. A thorough survey of the different ordering strategies possible with this algorithm was given by Hays [1967]. This algorithm was subsequently developed by Cocke (among others) into a type 1 bottom-up parallel algorithm named "nodal spans" and subsequently into a type 1 top-down parallel algorithm called "improved nodal spans" (see Cocke [1970] for a description of these algorithms). The latter is very similar to a parsing algorithm described by Earley [1970]. These type 1 algorithms have, to the best of our knowledge, not yet been used in natural language parsing.

In closing a few remarks are in order on the practical importance of the differences between the various algorithms. How significant is the difference between type 2 and type 1, between top-down and bottom-up, between serial and parallel? There has been no systematic study of these questions, and the answers to them are in all likelihood quite grammar specific.

For example, the advantage of the top-down parser is that, in working on word n+1, it eliminates from consideration those symbols which could not occur in any parse tree whose first n terminal symbols are the first n words of the sentence. Is this a large effect? Although I am not aware of any measurement of this quantity, the factor seems to be relatively small for large-coverage English grammars -- perhaps reducing the number of symbols in half.

The advantage of type 1 over type 2 algorithms depends on the degree of ambiguity of the grammar. How frequently can a portion of a sentence be analyzed as a particular symbol in several ways? For unaugmented context-free grammars the answer in general has been very frequently -- this was one of the problems of the context-free systems. For such grammars, type 1 algorithms would be much more efficient. When restrictions are added, however, they discriminate some of the analyses from others.

A rich set of syntactic, semantic, and pragmatic restrictions (available so far only for small subsets of English in limited areas of discourse) would presumably eliminate almost all ambiguity, so that the advantage of a type 1 algorithm would then be small.

Finally, we should mention the difference between serial and parallel parsers. Since serial and parallel algorithms will have created the same number of nodes by the time parsing is complete, the difference in time is probably quite small. The parallel algorithm may have the edge because "bookkeeping" is simpler. Also, the parallel algorithm can handle left recursion naturally, whereas a special mechanism is required for top-down serial parsers. On the other hand, a serial algorithm may be preferable if only the first parse of a sentence is required. In addition, the serial algorithms can more simply handle the situation where memory space is marginal. Normally most of the space in algorithms D and E is used by the set WFS, not by TREE. Consequently a type 1 or 2 serial parser can "rescue itself" when memory is almost exhausted by reverting to a type 0 algorithm; this simply means that it stops saving nodes in WFS. In terms of the SETL programs given above this requires, in addition to a change to ADDWFS, only that elements of EXPANDED no longer be set to true once saving has been terminated.

3.2. A Parser for Unrestricted Rewriting Rule Grammars

A number of natural language systems, such as REL (at the California Institute of Technology) and the Q-system (at the University of Montreal) have used unrestricted phrase structure grammars. In such grammars, each rule specifies that some sequence of symbols be rewritten as some other sequence of symbols. The parsing algorithm used in these systems was described by Kay in 1967 ("Experiments with a Powerful Parser," Martin Kay, in 2ème Conference Internationale sur le Traitment Automatique des Langues, Grenoble).

Kay added quite a few features to the basic parsing procedure to create his "powerful parser". These included rule ordering and conditions on rule application. Other unrestricted rewriting rule systems have also included some such features to permit the parsimonious description of complex natural language grammars. In this newsletter, however, we shall not be concerned with these additional features; only the basic parsing procedure will be described below.

The parser to be presented represents only a small modification to the context-free parser B (the "immediate constituent analyzer") given earlier. To understand this modification, consider the following example. We are given a context-free grammar which includes the productions

$$a \rightarrow def$$

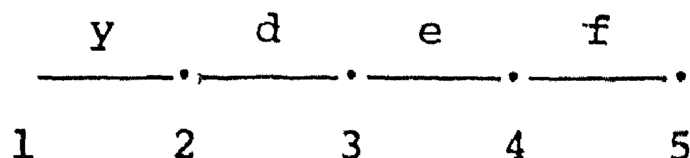
and

$$x \rightarrow y\ a$$

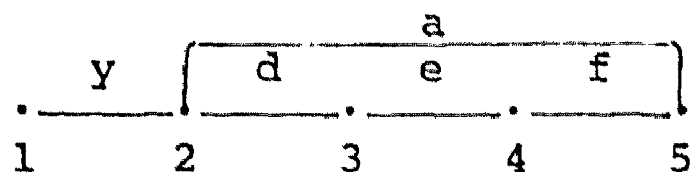
and the sentence

$$y\ d\ e\ f$$

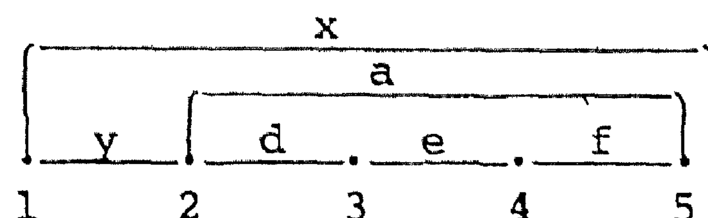
We shall create a diagram for the sentence by making each word into an arc connecting two nodes, which are labeled with the number of the word in the sentence and the number +1:



Context-free parser B would first apply the product a → def in reverse, to obtain a span "a"; we can indicate this thus:



Note that the arc for the span connects nodes corresponding to the first word and the last word + 1 subsumed by the span. The parser would then apply x → y a in reverse:

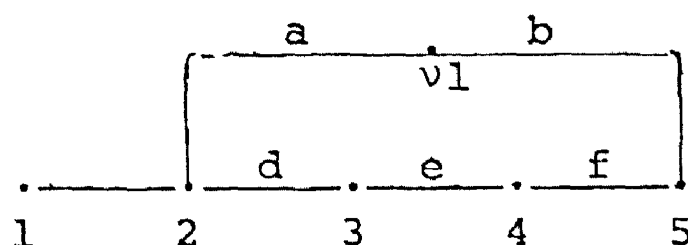


getting a span which subsumes the entire sentence.

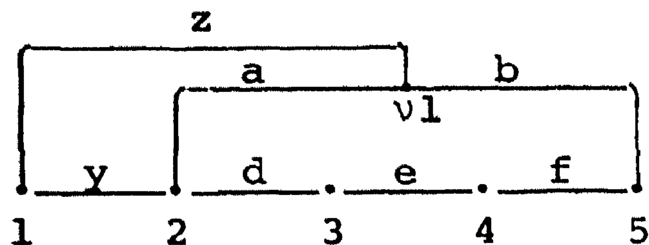
Now consider analyzing the same sentence with the unrestricted phrase structure grammar

$$a\ b \rightarrow d\ e\ f$$
$$z \rightarrow y\ a$$
$$x \rightarrow z\ b$$

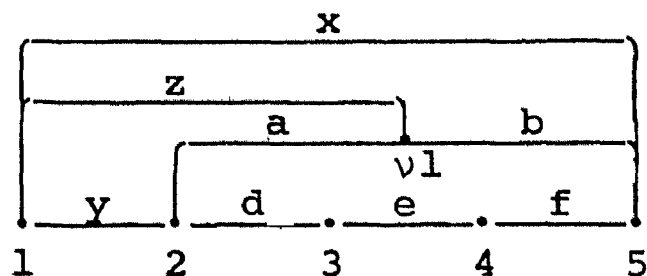
We begin by using the first production to reduce the sentence to y a b. This raises the problem of how to label the node between arcs a and b. Although a and b together subsume the last three words of the sentence, no fraction of this can be assigned individually to a or to b; hence we cannot label this new node with the number of a sentence word. Instead, we assign a new, unique label (here, ν1) to the node:



We can then reverse the second production to get a span z **from 1 to** ν1:



Finally we reverse the third production to get a span x **subsuming** the entire sentence:



In the program below, new node names are created by **calls on the** SETL function <u>newat</u>, which returns a different **unique symbol** (a "blank atom" in SETL terminology) each time **it is called.** We have retained the span component names **FW and LW+1** for the labels of the nodes at the ends of the arc, **though** their values may now be blank atoms instead of numbers.

A production of the form

$$a_1 \ldots a_n \rightarrow b_1 \ldots b_n$$

is represented by the structure

$$<<a_1,\ldots,a_n>, b_1,\ldots,b_n>$$

GRAMMAR is a set of such structures. For example, the **unrestricted** rewriting rule grammar given above would be **encoded**, as

$$\{ < < a, b>, d, e, f>,$$
$$< < z>, y, a>,$$
$$< < x>, z, b > \}$$

The fact that spans numbered $s_1, \ldots s_n$ were formed by **applying an** inverse production to spans numbered $d_1 \ldots d_n$ **is recorded by** assigning the component CONSTITUENTS with value

$$<<s_1,\ldots,s_n> <d_1,\ldots,d_n>>$$

to span s_n.

```
define URRPARSE(GRAMMAR, SENTENCE)
local SPAN, SPANS, WORD, TODO, CURRENT, DEF, DEFNAME,
     DEFELIST, MS;
                            /* initialization */
     SPAN = nl;
     SPANS = 0;
     TODO = nult;
                            /* iterate over WORD=last word
                               subsumed by spans being
                               constructed */
     (1 <= ∀WORD <= #SENTENCE)
          /* add span whose name is sentence word */
        ADDSPAN(SENTENCE(WORD), WORD, WORD+1, nult);
          /* TODO contains the numbers of spans which were just
             created and for which we have not yet checked
             whether they can be used as the last daughter span
             in building some more spans */
        (while TODO ne nult)
          /* select a span from TODO */
        CURRENT = hd TODO;  TODO = tl TODO;
          /* loop over all productions whose last element
             = name of current span */
        (∀DEF ∈ GRAMMAR|DEF(#DEF) eq SPAN(CURRENT,'NAME'))
          /* separate left and right sides of production */
               DEFNAME = hd DEF;
               DEFELIST = tl DEF;
          /* if elements preceding last element of production
             can be matched by spans, add new spans whose
             names = left-hand side of production for each match*/
           (∀REM ∈ MATCH(DEFELIST(1:(#DEFELIST)-1),
                            SPAN(CURRENT,'FW')))
              ADDSPANS(DEFNAME, hd REM, SPAN(CURRENT, 'LW+1'),
                            (tl REM) → <CURRENT>);
           end ∀REM;  end ∀DEF;
        end while TODO;
     end 1 <= ∀WORD;
     return SPAN;
end URRPARSE;
```

```
define MATCH(ELIST, ENDWDP1);
     local I, NTUP;
          /* MATCHfinds all n-tuples of spans whose names
             match the elements of the n-tuple ELIST and
             which span  a portion of the sentence whose
             last word + 1 = ENDWDP1; returns a set, each
             element of which is an (n+1)-tuple, whose tail
             is one of the n-tuples of spans and whose head
             is the number of the first word spanned by the
             n-tuple of spans */
     if ELIST eq nult
        then return {<ENDWDP1>};
        else return [∪: 1 <= I <= NODES]
          (if(SPAN(I,'NAME') eq ELIST(#ELIST))
             and (SPAN(I, 'LW+1') eq ENDWDP1)
            then {NTUP + <I>, NTUP ∈
             MATCH(ELIST(1:(#ELIST)-1),SPAN(I,'FW'))}
            else nl);;
end MATCH;

define ADDSPANS(LHS, FW, LWP1, CONSTITUENTS);
 /* builds a sequence of spans whose names are given by tuple
    LHS, with the first span beginning at FW and the last one
    ending at LWP1 */
local W, I, TUP;
W = FW;
TUP = nult;
(1 <= ∀I <= #LHS)
     SPANS = SPANS + 1;
     TUP = TUP + <SPANS>;
     SPAN(SPANS,'NAME') = LHS(I);
     SPAN(SPANS, 'FW') = W;
     W = if I eq #LHS then LWP1 else newat;
     SPAN(SPANS, 'LW+1') = W;
     end 1 <= ∀I;
SPAN(SPANS,'CONSTITUENTS') = <TUP, CONSTITUENTS>;
TODO = TUP + TODO;
```

```
return;
end ADDSPANS;
```

The unrestricted rewriting rule parser has the power of a Turing machine. The user is afforded great flexibility in the manipulation of sentence strings. One drawback of such power, however, is the absence of a decision procedure -- no parser can determine, for an arbitrary grammar of this type, that a given sentence string is ungrammatical. The user must therefore be careful to design grammars so that the parser will terminate (in a reasonable amount of time) for any input sentence.

* * * * *

3.3. Parsing Procedures for Transformational Grammars

Most linguistic research over the past fifteen years has been conducted within the framework of transformational grammar developed by Chomsky and Harris. In the early 1960's, a few years after the blossoming of transformational grammar, several efforts were begun to develop parsers which could operate fairly directly from a transformational grammar. Two of these achieved some measure of success: a project at MITRE led by Donald Walker [Zwicky 1965, Walker 1966] and work at MIT by Stanley Petrick [1965].

These two efforts had quite different objectives. The MITRE group was concerned with a specific practical application: development of a natural language interface for a military information retrieval system. They developed a grammar for a subset of English meeting their requirements and were primarily concerned with designing a parser which could handle this particular grammar. Petrick, in contrast, developed a general parsing procedure which would work with any member of a class of transformational grammars. This difference in objective affected a number of design decisions regarding the parsing procedure, as we shall see later on. Petrick and his coworkers, at the Air Force Cambridge Research Laboratory and now at IBM, Yorktown Heights, have modified the parser to reflect changes in transformational grammar and to adapt it for use as the front-end in an information retrieval system. [Petrick 1966,1973,1975; Keyser 1967; Plath 1974a,1974b]. Interestingly enough, these modifications have brought Petrick's parser much closer to the original MITRE design.

Since the structure of transformational grammar has varied in time and between different schools of linguistic theory, the notion of a transformational parser is not well defined. In order to present a parsing algorithm, we have selected a particularly simple grammar formulation. This formulation corresponds approximately to the early work of Chomsky (e.g., *Syntactic Structures*) and the theory used in the early versions of the MITRE and Petrick systems. Complicating factors, such as features and content-sensitive rules for lexical insertion, have been omitted.

The grammar consists of a *base component* and a *transformational component*. The base component is a context-free grammar which produces a set of *deep structure* trees. The transformational component is a set of tree-rewriting rules which, when applied to a deep structure tree, produces one or more *surface structure* trees. The frontiers (terminal node sequences) of the surface structure trees are the sentences of the language.

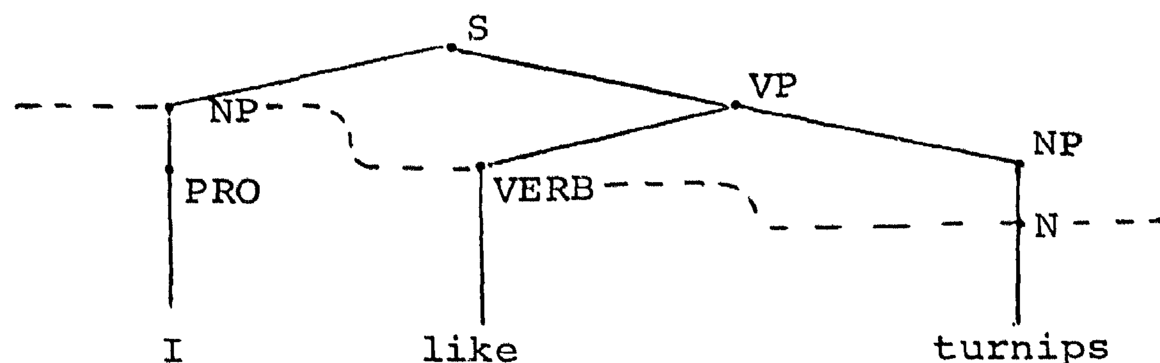
The root symbol of the base component is named S. The base component also contains a distinguished symbol COMP which appears on the left side of only one production:

$$\text{COMP} \rightarrow \# \; S \; \#$$

is referred to as the sentence boundary marker. With the exclusion of this production, the grammar is not recursive.

Each transformation consists primarily of a *structural index* and a *structural change*. The structural index is a tuple (vector), $\langle si_1, \ldots, si_n \rangle$, each of whose components is either a symbol (name of a node) or "X". The structural change is a tuple $\langle sc_1, \ldots, sc_n \rangle$ of the same length as the structural index. Each of its components is in turn a tuple $sc_i = \langle sc_{i1}, \ldots, sc_{in_i} \rangle$, possibly empty ($n_i = 0$). Each of the sc_{ij} is either a terminal symbol or an integer between 1 and n.

The application of transformational rules is based on the notion of a *proper analysis*, which is in turn based on the concept of a *cut* of a tree. Roughly speaking, a cut is defined by drawing a line from left to right through a tree, passing only through nodes (not through the lines connecting nodes); the nodes thus passed through form the cut. For example, for the tree



the sequence of nodes NP, VERB, N form a cut. More formally (Aho and Ullman, *The Theory of Parsing, Translation, and Compiling*, Vol. I, pg.140), a cut is a subset C of the nodes D of the tree such that

1. no node in C is on a successor path from some other node in C
2. no other node of D can be added to C without violating rule 1

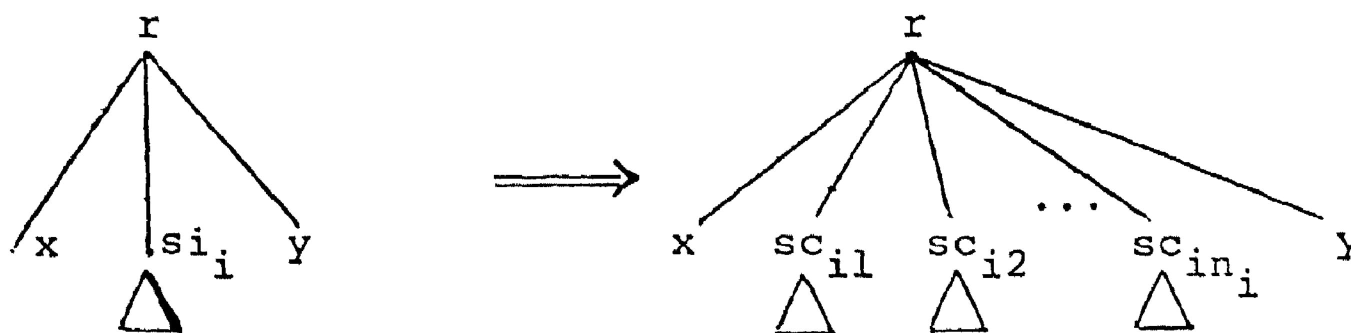
If the names of the nodes in the cut, arranged in sequence from left to right, match the structural index of the transformation, the cut is a proper analysis of the tree with respect to this transformation. A structural index matches the sequence of node names if there exists a substitution of sequences of symbols (possibly null and not including #) for the occurrences of "X" in the structural index which will make the structural index identical to the sequence of names. For example, the cut

NP VERB N

would be matched by any of the structural indices

NP VERB N
X NP VERB N X
NP X
X

The proper analysis associates with each element of the structural index (except possibly "X"s) a node in the tree and hence a subtree, the tree dominated by that node. The structural change indicates how these subtrees are to be shuffled to effect the transformation. sc_i specifies what is to go into the position occupied by the node matching sc_i. If sc_i is a 1-tuple, we simply have a case of one node (and the subtree it dominates) replacing another; if sc_i is an n tuple, n > 1, we first substitute sc_{i1} for the original node and then insert $sc_{i2},\ldots,sc_{in_i}$ as right siblings of that node:



If sc_{ij} is an integer, between 1 and n, the new node is the node matched to the sc_{ij}-th element of the structural index; if SC_{ij} is a terminal symbol, the new node is a terminal node with that name.

Because the value of sc_i may be the null tuple < >, it is possible for a node in the tree to be left with no successors. We therefore "clean up" the tree after applying the transformation by deleting any nonterminal node not dominating at least one terminal node.

The prescription just given is inadequate for components of the structural index equal to "X", since these may match zero or more than one node. We shall constrain the transformations so that nodes in the cut which are matched by "X"'s do not take part in the transformation. In terms of the structural

change, if si_k = "X" then sc_k = <k> and no other sc_{ij} = k.

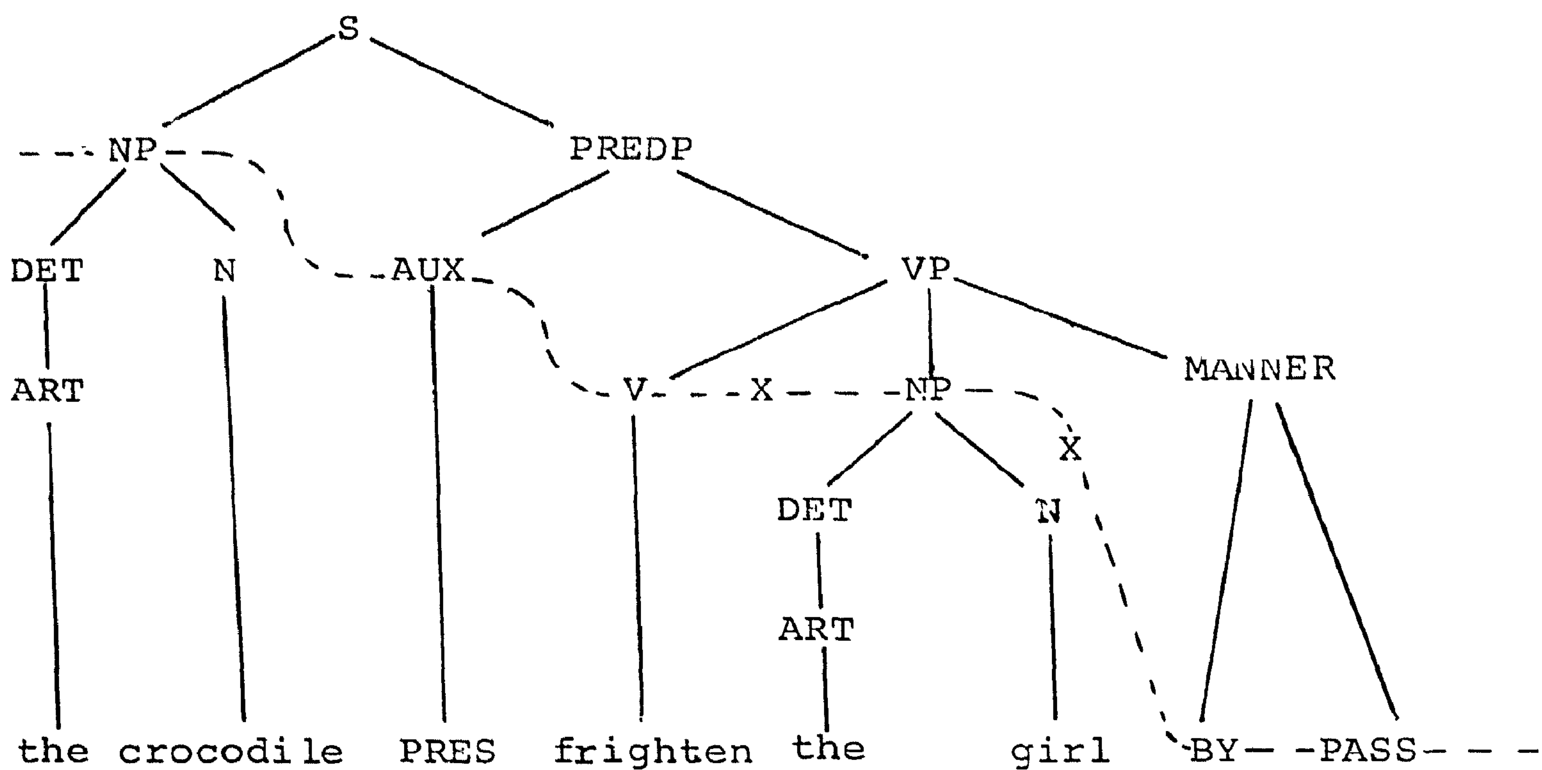
As an example (a simplfication of the example in Keyser and Petrick, *Syntactic Analysis*, p. 9), consider the *passive* transformation. Its structural index is

<NP, AUX, V, X, NP, X, BY, PASS>

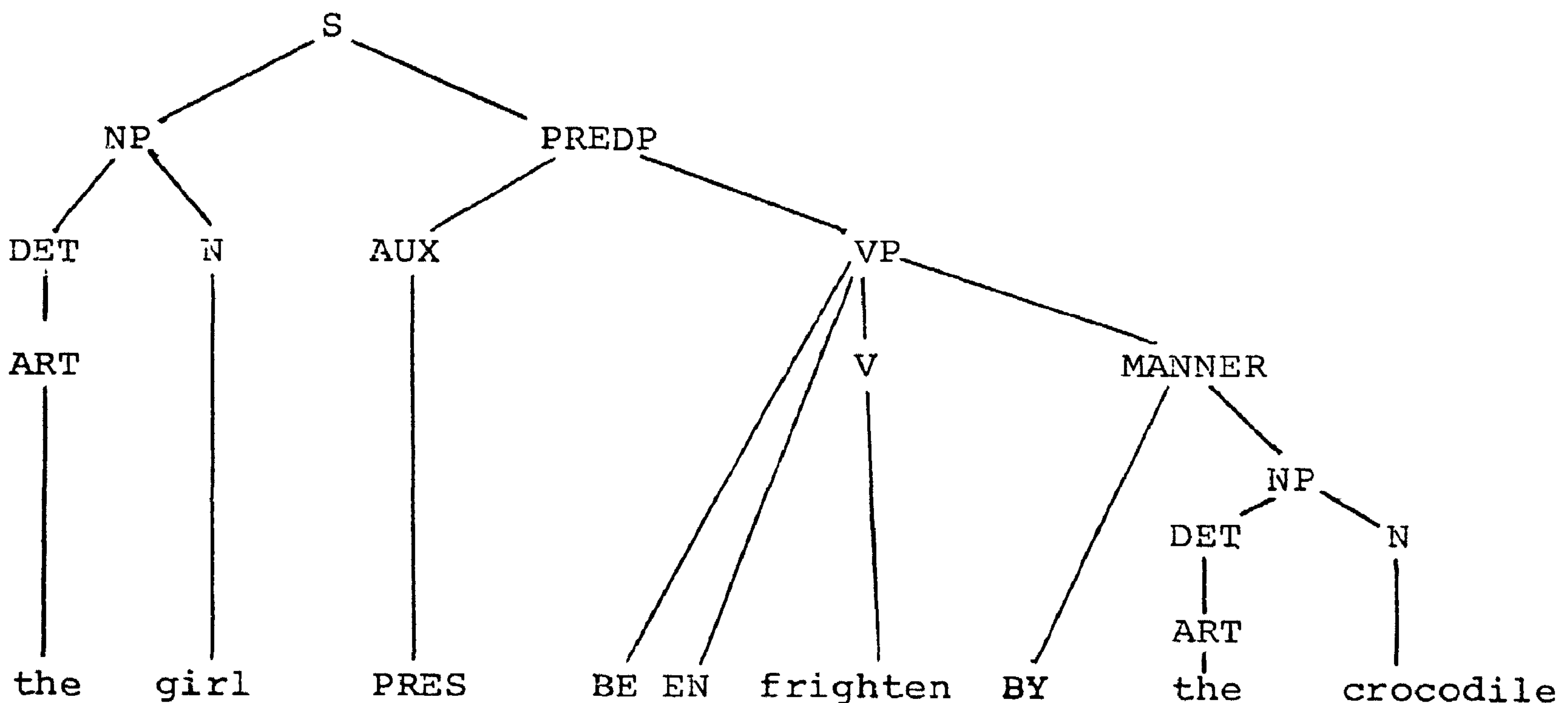
and its structural change

<<5>, <2>, <BE EN 3>, <4>, < >, <6>, <7>, <1>>

Applied to the tree

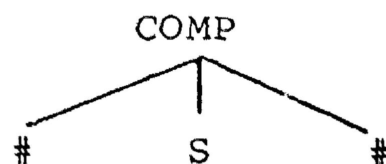


it produces the proper analysis indicated by the dotted line Applying the transformation yields the tree



In addition to the structural index and structural change, some transformations may have an identity condition, requiring that the subtrees matched by two elements of the structural index be identical for the transformation to apply.

The rule COMP → # S #, which makes the base component recursive, also plays a special role in the transformations. If the structure



appears in the parse tree, we call the tree dominated by that S a *constituent* (or embedded) sentence, and the tree dominated by the next S above COMP the *matrix* sentence parse tree.

The transformations are of two types, *singulary* and *binary* (or embedding). In a singulary transformation, the structural index does not contain the symbol #. In a binary transformation, the structural index is of the form $\alpha \# \beta \# \gamma$, where α, β, and γ are strings of symbols not containing #. The binary transformation deletes these boundary markers (if # are the ith and jth components of the structural index, then none of the $sc_{k\ell}$ = i or j), thus combining a constituent sentence with its matrix sentence.

The transformations are also cl ssed as *optional* or *obligatory*. Just like the generation of sentences with a context-free grammar, the application of transformations to a base structure may be viewed as a nondeterministic process. Depending on the choices made, one of several possible surface structures may be obtained from a single deep structu.e.

The transformations are considered in a fixed order to be described momentarily. If there exists no proper analysis for a transformation, the transformation is skipped. If there exist several proper analyses, one is chosen. If the transformation is obligatory, it is then applied; if it is optional, a choice is made to apply it or not.

The singulary and binary transformations are separately ordered. The transformational process begins by selecting an embedded sentence tree not including any other embedded sentence. The singulary transformations are applied in sequence to this tree; structural indices are matched against the embedded tree, not the entire parse tree. The binary transformations are then applied to this tree and its matrix sentence tree; one of these should actually transform the tree, deleting the embedded sentences (if none applied, we would eventually be left with a surface structure containing #'s, which would be rejected). Another deepest embedded sentence is selected and the process repeats until no embedded sentences remain. The singulary transformations are then applied to the entire tree, completing the generating process (if the base structure contained no embedded sentences, this would be the only step).

In order to parse a sentence -- obtain its deep structures -- we would like to reverse the process just described. First build one or more potential surface structure parse trees for a sentence and then, by applying the transformations in reverse, try to obtain a valid deep structure from each of these. We shall deal with these two steps in turn.

The surface structure parse tree will, in general, contain many structures which could not be directly generated by the base component. If we want to produce all the surface structure trees for a sentence using a context-free grammar, it will be necessary to augment the base component. For example, if the base component contains the production

$$A \rightarrow X\ Y$$

and there is a transformation which interchanges X and Y, the rule

$$A \rightarrow Y\ X$$

must be included in the grammar which is used to produce the surface structure trees. Petrick has described (in his Ph.D. thesis) a procedure which can determine from the base and transformational components, how the base must be augmented in order to obtain all surface structure trees.

Because a transformation can replace one node with two, it is possible for repeated application of such a transformation to produce a node in the surface structure with an arbi-arbitrary number of immediate descendants. This means that an infinite number of rules must be added to the base component. Petrick noted, however, that if a limit is placed on the length of sentences to be analyzed (and certain minimal assumptions are made about the grammar), only a finite number of rules are required. (Alternatively, it seems, a recursive transition network could be used to obtain the surface structure, since such a device allows a node to have an arbitrary number of immediate descendants.)

This augmented grammar will produce all the valid surface structure parse trees but it will also produce, in general, many spurious trees (trees not derivable from deep structures) This is unavoidable, since a context-free grammar is a much weaker computational device than a transformational grammar. Because the language defined by the augmented base component is larger than that defined by the transformational grammar, the augmented base component is called a *covering grammar*.

Since each spurious surface analysis will have to undergo a lengthy reverse transformational process before it is recognized as invalid, it is important to minimize the number of such parses. The seriousness of this problem is indicated

by some early results obtained by the MITRE group. The MITRE system did not have a procedure for automatically augmenting the base component; theirs was assembled manually. Using a small grammar, one of their 12-word test sentences obtained 48 surface analyses, almost all of them spurious. Petrick had similar experience: he found that the covering grammars produced by his procedure were too broad, producing too many surface parses. He has instead, like the MITRE group, produced his surface grammars manually, by analyzing constructions which appear in the surface structure of input sentences to determine which productions are required. In this way, he has been able to produce a practically useful covering grammar for a limited area of discourse.

These practical difficulties do not negate the value of an automatic procedure, such as that described by Petrick, which will produce a covering grammar we can be sure is complete (will produce all valid surface analyses). They do indicate, however, the value of developing procedures which produce "tighter" surface grammars, perhaps by observing that certain sequences of transformations are impossible and hence suppressing the corresponding surface grammar productions. They also suggest that a more powerful device than a context-free grammar -- such as an "augmented* context-free grammar" should perhaps be used to generate the surface analyses. This view is held by a number of workers, such as Sager and Woods, who are also aiming at a transformational decomposition.

Armed with a covering grammar, we turn now to the construction of a reverse transformational component. This component should produce, from a surface structure, all base structures which can generate the surface structure (and for a spurious surface structure, indicate that there are no base structures).

---

* "augmented" meaning here that the grammar may contain predicates which are arbitrary computable functions.

The first problem is that it is not always possible to construct such a component. If a (forward) transformation simply deletes a portion of the parse tree, it will in general be impossible to reconstruct that portion when working backwards from the surface structure; there may be an infinite number of deep structures which produce one surface structure. Such a situation is called *irrecoverable deletion*. (This is in contrast to recoverable deletions, which make use of a component of forward transformations briefly mentioned earlier: identity conditions. An identity condition specifies two or more components of the structural index; the transformation may be applied only if the trees dominated by the nodes matched by these elements are identical. If some -- but not all -- of these trees are deleted by a transformation, the deletion is recoverable: the reverse transformational component may restore the deletion by copying another part of the tree.) So, to be able to construct a reverse component at all, the grammar may contain no irrecoverable deletions.

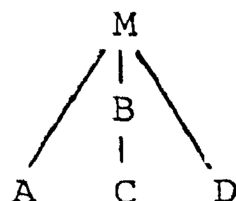
Life would be relatively easy if, for each transformation, one could produce an inverse transformation which undoes the change wrought in the tree. Unfortunately, for the form of transformation we have chosen, this is not possible. Consider, for example, a transformation with structural index

<A, C, D>

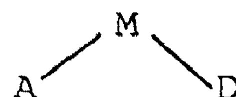
and structural change

<<1>, < >, <3>>

Suppose that the only structure to which it ever applies in the generation of a sentence is



producing



Reversing this transformation seems straightforward enough. The reverse transformation need not be a true inverse; it does not have to be able to reconstruct any input given to the forward transformation. It need only be able to reconstruct inputs which occur in the derivation of sentences. Thus, in this case, it must insert a B dominating a C below M.

This operation cannot be performed in the transformational formalism described above (unless a B dominating a C is already present in the tree). In terms of elementary changes to a parse tree, this formalism permits only deletion (of a node), replacement (of one node by another), and sister adjunction (insertion of one node "next" to another, with both dominated by the same node); it does not allow insertion of one node below another. This formalism was used in Petrick's original system. Most more recent systems, including the MITRE system and Petrick's later systems, have allowed a larger set of elementary operations, capable of making an arbitrary change to a tree.

Even if the set of operations is sufficient to form a set of reverse transformations, their formation is not trivial. In cases such as the one just considered, the reverse transformation cannot be generated from an examination of the forward transformation alone One must examine the entire grammar to see how the transformation is used in sentence generation. This is a complex process which has (to the author's knowledge) never been programmed. In the MITRE group, the reverse transformations were all produced manually.

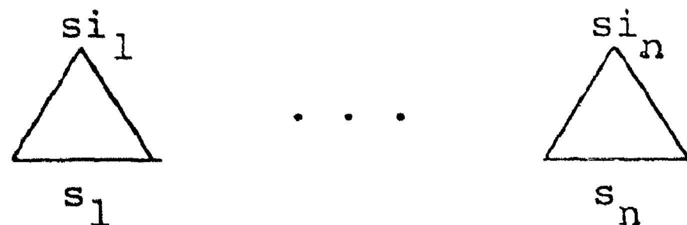
Petrick, seeking originally a procedure which would work automatically from the transformational grammar took a different tack. He developed a reverse transformational component which mapped the surface string (the sentence) into a set of potential deep structure strings; the latter were then parsed by the base component. The individual transformations of the reverse component are string and not tree rewriting rules.

The advantage of this approach lies in the simplicity of forming the individual reverse transformations. The reverse transformations will be in one-to-one correspondence with the forward ones, and each reverse transformation T' can be computed on the basis of the corresponding forward transformation T alone. These reverse transformations will satisfy the following property: for any tree t with frontier s, if T maps t into t' with frontier s', then T' maps s' into s.

Suppose we are given a forward transformation with structural index si and structural change sc. Since we are interested only in the frontier and not in the internal structure of the tree, we shall use a reduced structural change

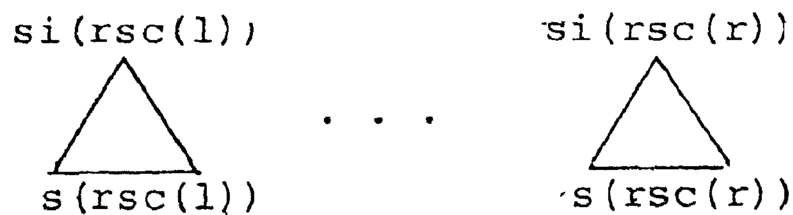
$$rsc = [+: 1 <= i <= \#sc]\ sc(i)$$

obtained by concatenating the elements of the structural change. The fact that a proper analysis exists for a tree with frontier s implies that s can be divided into substrings $s_1,\ldots,s_n$ such that, for all j from 1 to n, a tree can be built with root si_j and frontier s_j * (unless si_j = 'X', in which case there is no restriction on s_j):



The transformation rearranges the string into a set of substrings s_j' given by (for j from 1 to r, r = #rsc)

$$s'(j) = \text{if } rsc(j) \text{ is an integer then } s(rsc(j)) \text{ else } rsc(j)$$



* using the covering grammar.

How can this shuffle be reversed? We begin by creating an *inverse structural index* isi_j (1 <= j <= r) according to

isi(j) = if rsc(j) is an integer then si(rsc(j))
else rsc(j)

and an *inverse structural change* isc_j (1 <= j <= n) according to

isc(j) = if ∃k | rsc(k) eq j then k
else si(j)

Then, given a string s', we divide it into r substrings, requiring that the j-th substring be the frontier of some tree with root isi_j (again, unless isi_j = 'X'). One of these divisions will be the s_j produced by the forward transformation (there may be others). These substrings are then rearranged according to the isc, producing the original string s_j

s(j) = if isc(j) is an integer then s'(isc(j))
else isc(j)

If there are several matches to the isi, the transformation must be applied to all; we can only be sure that one of the resulting strings will be s. If the forward transformation is a recoverable deletion involving identity conditions, the formulas given above are somewhat more complicated.

Given a set of reverse transformations, we must finally specify the sequencing among them. The reverse transformations should be considered in precisely the reverse order from that of the corresponding forward transformations. The sequencing is again cyclic, with each iteration now creating an embedded sentence.

Even if a reverse transformation matches the sentence being decomposed, one cannot be sure that the corresponding forward transformation was involved in the generation of the sentence. Undoing the transformation may lead to a dead end

(no other reverse transformations apply), and another transformation may also have produced the current structure. Consequently, both possibilities -- undoing and not undoing the transformation -- must be followed. In analogy with the forward transformations, one can say that all reverse transformations are optional.

This implies, unfortunately, that parsing time can increase exponentially with the number of applicable transformations. Such a procedure has therefore proved impracticable for all but the smallest grammars and sentences. To avoid this exponential growth, the parser must have some way of determining directly from a tree the last transformation which applied to produce the tree. An analysis must be made of the possible intermediate structures which can arise in sentence generation, and the resulting information translated into conditions on the reverse transformations. Such an analysis has not been automated, but it is normally a straightforward and integral part of the manual construction of a reverse transformational component. The MITRE group was able to specify the appropriate conditions for all their reverse transformations; their system provided for optional reverse transformations but their grammar did not utilize this facility.

Eliminating optional reverse transformations is more difficult in a reverse component using string rewriting rules, not retaining any tree structure between transformations. * Most of the information which is needed to determine which transformation to undo is not available. In any case, the original impetus for using string rewriting rules -- providing a procedure which can operate directly from the transformational grammar -- is lost when we seek to add, for reasons of efficiency, restrictions which are not automatically generated from the grammar.

* Petrick's original system, using string rewriting rules, did retain some low-level tree structures between transformations, but his later systems did not.

Petrick's current parser, part of the REQUEST system, is much closer in overall structure to the MITRE design. A set of potential surface structure trees are operated upon by a reverse transformational component consisting of tree rewriting rules. The reverse transformations are prepared manually, not obtained automatically from corresponding forward transformations. The conditions on the reverse transformations are sufficiently tight to obviate the need for optional reverse transformations. As a result, they are able to operate efficiently with a moderately large set of reverse transformations (about 130).

Once all reverse transformations have been applied, the resulting structures must be checked to determine which are valid deep structures. If the reverse transformations work on trees, each tree must be examined for productions not in the base component. If the reverse transformations work on strings, each string must be parsed using the base component.

The original Petrick and MITRE procedures envisioned a final synthesis phase. This phase would apply to each deep structure produced by the reverse transformational component. It would apply the corresponding forward transformations to determine whether the original sentence can be recovered; if it cannot, the deep structure is rejected. Such a check is necessary if the reverse transformations can produce deep structures which do not lead back to the original sentence and perhaps do not lead to any sentence at all. This is certainly the case with the reverse transformations applied to strings; such transformations are unable to capture many constraints present when applying the forward transformations to trees. It can also be true with reverse transformations working on trees, if the constraints on the reverse transformations are too loose. With reverse transformations on trees, however, it should be possible to formulate constraints sufficiently tight as to obviate the need for a synthesis phase.

A synthesis check is optional in the current Petrick-Plath system. Instead of applying the forward transformations in a separate synthesis phase after a deep structure is obtained, however, they are applied during analysis after each corresponding inverse transformation is applied.

THE PROCEDURE

We present below a SETL version of one of Petrick's early transformational parsers. As was noted earlier, this algorithm is of importance because it is the only procedure which can work directly from a forward transformational grammar. The SETL program has been adapted from the LISP program developed by Petrick at the Air Force Cambridge Research Laboratories (1966), and is somewhat simpler than the version presented in Petrick's thesis. In particular, the 1966 version preserves no tree structure between reverse transformations.

Considerable liberty has been taken in rewriting the program for presentation here. Features which were not deemed essential to an understanding of the basic procedure were deleted. Specifically, the procedure for converting forward to reverse transformations was not included; identity conditions in transformations were not provided; optional elements in structural indices were not allowed. On the other hand, the gross flow of control of the LISP program has been preserved.

The main procedure of the parser, XFPARSE, takes five arguments:

SENTENCE	the string to be analyzed
XFMN	the set of reverse transformations
NUMXFMNS	the number of transformations
BASEGR	the (context-free) base component
AUXRULES	the additional rules which must be added to the base component to form the covering grammar

The context-free grammars have the form described in Sec. 3.1.
Each transformation has the following components:

XFMN(i,'ISI')	inverse structural index
XFMN(i,'NAME')	name of transformation
XFMN(i,'TYPE')	type of transformation: 'UNARY' or 'BINARY'

if the transformation is unary, it also has the component

XFMN(i,'ISC')	inverse structural change

if the transformation is binary, the inverse structural change will contain a pair of sentence boundary markers, with the component sentence inside the markers and the matrix sentence outside (the general form is m m m # c c c # m m m , with the m's part of the matrix sentence and the c's part of the component). For the parser, it is more convenient to represent this as two separate tuples, one with the boundary markers and the elements between them replaced by the symbol 'COMP' (m m m 'COMP' m m m) the other with the elements between the markers (c c c) In the transformation, these are components

XFMN(i,'ISC-MATRIX')	inverse structural change for matrix sentence
XFMN(i,'ISC-COMP')	inverse structural change for component sentence

The value of XFPARSE is a set, with each element giving one possible deep structure frontier for the sentence, and the reverse transformations which were applied to obtain that deep structure. To determine whether these are in fact deep structures for the sentence, it would be necessary to go through the generative transformational phase for each potential deep structure and verify that the original sentence can be obtained.

If the deep structure contains no embedded sentences, the structure of an element in the returned set is

<deep-structure-frontier, transformations-applied>

where deep-structure-frontier is a tuple whose elements are the symbols in the frontier of the possible deep structure. Transformations-applied is a tuple whose elements are the transformations applied to obtain this deep structure; the first element gives the *last* transformation applied in the decomposition, and hence the first which would be applied in generation. If the deep structure contains an embedded sentence, the element will still be a pair as just described; deep-structure-frontier, however, will not include as elements two boundary markers and the intervening embedded sentence. Instead at that point in the tuple will be an element which is itself a pair, with the first element the frontier of the embedded sentence and the second the transformations applied to decompose the embedded sentence. The transformations-applied element of the top-level pair will include only the embedding transformation and the transformations applied to the sentence before it was divided into matrix and constituent.

Since each reverse transformation is optional and may apply in several ways, there will usually be many paths to follow during the decomposition process. In XFPARSE, each such path is recorded as an element in the set TODO; the element is a frontier/history pair, just like those produced as output. The main loop of the parser runs over the transformations. For each element in TODO, if the transformation applies all possible transforms of the element are added to TODO; since the transformation is optional, the original element remains as well. When an inverse embedding transformation applies, XFPARSE is called recursively to decompose the embedded sentence.

```
definef XFPARSE(SENTENCE, XFMN, NUMXFMNS, BASEGR, AUXRULES);
local TODO, DONE, PARSES, SGRAMMAR, XFMNNO, CONT, SENT, XFAPPLD,
     MATCHSET, MATCH, COMPPARS, KOMP, MATRIX, P;
DONE ={<SENTENCE, nult>};
PARSES = nℓ;
     /* compute covering  grammar */
SGRAMMAR = BASEGR ∪ AUXRULES;
     /* iterate over transformations */
( 1 <= ∀XFMNNO <= NUMXFMNS)
     TODO = DONE;
     DONE = nℓ;
     /* iterate over active continuations */
     (∀CONT ∈ TODO)
          SENT = CONT(1);
          XFAPPLD = CONT(2);
          MATCHSET=PROCES(SENT,XFMN(XFMNNO,'ISI'),1,SGRAMMAR);
          /* iterate over matches to structural index */
          (∀MATCH ∈ MATCHSET)
               if XFMN(XFMNNO, 'TYPE') eq 'BINARY' then
                    /* for binary transformations, first try
                       to analyze embedded sentence */
                    COMPPARS=XFPARSE(IMPOSE (MATCH, XFMN(
                       XFMNNO,'ISC-COMP')),XFMN,XFMNNO,BASEGR,
                       AUXRULES);
                    /* iterate over analyses of embedded
                       sentence, adding matrix with analyzed
                       embedded sentence to continuations */
                    (∀KOMP ∈ COMPPARS)
                       MATRIX=IMPOSE(MATCH,XFMN(XFMNNO,
                                   'ISC-MATRIX'));
                       <MATRIX,<XFMNNO> + XFAPPLD> in DONE;
                       end ∀KOMP;
               else /* unary transformation */
                    /* add transformed sentence to continuations*/
```

```
                NEWSENT = IMPOSE(MATCH,XFMN(XFMNNO, 'ISC'));
                <NEWSENT,<XFMNNO> + XFAPPLD> in DONE;
            end ∀MATCH;
            /* include untransformed sentence in continuations,
               since reverse transformation is optional */
        CONT in DONE;
        end ∀CONT;
    end 1 <= ∀ XFMNNO;
      /* all transformations have been tried */
      /* select  and return those strings which can be
         analyzed by the base component */
return {P∈DONE|PARSE(BASEGR,'S',P(1)) ne nℓ};
end XFPARSE;
```

Most of the work of the parser is done by the two routines PROCES and IMPOSE. PROCES matches the current string against the inverse structural index, and IMPOSE computes the effect of the inverse structural change.

PROCES takes four arguments:

SENTENCE	the string to be matched
SI	the structural index
STARTWD	the number o the first word in the string to be matched by the structural index (this argument is required because the procedure operates recursively; its value is 1 for calls from XFPARSE)
GRAMMAR	the context-free grammar used in matching the structural index to the string.

The value of PROCES is a set, with each element giving one match of the structural index to the string. Each element is a forest, i.e., a tuple of trees, where each tree is represented as described in Sec. 3.1A. The nth tree of the forest has as its root Symbol the nth element of the structural index. Successive trees subsume contiguous segments of the string being matched. The following routine differs from Petrick's routine of the same name in using recursion instead of iteration.

```
definef PROCES(SENTENCE,SI,STARTWD,GRAMMAR);
local R, RMDRSI, MATCHES, ENDWD, P, M, RMDRMATCH, PARSES;
     /* split off first element of structural index */
R = hd SI;
RMDRSI = tl SI;
     /* parse part of remainder of sentence with this element*/
PARSES = PARTPARSE(GRAMMAR,R,SENTENCE,STARTWD);
MATCHES = nℓ;
(∀P ∈ PARSES)
     /* set ENDWD = next word in  sentence to be matched */
          ENDWD = P(1, 'LW+1');
          if RMDRSI eq nult then
               /* if at end of s.i., accept parse tree for last
                  element of s.i. only if it extends to end of
                  sentence */
               if ENDWD eq ((#SENTENCE) + 1) then
                  <P> in MATCHES;
                  end if ENDWD;
          else /* not at end of s.i., call PROCES recursively
                  to process next element */
               RMDRMTCH = PROCESS(SENTENCE,RMDRSI,ENDWD,GRAMMAR);
               /* concatenate parse tree for current element
                  to each forest of matches to succeeding elements*/
               (∀M ∈ RMDRMTCH)
                  <P> + M in MATCHES;
                  end ∀M ∈ RMDRMTCH;
               end if RMDRSI;
```

```
    end ∀P ∈ PARSES;
return MATCHES;
end PROCESS,
```

The transformational parser uses two varieties of context-free parser. The first, called simply PARSE, has the external specifications given in Sec. 3.1A. The other, PARTPARSE, differs in three respects:

1. the routine takes a fourth argument, STARTWD, specifying the first word in the sentence to be matched by the parser (preceding words are ignored); the returned value includes parse trees which do not extend to the end of the sentence.
2. if the specified root symbol is 'X', the routine creates a set of parse trees, each containing a single node, named X, spanning all possible substrings with first word = STARTWD
3. the symbol COMP will match an element of the string which is a tuple (since, as noted earlier, an embedded sentence is represented as a tuple).

The first two changes are effected by modifying the main routine of the first parser presented in Sec. 3.1.

```
definef PARTPARSE(GRAMMAR, ROOT, SENTENCE, STARTWD);
local PARSES, TREE, NODES, WORD, LW;
if ROOT eq 'X' then
     return {{<1, 'NAME', 'X'>, <1,'FW',STARTWD>, <1,'LW+1',LW+1>},
                (STARTWD+1) <= LW <= #SENTENCE};
     end if ROOT;
if STARTWD gt #SENTENCE then return nℓ; end if STARTWD;
TREE = nℓ;
PARSES = nℓ;
WORD = STARTWD;
NODES = 1;
```

```
TREE(1,'NAME') = ROOT;
TREE(1,'FW') = WORD;
(while EXPAND(1,GRAMMAR,SENTENCE))
     PARSES = PARSES ∪ {TREE};
     end while EXPAND;
return PARSES;
end PARTPARSE;
```

The third change is accomplished by modifying one line in the EXPAND routine of the first parser in Sec. 3.1. The line which tests for a match against the current sentence word is changed from

```
          if SENTENCE(WORD) eq TREE(X,'NAME')
```

to

```
          if (SENTENCE(WORD) eq TREE(X,'NAME'))
                              or
           ((type SENTENCE(WORD) eq tupl) and
            (TREE(X,'NAME') eq 'COMP'))
```

IMPOSE, which computes the effect of the structural change, is a routine with two arguments.

FOREST	is a tuple of trees representing the match to the structural index (one element of the value returned by PROCES)
SC	is the structural change component of a transformation

In addition, KOMP, which (for binary transformations) holds the embedded sentence and its transformational history, is passed from XFPARSE to IMPOSE as a global variable. IMPOSE returns a tuple, the frontiers of the trees in the forest as rearranged in accordance with the structural change.

```
definef IMPOSE(FOREST,SC)
local I;
return [+: I ∈ SC]
     if (type I) eq int then FRONTIER(FOREST(I))
     else if I eq 'COMP' then <KOMP>
     else <I>;
end IMPOSE;
```

The function FRONTIER takes as its argument a tree and returns the frontier of the tree. Since the root node of the tree specifies the words spanned in the sentence (variable SENT, declared in XFPARSE) this is trivial:

```
definef FRONTIER(TREE);
local FW, LWP1;
FW = TREE(1,'FW');
LWP1 = TREE(1,'LW+1');
return SENT(FW: LWP1-FW);
end FRONTIER;
```

Appendix. A Very Short Introduction to SETL
(Prepared in collaboration with Norman Rubin, CIMS.)

SETL is a programming language designed around set-theoretic constructions and developed at NYU by a group led by Jack Schwartz. The rich set of operators and control structures provided in SETL are intended to make the specification of algorithms considerably easier in SETL than in other higher-level languages. To facilitate the reading of our artificial intelligence surveys, we have used only a subset of SETL; this subset is described in the pages which follow. Those few points where we have deviated from a true SETL subset are marked on the left with asterisks and noted at the end of this section.

Further information on SETL is available in

J. T. Schwartz, On Programming: An Interim Report on the SETL Project, Parts I and II. Courant Computer Science Notes, Courant Institute of Mathematical Sciences, New York Univ.

K. Kennedy and J. Schwartz, "An Introduction to the Set Theoretical Language SETL." Comp. and Math. with Appl. 1 97.

General Features

Much of the expression semantics of SETL is modeled on that used in the mathematical theory of sets, and many of the syntactic conventions used reflect notions which are standard in that theory. However, for programming purposes a set theory including atoms which themselves have no members but may freely be members of sets is more convenient than pure set theory. Thus SETL contains both a general data object (the set) and other operationally more efficient structures known as atoms; among these is a vector-like object known as a tuple.

Atoms

(1) integers (1276)

(2) character strings ('hello, polly')

(3) bit strings (10101b)

(4) boolean constants (true ≡ 1b, false ≡ 0b)

(5) blank atoms (created by the function NEWAT, the SETL equivalent of the LISP GENSYM)

(6) Ω, the undefined atom

(7) labels (labels precede statements, separated by a colon)

(8) subroutines and functions

(9) tuples (one-dimensional arrays of variable length; i.e., ordered lists; written as <1,2,'three'>; the null tuple is designated nult)

Sets

are unordered collections of elements, written as {1,2,'three'} the null set is designated nℓ

Operators

Integers

arithmetic (+,-,*,/), comparison (eq,ne,lt,gt,le,ge) max,min, abs

Booleans

and, or, not (abbreviated n), eq, ne

Character and bit strings

concatenation (+), length (#), substring (S(I:K) is the string of length K starting with the Ith element, like the PL/I substring function)

Tuples

A tuple is a sequence of components $C_1, C_2, C_3, \ldots$, all but a finite number of which are equal to Ω, the undefined atom.

(1) T(K) is the Kth component of T

(2) T(I:K) is the tuple of length K starting with the Ith element

(3) #T is the index of the last defined component of T

(4) hd T ≡ T(1)

(5) tℓ T ≡ T(2:(#T-1))

(6) T + U is the concatenation of tuples T and U

Tuples are often used as push-down stacks, using T(#T+1) = X to push X on the stack and T(#T) = Ω to pop the stack.

Sets

(1) X ∈ S (true if X is an element of X, otherwise false)

(2) arb S (an arbitrary element of set S)

(3) A ∪ B (union of sets A and B)

(4) A ∩ B (intersection of sets A and B)

(5) A / B (symmetric difference of sets A and B).

(6) A - B (difference set)

(7) #S (number of elements in set S)

(8) A with B ≡ A ∪ {B}

(9) A less B ≡ A - {B}

(10) sets may be compared using eq, ne, incs (includes)

Precedence

There are three levels of precedence:

(1) (highest built-in binary operators producing Boolean from non-Boolean values (eq, ne, lt, gt, le, ge, incs, ∈)

(2) unary operators.

(3) other binary operators

Within each level, evaluation is left-to-right.

Set Definition

A set may be defined by enumeration {A,B,C} or by a set-former:

$$\{EXPR(X_1,\ldots,X_n), \text{range-restriction} \mid C(X_1,\ldots,X_n)\}$$

which forms the set of values $EXPR(X_1,\ldots,X_n)$ for those $X_1,\ldots,X_n$ within the range-restriction for which $C(X_1,\ldots,X_n)$ is true. The range-restriction is a series of items, one for each of the X_i, of the form

$$X_i \in S(X_1,\ldots,X_{i-1})$$

$$M \leq X_i \leq N$$

$$M < X_i \leq N$$

$$M \leq X_i < N$$

$$M < X_i < N$$

$$N \geq X_i \geq M$$

$$N > X_i \geq M$$

$$N \geq X_i > M$$

$$N > X_i > M$$

For example,

```
{<X,X*X>, 1 ≤ X ≤ 4 | X lt 4}
```

is

```
{<1,1>, <2,4>, <3,9>}
```

Two abbreviated forms are allowed:

```
{X ∈ S | C(X)}  for  {X, X ∈ S | C(X)}
```

and

```
{EXPR(X), X∈S}  for  {EXPR(X), X∈S|true}
```

Conditional Operators

if BOOL then $EXPR_1$ else $EXPR_2$

has the value $EXPR_1$ if BOOL is true, the value $EXPR_2$ if BOOL is false. The else is considered a unary operator, so that

```
if X gt 0 then Y else X + Y
```

is analyzed as

```
(if X gt 0 then Y else X) + Y
```

Functional Application and Sets

If F is a set and A any set or atom then

```
(1) F{A} ≡ {if #P gt 2 then tℓ P else P(2), P ∈ F |
           ("P is a tuple") and (#P gt 2) and (P(1) eq A)}
```

e.g., if F is a set of pairs, F{A} is the set of all X such that <A,X> $\in$ F.

(2) F(A) $\equiv$ if #F{A} eq 1 then arb F{A} else Ω.
i.e., the unique image of A under F.

(3) F[A] $\equiv$ the union of the elements of F{A}.

Sets are often used to define complex mappings. For instance

F = {<1,1>, <2,4>, <3,9>},

or alternatively

F(1) = 1; F(2) = 4; F(3) = 9;

defines a set mapping the first three integers into their squares. In either case, F(2) is 4 and F(4) is Ω.

Quantified Boolean Expressions

The basic forms for quantified boolean expressions are

$$\exists x \in s \mid C(x) \qquad \forall x \in s \mid C(x)$$

The first is true if C(x) is true for some x in s, and furthermore sets x to the first value found for which C(x) is true; the second is true if C(x) is true for all x in s. Several range restrictions may be combined:

$$\exists x_1 \in s_1,\ x_2 \in s_2(x_1),\ \forall x_3 \in s_3(x_1,x_2), \ldots \mid C(x)$$

The alternate, numerical, forms of range restrictions, such as $\min \le \exists x \le \max$ and $\min \le \forall x \le \max$, may also be used.

Compound Operators

Compound operators have the form

$$[op\text{: range-restrictions} \mid C(X_1,\ldots,X_n)]\ EXPR(X_1,\ldots,X_n)$$

where op is a binary operator and the range-restrictions have the form described earlier. The value of this expression is the value of variable VALUE after executing:

```
PILE = {EXPR(X1,...,Xn), range-restrictions|C(X1,...,Xn)};
VALUE from PILE;
(while PILE ne nl)
   X from PILE;
   VALUE = VALUE op X;
   end while PILE;
```

For example,

$$[\max: X \in \{1,2,3\}](X+1) \quad \text{is} \quad 4$$

$$[+ \; : 1 \le N \le K]\, A(N) \quad \text{is} \quad \sum_{1}^{K} A_N$$

Statements

All statements are terminated by a semicolon.

Assignment

```
A = EXPR;
<A,B,C> = EXPR; is the same as A=EXPR(1);B=EXPR(2);C=EXPR(3);
```

Assignment may also be done with the operator "is":

"EXPR is V" is an expression with value EXPR

and the side effect of assigning this value to V

X in S;	is	S = S with X;
X from S;	is	X=arb S; S = S less X;
X out S;	is	S = S less X;

Transfer

```
go to LABEL;
```

Conditional

if $BOOL_1$ then $BLOCK_1$ else if $BOOL_2$ then ... else $BLOCK_n$;

Iteration

(while BOOL) BLOCK;

$(\forall X_1 \in S_1,\ X_2 \in S_2(X_1), \ldots | C(X_1,\ldots,X_n))$ BLOCK;

$(M \le \forall X \le N)$ BLOCK;

$(N \ge \forall X \ge M)$ BLOCK;

etc.

Scope of Conditionals and Iterators

The BLOCK indicated above as the scope of a SETL conditional statement or iterator is any sequence of SETL statements. Note that the semicolon terminating the block is in addition to the semicolon terminating the final statement of that block. This semicolon may be replaced by an end statement such as end ∀; end ∀X; end while; end while BOOL; to indicate to the reader which scope is being closed.

Ex.:

```
(1 ≤ ∀X ≤ 100 | P(X)) Y(#Y+1) = X;  end ∀X;
if Y gt 0 then S = S with X; else N = N+1; end if Y;
```

Output

print $EXPR_1$,$EXPR_2$,...,$EXPR_n$;

Subroutines and Functions

Subroutines

defined by define SUB(X,Y,Z); BLOCK end SUB;

invoked by SUB(A,B,C);

exit from subroutine by return;

Functions

defined by definef FCN(X,Y,Z); BLOCK end FCN:

invoked by FCN(A,B,C)

exit from function by return EXPR;

Infix operator definition

defined by X infixop Y; BLOCK end A infixop B;

Name Scoping

local X,Y,Z; defines X,Y, and Z as local to the current subroutine or function (these variables are allocated on entry to the routine). Name scoping is dynamic, as in LISP; a variable declared local by procedure p is available to all procedures invoked by p which do not themselves declare the variable local. Thus

```
define P;
   local X;
   Q(1);
   print X;
   return;
   end P;
define Q(Y);
   X = Y;
   return;
   end Q;
```

will print a 1.

Differences from Standard SETL

Standard SETL uses a somewhat smaller character set than we have adopted in this survey. Thus + (addition), + (concatenation) and ∪ (union) are all written as + in standard SETL; * (multiplication) and ∩ (intersection) are written as *. We have adopted the simple and familiar name-scoping rules of the current SETL implementation in place of the relatively complex ones of the SETL standard.

We have written all variables, function names, and subroutin names in upper case, operator names and other tokens in lower case.

BIBLIOGRAPHY

[Aho 1972] A. V. Aho and J. D. Ullman. The Theory of Parsing, Translation, and Compiling, Vol. I. Prentice-Hall, Englewood Cliffs, N. J.

[Bobrow 1969] D. Bobrow and B. Fraser, "An Augmented State Transition Network Analysis Procedure," Proc. International Joint Conference on Artificial Intelligence.

[Borgida 1975] Alexander Borgida, Topics in the Understanding of English Sentences by Computer. Tech. Rep. 78, Dept. of Computer Science, Univ. of Toronto.

[Bross 1968] Irwin Bross, "A Syntactic Formula for English Sentences: Application to Scientific Narrative," Computers and Biomedical Research 1, 565.

[Cautin 1969] Harvey Cautin, Real English: A Translator to Enable Natural Language Man-Machine Conversation. Thesis, Moore School of Electrical Engineering, Univ. of Pennsylvania.

[Cocke 1970] J. Cocke and J. T. Schwartz, Programming Languages and their Compilers. Lecture Note series, Courant Institute of Mathematical Sciences, New York Univ.

[Colmerauer 1970] Alain Colmerauer, "Les Systems-Q ou un formalisme pour analyser et synthetiser des phrases sur ordinateur. Publ. interne no. 43, Faculté des sciences, Université de Montréal.

[Craig 1966] J. A. Craig, S. C. Berenzner, H. C. Carney, and C. R. Longyear, "DEACON: Direct English Access and Control." Proc. 1966 Fall Joint Computer Conf., Thompson Books, Washington, D. C.

[Culicover 1969] P. Culicover, J. Kimball, C. Lewis, D. Loveman, J. Moyne, An Automated Recognition Grammar for English, IBM Technical Report FSC 69-5007.

[de Chastellier 1969] G. de Chastellier and A. Colmerauer, "W-Grammar," Proc. 24 National Conf. Assn. for Comp. Mach..

[Dewar 1969] H. Dewar, P. Bratley, and J. P. Thorne, "A Program for the Syntactic Analysis of English Sentences," Comm. Assn. Comp. Mach. 12, 476.

[Dostert 1971] B. Dostert and F. Thompson, "How Features Resolve Syntactic Ambiguity," Proc. Symposium on Information Storage and Retrieval.

[Earley 1970] J. Earley, "An Efficient Context-Free Parsing Algorithm. Comm. Assn. Comp. Mach. 13, 94.

[Grishman 1973a] Ralph Grishman, "Implementation of the String Parser of English," in Natural Language Processing, ed. R. Rustin, Algorithmics Press, New York.

[Grishman 1973b] R. Grishman, N. Sager, C. Raze, and B. Bookchin, "The Linguistic String Parser." Proc. 1973 Natl. Computer Conf., AFIPS Press, Montvale, N. J.

[Harris 1965] Zellig Harris, String Analysis of Sentence Structure. Mouton, The Hague.

[Hays 1967] David Hays, Introduction to Computational Linguistics. American Elsevier, New York.

[Hiz 1967] D. Hiz and A. Joshi, "Transformational Decomposition: A Simple Description of an Algorithm for Transformational Analysis of English Sentences." 2ème Conf. Internationale sur le Traitement Automatique des Langues, Grenoble.

[Hobbs 1974] Jerry Hobbs, A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language. Courant Computer Science Report #2, Courant Inst. Math. Sci., New York Univ.

[Hobbs 1975] J. Hobbs and R. Grishman, "The Automatic Transformational Analysis of English Sentences: An Implementation." Submitted to Int'l J. Computer Math.

[Irons 1963] N. Irons, "Error-Correcting Parse Algorithm." Comm. Assn. Comp. Mach. 6, 669.

[Joshi 1962] Aravind Joshi, A procedure for transformational decomposition. Transformations and Discourse Analysis Papers #42, Univ. of Pennsylvania.

[Joshi 1973] Aravind Joshi, "A Class of Transformational Grammars." In The Formal Analysis of Natural Languages, ed. M. Gross, M. Halle, and M.-P. Schützenberger, Mouton, The Hague.

[Kay 1967] Martin Kay, "Experiments with a Powerful Parser." In 2ème Conf. Internationale sur le Traitement Automatique des Langues, Grenoble.

[Keyser 1967] S. J. Keyser and S. R. Petrick, Syntactic Analysis. Air Force Cambirdge Research Laboratories, AFCRL-67-0305.

[Kittredge 1973] Richard Kittredge et al., TAUM 73. A report of the Projet de Traduction Automatique de l'Université de Montréal.

[Kuno 1962] S. Kuno and A. G. Oettinger, "Multiple-Path Syntactic Analyzer." Information Processing 1962, North-Holland, Amsterdam.

[Kuno 1963] Susumo Kuno, "The Multiple-path Syntactic Analyzer for English." Report No. NSF-9 in Mathematical Linguistics and Automatic Translation of the Computation Lab., Harvard Univ.

[Kuno 1965] Susumo Kuno, "The Predictive Analyzer and a Path Elimination Technique." Comm. Assn. Comp. Mach. 8, 453.

[Loveman 1971] D. Loveman, J. Moyne, and R. Tobey, "CUE: A Preprocessor System for Restricted, Natural English." In Proc. Symposium on Information Storage and Retrieval.

[Owens 1975] Phillip Owens, A Comprehensive Survey of Parsing Algorithms for Programming Languages, Courant Computer Science Report #4, Courant Inst. Math. Sci., New York Univ. (Forthcoming).

[Paxton 1973] W. H. Paxton and A. E. Robinson, "A Parser for a Speech Understanding System." Advance Papers of the Third Intl. Joint Conf. on Artificial Intelligence, Stanford Research Institute, California.

[Petrick 1965] Stanley R. Petrick, A Recognition Procedure for Transformational Grammars. MIT Doctoral Dissertation.

[Petrick 1966] Stanley R. Petrick, A Program for Transformational Syntactic Analysis, Air Force Cambridge Research Laboratories, AFCRL-66-698.

[Petrick 1973] Stanley R. Petrick, Transformational Analysis." In Natural Language Processing, ed. R. Rustin, Algorithmics Press, N. Y.

[Petrick 1975] Stanley R. Petrick, "Design of the Underlying Structure for a Data Base Retrieval System." In Directions in Artificial Intelligence: Natural Language Processing, ed. R. Grishman, Courant Computer Science Report #7, Courant Institute of Mathematical Sciences, New York Univ.

[Plath 1974a] Warren J. Plath, "Transformational Grammar and Transformational Parsing in the REQUEST System." In Computational and Mathematical Linguistics, Proc. Intl. Conf. on Computational Linguistics, ed. A. Zampolli.

[Plath 1974b] Warren J. Plath, String Transformations in the REQUEST System. IBM T. J. Watson Research Center, RC 4947 (#21963).

[Raze 1974] Carol Raze, A computational treatment of coordinate conjunctions. Talk at 12 Ann. Meeting of Assn. of Computational Linguistics, Amherst, Mass., July 26, 1974.

[Sager 1967] Naomi Sager, "Syntactic analysis of natural language." In Advances in Computers, No. 8, ed. F. Alt and M. Rubinoff, Academic Press, N. Y.

[Sager 1973] Naomi Sager, "The string parser for scientific literature." In Natural Language Processing, ed. R. Rustin, Algorithmics Press, N. Y.

[Sager 1975] N. Sager and R. Grishman, "The Restriction Language for Computer Grammars of Natural Language." Comm. Assn. Comp. Mach. 18, 390.

[Shapiro 1971] P. Shapiro and D. Stermole, "ACORN (Automatic Coder Report Narrative): An Automatied Natural-Language Question-Answering System for Surgical Reports." Computers and Automation, Feb. 1971, p. 13.

[Simmons 1975] R. Simmons and G. Bennett-Novak, "Semantically Analyzing an English Subset for the Clowns Micro-world." Tech. Report NL-24, Dept. of Computer Sciences, Univ. of Texas at Austin.

[Thompson 1969] F. B. Thompson, P. C. Lockeman, B. Dostert, and R. S. Deverill, "REL: A Rapidly Extensible Language System." Proc. 24 Natl. Conf. Assn. Comp. Mach.

[Thorne 1968] J. P. Thorne, P. Bratley, and H. Dewar, "The Syntactic Analysis of English by Machine." Machine Intelligence 3.

[Walker 1966] D. Walker, P. Chapin, M. Geis, and L. Gross, Recent Developments in the MITRE Syntactic Analysis Procedure. MITRE Report MTP-11.

[Walker 1973] Donald Walker, "Automated Language Processing." In Annual Review of Information Science and Technology, Vol. 8, ed. C. Cuadra, American Society for Information Science, Washington, D. C.

[Wilks 1975] Yorick Wilks, "An Intelligent Analyzer and Understander of English." Comm. Assn. Comp. Mach. 18, 264.

[Winograd 1971] Terry Winograd, Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. MIT Report MAC TR-48.

[Woods 1970a] William A. Woods, "Context-Sensitive Parsing." Comm. Assn. Comp. Mach. 13, 437.

[Woods 1970b] William A. Woods, "Transition Network Grammars for Natural Language Analysis." Comm. Assn. Comp. Mach. 13, 591.

[Woods 1972] W. A. Woods, R. M. Kaplan, B. Nash-Webber, The Lunar Sciences Natural Language Information System: Final Report. Report #2378, Bolt Beranek and Newman, Cambridge, Mass.

[Woods 1973] William A. Woods, "An Experimental Parsing System for Transition Network Grammars." In Natural Language Processing, ed. R. Rustin, Algorithmics Press, New York.

[Zwicky 1965] A. Zwicky, J. Friedman, B. Hall, and D. Walker, "The MITRE Syntactic Analysis Procedure for Transformational Grammars." Proc. 1965 Fall Joint Computer Conf., Thompson Books, Washington, D. C.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Value
1. REPORT NUMBER	NSO-8
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	A Survey of Syntactic Analysis Procedures for Natural Language
5. TYPE OF REPORT & PERIOD COVERED	Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Ralph Grishman
8. CONTRACT OR GRANT NUMBER(s)	N00014-67A-0467-0032
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Courant Institute of Math. Sci. New York University 251 Mercer St., New York, N.Y. 10012
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Office of Naval Research Department of the Navy Arlington, Virginia 22217
12. REPORT DATE	August 1975
13. NUMBER OF PAGES	94
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	
15. SECURITY CLASS. (of this report)	unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report 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, syntax, parsing, grammar, computational linguistics

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

This report includes a brief discussion of the role of automatic syntactic analysis, a survey of parsing procedures, past and present, and a discussion of the approaches taken to a number of difficult linguistic problems, such as conjunction and graded acceptability. It also contains precise specifications in the programming language SETL of a number of parsing algorithms.