EXPERIMENTS WITH A POWERFUL PARSER

by

Martin KAY

The Rand Corporation
1700 Main Street
SANTA-MONICA - California 90406 - U.S.A.

EXPERIMENTS WITH A POWERFUL PARSER

This paper describes a sophisticated syntactic-analysis program for the IBM 7040/44 computer and discusses some of the problems which it brings to light. Basically the program is a nondeterministic device which applies unrestricted rewriting rules to a family of symbol strings and delivers as output all the strings that can be derived from members of the initial family by means of the rules provided. A subsidiary mechanism deals with the relation of dominance in the sense common in linguistics. This makes it possible for rules to refer to complete or partial syntactic structures, or P-markers, so that the program can be used at least to some extent for transformational analysis.

A program of this kind, which is intended for analysing natural languages, must be capable of operating on a family of strings as a single unit because of the grammatical ambiguity of words. Take, for example, the famous sentence "Time flies like an arrow." These five words are not, themselves, the primary data on which a parsing program can be expected to operate. Instead, each word is replaced by one or more symbols representing the grammatical categories to which it belongs. The assignments for this example might be somewhat as follows:

Word	Grammatical category
Time	Noun, verb, adjective
flies	Plural noun, 3rd person verb
like	Singular noun, preposition, verb
an	Indefinite article
arrow	Singular noun, adjective.

Taking one category symbol for each word, it is possible to form 30 different strings, preserving the order of the original sentence. These 30 strings constitute the family on which the program would operate if set to analyze this sentence.

The program is said to perform as a non-deterministic device because whenever two mutually incompatible rules are applicable to the same string neither is given any priority; both are applied, and the resulting strings developed independently. Given the string "A B C" and the rules

$$A\ B \rightarrow X\ Y$$

$$B\ C \rightarrow Z$$

the program will therefore produce two new strings:

X Y C

A Z

The program contains no mechanism for guarding against sequences of rules which do not terminate. If the grammar contains the following rules

$$A\ B \rightarrow B\ A$$

$$B\ A \rightarrow A\ B$$

and the string to be parsed contains either "A B" or "B A," then the program will continue substituting these sub-strings for one another until the space available for intermediate results is exhausted. This may not seem to present any particularly severe problem because a pair of rules such as these would never appear in any properly constructed grammar. But, as we shall shortly see, entirely plausible grammars can be constructed for which this problem does arise.

1. THE FORM OF RULES

In order to get a general idea of the capabilities of the program, it will be useful first to consider the notation used for presenting rules to it and the way this is interpreted by the machine. In what follows, we shall assume that the reader is familiar with the terminology and usual conventions of phrase-structure and transformational grammar. An example of the simplest kind of rewrite rule is

```
VPRSG  =  PRES SG VERB
```

The "equals" sign is used in place of the more familiar arrow to separate the left and right-hand sides of the rule. The symbols on which the rules operate are words consisting of between one and six alphabetic characters. The above rule will replace the symbol "VPRSG" by a string of three symbols "PRES SG VERB" whenever it occurs. The following rule will invert the order of the symbols "VERB" and "ING"

```
VERB  ING  =  ING  VERB
```

The simplest way to represent a context free phrase structure rule is as in the following example:

NP AUX VP = S

Notice that the normal order of the left and right-hand sides of the rule is reversed because the recognition process consists in rewriting strings as single symbols; the rules must therefore take the form of reductions rather than productions.

The program will accept phrase structure rules in the form we have shown, but, in applying them, it will not keep a record of the total sentence structure to which they contribute. In other words, it will cause a new string to be constructed, but will not relate this string in any way to the string which was rewritten. One way to cause this relationship to be preserved is to write the rule in the following form:

NP.1 AUX.2 VP.3 = S(1 2 3)

The number following the symbols on the left-hand side of the rule function very much like the numbers frequently associated with structural indices in transformational rules. When the left-hand side of the rule is found to match a particular sub-string, the number associated with a given symbol in the rule becomes a pointer to, or a temporary name for, that symbol. With this interpretation, the left-hand side of the above rule can be read somewhat as follows "Find an NP and call it 1; Find an AUX following this and call it 2; Find a VP following this and call it 3."

The numbers in parentheses after a symbol on the right-hand side of a rule are pointers to items identified by the left-hand side, and which the new symbol must dominate. In the example, the symbol "S" is to dominate all the symbols mentioned on the left-hand side.

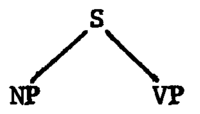
A pointer may refer to a single symbol, as we have shown, or to a string of symbols. The following rule is equivalent to the one just described:

NP.1 AUX.1 VP.1 = S(1)

Furthermore, the string to which a pointer refers need not be continuous. Consider the following example

NP.1 AUX VP.1 = S(1)

This will cause any string "NP AUX VP" to be re-written as "S", but the "S" will dominate only "NP" and "VP." There will be no evidence of the intervening "AUX" in the final P-marker which will contain the following phrase:

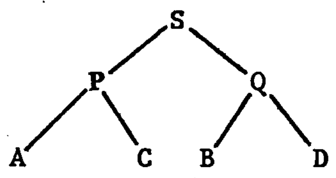


Consider now the following pairs of rules:

A.1 B.2 C.1 D.2 = P(1) Q(2)

P.1 Q.1 = S(1)

If these rules are applied to the string "A B C D" the following P-marker will be formed:



Notice that the first rule in the pair not only re-orders the symbols in the P-marker but forms two phrases simultaneously.

A different way of using pointer numbers on the right-hand side can be illustrated by comparing the effects of the following two rules:

N.1 SG.1 V.2 SG.3 = NOUN(1) V(2) SG(3)

N.1 SG.1 V.2 SG.2 = NOUN(1) 2

What is required, we assume, is a context sensitive phrase structure rule which will rewrite "N SG" as "NOUN" in the environment before "V SG". The first rule achieves this effect but also introduces a new "V" dominating the old one, and a new "SG". The second rule does what it really wanted: It constructs phrase labeled "NOUN" as required, and leaves the symbols referred to by pointer number 2 unchanged.

The context sensitive rule just considered is presumably intended to insure that singular verbs have only singular subjects. A second rule in which "SG" is replaced by "PL" would be required for plural verbs. But, since agreements of this kind may well have to be specified in other parts of the grammar, the situation might better be described by the following three rules:

```
SG.1 = NUM(1)

PL.1 = NUM(1)

N.1 NUM.2 V.3 2 = NOUN(1 2) 3 2
```

The first two rules introduce a node labeled "NUM" into the structure above the singular and plural morphemes. The third rule checks for agreement and forms the subject noun phrase. Pointer number 2 is associated with the symbol "NUM" in the second place on the left-hand side, and occurs by itself in the fourth place. This means that the fourth symbol matched by the rule must be "NUM," and also that it must dominate exactly the same sub-tree as the second. In the example we are assuming that "NUM" governs a single node which will be labeled either "SG" or "PL" and the rule will ensure that whichever of these is dominated by the first occurrence of "NUM" will also be dominated by the second occurrence. Notice that noun and verb phrases could be formed simultaneously by the following rule:

```
N.1 NUM.2 V.3 2 = NOUN(1 2) VERB(3 2)
```

The symbols "ANY" and "NULL" are treated in a special way by this program and should not occur in strings to be analyzed. The use of the symbol "NULL" is illustrated in the rule:

```
PPH = NULL
```

This will cause the symbol "PPH" to be deleted from any string in which occurs. The program is non-deterministic in its treatment of rules of this kind, as elsewhere, so that it will consider analyses in which the symbol is deleted, as well as any which can be made by retaining it. The symbol "NULL" is used only on the right-hand sides of rules.

The symbol "ANY" is used only on the left-hand sides of rules and has the property that the word implies, namely that it will match any symbol in a string. The use of this special symbol is illustrated in the following rule:

VERB.1 ANY.1 NP.1 = VP(1)

This will form a verb phrase from a verb and a noun phrase, with one intervening word or phrase, whose grammatical category is irrelevant.

Elements on the left-hand sides of rules can be specified as optional by writing a dollar sign to the left or right of the symbol as in the following rules:

DET.1 ADJ$.1 NOUN.1 = NP(1)

VERB.1 $ANY.1 NP.1 = VP(1)

The first of these forms a noun phrase from a determiner and a noun, with or without an intervening adjective. The second is a new version of a rule already considered. A verb phrase is formed from a verb and a noun phrase, with or without an intervening word or phrase of some other type.

Elements can also be specified as repeatable by writing an asterisk against the symbol, as in the following example:

VERB.1 *NP.1 = VP(1)

This says that a verb phrase may consist of a verb followed by one or more noun phrases. It is often convenient to be able to specify that a given element may occur zero or more times. This is done in the obvious way by combining the dollar sign and the asterisk as in the following rule:

$DET.1 *$ADJ.1 N.1 *PP$.1 = NP(1)

According to this, a noun may constitute a noun phrase by itself. However the noun may be preceeded by a determiner and any number of adjectives, and followed by a prepositional phrase, and all of these will be embraced by the new noun phrase that is formed. Notice that the asterisk and the dollar sign can be placed before or after the symbol they refer to. The combination is often useful with symbol "ANY" in rules of the following kinds

N.1 NUM.2 *$ANY.3 V.4 2 = NOUN(1 2) 3 VERB(4 2)

This is similar to an earlier example. It combines the number morpheneme with a subject noun and with a verb, provided that the two agree, and allows for any number of other symbols to intervene. The symbol "ANY" with an asterisk and a dollar sign corresponds in this system to the so called variables in the familiar notation of transformational grammar.

Consider now the following rule:

```
SCONJ.1 NP(S).1 = NP(1)
```

This will form a noun phrase from a subordinating conjunction followed by a noun phrase, provided that this dominates only the symbol "S." Any symbol on the left-hand side of the rule may be followed by an expression in parentheses specifying the string of characters that this symbol must directly dominate. This expression is constructed exactly like the left-hand sides of rules. In particular, it may contain symbols followed by expressions in parentheses. The following rule will serve as an illustration of this, and of another new feature:

```
NP($DET.1 $*ANY.1 ADJ(PRPRT.2) $*ANY.3 N.4

   $ PP.5)  1 3 4 WH DEF 4 BE ADJ((2)) 5
```

This rule calls for a noun phrase consisting of a noun, a preceding adjective which dominates a present participle and, optionally, a number of other elements. This noun phrase is replaced by the determiner from the original noun phrase, if there is one, the elements preceding the noun except for the present participle, the noun itself, the symbol "WH," the symbol "DEF," another copy of the noun, the symbol "BE," the symbol "ADJ" dominating exactly those elements originally dominated by "PRPRT" and, finally, any following prepositional phrases the original noun phrase may have contained. The number "2" in double parentheses following "ADJ" on the right-hand side of this rule specifies that this symbol is to dominate, not the present participle itself, but the elements, if any, that it dominates. This device turns out to have wide utility.

Double parentheses can also be used following a symbol on the left-hand side of a rule, but with a different interpretation. We have seen how single parentheses are used to specify the string immediately dominated by a given symbol. Double

parantheses enclose a string which must be a pro-
per analysis of the sub-tree dominated by the given symbol. A string is said to be a proper analysis of a sub-tree if each terminal symbol of the sub-tree is dominated by some member of the string. As usual, a symbol is taken to dominate itself. As an example of this, consider the following rule:

```
ART.1 S((ART N.2 ANY*)).1 2 = DET(1) 2
```

This rule applies to a string consisting of an article, a sentence, and a noun. The sentence must be analysable, at some level, as an article followed by a noun, followed by at least one other word or phrase. The noun in the embeded sentence, and the sub-tree it dominates, must be exactly matched by the noun corresponding to the last element on the left-hand side of the rule. The initial article and the embeded sentence will be collected as a phrase under the symbol "DET" and the final noun will be left unchanged.

The principal facilities available for writing rules have now been exemplified. Another kind of rule is also available which has a left-hand side like those already described but no equal sign or right-hand side. However it will be in the best interests of clarity to defer an explanation of how these rules are interpreted.

The user of the program may write rules in exactly the form we have described or may add information to control the order in which the rules are applied. This additional information takes the form of an expression written before the rule and separated from it by a comma. This expression, in its turn, takes one of the following forms:

n_1,

n_1/n_2,

$n_1/n_2/n_3$,

$n_1//n_3$,

n_1 in an integer which orders this rule relative to the others. Since the same integer can be assigned to more than one rule, the ordering is partial. Rules to which no number is explicitly assigned are given the number 0 by the program.

n_2 and n_3, when present, are interpreted as follows: Every symbol in the sub-string matched by the left-hand side of the rule must have been produced by a rule with number i, where $n_2 \geq i \geq n_3$. For these purposes the symbols in the original family of strings offerred for analysis are treated as though they had been produced by a rule with number 0.

2. PHRASE-STRUCTURE GRAMMAR

It will be clear from what has been said already that this program is an exceedingly powerful device capable of operating on strings and trees in a wide variety of ways. It would clearly be entirely adequate for analyzing sentences with a context-free phrase-structure grammar. But this problem has been solved before, and much more simply. We have seen how the notation can be used to write context-sensitive rules, and we should therefore expect the program to be able to analyze sentences with a context-sensitive grammar. However in the design of parsing algorithms, as elsewhere, context-sensitive grammars turn out to be surprisingly more complicated than context-free grammars.

The problem that context-sensitive grammars pose for this program can be shown with a simple example.[1] Consider the following in grammar:

$$S \rightarrow \left\{\begin{array}{l} A\ B\ C \\ D\ E\ (S) \end{array}\right\} \quad \begin{array}{l} (1) \\ (2) \end{array}$$

$$B \rightarrow \left\{\begin{array}{l} D/A\_\_\_ \\ F/\_\_\_E \end{array}\right\} \quad \begin{array}{l} (3) \\ (4) \end{array}$$

$$D \rightarrow \left\{\begin{array}{l} G/A\_\_\_ \\ B/\_\_\_E \end{array}\right\} \quad \begin{array}{l} (5) \\ (6) \end{array}$$

This grammar, though trivial, is well behaved in all important ways. The language generated, though regular and unambigious, is infinite.

[1] I am indebted for this example, as for other ideas too numerous to document individually, to Susumu Kuno of Harvard University.

Furthermore, every rule is useful for some derivation. Since the language generated is unambigious, the grammar is necessarily <u>cycle-free</u>, in other words, it produces no derivation in which the same line occurs more than once. Suppose, however, that the grammar is used for analysis and is presented with the string"A D E" — not a sentence of the language. The attempt to analyze this string using rules of the grammar results in a rewriting operation that begins as follows and continues indefinitely:

```
A D E
A B E (by rule 3)
A D E (by rule 6)
A B E (by rule 3)
  etc.
```

It would clearly be possible, in principal, to equip the program with a procedure for detecting cycles of this sort, but the time required by such a procedure, and the complexity that it would introduce into the program as a whole, are sufficient to rule it out of all practical consideration. It might be argued that the strings which have to be analyzed in practical situations come from real texts and can be assumed to be sentences. The problem of distinguishing sentences from nonsentences is of academic interest. But, in natural languages, the assignment of words to grammatical categories is notoriously ambigious and for this problem to arise it is enough for suitably ambigious words to come together in the sentence. A sentence which would be accepted by the above grammar, but which would also give rise to cycles in the analysis, might consist of words with the following grammatical categories:

Word	Grammatical Category
1	A
2	B
3	C, E

The program, as it stands, contains no mechanism which automatically guards against cycles. However, if the user knows where they are likely to occur or discovers them as a result of his experience with the program, he can include some special rules in his grammar which will prevent them from occurring. These rules, which we have already eluded to, are formally similiar to all others except that they contain no equals sign and no right-hand side. When a P-marker is found to contain a string which matches the left-hand side of one of these rules, the program arranges that, thence forward, no other rule shall be allowed to apply to the whole string. The cycle in this latest example could not occur if the grammar contained the rule:

A B E

3. TRANSFORMATIONAL GRAMMAR

We now come to the main concern of this paper which is to discuss the extent to which the program we have been describing can be made to function as a transformational analyzer. The main purpose of the examples that have been given is to show the great power of the program as a processor of symbol strings. The notion of dominance is provided for, but only in a rudimentary way. It certainly could not be claimed that the program is a tree processor in any really workable sense. But grammatical transformations are operations on trees and our investigation therefore must take the form of showing that these operations can frequently, if not always, be mimicked by string rewriting rules.

We shall take it that a transformational grammar consists of a context-free or context-sensitive phrase-structure component and a set of transformations ordered in some way. To begin with, very little will be lost if we assume that the transformational rules are simply ordered.

Consider now the first transformation in the list. In general, this may be expected to introduce phrases into the P-markers to which it applies which could not have been generated by the phrase-structure component. Let us now write some additional phrase-structure rules capable of generating these new phrases. Let us insert these rules into the grammar immediately following the first transformational rule and establish the convention that, when they are used in the analysis of the string, their output will be used only as input to the first transformation. Now treat the second transformational rule in the same way. It also can be expected to create new kinds of phrase and phrase-structure rules can be written which would recognize these. It may be that some of the phrases formed by the second rule could also be formed by the first, and in this case, it may be possible to move the appropriate rule from its position after the first transformation to a position after the second and to mark it as providing input only for these two rules.

Notice that the rules we are proposing to construct will not constitute what has sometimes been called a surface grammar. The phrases they describe certainly do not belong to the base structure and many of them may not be capable of surviving unchanged into the surface structure. In general these rules describe phrases which can only have transititory existence somewhere in the generative process. Notice also that in order to describe these phrases adequately it may sometimes be necessary to extend the notion of phrase structure grammar somewhat. Consider for example the following transformation:

```
X  -  A  -  B  -  Y

1     2     3     4
```

Adjoin 2 as right daughter of 3

If we make the usual assumption that a rule is applied repeatedly until no proper analyses of the P-marker remain which can be matched by its structural index, then this transformation, and many others, may produce phrases of indefinitely many types. Let us suppose that, before this transformation is applied for the first time, all possible phrases that can be dominated by the symbol "B" are describable by context free phrase structure rules of the following form

$$B \rightarrow \left\{ \begin{matrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{matrix} \right\}$$

where the α_i are any strings. The phrase structure grammar needed to describe all the phrases that can exsist after the operation of this transformation must contain the following rules, or more accurately rule schemata

$$B \rightarrow \left\{ \begin{matrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{matrix} \right\} A^*$$

Where the asterisk indicates one or more repetitions of the symbol "A". If the left and right-hand sides of these rules are reversed and they are presented to the program in the proper notation, then the transformation itself can be represented by the following pair of rules:

```
B(*$ANY.1 *A.2) = 2 B+ 1 +B

B+ B.1 +B = 1
```

Since there are no facilities for specifying dominance relations among elements on the right-hand sides of these rules, it is necessary to resort to subterfuge. The phrase dominated by the symbol "B" is reproduced in the output of this rule with copies of the symbol "A" removed from the right-hand end and the remainder bounded by the symbols "B+" and"+B". These symbols serve to delimit a part of the string which can only figure in the complete analysis of the sentence if it constitutes a phrase of type "B". The second rule removes these boundry symbols from the phrase of type "B" and, since no pointer is assigned to them, they will leave no trace in the final P-marker.

Another, and perhaps more economical, way to write recognition rules corresponding to this transformation involves conflating the additional phrase-structure rules with the reverse of the transformational rule itself to give rules of the following kind:

α_i1 *A.2 = 2 B+ 1 +B $(1 \le i \le n)$

B+ B.1 + B = 1

In fact, the elementary transformation for daughter adjunction that we are providing for here is more general than that often allowed by transformational grammarians. It is common to require that if some element a is adjoined as a daughter of another element b then b must have no daughters before the transformation takes place.

Sister adjunction can be treated in an analogous manner. Consider the following transformation:

$$\begin{array}{ccccccc} X & - & A & - & B & - & Y \\ 1 & & 2 & & 3 & & 4 \end{array}$$

Adjoin 2 as right sister of 4.

The phrases exsisting before this transformation is carried out, and which have "B" as a constitutent, can be thought of as being described by a set of rules as follows:

$$\begin{aligned} a_1 &\longrightarrow B\ \alpha_1 \\ a_2 &\longrightarrow B\ \alpha_2 \\ &\ \vdots \\ a_n &\longrightarrow B\ \alpha_n \end{aligned}$$

Here the a_i are nonterminal symbols and the α_i are strings, possibly null. The grammar which describes the phrases existing after the operation of this transformation must contain, in addition, the following rules:

$$\begin{aligned} a_1 &\longrightarrow B\ \alpha_1\ A^* \\ a_2 &\longrightarrow B\ \alpha_2\ A^* \\ &\ \vdots \\ a_n &\longrightarrow B\ \alpha_n\ A^* \end{aligned}$$

The reverse transformation itself can now be represented by a set of rules as follows:

$$B.1\ \alpha_i.1\ A^*.2 = 2\ B{+}\ 1\ {+}B$$

Notice that the strings referred to by the symbols "X" and "Y" in both of the above examples are unchanged by the transformation and are therefore not mentioned at all in the analysis rules. Experience shows that it is in fact rarely necessary to write separate rules for each α_i. In most cases, a transformation of this kind could be handled in the program with a rule of the following form:

```
B.1 ANY.1 A*.2 = 2 B+ 1 +B
```

This is one of a large number of cases in which it has been found that the analysis rules can be made more permissive than the original grammar suggests without introducing spurious structures and without seriously increasing the amount of time or space used by the program.

While it is possible that transformational analysis can be done in an interesting way with a program of this sort there seems to be little hope of finding an algorithm for writing analysis rules corresponding to a given transformational grammar. The following rule also involves sister adjunction but poses much more serious problems than the previous example:

```
X - A - Y - B - Z

1   2   3   4   5

Adjoin 2 as right sister of 4
```

The problem here is that a variable "Y" intervenes between "A" and "B". On the face of it, the analysis rule corresponding to this transformation would have to be somewhat as follows:

```
*$ANY.1 B.2 *A.3 = 3 1 2
```

And in principal the program could carry out a rule of this kind. However the first symbol on the left-hand side of this rule will match any string whatsoever, so that, if the rule can be applied at all, it can be applied in a prodigious number of ways. But, with real grammars, it usually turns out that quite a lot can be said about the part of the sentence covered by the variable "Y" so that analysis rules can be written which are sufficiently specific to be practiable.

Deletions are notoriously troublesome in grammars of any kind because they can so easily give rise to cycles and undecidable problems. Transformational grammarians require that lexical items should only be deleted from a P-marker if there is some other copy of the same item which remains. This condition insures what they call the recoverability of the transformation. However, it is very important to realize that recoverability, in this sense is a very weak condition. The requirement is that, knowing that an item has been deleted from a certain position in the P-marker, it should be possible to tell what that item was. But there is no requirement that a P-marker should contain evidence that it was derived by means of a deletion transformation or of the places in it where deletions might have taken place.

Deletions are more easy to cope with in certain situations than others. Consider for example the following transformation:

```
X - A - B - A - Y

1   2   3   4   5

Delete 4.
```

The recoverability requirement is satisfied because of the identity of the second and fourth elements in the structural index. The corresponding rule for the program might be as follows:

```
23/22, A.1 B.2 = 1 2 1
```

It is necessary to provide ordering information with a rule of this kind because it would otherwise be capable of operating on its own output and cycling indefinately. But presumably this transformation can be carried out any number of times and the same therefore should be true of the corresponding analysis rule. Once again, experience shows that the grammarian almost invariably knows more about the environment in which a deletion takes place than is stated in the rule, and if this information is used carefully, analysis rules can be written which do not lead to cycles.

In principle the situation is even worse in rules of the following kind:

X - A - Y - A - Z

1 2 3 4 5

Delete 4

Here the third element is a variable which can cover any number of nodes in the P-marker. In analysis we are therefore not only without information about how many times the rule may have been applied but we know nothing about where to insert new copies of the symbol "A", except that they must be to the right of the existing copy.

The other commonly used elementary transformations (substitutions and Chomsky-adjunction) do not present special problems. The main outstanding difficulty comes from the fact that transformational rules are ordered. We have already said that the theory of transformational grammar is in the state of continual change and this is particularly true of the part that concerns the ordering of rules. For this reason we have assumed that the rules are simply ordered in the hope that other possibilities will not be notably more difficult to deal with. We shall also make the assumption that transformational rules are all obligatory.

Consider now the following grammar

<u>Phrase structure</u>

1. $S \rightarrow A\ (D)\ B\ C$

2. $C \rightarrow D\ E$

<u>Transformations</u>

1. A - B - X

   1 2 3

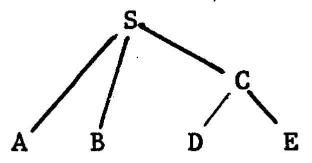
   0 2+1 3

and suppose that the program is required to analyze the string "A D B E". Since, in generation, the list of transformations is read from top to bottom it is reasonable to suppose that in analysis it should be read from bottom to top.

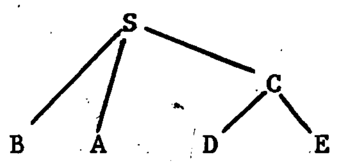
We may take it that the analysis rule corresponding to the second transformation is somewhat as follows:

D.1 B.2 = 2 1

This, together with the two phrase-structure rules, is sufficient to give a complete analysis of the string with this underlying P-marker:



But if this is an underlying P-marker, the second transformational rule could not possibly be used to produce a derived structure from it because the first transformation, which according to our assumption is obligatory, can be applied to it giving the following result:



It is in fact not sufficient to scan the list of transformations from bottom to top because this procedure does not make allowance for the fact that the transformations are obligatory. To regard transformations as optional which were intended to be obligatory is in general to associate spurious base structures to some sentences. The solution for the present grammar is to use the following set of analysis rules:

```
1/0, B D
2/1, D.1 B.2 = 2 1
3/2, A B
4/3, B.1 A.2 = 2 1
     D.1 E.1 = C(1)
     A.1 $D.1 B.1 C.1 = S(1)
```

The first and third rules contain, in effect, the structural indices from the second and first transformations respectively. The first rule says that no string is acceptable as a sentence which contains "B D" as a sub-string because to this it would have been possible to apply transformation 2. The second rule reverses the effect of transformation 2. The third rule, excludes any P-marker existing at this stage with a proper analysis containing "A B" as a sub-string. This is the structural index of transformation 1 which therefore should have been applied to any P-marker containing it. The fourth rule reverses the effect of transformation 1 and the remaining rules are the phrase-structure component of the grammar. Once again it turns out that what may be necessary in theory is only rarely needed in practice. Experience with this program is, so far, very limited but no cases have so far been found in which incorrect analysis have resulted from omitting rules such as those numbered one and three above.

CONCLUSIONS

It requires skill to write rules for analyzing natural sentences with the program described in this paper. A program can only properly be called a transformational parser if it can work directly with the unedited rules of the transformational grammar. But no algorithm is known, nor is it likely that one will shortly be found, which will produce from a transformational grammar a set of corresponding rules of the kind required by this program. It is not difficult to construct a transformational grammar for which no exactly corresponding set of analysis rules can be written. However, other programs have been written which, though they are still in many ways imperfect, can more reasonably be called transformational parsers. What then are the advantages of the present program?

The current version of the program is written in ALGOL and with very little regard for efficiency. But the basic algorithm is inherently a very great deal more efficient than any of its competitors. The various interpretations of an ambiguous sentence, or a sentence which seems likely to be ambiguous in the early stages of analysis, are all worked on simultaneously. At no stage can the program be said to be developing one interpretation of a sentence rather than another. If two interpretations differ only in some small part of the P-marker, then only one complete P-marker is stored with two versions of the ambiguous part. Work done on the unambiguous portion is done only once for both interpretations.

The program, though undoubtably very powerful, seems naive from the point of view of modern linguistic theory. The program embodies very little of what we know or believe to be true about the structure of natural languages. It might well be said that a computer program for analyzing natural languages is only interesting to the extent that it makes a claim about the basic form of those languages. But the program described here is intended as a tool and not as a linguistic hypothesis. There is much to be learned about natural language from ruminating on the form of universal generative grammar and trading counter-example for example. But there is also much to be learned from studying text as it actually occurs. The small amount of work that has so far been done with this program has been sufficient to suggest strongly that a set of rules derived algorithmically from a transformational grammar is unlikely to be the most effective or the most revealing analytic device.