A Parsing Algorithm That Extends Phrases

Daniel Chester

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

It is desirable for a parser to be able to extend a phrase even after it has been combined into a larger syntactic unit. This paper presents an algorithm that does this in two ways, one dealing with "right extension" and the other with "left recursion". A brief comparison with other parsing algorithms shows it to be related to the left-corner parsing algorithm, but it is more flexible in the order that it permits phrases to be combined. It has many of the properties of the sentence analyzers of Marcus and Riesbeck, but is independent of the language theories on which those programs are based.

1. Introduction

To analyze a sentence of a natural language, a computer program recognizes the phrases within the sentence, builds data structures, such as conceptual representations, for each of them and combines those structures into one that corresponds to the entire sentence. The algorithm which recognizes the phrases and invokes the structure-building procedures is the *parsing algorithm* implemented by the program. The parsing algorithm is combined with a set of procedures for deciding between alternative actions and for building the data structures. Since it is organized around phrases, it is primarily concerned with syntax, while the procedures it calls deal with non-syntactic parts of the analysis. When the program is run, there may be a complex interplay between the code segments that handle syntax and those that handle semantics and pragmatics, but the program organization can still be abstracted into a (syntactic) parsing algorithm and a set of procedures that are called to augment that algorithm.

By taking the view that the parsing algorithm recognizes the phrases in a sentence, that is, the components of its surface structure and how they can be decomposed, it suffices to specify the syntax of a natural language, at least approximately, with a context-free phrase structure grammar, the rules of which serve as phrase decomposition rules. Although linguists have developed more elaborate grammars for this purpose, most computer programs for sentence analysis, e.g., Heidorn (1972), Winograd (1972) and Woods (1970), specify the syntax with such a grammar, or something equivalent, and then augment that grammar with procedures and data structures to handle the non-context-free components of the language. The notion of parsing algorithm is therefore restricted in this paper to an algorithm that recognizes phrases in accordance with a context-free phrase structure grammar.

Since the parsing algorithm of a sentence analysis program determines when data structures get combined, it seems reasonable to expect that the actions of the parser should reflect the actions on the data structures. In particular, the combination of phrases into larger phrases can be expected to coincide with the combination of corresponding data structures into larger data structures. This happens naturally when the computer program is such that it calls the procedures for combining data structures at the same time the parsing algorithm indicates that the corresponding phrases should be combined.

1.1 Other Parsing Algorithms

According to one classification of parsing algorithms (Aho and Ullman 1972), most analysis programs are based on algorithms that are either "top-down", "bottom-up" or "left-corner", though according to a recent study by Grishman (1975), the top-down and bottom-up approaches are dominant. The principle of *top-down* parsing is that the rules of the controlling grammar are used to generate a sentence that matches the one being analyzed. A seri-

Copyright 1980 by the Association for Computational Linguistics. Permission to copy without fee all or part of this material is granted provided that the copies are not made for direct commercial advantage and the *Journal* reference and this copyright notice are included on the first page. To copy otherwise, or to republish, requires a fee and/or specific permission.

0362-613X/80/020087-10$01.00

ous problem with this approach, if computed phrases are supposed to correspond to natural phrases in a sentence, is that the parser cannot handle left-branching phrases. But such phrases occur in English, Japanese, Turkish and other natural languages (Chomsky 1965, Kimball 1973, Lyons 1970).

The principle of *bottom-up* parsing is that a sequence of phrases whose types match the right-hand side of a grammar rule is reduced to a phrase of the type on the left-hand side of the rule. None of the matching is done until all the phrases are present; this can be ensured by matching the phrase types in the right-to-left order shown in the grammar rule. The difficulty with this approach is that the analysis of the first part of a sentence has no influence on the analysis of latter parts until the results of the analyses are finally combined. Efforts to overcome this difficulty lead naturally to the third approach, left-corner parsing.

Left-corner parsing, like bottom-up parsing, reduces phrases whose types match the right-hand side of a grammar rule to a phrase of the type on the left-hand side of the rule; the difference is that the types listed in the right-hand side of the rule are matched from left to right for left-corner parsing instead of from right to left. This technique gets its name from the fact that the first phrase found corresponds to the left-most symbol of the right-hand side of the grammar rule, and this symbol has been called the *left corner* of the rule. (When a grammar rule is drawn graphically with its left-hand side as the parent node and the symbols of the right-hand side as the daughters, forming a triangle, the left-most symbol is the left corner of the triangle.) Once the left-corner phrase has been found, the grammar rule can be used to predict what kind of phrase will come next. This is how the analysis of the first part of a sentence can influence the analysis of later parts.

Most programs based on *augmented transition networks* employ a top-down parser to which registers and structure building routines have been added, e.g., Kaplan (1972) and Wanner and Maratsos (1978). It is important to note, however, that the concept of augmented transition networks is a particular way to *represent* linguistic knowledge; it does not *require* that the program using the networks operate in top-down fashion. In an early paper by Woods (1970), alternative algorithms that can be used with augmented transition networks are discussed, including the bottom-up and Earley algorithms.

The procedure-based programs of Winograd (1972) and Novak (1976) are basically top-down parsers, too. The NLP program of Heidorn (1972) employs an augmented phrase structure grammar to combine phrases in a basically bottom-up fashion. Likewise, PARRY, the program written by Colby (Parkison, Colby and Faught 1977), uses a kind of bottom-up method to drive a computer model of a paranoid.

1.2 A New Parsing Algorithm

This paper presents a parsing algorithm that allows data structures to be combined as soon as possible. The algorithm permits a structure A to be combined with a structure B to form a structure C, and then to enlarge B to form a new structure B'. This new structure is to be formed in such a way that C is now composed of A and B' instead of A and B. The algorithm permits these actions on data structures because it permits similar actions on phrases, namely, phrases are combined with other phrases and afterward are extended to encompass more words in the sentence being analyzed. This behavior of combining phrases before all of their components have been found is called *closure* by Kimball (1973). It is desirable because it permits the corresponding data structures to be combined and to influence the construction of other data structures sooner than they otherwise could.

In the next section of this paper the desired behavior for combining phrases is discussed in more detail to show the two kinds of actions that are required. Then the algorithm itself is explained and its operation illustrated by examples, with some details of an experimental implementation being given, also. Finally, this algorithm is compared to those used in the sentence analysis programs of Marcus and Riesbeck, and some concluding remarks are made.

2. Phrase Extension

A parser that extends phrases combines phrases before it has found all of their components. When parsing the sentence

(1) This is the cat that caught the rat
that stole the cheese.

it combines the first four words into the phrase "this is the cat", in which "the cat" is a noun phrase, then extends that noun phrase to "the cat that caught the rat", in which "the rat" is a noun phrase, and finally extends *that* noun phrase to "the rat that stole the cheese." This is apparently how people parse that sentence, because, as Chomsky (1965) noted, it is natural to break up the sentence into the fragments

this is the cat
that caught the rat
that stole the cheese

(by adding intonation, pauses or commas) rather than at the boundaries of the major noun phrases:

this is
the cat that caught
the rat that stole
the cheese

Likewise, when the parser parses the sentence

(2) The rat stole the cheese in the pantry by the bread.

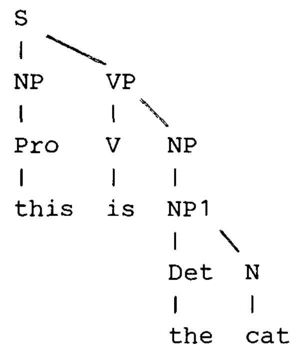
it forms the phrase "the rat stole the cheese", then extends the noun phrase "the cheese" to "the cheese in the pantry" and extends *that* phrase to "the cheese in the pantry by the bread".

These two examples show the two kinds of actions needed by a parser in order to exhibit the behavior called closure. Each of these actions will now be described in more detail, using the following grammar, G1:

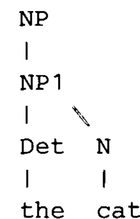
```
      S -> NP VP
     NP -> Pro
     NP -> NP1
     NP -> NP1 RelPro VP
     VP -> V NP
    Pro -> this
    NP1 -> Det N
    NP1 -> NP1 PP
 RelPro -> that
      V -> is
      V -> caught
      V -> stole
    Det -> the
      N -> cat
      N -> rat
      N -> cheese
      N -> pantry
      N -> bread
     PP -> Prep NP
   Prep -> in
   Prep -> by
```

2.1 Right Extension

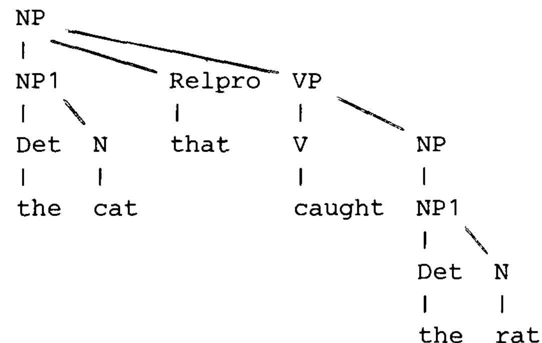
The first kind of action needed by the parser is to add more components to a phrase according to a rule whose right-hand side extends the right-hand side of another rule. This is illustrated by sentence 1. With grammar G1, the phrase "this is the cat" has the phrase structure



In order to extend the NP phrase "the cat", the substructure



must be changed to the substructure



The parser, in effect, must extend the NP phrase according to the rule NP --> NP1 RelPro VP, whose right-hand side extends the right-hand side of the rule NP --> NP1.

There is a simple way to indicate in a grammar when a phrase can be extended in this way: when two rules have the same left-hand side, and the right-hand side of the first rule is an initial segment of the right-hand side of the second rule, a special mark can be placed after that initial segment in the second rule and then the first rule can be discarded. The special mark can be interpreted as saying that the rest of the right-hand side of the rule is optional. Using * as the special mark, the rule NP --> NP1 * RelPro VP can replace the rules NP --> NP1 and NP --> NP1 RelPro VP in the grammar G1.

Our algorithm therefore requires a *modified grammar*. For the general case, a grammar is modified by execution of the following step:

1) For each rule of the form X --> Y1 ... Yk, whenever there is another rule of the form X --> Y1 ... Yk Yk+1 ... Yn, replace this other rule by X --> Y1 ... Yk * Yk+1 ... Yn, provided Yk+1 is not *.

When no more rules can be produced by the above step, the following step is executed:

2) Delete each rule of the form X --> Y1 ... Yk for which there is another rule of the form X --> Y1 ... Yk Yk+1 ... Yn.

The mark * tells the parser to combine the phrases that correspond to the symbols preceding * as if the rule ended there. After doing that, the parser tries to extend the combination by looking for phrases to correspond to the symbols following *.

2.2 Left Recursion

The second kind of action needed by the parser is to extend a phrase according to a *left-recursive* rule. A left-recursive rule is a rule of the form X --> X Y1 ... Yn. The action based on a left-recursive rule is illustrated by sentence 2 above. The phrase "the rat stole the cheese" has the phrase structure

```
S
|  \
NP       VP
|        |  \
NP1      V      NP
|  \     |      |
Det  N   stole  NP1
|    |          |  \
the  rat        Det  N
                |    |
                the  cheese
```

In order to extend the phrase "the cheese", the substructure

```
NP1
|  \
Det  N
|    |
the  cheese
```

must be changed to the substructure

```
NP1
|   \
NP1          PP
|  \         |  \
Det  N       Prep  NP
|    |       |     |
the  cheese  in    NP1
                   |  \
                   Det  N
                   |    |
                   the  pantry
```

The parser, in effect, must extend the NP1 phrase according to the left-recursive rule NP1 --> NP1 PP.

In the general case, after finding a phrase and combining it with others, the parser must look for more phrases to combine with it according to some left-recursive rule in the grammar. Left-recursive rules, therefore, play a special role in this algorithm.

It might be thought that left-recursive rules are the only mechanism needed to extend phrases. This is not true because phrases extended in this way have a different property from those extended by right extension. In particular, left-recursive rules can be applied several times to extend the same phrase, which is not always the desired behavior for a given language phenomenon. For instance, any number of PP phrases can follow an NP1 phrase *without* connecting words or punctuation because of the left-recursive rule NP1 --> NP1 PP. If, on the other hand, NP --> NP RelPro VP were used instead of NP --> NP1 RelPro VP, any number of relative clauses (RelPro and VP phrase pairs) could follow an NP phrase, which is generally ungrammatical in English unless the clauses are separated by commas or connective words like "and" and "or".

3. Details of the Parsing Algorithm

The parsing algorithm that provides for both right extension and left recursion will now be described. It works with structures called phrase status elements to keep track of the status of phrases during the sentence analysis. In this section, after these elements are defined, the algorithm is presented as a set of operations to be performed on a list of phrase status elements. Then the algorithm is applied to sentences 1 and 2 to show how it works, and some refinements that were added in an experimental implementation are discussed.

3.1 Phrase Status Elements

In order to combine and extend phrases properly, a parser must keep track of the status of each phrase; in particular, it must note what kind of phrase will be formed, what component phrases are still needed to complete the phrase, and whether the phrase is a new one or an extension. This information can be represented by a *phrase status element.* A phrase status element is a triple of the form (Yk ... Yn,X,F) where, for some symbols Y1, ..., Yk-1, there is a grammar rule of the form X --> Y1 ... Yk-1 Yk ... Yn, and F is a flag that has one of the values n, e, or p, which stand for "new", "extendible" and "progressing", respectively. Intuitively, this triple means that the phrase in question is an X phrase and that it will be completed when phrases of types Yk through Yn are found. If F = n, the phrase will be a new phrase. If F = e, the phrase is ready for extension into a longer X phrase, but none of the extending phrase components have been found yet. If F = p, however, some of the extending phrase components *have* been found already and the rest must be found to complete the phrase. The status of being ready for extension has to be distinguished from the status of having the extension in progress, because it is during the ready status that the decision whether to try to extend the phrase is made.

3.2 The Algorithm

The parsing algorithm for extending phrases is embodied in a set of operations for manipulating a list of phrase status elements, called the *element list,* according to a given modified grammar and a given sentence to be analyzed. Beginning with an empty element list, the procedure is applied by performing the operations repeatedly until the list contains only the element (,S,n), where S is the grammar's start symbol, and all of the words in the given sentence have been processed.

The following are the six operations which are applied to the element list:

1. Replace an element of the form (* Y1 ... Yn,X,F) with the pair (Y1 ... Yn,X,e) (,X,F).
2. Replace an element of the form (,X,p) with the element (Y1 ... Yn,X,e) if there is a left-recursive rule of the form X --> X Y1 ... Yn in the grammar; if there is no such rule, delete the element.
3. Replace adjacent phrase status elements of the form (,X,n) (X Y1 ... Yn,Z,F) with the pair (,X,p) (Y1 ... Yn,Z,F'). If the flag F = e, then F' = p; otherwise, F' = F.
4. Replace an element of the form (,X,n) with the pair (,X,p) (Y1 ... Yn,Z,n) if there is a grammar rule of the form Z --> X Y1 ... Yn in the grammar, provided the rule is not left-recursive.
5. Get the next word W from the sentence and add the element (,W,n) to the front (left) end of the element list.
6. Delete an element of the form (Y1 ... Yn,X,e).

These operations are applied one at a time, in arbitrary order. If more than one operation can cause a change in the element list at any given time, then one of them is selected for actual application. The manner of selection is not specified here because that is a function of the data structures and procedures that would have to be added to incorporate the algorithm into a complete sentence analysis program.

These operations have a fairly simple purpose: the major goal is to find new phrases, which are represented by phrase status elements of the form (,X,n). Initially, by application of operation 5, a word of the sentence to be analyzed is made into a new phrase. When a new phrase is found, whether by operation 5 or some other operation, there are two ways that can be used to find a larger phrase. One way is to attempt to find a new phrase that begins with the just-found phrase; this is the purpose of operation 4. Once an X phrase is found, the element (,X,n) is replaced by (,X,p) (Y1 ... Yn,Z,n) for some grammar rule of the form Z --> X Y1 ... Yn and some Z different from X. The element (Y1 ... Yn,Z,n) represents an attempt to find a Z phrase, of which the first component, the X phrase, has been found.

The second way that can be used to make a larger phrase is to make the X phrase a component of a phrase that has already been started. In operation 3, the element (,X,n) represents a new X phrase that can be used in this way. An immediately preceding phrase has been made part of some unfinished Z phrase, represented by the element (X Y1 ... Yn,Z,F). Since the symbol X is the first symbol in the first part of this element, the Z phrase can contain an X phrase at this point in the sentence, so the X phrase is put in the Z phrase, and the result of this action is indicated by the new elements (,X,p) (Y1 ... Yn,Z,F').

In both operations 3 and 4, the element (,X,n) itself is replaced by (,X,p). The flag value p indicates, in this case, that the X phrase has already been added to some larger phrase. Operation 2 tries to extend the X phrase by creating the element (Y1 ... Yn,X,e) for some left-recursive grammar rule X --> X Y1 ... Yn, indicating that the X

phrase can be extended if it is followed by phrases of types Y1, ..., Yn, in that order. If a Y1 phrase comes next in the sentence, the extension begins by an application of operation 5, which changes the flag from e to p to indicate that the extension is in progress. If there is no Y1 phrase, however, the X phrase cannot be extended, so the element is deleted by operation 6, allowing the parser to go on to the next phrase.

In the event a phrase status element has the form (* Y1 ... Yn,X,F), the X phrase can be considered completed. It can also be extended, however, if it is followed by phrases of types Y1 through Yn. Operation 1 creates the new element (,X,F) to indicate the completion of the X phrase, and the new element (Y1 ... Yn,X,e) to indicate its possible extension. Again, if extension turns out not to be possible, the element can be deleted by operation 6 so that parsing can continue.

3.3 Examples

As examples of how the algorithm works, consider how it applies to sentences 1 and 2. Grammar G1 must be modified first, but this consists only of substituting the rule NP --> NP1 * RelPro VP for the two NP rules in the original grammar, as described earlier. Starting with an empty element list, the sentence "this is the cat that caught the rat that stole the cheese" is processed as shown in Table 1. By operation 5, the word "this" is removed from the sentence and the element (,this,n) is added to the list. By operation 4, this element is replaced by (,this,p) (,Pro,n), which is shortened to (,Pro,n) by operation 2. By two applications of operations 4 and 2, the element list becomes (,NP,n), then (VP,S,n). The element (,is,n) is now added to the front of the list. This element is changed, by operations 4 and 2 again, to (,V,n) and then to (NP,VP,n). The words "the" and "cat" are processed similarly to produce the element list (,N,n) (N,NP1,n) (NP,VP,n) (VP,S,n) at step 20.

At this point, operation 3 is applied to combine (,N,n) with (N,NP1,n), yielding (,N,p) (,NP1,n) in their place. By operation 2, element (,N,p) is removed. The first element of the list is now (,NP1,n), which by operation 4 is changed to (,NP1,p) (* RelPro VP,NP,n). When operation 2 is applied this time, because there is a left-recursive grammar rule for NP1 phrases, element (,NP1,p) is replaced by (PP,NP1,e). Operation 1 is applied to the next element to eliminate the special mark * that appears in it, changing the element list to (PP,NP1,e) (RelPro VP,NP,e) (,NP,n) (NP,VP,n) (VP,S,n) at step 25.

At this point, the element (,NP,n) represents the NP phrase "the cat". Operation 3 is applied to it and the next element, and then operation 2 to the result, reducing those two elements to (,VP,n). By operations 3 and 2 again, this element is combined with (VP,S,n) to produce (,S,n), which at this point represents the phrase "this is the cat." If this phrase were the whole sentence, operation 6 could be applied, reducing the element list to (,S,n) and the sentence would be successfully parsed. There are more words in the sentence, however, so other operations are tried.

The next word, "that", is processed by operations 5, 4 and 2 to add (,RelPro,n) to the front of the list. Since the grammar does not allow a PP phrase to begin with the word "that", operation 6 is applied to eliminate the element (PP,NP1,e), which represents the possibility of an extension of an NP1 phrase by a PP phrase. The next element, (RelPro VP,NP,e), represents the possibility of an extension of an NP phrase when it is followed by a RelPro phrase, however, so operations 3 and 2 are applied, changing the element list to (VP,NP,p) (,S,n). Note that the flag value e has changed to p; this means that a VP phrase *must* be found now to complete the NP phrase extension or this sequence of operations will fail.

By continuing in this fashion, the sentence is parsed. Since no new details of how the algorithm works would be illustrated by continuing the narration, the continuation is omitted.

The sentence "the rat stole the cheese in the pantry by the bread" is parsed in a similar fashion. The only detail that is significantly different from the previous sentence is that after the element (RelPro VP,NP,e) is deleted by operation 6, instead of (PP,NP1,e), a new situation occurs, in which a phrase can attach to one of *several* phrases waiting to be extended. The situation occurs after the sentence corresponding to the phrase "the rat stole the cheese" is represented by the element (,S,n) when it first appears on the element list. When the PP phrase "in the pantry" is found, the element (PP,NP1,e) changes to (,NP1,p), indicating that the NP1 phrasé "the cheese" has been extended to "the cheese in the pantry". By operation 2, the element (,NP1,p) is changed to (PP,NP1,e) so that the NP1 phrase can be extended again. But "the pantry" is an NP1 phrase also, which means that an element of the form (PP,NP1,e) has been created to extend *it* as well. Thus, when the next PP phrase, "by the bread" is found, it can attach to either of the earlier NP1 phrases. The parser does not decide which attachment to make, as that depends on non-syntax related properties of the data structures that would be associated with the phrases in a complete sentence analyzer. In this example the PP phrase can be attached to the NP1 phrase "the cheese", which

STEP	ELEMENT LIST	OPERATION
1	(,this,n)	5
2	(,this,p)(,Pro,n)	4
3	(,Pro,n)	2
4	(,Pro,p)(,NP,n)	4
5	(,NP,n)	2
6	(,NP,p)(VP,S,n)	4
7	(VP,S,n)	2
8	(,is,n)(VP,S,n)	5
9	(,is,p)(,V,n)(VP,S,n)	4
10	(,V,n)(VP,S,n)	2
11	(,V,p)(NP,VP,n)(VP,S,n)	4
12	(NP,VP,n)(VP,S,n)	2
13	(,the,n)(NP,VP,n)(VP,S,n)	4
14	(,the,p)(,Det,n)(NP,VP,n)(VP,S,n)	2
15	(,Det,n)(NP,VP,n)(VP,S,n)	4
16	(,Det,p)(N,NP1,n)(NP,VP,n)(VP,S,n)	4
17	(N,NP1,n)(NP,VP,n)(VP,S,n)	2
18	(,cat,n)(N,NP1,n)(NP,VP,n)(VP,S,n)	5
19	(,cat,p)(,N,n)(N,NP1,n)(NP,VP,n)(VP,S,n)	4
20	(,N,n)(N,NP1,n)(NP,VP,n)(VP,S,n)	2
21	(,N,p)(,NP1,n)(NP,VP,n)(VP,S,n)	3
22	(,NP1,n)(NP,VP,n)(VP,S,n)	2
23	(,NP1,p)(* RelPro VP,NP,n)(NP,VP,n)(VP,S,n)	4
24	(PP,NP1,e)(* RelPro VP,NP,n)(NP,VP,n)(VP,S,n)	2
25	(PP,NP1,e)(RelPro VP,NP,e)(,NP,n)(NP,VP,n)(VP,S,n)	1
26	(PP,NP1,e)(RelPro VP,NP,e)(,NP,p)(,VP,n)(VP,S,n)	3
27	(PP,NP1,e)(RelPro VP,NP,e)(,VP,n)(VP,S,n)	2
28	(PP,NP1,e)(RelPro VP,NP,e)(,VP,p)(,S,n)	3
29	(PP,NP1,e)(RelPro VP,NP,e)(,S,n)	2
30	(,that,n)(PP,NP1,e)(RelPro VP,NP,e)(,S,n)	5
31	(,that,p)(,RelPro,n)(PP,NP1,e)(RelPro VP,NP,e)(,S,n)	4
32	(,RelPro,n)(PP,NP1,e)(RelPro VP,NP,e)(,S,n)	2
33	(,RelPro,n)(RelPro VP,NP,e)(,S,n)	6
34	(,RelPro,p)(VP,NP,p)(,S,n)	3
35	(VP,NP,p)(,S,n)	2
	etc.	

Table 1. Trace of parsing algorithm on sentence 1.

means that the intervening element (PP,NP1,e) attempting to expand the NP1 phrase "the pantry" has to be deleted.

3.4 Constraints on the Operations

The algorithms for top-down, bottom-up and left-corner parsing are usually presented so that all operations are performed on the *top* of a stack that corresponds to our element list. We have not constrained our algorithm in this way because to do so would prevent the desired closure behavior. In particular, in the sentence "this is the cat that caught the rat that stole the cheese," the NP phrase "the cat" would not combine into the phrase "this is the cat" until the rest of the sentence was analyzed if such a constraint were enforced. This is because operation 1 would create an element for extending the NP phrase that would have to be disposed of first before the element (,NP,n), created also by that operation, could combine with anything else. Thus, constraining the operations to apply only to the front end of the element list would nullify the algorithm's purpose.

Our algorithm can be viewed as a modification of the left-corner parser. In fact, if a grammar is *not* modified before use with our algorithm, and if operation 4 is *not* restricted to non-left-recursive rules, and if operation 2 is modified to delete *all* elements of the form (,X,p), then our algorithm would actually *be* a left-corner parser.

3.5 Experimental Program

The algorithm has been presented here in a simple form to make its exposition easier and its operating principles clearer. When it was tried out in an experimental program, several techniques were used to make it work more efficiently. For example, operations 1 and 2 were combined with the other operations so that they were, in effect, applied as soon as possible. Operation 3 was also applied as soon as possible. The other operations cannot be given a definite order for their application; that must be determined by the non-syntactic procedures that are added to the parser.

Another technique increased efficiency by applying operation 4 only when the result is consistent with the grammar. Suppose grammar G1 contained the rule Det --> that as well as the rule RelPro --> that. When the word "that" in the sentence "this is the cat that caught the rat that stole the cheese" is processed, the element list contains the triple (,that,n) (PP,NP1,e) (RelPro VP,NP,e) at one point. The grammar permits operation 4 to be applied to (,that,n) to produce either (,Det,n) or (,RelPro,n), but only the latter can lead to a successful parse because the grammar does not allow for either a PP phrase or a RelPro phrase to begin with a Det phrase. The technique for guiding operation 4 so that it produces only useful elements consists of computing beforehand all the phrase types that can begin each phrase. Then the following operation is used in place of operation 4:

4'a. If an element of the form (,X,n) is at the (right) end of the list, replace it with the pair (,X,p) (Y1 ... Yn,Z,n) when there is a grammar rule of the form Z --> X Y1 ... Yn in the grammar, provided the rule is not left-recursive.

4'b. If an element of the form (,X,n) is followed by an element of the form (U1 ... Um,V,F), replace (,X,n) with the pair (,X,p) (Y1 ... Yn,Z,n) when there is a grammar rule of the form Z --> X Y1 ... Yn in the grammar, provided the rule is not left-recursive, and a Z phrase can begin a U1 phrase or Z = U1.

It is sufficient to consider only the first element after an element of the form (,X,n) because if operation 4'b cannot be applied, either that first element can be deleted by operation 6 or the parse is going to fail anyway. Thus, in our example above, operation 6 can be used to delete the element (PP,NP1,e) so that operation 4'b can be applied to (,that,n) (RelPro VP,NP,e). This technique is essentially the same as the selective filter technique described by Griffiths and Petrick (1965) for left-corner parsing (the SBT algorithm, in their terminology).

Another technique increased efficiency further by postponing the decision about which of several grammar rules to apply via operations 3 or 4' for as long as possible. The grammar rules were stored in Lisp list structures so that rules having the same left-hand side and a common initial segment in their right-hand side shared a common list structure. for example, if the grammar consists of the rules

```
X -> Y Z U
X -> Y Z V
W -> Y Z U
```

these rules are stored as the list structure

```
( (X (Z (U)
        (V) ))
  (W (Z (U) )) )
```

which is stored on the property list for Y. The common initial segment shared by the first two rules is represented by the same path to the atom Z in the list structure. The component (X (Z (U) (V))) in this list structure means that a Y phrase can be followed by a Z phrase and then either a U phrase or a V phrase to make an X phrase. When a Y phrase is found, and it is decided to try to find an X phrase, this component makes it possible to look for a Z phrase, but it postpones the decision as to whether the Z phrase should be followed by a U phrase or a V phrase until after the Z phrase has been found. The left-hand sides (X and W) of the rules are listed first to facilitate operation 4'b. This technique is similar to a technique used by Irons (1961) and described by Griffiths and Petrick (1965).

4. Related Work

There are two sentence analysis programs with parsing algorithms that resemble ours in many ways, though theirs have been described in terms that are intimately tied to the particular semantic and syntactic representations used by those programs. The programs are PARSIFAL, by Marcus (1976, 1978a,b) and the analyser of Riesbeck (1975a,b).

4.1 PARSIFAL (Marcus)

The basic structural unit of PARSIFAL is the *node*, which corresponds approximately to our phrase status element. Nodes are kept in two data structures, a pushdown stack called the *active node stack*, and a three-place *constituent buffer*. (The buffer actually has five places in it, but the procedures that use it work with only three places at a time.) The grammar rules are encoded into *rule packets*. Since the organization of these packets has to do with the efficient selection of appropriate grammar rules and the invocation of procedures for adding structural details to nodes, the procedures we want to ignore while looking at the parsing algorithm of PARSIFAL, we will ignore the rule packets in this comparison. The essential fact about rule packets is that they examine only the top node of the stack, the S or NP node nearest the top of the stack, and up to three nodes in the buffer.

The basic operations that can be performed on these structures include attaching the first node in the buffer to the top node of the stack, which corresponds to operation 3 in our algorithm, creating a new node that holds one or more of the nodes in the buffer, which corresponds to operation 4, and reactivating a node (pushing it onto the stack) that has already been attached to another node so that more nodes can be attached to it, which corresponds to the phrase extending operations 1,2, and 6. PARSIFAL has one operation that is *not* similar to the operations of our algorithm, which is that it can create nodes for dummy NP phrases called *traces*. These nodes are intended to provide a way to account for phenomena that would otherwise require transformational grammar rules to explain them. Our algorithm does not allow such an operation; if such an operation should prove to be necessary, however, it would not be hard to add, or its effect could be produced by the procedures called.

One of the benefits of having a buffer in PARSIFAL is that the buffer allows for a kind of look-ahead based on *phrases* instead of just *words*. Thus the decision about what grammar rule to apply to the first node in the buffer can be based on the *phrases* that follow a certain point in the sentence under analysis instead of just the *words*. The system can look further ahead this way and still keep a strict limit on the amount of look-ahead available. We can get a similar effect with our algorithm if we restrict the application of its operations to the first four or five phrase status elements in the element list. In a sense, the first five elements of the list correspond to the buffer in PARSIFAL and the rest of the list corresponds to the stack. In fact, in a recent modification of PARSIFAL (Shipman and Marcus 1979) the buffer and stack *were* combined into a single data structure closely resembling our element list.

4.2 Riesbeck's Analyzer

The basic structural unit of Riesbeck's analyzer is the *Conceptual Dependency structure* as developed by Schank (1973,1975). A Conceptual Dependency structure is intended to represent the meaning of a word, phrase or sentence. The details of what a Conceptual Dependency structure is will not be discussed here.

The monitor in Riesbeck's analyzer has no list or stack on which operations are performed; instead, it has some global variables that serve the same purpose. Only a few of these variables concern us here. The variable WORD holds the current word being looked at and can be thought of as the front element of our element list. The variable SENSE holds the sense of WORD or of the noun phrase of which WORD is the head. It is like the second element in our list. The equivalent to the third element in our list is the variable REQUESTS, which holds a list of pattern-action rules. There are some other variables (such as ART-INT) that on occasion serve as the fourth element of the list.

Unlike the controllers in many other analysis programs, Riesbeck's monitor is not driven explicitly by a grammar. Instead, the syntactic information it uses is buried in the pattern-action rules attached to each word and word sense within his program's lexicon. Take, for example, the common sense of the verb "give": one pattern-action rule says that if the verb is followed by a noun phrase denoting a person, the sense of that phrase is put in the recipient case slot of the verb. Another pattern-action rule says that if a following noun phrase denotes an object, the sense of the phrase is put in the object case slot of the verb. These pattern-action rules correspond to having grammar rules of the form VP --> give, and VP --> VP NP, where the pattern-action rules describe two different ways that a VP phrase and an NP phrase can combine into a VP phrase. There is a third pattern-action rule that changes the current sense of the word "to" in case it is encountered later in the sentence, but that is one of the actions that occurs below the syntactic level.

Noun phrases are treated by Riesbeck's monitor in a special way. Unmodified nouns are considered to be noun phrases directly, but phrases beginning with an article or adjective are handled by a special subroutine that collects the following adjectives and nouns *before* building the corresponding Conceptual Dependency structure. Once the whole noun phrase is found, the monitor examines the REQUESTS list to see if there are any pattern-action rules that can use the noun phrase. If so, the associated action is taken and the rule is marked so that it will not be used twice. The monitor is started with a pattern-action rule on the REQUESTS list that puts a beginning noun phrase in the subject case slot of whatever verb that follows. (There are provisions

to reassign structures to different slots if later words of the sentence require it.)

It can be seen that Riesbeck's analysis program works essentially by putting noun phrases in the case slots of verbs as the phrases are encountered in the sentence under analysis. In a syntactic sense, it builds a phrase of type *sentence* (first noun phrase plus verb) and then extends that phrase as far as possible, much as our algorithm does using left-recursive grammar rules. Prepositions and connectives between simple sentences complicate the process somewhat, but the process is still similar to ours at the highest level of program control.

5. Concluding Remarks

Since our parsing algorithm deals only with syntax, it is not complete in itself, but can be combined with a variety of conceptual representations to make a sentence analyzer. Whenever an operation that combines phrases is performed, procedures to combine data structures can be called as well. When there is a *choice* of operations to be performed, procedures to make the choice by examining the data structures involved can be called, too. Because our algorithm combines phrases sooner than others, there is greater opportunity for the data structures to influence the progress of the parsing process. This makes the resulting sentence analyzer behave not only in a more human-like way (the closure property), but also in a more efficient way because it is less likely to have to back up.

Although the programs of Marcus and Riesbeck share many of these same properties, the syntactic processing aspects of those programs are not clearly separated from the particular conceptual representations on which they are based. We believe that the parsing algorithm presented here captures many of the important properties of those programs so that they may be applied to conceptual representations based on other theories of natural language.

References

Aho, A. and Ullman J., 1972. *The Theory of Parsing, Translation and Compiling, Vol. 1,* Prentice-Hall Inc., New Jersey.

Chomsky, N. 1965. *Aspects of the Theory of Syntax,* MIT Press, Cambridge, Mass.

Griffiths, T. and Petrick S., 1965. On the relative efficiencies of context-free grammar recognizers. *CACM* 8, May, 289-300.

Grishman, R., 1975. A Survey of Syntactic Analysis Procedures for Natural Language. Courant Computer Science Report #8, Courant Institute of Mathematical Sciences, New York University, August. Also appears on *AJCL* Microfiche 47, *1976.*

Heidorn, G., 1972. Natural language inputs to a simulation programming system. Report No. NPS-55HD72101A, Naval Postgraduate School, Monterey, Calif., October.

Irons, E., 1961. A syntax directed compiler for ALGOL 60. *CACM* 4, January, 51-55.

Kaplan, R., 1972. Augmented transition networks as psychological models of sentence comprehension. *Artificial Intelligence* 3, 77-100.

Kimball, J., 1973. Seven principles of surface structure parsing in natural language. *Cognition* 2, 15-47.

Lyons, J., 1970. *Chomsky.* Fontana/Collins, London.

Marcus, M., 1976. A design for a parser for English. *ACM '76* Proceedings, Houston, Texas, Oct 20-22, 62-68.

Marcus, M., 1978a. Capturing linguistic generalizations in a parser for English. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence,* University of Toronto, Toronto, Ontario, July 19-21, 64-73.

Marcus, M., 1978b. A computational account of some constraints on language. *Theoretical Issues in Natural Language Processing-2,* D. Waltz, general chairman, University of Illinois at Urbana-Champaign, July 25-27, 236-246.

Novak, G., 1976. Computer understanding of physics problems stated in natural language. *AJCL* microfiche 53.

Parkison, R., Colby, K. and Faught, W., 1977. Conversational language comprehension using integrated pattern-matching and parsing. *Artificial Intelligence* 9, October, 111-134.

Riesbeck, C., 1975a. Computational understanding. *Theoretical Issues in Natural Language Processing,* R. Schank and B. Nash-Webber, eds., Cambridge, Mass., June 10-13, 11-16.

Riesbeck, C., 1975b. Conceptual analysis. In Schank (1975), 83-156.

Schank, R., 1973. Identification of conceptualizations underlying natural language. In Schank R. and Colby, K., eds., *Computer Models of Thought and Language,* W. H. Freeman and Company, San Francisco.

Schank, R., 1975. *Conceptual Information Processing.* American Elsevier, New York.

Shipman, D., and Marcus, M., 1979. Towards minimal data structures for deterministic parsing. *IJCAI-79,* Tokyo, Aug 20-23, 815-817.

Wanner, E., and Maratsos, M., 1978. *Linguistic Theory and Psychological Reality,* M. Halle, J. Bresnan, G. Miller, eds., MIT Press, Cambridge, Mass., 119-161.

Winograd, T., 1972. *Understanding Natural Language.* Academic Press, New York.

Woods, W., 1970. Transition network grammars for natural language analysis. *CACM* 13, October, 591-606.

Daniel Chester is an assistant professor in the Department of Computer Sciences of the University of Texas at Austin. He received his Ph.D. in mathematics from the University of California at Berkeley in 1973.