# A PARSER WITH SOMETHING FOR EVERYONE*

Eugene Charniak
Dept. of Computer Science, Brown University

**ABSTRACT**

We present a syntactic parser, Paragram, which tries to accommodate three goals. First it will parse, in a natural way, ungrammatical sentences. Secondly, it aspires to "capture the relevant generalizations", as in transformational grammar, and thus its rules are in virtual one-to-one correspondence with typical transformational rules. Finally, it promises to be reasonably efficient, especially given certain limited parallel processing capabilities.

## 1. Introduction

Syntactic parsing in Artificial Intelligence (AI) has always had its share of controversies. Many in AI have seen in this work "much wasted effort"[4] and suggested that "the heavily hierarchical syntax analyses of yesteryear may not be necessary" [6]. At the same time, syntactic parsers have been attacked by those in linguistics as "devoid of any principles which could serve as even a basis for a serious scientific theory of human linguistic behavior" [2]. And, while psychologists have been kinder, any psychologist must be uncomfortable with theories which, if taken literally, would predict that people cannot understand ungrammatical sentences — a prediction which are false.

In this paper we will propose a parser, named "Paragram", which goes some way to answering this criticism. In particular:

1) The parser is "semi-grammatical" in the sense that it takes a standard "correct" grammar of English and applies it so long as it can, but will accept sentences which do not fit the grammar, while noting in which ways the sentences are deviant. Thus it will parse (1) while still using grammatical rules for subject/verb agreement to distinguish (2) from (3).

   (1) *The boys is dying.[1]
   (2) The fish is dying.
   (3) The fish are dying.

2) The rules of the parser are intended to capture the relevant generalizations about language in much the same way as a good transformational grammar. Paragram's rules are nearly in one-to-one correspondence with those proposed in some versions of transformational grammar.[2] Despite the fact that augmented transition network (ATN) parsers are based upon transformational grammar, when examined closely typical ATN grammars [7] seem to be far from the above ideal.

3) The parser is reasonably efficient, (0.3 seconds/word for a group of test sentences) and would be very efficient if implemented on a machine with limited parallelism, so that the rules of the grammar all test the input in parallel, but only one is actually applied (estimated .04 seconds/word). Efficiency aspects will not be discussed further in this paper.

## 2. Handling Ungrammatical Sentences

### 2.1. Why We Need a Deterministic Parser

Paragram is based upon Marcus's parser "Parsifal"[3]. We will explain Parsifal shortly, but first let us explain why we chose it as a starting point.

Probably the best known parser in AI today is Woods' ATN parser [7]. However it would not be possible to base a Paragram type parser upon the ATN parsing model. To see why this is so, we need only consider that when Paragram finds an ungrammatical situation, it must simply recognize it as such, and continue as best it can. ATN's simply do not work this way. When an ATN finds an ungrammatical situation it takes it as evidence that it made an incorrect decision earlier in the sentence, and hence backs up to find the correct path. So, consider

   (4) Jack sold the ball.
   (5) Jack sold Sue the ball.

Suppose that an ATN parser initially decides to parse "Sue" in (5) as a direct object, just like "the ball" in (4). When it gets to the second noun phrase in (5), "the ball", it has no way to handle it, and hence it backs up and tries making "Sue" into a dative which has been moved before the direct object. But suppose we had the ungrammatical sentence,

   *Jack sold Sue ball.

Here the ATN would back up as well, but to no avail, since there is no way to get a grammatical sentence out of this.[3]

In a deterministic parser (one which does not back up) the parser may assume that it has parsed everything correctly up to the point where it runs into trouble. Thus Parsifal knows where the trouble lies. It is this property which makes it an ideal starting point for Paragram.

### 2.2. Parsifal

Parsifal has two basic data structures, a stack and a buffer. The stack contains the sentence constituents on which it is still working. If a constituent is complete, then it must reside in one of two places: first, it may simply hang off some larger constituent. So at the end of a sentence there is only one item on the stack, the topmost **s** node, and everything else hangs off it. Second, Parsifal may have a complete constituent, but not know yet where it should go. Such constituents are put in the *buffer* which is a storage area of limited size. An obvious example would be an individual word (which is clearly complete). A less obvious example would be a noun phrase which, while complete, might be attached at any one of several places in the tree.

Rules in Parsifal are of the typical situation/action type. To decide if it is applicable, a rule will most often look to see what is in the buffer, although, with some limitations, rules may also look at the stack. Two positions in the stack are special, the bottom of the stack, which is named **c**, and the lowest sentence node in the stack, which is named **s**. To take a simple example, in Parsifal the rule for recognizing passive constructions is this:

```
(rule passive-aux        ;The rule is named passive-aux
  [= be] [= en] *        ;It looks at first two buffers
  Attach 1st to c        ;It puts the "be" on the bottom-
       as passive.)      ;most node of the stack.
```

---

*This is an extended abstract of a much longer paper by the same name, available from the author. My thanks to Graeme Hirst, who commented on the original paper. This research was supported in part by the Office of Naval Research under contract N00014-79-C-0592, and in part by the National Science Foundation under contract #MCS-8013689.

[1]Unless we explicitly indicate to the contrary, this and all other examples in this paper can be handled by Paragram. When an example is ungrammatical, Paragram will recognize it as such, but produce a reasonable "deep structure" anyway. Furthermore, it will indicate what in the sentence it did not like.

[2]We will be using a version of transformational grammar which was current in the late sixties. The primary reason for this choice is its familiarity. It should not be assumed that Paragram must necessarily use a grammar of this type.

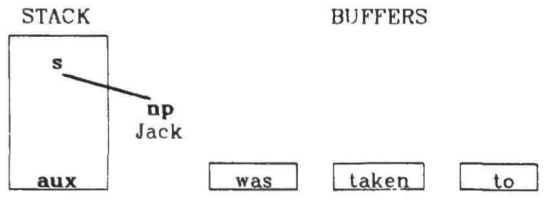
[3]Furthermore the time it takes an ATN to decide that the sentence is ungrammatical should go up roughly exponentially with the number of words. For some reason, those who tout ATNs as a model of human performance do not draw much attention to this "prediction".

The two square-bracket groupings indicate what the rule requires in the first and second buffer. In particular, the "=" indicates that what appears in the corresponding buffer must have the feature specified, such as being a form of the verb "be". Everything following the "→" is the action portion of the rule. These actions are specified in a language called "Pidgin", which is quite restricted but formulated to look like English.

Suppose we were applying this rule in the course of processing the sentence

Jack was taken to the house.

At the point where **passive-aux** is applicable, the state of the parser would be as follows:




Here the **np** "Jack" has been made a constituent of the top level sentence, but it is hanging off to the side to indicate that it is no longer on the stack, since it is a completed constituent. In the course of testing, Parsifal will see that "was" is a form of the verb "to be", while "taken" is an "en" form of the verb "to take" and thus the buffers match the rule test. At this point the action would be executed, which is to put the "was" on the **aux** which is currently the node Parsifal is working on. This will have the side effect of removing the "was" from the buffer, at which point the words further along in the sentence will move in to replace those which have been removed from the buffer.

However, not all rules of the grammar will actually be tested at any given point. Indeed most of the rules would be completely irrelevant; while parsing the auxiliaries of a verb we would hardly expect to find direct objects. To prevent Parsifal from even looking, each of its rules is found in one or more "packets" and only the rules which are in "active" packets will be tried. The active packets are those which are attached to the bottom node of the stack, **c**. Should this node be removed, the packets on the next higher node will be active. The idea is that if Parsifal is working on a noun phrase, then noun phrase rules will be active. Once Parsifal is done with it, it will be removed from the stack, and the rules on the next higher uncompleted constituent, say a verb phrase, will become active once more. Which packets are attached to a given node is explicitly controlled by the grammar rules themselves.

There are complications to this scheme, but this should due to give the reader a basic idea of how Parsifal works.

### 2.3. Parsing Ungrammatical Sentences

Naturally, Parsifal as currently constructed will only parse grammatical sentences. Should it be given an ungrammatical sentence, it will eventually come to a point where no rule applies, and it will simply give up.

Paragram differs from Parsifal in numerous ways, but allowing for ungrammatical input requires only a comparatively minor modification. Whereas Parsifal rules are tried sequentially until one works, active Paragram rules are to be thought of as being tested in parallel. Naturally, on current computers, they are really being tried sequentially, but it is useful to think of them as working in parallel. Furthermore, unlike Parsifal, the result of a test in Paragram is not a yes/no decision. Rather it is a numerical "goodness rating" which, the higher the number, the better the fit between the rule and the buffer/stack. Paragram then takes the rule with the highest number and runs it, allowing it to change the stack and buffers. It then repeats the process.

The goodness rating of a rule is the sum of the values returned by the rule's atomic tests. Each atomic test will add to the score if it succeeds, and subtract if not. No significance should be attached to the actual numbers. The basic idea is that the more tests succeeding, the higher the score, and failure is punished severely.

Now the crucial point in all of this is that for an ungrammatical sentence, the various ratings that we will get at the point of ungrammaticality will all be quite low, since none of the rules of grammar will exactly match the input. *Nevertheless, one rule must still have the highest score, and hence will apply, even though it does not really approve of the sentence as given.*[4] So, for example,

*The boys is dying.

will be given a low rating when Paragram starts to parse the auxiliary "is", at which point the best rule will be:

(rule subject-verb-agreement in parse-aux
[= verb] [test: The np of s agrees with 1st.] →
Create an aux. Activate build-aux.)

When applied to the above ungrammatical sentence this rule will have a poor goodness-of-fit rating since there is a match with **verb**, but the subject/verb agreement fails. Nevertheless, this is the best value at that point, so the rule is used anyway, and Paragram starts parsing the auxiliary verb, as intuitively it should. Note however that with sentences like

The fish is dying.
The fish are dying.

the above rule will succeed in each case, and in the process specify that the word "fish" is to be understood as singular and plural respectively. Some other ungrammatical sentences handled by Paragram are:

*Bill sold Sue book.
*Jack wants go to the store.

There are however, many ungrammatical, yet understandable, constructs which Paragram cannot currently handle. For example, extra constituents give it a problem. So more work needs to be done.

### 3. Parsing the Relevant Generalizations

The second goal set out for Paragram is that it capture appropriate generalizations about language in much the same way as a good transformational grammar. This has proved an elusive goal in parsing programs. The most well known of AI parsers, Woods's ATN parser, has been based upon transformational grammar, and Bresnan [1] points out that one could use the ATN framework to provide the link needed between her "realistic" grammar of English and an actual performance model of parsing. Nevertheless, while ATNs are inspired by transformational grammar, the single extended ATN grammar I have seen often required several special-case rules to handle what is a single rule in transformational grammar. While we will not pursue this point in any detail, let us take a single example, taken from the ATN grammar for English given in [7].

The rule of **there-insertion** in transformational grammar relates sentences like these:

There were barnacles on the ship.
Were there barnacles on the ship?
The ship on which there were barnacles sank.
The ship there were barnacles on sank.[5]

The statement of the rule is something like this:

[4] There is still the possibility of ties. However in practice this has not come up, and it can be argued that baring problems with the basic idea of deterministic parsing, ties should simply not occur.

[5] Stylistically this ain't so hot, but presumably it is grammatical. At any rate, the ATN grammar has a rule to handle it.

**np(-def) exist-verb** → There **exist-verb np** . .

This rule handles all cases of unstressed "there" (as opposed to the "there" in "There is Jack"). However, Woods's ATN has four separate rules for **there-insertion**, one for handling each of the above cases. So, despite the inspiration of transformational grammar, this ATN grammar has not done as well as one would like in the elegance of the rules it embodies [6]

Now in many respects, Parsifal does better. In particular, it needs only one rule of **there-insertion**. But this is not to say that all of Parsifal's rules are this elegant. *Indeed, of the 57 or so rules which I have looked at in depth from the detailed grammar at the end of Marcus' book [3], only twenty or so correspond to transformational rules.* Of the rest, they may be categorized into three groups, depending on the particular deficiencies which required them to appear in the grammar.

**Miscellaneous problems.** Six of the rules are needed because of various peculiarities of the grammar and the parser. While some of these are interesting in their own right, (particularly two, which deal with the rule of **raising**, a controversial rule in Linguistics) we shall say no more about them here.

**Phrase Structure Rules.** The majority of the 31 other rules, 21 in all, are there because Parsifal must have explicit rules in its grammar for the placement of phrase structure constituents. Thus, a rule like

**s → np vp**

is implemented by four separate rules in the grammar: one for creating an **s** when needed, one for attaching the **np** at the right spot, another for the **vp**, and finally one which says to stop parsing the **s**. Not only would it would be preferable to have a single rule, but the packet mechanism in Parsifal is really a phrase structure mechanism in disguise, thus these twenty one rules are redundant, at least in principle.

Paragram solves this problem by explicitly using phrase structure rules to handle packet switching, as well as replace many of the aforementioned rules. However, it has not proved possible as of yet to replace the rules which create new constituents of the appropriate type. This is because such rules typically differ widely from one another in what they are looking for in the buffer to clue them in that the new constituent is needed. These creation rules are currently the only rules which are not in one-to-one correspondence with typical transformational rules.

**Wh-movement.** Next, the ten remaining rules are involved in the implementation of the **wh-movement** rule, as used in

Who did Jack give the ball to?

Some of these rules are needed because Parsifal has two sets of verb-phrase rules. There is a normal set, which handles most verb phrases, but as soon as we must worry about gaps, we have a completely different set. This second set differ from the first in two ways. First, because they must worry about gaps, this second set continually checks the semantics to insure that it has not gone astray. In fact, there are cases where it is only semantics that can tell the parser what to do. For example

What did Bob give the girl?
Who did Bob give the book?[7]

Second, this second set contain many rules, each looking for a different configuration of things in the buffer which, in turn, will suggest where to locate the "gap" left behind by **wh-movement**.

Paragram does without all of this by making two changes in the parsing mechanism. The first is a technical change in the way the parser decides to postulate that a **wh** might have been moved from a particular location. The second, and perhaps more interesting change, is to avoid needing two sets of verb-phrase rules (with and without calls to semantics). Paragram *always* checks to see if a constituent is semantically reasonable before it will add it to the syntactic tree. Note that the user need not explicitly specify that a call to semantics is required here. Rather, Paragram automatically adds such calls to the testing section of any rule which adds a constituent to the tree [7]

## 4. Conclusion

While we have given no reason to take Paragram seriously as a model of human cognition, if we were to do so we would want answers to two important questions. First and foremost, how does syntax fits into the overall parsing process? Paragram essentially takes the conservative view that syntax is the initial mechanism which takes the word string and produces as its output some "deeper" representation that has properties that make it useful for the further pragmatic processing. It differs slightly from this however in that it requires that semantics be performed on a constituent before it can be attached to the tree.

Once we have decided to adopt a model in which syntactic analysis is done, and done as a separate process, we must then answer the second major question: what is the relation of the syntactic parsing process to standard "competence" models of syntax? Again, Paragram takes the old fashion view that this relationship is reasonably direct. Paragram argues that it may be possible to have a one-to-one correspondence between parsing rules and rules of grammar in a parser which is computationally efficient.

These views were, of course, very common in the sixties. They are less common now, in part because of various psychological results such as those of Slobin [5]. It is not my intent to try to refute the interpretation placed upon these results. Rather I hope that the existence of Parsers like Paragram can reopen the debate on these crucial subjects.

## References


1. Joan Bresnan, "A Realistic Transformational Grammar," in *Linguistic Theory and Psychological Reality*, ed. M. Halle, J. Bresnan, and G. Miller, M.I.T. Press, Cambridge Mass. (1978).
2. B. Elan Dresher and Norbert Hornstein, "On Some Supposed Contributions of Artificial Intelligence to the Scientific Study of Language," *Cognition*, (4) pp. 321-398 (1976).
3. Mitchell P. Marcus, *A Theory of Syntactic Recognition for Natural Language*, M.I.T. Press, Cambridge, Mass. (1980).
4. Christopher K. Riesbeck, "Computational Understanding," pp. 11-15 in *Proceedings of the First Workshop on Theoretical Issues in Natural Language Processing*, ed. R. Schank and B. L. Nash-Webber, (1975).
5. Dan I. Slobin, "Grammatical Transformations and Sentence Comprehension in Childhood and Adulthood," *Journal of Verbal Learning and Verbal Behavior* 5 pp. 219-227 (1966).
6. Yorick Wilks, "An Intelligent Analyzer and Understander of English," *Communications of the ACM* 18(5) pp. 264-274 (1975).
7. William A. Woods, "An Experimental Parsing System for Transition Network Grammars," pp. 113-154 in *Natural Language Processing*, ed. R. Rustin, Algorithmics Press, New York (1972).


[6] It is important, however, to keep in mind the distinction between limitations in the parser and limitations in the particular grammar. It is possible that a more clever grammar writer might have avoided this problem. Indeed Woods has claimed (personal communication) that Parsifal is simply one kind of ATN. This may be so, but only in the uninteresting sense that both are one kind of Turing machine. In particular, the only ways I can see of simulating Parsifal with an ATN would completely ignore all of the built-in features that make ATNs what they are. At any rate, if it is true that Parsifal is one kind of ATN, then it must surely be the case that any improvement in Parsifal's grammar over a particular ATN grammar must only indicate deficiencies in the ATN grammar.

[7] There are two exceptions, due to problems in our handling of **dative-movement**.