LINGOL — A Progress Report

Vaughan R. Pratt

Massachusetts Institute of Technology
Artificial Intelligence Laboratory
Working Paper 89
January, 1975

Abstract

LINGOL is a linguistics oriented programming language. We describe briefly work in progress on a large-scale LINGOL program designed to demonstrate the value of LINGOL as a research tool in the writing of natural language programs. Attention is then focused on the parsing algorithm used in LINGOL. It is shown that for the class of interpretive context-free parsers that do no backing up or lookahead, the algorithm is optimal with respect to discovery of phrases in that no phrase is discovered (or "state" built) that cannot be used in some continuation of the input seen so far. This constitutes an improvement in space and time of up to a factor proportional to the size of the grammar over previously known general context-free parsing algorithms. For small grammars of English, a factor of five has been measured. It is shown that the parsing algorithm can accept context-dependent "advice" in such a way as to facilitate the writing of "intelligent" grammars. Finally, the role of syntax in contributing to efficient parsing is discussed.

1. INTRODUCTION

LINGOL (LINGuistics Oriented programming Language) is a language to facilitate the writing of natural language processing programs. In a previous paper [Pratt1973], the LINGOL system was described, examples of output from several small-scale LINGOL programs were given, and the reader was led through a console session as a trivial-scale French translator was developed. That paper sought to establish LINGOL's credentials as a pedagogical device for newcomers to the art of writing natural language "front ends".

LINGOL was originally conceived as a language intended for use by serious researchers. Due to its author's preoccupation with more mathematical pursuits during the past few years, LINGOL has not been exercised with anything but small-scale student-generated programs. Since the appearance of [Pratt1973], several more small-scale LINGOL programs have been written. More recently, the author has begun work on a large-scale program to see whether LINGOL really can be used as the research tool it was originally designed to be, without compromising those features that made it attractive to beginners (ease of use and low resource consumption). In this paper we briefly describe work to date on this project, and then concentrate on the implementation of the current version of LINGOL. This latter discussion will complement the one in [Pratt1973], which concerned the motivation for the language, that part of the LINGOL system accessible to the user. Inter alia, we shall present a new context-free parsing algorithm that, paradoxically perhaps, is responsible for the efficiency of the very context-dependent parser in LINGOL's cognitive component. The idea behind this algorithm is one that may be of value in other structure-eliciting type problems besides parsing.

2. WORK IN PROGRESS

To establish the viability of LINGOL as a useful research tool, it was decided to begin by seeing whether Terry Winograd's [Winograd1971] Blocks program (affectionately called SHRDLU) could be written in LINGOL. Several reasons make this a suitable choice of target. Firstly, SHRDLU constitutes a well-documented bench-mark for the fore-front of computational linguistics research. If SHRDLU can be written in LINGOL, the potential LINGOL user need not be deterred by prospects of LINGOL getting in his way. Secondly, the size of SHRDLU, its resource consumption and the work required to implement it are all well-known. Thirdly, though it is tempting to want to begin with a more exotic domain, the blocks world remains a challenge inasmuch as there still exist no programs really competent to play with blocks the way a five-year-old might (see [Fahlman1973] for recent progress in building with blocks). Moreover, graduation from blocks to LEGO or Fischer-Technik toys is a natural and interesting extension. Fourthly, since Winograd wrote his thesis, the situation at the MIT AI laboratory has improved with respect to robot arms and their software support, making it feasible to have the program drive a real arm instead of the simulated arm shown on a display in Winograd's program. Such a system would make an excellent demonstration program quite apart from the issue of how it compared with SHRDLU. Fifthly, there is the tradition of repeating experiments to verify that the results really are valid. If SHRDLU were to become an irreproducible result, that might enhance the reputation of Winograd, but it would be a disaster for Computational Linguistics. Since 1971, no programs on the same scale as SHRDLU have been written. Sixthly, SHRDLU is a present in a state of disrepair, and this author has never seen it in action, despite two years of waiting for it to be resuscitated. This can be something of a frustration to someone whose top-level goal is to see such a program working. Under such circumstances one can be forgiven for giving up and building a new one.

SHRDLV is the obvious choice of names for such an imitation of SHRDLU, being its alphabetic successor (forbidding longer names). It has nothing to do with the author's first initial.

SHRDLV takes English sentences and translates them into a program which takes the action appropriate to the sentence, answering queries, noting assertions and obeying commands in a mindlessly dutiful way. For obeying commands to perform operations on the blocks accessible to the robot, there exist a set of high level operations including "pickup b", "put b on c", "grasp", "ungrasp", "tower(b,c,d,...)" and the like which know about such issues as finding places to put things and making room for things. What has been built to date are the translator and the blocks world operations. The interpreter for the output of the translator is in its initial stages of construction.

2.1 The SHRDLV Translator

The output of the translator should be more or less self-explanatory given the following examples taken straight from the machine. The output is in the form of a program which is forwarded to an evaluator (not LISP's) for execution. The top-level functions ACHIEVE, QUERY and ASSERT take one argument, a fact, and either try to make the fact come true, test it (filling in and returning values when there are items of the form (WHAT ...)) or merely add it to the data base. Other top-level functions include ASSUMING, which takes two arguments, one of which it makes true in a hypothetical world and the other of which it executes (normally it will be a QUERY) and CHOICE, which takes two arguments, the first of which is a query and the second a list to choose from (as in "What do you like better, red blocks or blue?").

Note that the translator has left some of the so-called "deep-structure" analysis of the sentences up to the interpreter. In particular, the interpreter is aware of failings of the translator; it turns out to be simpler to have the programs running under the interpreter compensate for such things as an "OBJ" appearing somewhere and denoting an unfilled object slot than to make the translator go to the trouble of seeing that everything it produces is in perfect health. Also, anaphoric reference is resolved by the interpreter, although we suspect that this is a bad thing and propose to shift this back to the translator.

Examples:

```
PICK UP A BLOCK.
(ACHIEVE (PICK YOU (CHOOSE BLOCK) UP))
536. MILLISECONDS.

WHAT ARE YOU HOLDING?
(QUERY (GRASP YOU (WHAT)))
928. MILLISECONDS.

WHAT COLOR IS IT?
(QUERY (BE IT (WHAT COLOR)))
853. MILLISECONDS.

PUT IT BESIDE THE PYRAMID.
(ACHIEVE (PUT YOU IT (BESIDE (DEF PYRAMID))))
560. MILLISECONDS.

WHERE IS THE RED BLOCK?
(QUERY (BE (DEF RED BLOCK) OBJ (WHAT LOC)))
594. MILLISECONDS.

WHAT IS IT ON?
(QUERY (BE IT (ON (WHAT))))
542 MILLISECONDS.

WHAT SHAPE IS THE BIG RED THING?
(QUERY (BE (DEF BIG RED) (WHAT SHAPE)))
1176. MILLISECONDS.

IF YOU PUT ONE BLOCK ON ANOTHER, CAN YOU THEN PICK UP THE LATTER?
(ASSUMING (PUT YOU (CHOOSE BLOCK) (ON (CHOOSE OTHER)))
          (QUERY (POSSIBLE (PICK YOU (DEF LATTER) UP))))
1759. MILLISECONDS.
```

```
CAN YOU PUT A BLOCK ON A PYRAMID?
(QUERY (POSSIBLE (PUT YOU (CHOOSE BLOCK) (ON (CHOOSE PYRAMID)))))
976. MILLISECONDS.

HOW MANY BLOCKS CAN YOU PICK UP?
(QUERY (POSSIBLE (PICK YOU (CHOOSE (WHAT QTY) BLOCK) UP)))
1114. MILLISECONDS.

WHY DID YOU DO THAT?
(QUERY (PAST (DO YOU (DEF) (WHAT REASON))))
703. MILLISECONDS.

WHICH IS TALLER, THE RED CUBE OR THE GREEN ONE?
(CHOICE (BE (WHAT) (RANK HEIGHT 1)) (OR1 (DEF RED CUBE) (DEF GREEN)))
2244. MILLISECONDS.
```

2.2 Evaluating the Translator's Output

With some imagination, one may infer what sort of programs are required to run the output of the translator. These programs are under construction at present, and already allow one to say "PUT THE BIG RED BLOCK ON THE GREEN CUBE" and have it translated as the program (PUT B1 B5), where B1 and B5 are the internal names of the designated blocks, and PUT is one of the blocks world functions implemented by the author to manipulate blocks using David Silver's "Little Robot". (This interpreter represents a few hours of programming effort, so things are not yet particularly advanced here.)

For example, the program for DEF expects a list of predicates, some of which are atomic, like "RED" or "BLOCK", and some of which are implicit lambda-expressions (with bound variable X), like "(OWN I X)". It searches for an item satisfying all of the predicates, and takes obvious actions depending on how many it finds. Some predicates, like "(QTY 2)" or "(RANK HEIGHT 1)" are in the nature of global predicates rather than local ones. The former means that two are expected ("the two red cubes"), while the latter means maximize with respect to height among all candidates. DEF is intelligent enough to conduct the search efficiently, preferring to begin with predicates known or likely to apply to few items.

A similar operation is CHOOSE, which differs from DEF in that it does not suppose that the speaker has a particular item in mind, but only wants it to have certain properties. Thus CHOOSE, unlike DEF, feels free to carry out its own optimization of choice with respect to what the item is to be used for. This avoids such foolish things as having the arm "pick up a block" from the far end of the table when there was one right beside the hand.

2.3 The Blocks World Program

The blocks world itself is perhaps in better shape than either the translator from English to the intermediate language or the processor of the intermediate language, despite the fact that an order of magnitude more work has been invested in the translation program. This says something about English as being the hard part of building a SHRDLU.

Space does not permit here any sort of demonstration of the Little Robot block manipulator, and the subject matter of this paper makes it somewhat irrelevant here. However, visitors to the MIT AI lab are more than welcome to watch it in action -- it makes an interesting demonstration to see a mechanical device solving complex logistics problems in finding space for things, navigating around things and dealing with several sub-goals nested simultaneously.

3. THE CURRENT LINGOL IMPLEMENTATION

3.1. General Overview

We have already discussed elsewhere [Pratt1973] the rationale for those features of LINGOL available to the user. In this section we shall talk about what goes on behind the scenes. The distinction being drawn here is exactly that of the programmer's manual for a language versus the implementation of a particular compiler for that language. In the case of LINGOL the author has found on occasion people who are unwilling or unable to draw the distinction; the result is a misconception of what the LINGOL user has to put up with in writing his programs, as opposed to what performance he may expect when running his programs. The distinction has in fact to be drawn even more carefully for LINGOL than for conventional programming languages because for the latter, the usual choice of operations (arrays, lists, block structure, arithmetic and other operations with well-understood implementations) suggests, to within details of little interest to the programmer, the appropriate implementation of the run-time support. Since the programs one writes for LINGOL are highly non-deterministic, and are organized as modules (actors, to use a term in vogue) that do not know with whom they will be communicating until LINGOL connects them together during the processing of a sentence, the LINGOL run-time system's task is appreciably more difficult than that of FORTRAN or ALGOL. So far, no one has proposed an "obviously" good way to tackle this problem for English, and LINGOL users should be prepared to accept radical changes in LINGOL's internal operation (i.e. parsing algorithm) as progress is made in this area. Since the only legitimate effect of such changes is to improve overall resource consumption, rather than correctness of the user's program with respect to a machine with unlimited resources, this is not really a burden on the user.

For the benefit of non-readers of [Pratt1973] we describe briefly the overall organization of LINGOL. The LINGOL system is envisaged as a translator from some natural source language to some target language (natural or artificial) of the user's choice, not necessarily a different language from the source language. There are two phases, one that elicits the surface structure of a sentence and one that produces the desired translation(s). The intention is that issues relevant to determining the intended surface structure versus those of translation should be separated out. This corresponds to the recommended practice when translating from English to French, say, of understanding each sentence (naturally taking into account previous sentences) before attempting the translation. No restrictions are made on where the LINGOL programmer draws this boundary, or to what extent the information in the two components is duplicated. In fact, he can omit the generative component entirely and put everything into the cognitive component, though at some cost in resource consumption. At run time, no firm commitment is made by the cognitive component to a particular choice of surface structure of an ambiguous sentence, allowing the generative component to pick and choose when the cognitive component has not had enough information to decide. At present LINGOL users are encouraged to try to make their cognitive component intelligent enough to make the right decision, and so far no LINGOL programs have attempted disambiguation in the generative component. One would expect this to change as people attempt more sophisticated programs.

A LINGOL program is a set of modules each having three components, a context-free rule, a cognitive function and a generative function. Their respective roles are as follows. The CF rule specifies a general English construction, the cognitive component (or "critic") supplies the expertise about that construction and the generative component supplies the information about the target language that may be relevant to this English construction. (Our tacit assumption of English as the source language reflects LINGOL's applications to date.)

It is fashionable these days to want to avoid all reference to context-free grammars beyond warning students of computational linguistics that they are unfit for computer consumption as far as computational linguistics is concerned. (See [Pratt1973] for arguments attacking context-free grammars as being too powerful for specifying programming languages, and [Pratt1969] for arguments attacking them as too weak for computational linguistics.) In LINGOL, their role is different from that in, say, the Harvard Predictive Analyzer [Kuno1965]. Instead of being used to encode all information about English, they form the basis of a pattern-directed non-deterministic programming language. This strategy has several advantages.

(i) It allows the programmer to structure his program as a set of relatively self-contained modules, thereby decreasing the number of things he has to keep in his head at once when looking at a particular part of his program.

(ii) It eliminates much of the testing-for-cases control structure the programmer would need in a non-pattern-driven language.

(iii) Flow of control between modules is confined to the surface structure, radically simplifying tracing of a computation.

(iv) Instead of having to identify each possible source of ambiguity and thing up a way to deal with it, the user writes "critics" of individual situations and lets LINGOL compare the results of the criticisms as applied to competing situations when an ambiguity arises. This reduces the order of magnitude of programming effort in the resolution of ambiguity from order n^2 to order n, where n is the number of situations that may need to be compared.

(v) LINGOL can optimize the user's program much more effectively if it can identify the context-free component by itself. If this component were to be incorporated into the cognitive component, a popular practice these days, the system would not be able to do its own optimization as effectively, and the burden would fall back on the user.

The reader wanting more information on items (i) to (iii) is referred back to [Pratt1973]. The worked example illustrates each of these advantages. The remaining two items are covered in the following sections.

3.2 The Cognitive Component

In this section we describe the LINGOL parser. We first present the algorithm on which it is based, and then show how to use this algorithm to assemble the user's modules and set up communication between them.

Before immersing ourselves in the technical details of the algorithm, let us consider the options open to us. The goal for the parser is to build the surface structure intended by the sentence's speaker. All it has to go on is the top and bottom of this tree, and the rules (grammar) constraining plausible surface structures. A decision must be made as to where growth should begin. Two extremes are the top-down approach, in which the tree is grown from its root, and the bottom-up, where growth begins from the bottom, that is, from the words of the sentence. Methods with the flavor of either (or both) of these extremes inherit their name(s). These methods have other aliases in the literature. Any scheme that claims to be doing "predictive" analysis, that is, that has expectations about what is coming next and uses those expectations to "drive" the program is essentially a top-down method. A program that finds substructures (say conceptual dependency structures [Schank1970]) and uses them to build bigger structures is a bottom-up method. Terms suggested to the author by R. Moore are "hypothesis-driven" for "top-down" and "data-driven" for "bottom-up". These concepts transcend phrase-structure grammars and may be applied to any system responsible for building an hierarchically organized system.

No matter where the construction begins, we do not know how to carry it out in any straightforward way, even if we want nothing more than to satisfy the context-free rules of our grammar. We always run the risk of letting the construction wander down blind alleys. For some grammars and some sentences, the top-down method is less likely to run into cul-de-sacs, but the dual case can also arise. It is hardly surprising, given this state of affairs, to hear people wish that they could build structures both top-down and bottom-up in a way that somehow reduced the overhead. One seemingly ambitious form of this wish is to request a single algorithm that builds no node a bottom-up method would not consider, nor anything a top-down method would not build. An alternative and even more ambitious desideratum might be that no node N be built unless that part of the text seen to date is part of some sentence having a surface structure in which N participates. For a backup-less parser like LINGOL's present one, this is the strongest possible thing one could ask for as far as exploring cul-de-sacs is concerned. One might add to the above the requirement that the parser be able to cooperate with other processes such as routines written by the user, but with so many wishes stacked up the situation seems hopeless.

The current implementation of LINGOL achieves all of the above goals. To be more precise, every node it builds is built by both the Cocke-Kasami-Younger bottom-up algorithm and the ingenious Earley top-down algorithm, the two algorithms cited in [AhoUllman1972] as the canonical methods for parsing general context-free languages. That is, the work performed is the intersection of the work done by each of these methods, at least with respect to proposed phrases. Moreover, as each phrase is discovered, LINGOL is able to accept advice from other sources (namely the user's cognitive component) and use it to guide the parse.

Roughly speaking, the minimization of searching is accomplished by running the Cocke-Kasami-Younger algorithm and as each phrase is discovered asking an "oracle" whether the Earley algorithm would have discovered it. The remarkable thing is that this question can be answered in time independent of the length of the input, without having to go and actually run the Earley algorithm to see what it would have done. (The time is proportional to the size of the grammar, but in the LINGOL implementation, asking the question involves no more than forming the logical and of three 36-bit vectors for a grammar of a hundred non-terminals, a size that SHRDLV attains with the help of its cognitive component, which takes over much of the work the large CF grammars of the 60's used to do.)

Before discussing the construction of the oracle, let us sketch the version of the Cocke-Kasami-Younger algorithm we shall use. We assume that all rules are of the form either A→B or A→B C, where A, B and C are non-terminals, or of the form A→a where a is a terminal. The presence of A→B means that this is not really Chomsky normal form, and allows either the user or some preprocessor to turn an arbitrary grammar into this form using only the trick of replacing all but the first item on the right side of a rule having three or more items by a non-terminal which is itself rewritten to be the replaced non-terminals, and so on until all rules have right sides of length 1 or 2. (In [Earley1968] the notion of "state" is introduced, which elegantly plays the role of these introduced non-terminals. In Earley's notation, the state AB.CDE plays the role of the non-terminal that replaces the CDE. Everything we say in terms of our restricted grammars can be rephrased more elegantly in terms of Earley's states. The main advantage of our notation is that things look less complicated if the reader is not required to think about arbitrarily long states.) With this form of grammar we can and shall talk about the left and right sons of binary nodes and the "only" sons of unary nodes. For our purposes it will be convenient to refer to only sons as left sons.

In the following version of the algorithm, we use the notation ?x (where x is a variable) to mean "find all x such that". Thus the code

```
for ?i+?j=5 do print(i,j)
```

will print all pairs of numbers summing to 5. This avoids cluttering up the algorithm with details of searching that the programming reader will have no difficulty filling in.

We shall employ "between-word" notation for positions rather than "at-word". That is, rather than saying that the first word in the sentence is at position 1, we will say it lies between positions 0 and 1. This avoids any possible ambiguity when referring to the string lying between positions i and j, and also simplifies naming the common boundary of two concatenated strings.

The Cocke-Kasami-Younger algorithm is:

```
parse: for each word w between positions j-1,j (scanning left to right) do
            (for each rule ?A->w do note(A,j-1,j);
             for each phrase (?B,?i,j) in turn on the queue do
                 (for each rule ?A->B do note(A,i,j);
                  for each rule ?A->?C B such that (C,?k,i) has been built do
                      note(A,k,j))).

note(A,i,j): build(A,i,j); enqueue(A,i,j).
```

We assume as given the primitives that build a node (representing a phrase consisting of a non-terminal, a starting position and an ending position -- we shall use "phrase" and "node" interchangeably) and that enqueue and dequeue items.

To make this algorithm run in polynomial time ($O(n^3)$) to be exact) "note" is usually written as:

```
note(A,i,j): if (A,i,j) not yet built then (build(A,i,j); enqueue(A,i,j)).
```

The above suffices for context-free recognition. For parsing, nodes record (in addition to the three items type, start and end) two additional items, namely which rule was invoked when noting that node, and which phrases are its sons. The former allows us to access the cognitive and generative components associated with the rule at a later date, while the latter allows us to recover the surface structure. (Having the rule present makes the syntactic category of the node redundant, and in fact LINGOL omits it.)

We now introduce the oracle. Two things are required to construct this oracle: a readily accessible representation of the left-most-character relation, and the notion of a goal. We first deal with the goals. Associated with each position in the sentence is a set of goals. A goal is a desired non-terminal. If a phrase of the same type as some goal is discovered starting in the position with which the goal is associated, then that phrase can be used as a right son, in conjunction with some phrase discovered earlier, to form a new phrase. For example, if there is a rule A→B C and a B has been found ending in position i, then a goal C associated with position i will be created. Later, if a C is found starting in position i, it can be used with B to form an A.

Initially there is one goal, to find a Sentence (assuming this is the distinguished non-terminal of the grammar), which is associated with position 0. As the algorithm runs, other subgoals will be generated. While this sounds like a top-down method, no structures are built top-down -- they grow bottom-up and the goals are associated with the frontiers of the growing tree (with the exception of the Sentence goal). The goals appear just when and where you would expect them to; for example, when reading the sentence "Secretary of State Henry Kissinger announced today the cessation of ware in the Middle East", after reading the fifth word it will have set up (inter alia) a goal associated with position 5 and wanting a verb phrase. In fact all goals for a given position are created at once, just before reading the word at that position. We defer for a bit the question of where goals come from, and look first at how they are used.

Let R be a binary relation on the non-terminals of the grammar such that A R B if and only if there exists a rule of the form A→B … . Then R^*, the transitive closure of R, is the left-most-character relation alluded to above. Thus A R^* B can hold just when there exists a derivation A $=>^*$ B …, or for those who prefer trees (myself included) it should be possible for B to occur on the left edge of some tree with root A that satisfies the productions of the grammar. (See [GriffithsPetrick1965] for other applications of this relation.)

It is straightforward to incorporate this oracle and the goal machinery into the algorithm. We assume primitives for storing and fetching goals which are pairs (A,i) of non-terminals and positions. To introduce goals, append to the innermost loop of "parse" the code:

```
for each rule ?A->B? C
    such that goal (?D,i) exists satisfying D R* A do
        setgoal(C,j)
```

and precede the code for "note" with the test:

```
if goal(?B,i) exists such that B R* A holds then
```

To see that the algorithm as modified does no less than it has to (i.e. that it overlooks nothing), suppose we have an initial segment of a sentence of the language of the given grammar. Then in any tree for this sentence, we claim that all phrases in the tree contained within the initial segment will have been proposed by the time we have reached the end of the segment. We also claim that every right son in the tree will have been generated as a goal just before the parser reached the starting position of that phrase. (The induction proofs of these claims are messy and probably inappropriate for this paper -- the interested reader is encouraged to fill in his own details.) It follows from these two claims that the oracle will always answer in the affirmative when a phrase that appears in the tree is proposed to the oracle, because every phrase is the left-most character of either some non-left-son or of the root, and all such possible goals will have been created by the time we propose this phrase. Hence the phrase is correctly built.

To see that no node is built that could not participate in some surface structure of some completion of the sentence, we must assume that for each non-terminal of the grammar there exists a derivation starting with that non-terminal and ending with a string of all terminals. This follows immediately if we require that every non-terminal appear in at least one surface structure of some sentence of the language, a perfectly reasonable requirement. Suppose that we have just built some node (B,i,j). Then there exists some goal (A,i) such that A R^* B holds. Hence there exists a tree with A at the root and B on the left edge. We can therefore extend the sentence so that the part starting with the B reduced to A. One more reduction is now possible, using the rule that gave rise to the goal in the first place. We continue up the tree in this fashion, progressively extending the sentence and satisfying more goals, until we satisfy the Sentence Goal. At this point we have the desired sentence. This completes the proof of the claim that every node built has a chance of being used in the final surface structure.

What does this fancy algorithm buy as far as the practically minded user is concerned? One thing we do not claim is any improvement over the traditional $O(n^3)$ speed limit for parsing sentences of length n. (See [Valiant1974] for an improvement to this situation -- he offers $O(n^{2.81\cdots})$, though practical considerations make it much worse than most $O(n^3)$ algorithms with respect to both speed and ability to take advice when parsing "typical" English sentences.) However, we do claim what amounts to an even better improvement in practice than going from $O(n^3)$ to $O(n^2)$, namely an improvement proportional to the number of non-terminals in the grammar. That is, there exist grammars for which the Earley algorithms may generate large state sets in its operation when our algorithm builds very many fewer nodes. (However, there do not exist any grammars where Earley's algorithm runs in time $O(n^\alpha)$ while ours runs in time $O(n^\beta)$ for $\beta<\alpha$. There do exist such grammars when comparing our algorithm with the Cocke-Kasami-Younger algorithm.)

In even more practical terms, how does this affect parsers working with a purported grammar of English? We conducted an experiment to compare our algorithm with the Cocke-Kasami-Younger algorithm by the simple expedient of suppressing the test in the procedure "note" that makes our algorithm different from the other. Working with the grammar of English used in the "9-hour" French translator exhibited in [Pratt1973], we found an improvement of a factor of five in the number of nodes built altogether! In fact, with the new algorithm almost all of the nodes built were used in the final surface structures of the sentences we tried. Lack of time has not allowed us to compare the algorithm with Earley's, but the sort of thing that makes our parser better than Earley's in some grammars is exactly what arises in English grammars. For example, if one has the rules Sentence→Np Vp, Sentence→Wh Vp and Sentence→Vp, then Earley's algorithm will generate states corresponding to each of these rules even when the sentence begins with, say, "Why". Earley's algorithm is not smart enough to realize that the first and third rules can be ruled out here (we are making some obvious assumptions about what the rest of this simple grammar might look like.)

3.3 Taking Advice

One attractive feature of the above technique is that we did not need to "compile" the grammar; we retained the interpretive nature of the Earley and Cocke algorithms. This makes it simple for the user to contribute to the operation of the parser, since all the parser is doing at each step is recognizing that some combination of phrases forms a new phrase. The user is given the opportunity at each step to look at the constituents of those phrases, to consult his model of the world, or to perform deductions. His conclusions are summarized numerically for LINGOL's benefit, and in pursuing any particular structure, LINGOL accumulates these numbers as a measure of its confidence in that structure. These confidence numbers are used to choose between alternative ambiguous structures. The winning structures are made readily available to the generative component while the losing structures are kept around (on the end of a list of alternatives) in case the generative component becomes dissatisfied with the choice made by the cognitive component and wants to try some of the others.

The style of programming used in the cognitive component is analogous to that described in [Pratt1973] for the generative component. The primary difference is that, since the structures are being built bottom-up at the time the cognitive component is being built, it is not possible to declare variables high up in the tree for use by routines lower down, a facility that gives the generative component considerable power. This inability is inherent in the nature of any system that wants to do criticism on the spot without waiting for the rest of the sentence. If you don't know what's coming, you can't (other than by guessing) make assumptions about what the higher nodes using the one in question will look like. (This is not altogether true -- if the relation R^* were represented explicitly as a collection of paths in R, it might be possible to set up variables on high as the goals are being generated, provided nodes being discovered below set their own sights on only one goal. It is likely that this would add substantially to the overhead of the system, however.)

3.4 Role of the Context-free Component.

Since the trend these days is away from explicit context-free grammars and towards encoding all syntactic information inside procedures, it is reasonable to ask why separate the context-free component. The issue is one of efficiency, among other things. It has yet to be demonstrated that English is easy to parse. As far as we know, a program with a lot of expertise about English must have at the very least the potential to be aware of the enormously many possibilities for misinterpreting typical sentences, and to be able to apply sensible criteria rejecting most of these interpretations. There is much wishful thinking these days about being able to ignore entirely the sorts of phrases discovered by the Harvard Predictive Analyzer [Kuno1965] for reasonable English sentences. To this author's knowledge, no such wishful thinking has been realized as a program having anywhere near the linguistic competence of, say, SHRDLU, or for that matter the Predictive Analyzer. The problem may be that one cannot dismiss these obscure parses on trivial grounds without also eliminating perfectly good parses of other sentences where the corresponding structure is not so peculiar. Unfortunately, there is no evidence to support this one way or the other, and the author wishes to sit on the fence for the time being as far as whether the above wishful thinking can be put into practice. (He would like to do so himself, but along with the rest of the world has no idea how.)

Given that one's program can be expected to encounter many competing interpretations of a sentence, and that in many cases it will have to pass non-trivial judgement on these cases, it can make very difficult the task of writing a program to deal with much of English. LINGOL allows the user to organize his program so that the burden of the bookkeeping associated with discovering and comparing all these possibilities is shifted to the system, allowing the user to concentrate on writing code to criticize individual situations. M. Marcus has suggested to the author that in so doing he is allowing the user to concentrate on the competence aspects of English by supplying him with packaged performance.

A long range goal in this regard is to develop a high level language version of LINGOL such that grammars may be given to either computers or people. Then if the computer "understood" English on the basis of that grammar, and if people could read it painlessly, it would make an ideal theory of English. Since people read procedures painfully slowly if at all, the high level language will have to be considerably less procedural than at present.

In our approach, the CF rules function as a crude approximation to English that permits LINGOL to rapidly select from a huge set of possible structures for the sentence a small plausible set for more detailed (and expensive) criticism by the cognitive component. The context-free representation for the "crude approximation" is chosen partly because it is not difficult to construct quite good approximations using context-free grammars, and partly because there exist remarkably efficient algorithms for exploring the space of possible surface structures for sentences of context-free grammars, yet that can accept advice on a step-by-step basis.

This situation of having the system do one's bookkeeping was what obtained in the hey-day of context-free parsers, of course. One side-effect of the later disenchantment with and abandonment of context-free grammars was to throw the baby out with the bath-water by reverting to doing much or all of the bookkeeping oneself. Not only does this require an indefinitely larger programming effort, it also requires of the programmer considerable sophistication in parsing techniques if his code is to operate as efficiently as one of the better context-free parsing algorithms, especially when that algorithm can cooperate effectively with the user's code in assisting it to reduce the search space even further.

A comparison of our system with that of Woods [Woods1969] is inevitable. Where Woods has augmented transition networks, we have augmented context-free grammars. One difference is in our parsing algorithm, a property of the implementation rather than of the LINGOL language. We would like to claim that it is more efficient than Woods', but in fairness we cannot tell without trying them out on comparable grammars, a test yet to be arranged. Another difference is in the language -- we take a static view of English, Woods a dynamic view. Both have their merits and demerits, but mainly from a human engineering than a computer science point of view. It all depends on how the programmer likes to think about English. One notable difference is that LINGOL has far fewer of its own concepts than does Woods language, without losing any of the features of Woods system. The idea here is that the constructs provided explicitly in Woods system are provided in LISP anyway, so why duplicate them? Another difference is our separation of the cognitive and generative components, which we feel is valuable from an economic point of view.

4. SYNTAX AND EFFICIENCY

LINGOL has had a chronic identity crisis over the issue of whether it should predominantly rely on syntax in its initial phase. This brief sections addresses the issue of whether syntax is a necessary part of a computational linguistics program. We have in mind here R. Schank's claim that "syntax is not needed to do parsing". For some variety in the usual replies to this sort of claim, we propose that even if Schank is right (this assumption is local to this section) syntax may be of value in improving the efficiency of the parsing process. This point can be easily overlooked both in informal introspection about whether one felt one needed any syntax to parse the sentence and in formal experiments, e.g. with the tachistoscope, designed to show that scrambled sentences shown briefly are recalled in their unscrambled form. What is being overlooked here is how long it took to unscramble the sentence.

Given grammar's language's sentence's initial segment assumed given in order to see no less overlooked than necessary. (Translation: To see that no less is overlooked than necessary, assume we are given an initial segment of a sentence of the language of the given grammar.) If you stumbled over this sentence then perhaps it is because the syntax is not there to speed things up for you. Conventional groupings of words have been rearranged in relatively unfamiliar, though not entirely ungrammatical or meaningless, ways and some "noise" words are gone. Nevertheless, with a little extra effort you should at least be able to parse the sentence correctly, and after a few passes you will begin to wonder why you ever had any trouble with it at all. Moreover, there seems no obvious reason why a program that could handle the original sentence could not equally well handle the above version. (I had difficulty restraining myself from replacing it with a more difficult sentence by the time I had typed it up.) The claim is that in the original version, part of the reason why you had less trouble with it was that it was phrased in a very conventional style that you have encountered frequently, allowing you to go straight to the places where you expect to find the information in the sentence. There are "noise" words all along the way, but to see that they are not really all that noisy, try replacing them with other noise words; the effect will be somewhat like switching all the street signs when navigating in one's car.

This then is a case for even more attention to syntax than is paid by either LINGOL or any other system known to this author, who strongly believes that extremely fast algorithms that waste no time at all in searching down blind alleys are possible. We have only to elicit the (presumably large) number of syntactic rules that seem to have gone unnoticed by traditional grammarians along with computational linguists but which our subconscious parsers presumably use constantly to make the right choice every time. (We append the usual speculation disclaimer here.)

(The "even more attention" in the case of LINGOL refers more to attention to previously unnoticed constructions rather than to attention to noise words. This author has noticed that LINGOL runs considerably more slowly on sentences containing relatively few "closed class" words in the sense of [Thorne1968].)

Thus while it is conceivable that one can get by without syntax (though what this means exactly is surely open to debate), even if one does so one is faced with culling out the structure desired from a huge choice without the benefit of syntactic information to reduce the search space.

BIBLIOGRAPHY

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

Earley, J. 1968. An Efficient Context-free Parsing Algorithm. Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa.

Fahlman, S. 1973. A Planning System for Robot Construction Tasks. AI TR-283, MIT, Cambridge, Massachusetts.

Griffiths and Petrick, S. 1965. On the Relative Efficiencies of Context-free Grammar Recognizers, CACM 8, 5, 289.

Kuno, S. 1965. The Predictive Analyzer and a Path Elimination Technique. CACM 8, 7, 453-462.

Pratt, V.R. 1969. Translation of English into Logical Expressions. M.Sc. Thesis, University of Sydney.

Pratt, V.R. 1973. A Linguistics Oriented Programming Language. Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, Calif., August 1973.

Schank, R., L. Tesler and S. Weber, 1970. "Spinoza II—Conceptual Case Based Natural Language Analysis", AI-109, Stanford University, Stanford, California.

Thorne, J., P. Bratley and H. Dewar, 1968. The Syntactic Analysis of English by Machine. In Machine Intelligence 3, Michie, D. (ed.)

Valiant, L. 1974. Technical Report on a fast context-free parsing algorithm, Carnegie-Mellon University, Pittsburgh, Pa.

Winograd, T., 1971. "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language", Project MAC TR-84, MIT, Cambridge, Massachusetts.

Woods, W.A., 1969. "Augmented Transition Networks for Natural Language Analysis", Report No. CS-1 to the NSF, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts.