DATALOGILABORATORIET
Inst f informationsbehandling
Uppsala universitet

1974-07-05 DLU 74/20

Program structure and process look-ahead in language understanding programs

by

Erik Sandewall

Abstract: The parsing and assimilation (= internalization) aspects of language understanding programs are considered. Several approaches for writing such programs are discussed with respect to the structure that they support and encourage, in the program and the grammar. A new feature in programming languages, called process look-ahead, is suggested to be useful in some cases for structuring the grammar/program.

Sponsor: This research was supported in part by the Swedish Natural Science Research Council (Dnr 2654-009)

A program which analyzes sentences in a non-trivial subset of natural language, and relates them to a data base, must necessarily be quite complex. It is therefore an important issue how to structure language understanders so that they are easy to write, change, and extend. Some of the more specific problems involved are:

1. Subdivision of the task into separate sub-tasks. For example, one might decide to separate out "parsing" or "inference", although smaller and more specific modules might also be useful. Regardless of how the modules are chosen, there necessarily must be a large number of interdependencies between them. It is desirable that the interdependencies should be few, but it is more important that both modules and interdependencies should be clearly defined. It is also well known that efficiency, as well as good structure within the modules, may be hampered if the task is broken down in an unsuitable way.
2. Choice of notation, e.g. a programming language, which encourages well-structured programs, and avoiding notations which are supposed to encourage "bad" programs. The goto statement is of course the primary example of the latter, in the opinion of many. It has also been suggested (Sussman 1972) that fully automatic backtracking encourages programs which, amongst other vices, are hard to debug.
3. Understanding the "character" of the problem in the right way, and choosing a notation which supports it. For example, suppose a programming task which naturally requires recursion such as formal differentiation, is tackled by a programmer who has not heard about recursion and the techniques for implementing it, and who does not

invent it. He is then likely to come up with a complex and badly-structured program, compared to the recursively written program that would have been possible.

This paper considers some approaches to language understanding systems with respect to program structure. In particular, the paper proposes one language feature which, although much less dramatic than recursion, still is claimed to help structuring certain problems in an appropriate way. The feature can be characterized as look-ahead in the computation. One example is given to show the proposed feature in a practical context.

There have been several approaches to language understanding programs. In the <u>classical</u> approach the program was organized in several "passes". The first pass was a <u>parser</u> which used a grammar, and usually also a lexicon, but nothing else, to transform the sentences to a parse tree. It was followed by an <u>assimilation</u> (or internalization) routine, which related the parse tree to the data base, found referents to anaphora, resolved ambiguities that could not be handled within the sentence, etc. Both the first and the second pass had to scan through the whole sentence, or its associated structure descriptor. They were followed by a third, <u>operative</u> step which took an action on the assimilated parse. In the simple-minded question-answering application, the operative step stored assertions in the data base, and retrieved the answer for questions. It was often not necessary for the third pass to scan the whole sentence. The classical approach has been described e.g. by Palme (1971).

The classical approach has some advantages, for example that one can rely directly on linguistic models such as transformational grammars for the parsing step. However, it also has some obvious disadvantages, mainly that it assumes that the parser should operate without contact with the data base, generate some proposed "readings" of the sentence, and refer them all to assimilation. Several of the branches traversed by the parser in its search, might have been cut much earlier if parsing and assimilation could be done in an interleaved rather than a sequential manner. Conventional parsers quickly become overloaded because of the combinatorial explosion of possible "readings", and the only efficient way to avoid the explosion is through the use of semantic knowledge of various kinds (semantics of individual words and phrases, knowledge of the domain of discourse, etc.). Such information should be brought to bear on each part of the sentence as soon as it has been parsed.

Such considerations led to the programmed grammar approach, where Wood's Augmented Transitional Network Grammars (Woods, 1969) and Winograd's PROGRAMMAR (see Winograd, 1972) led the way. In this approach, the "grammar" is expressed in a specialized programming language. Teh grammar/program operates on a sentence and may make decisions about structure (parsing steps), query the lexicon or the data base (probably part of parsing), and extend or modify the data base (assimilation) and its discretion.

What program structure does such a system encourage? According to Woods (personal communication), a recommended method for writing an augmented transitional network (ATN) grammar is:

1. describe an approximation to your desired input language using a context-free grammar. Often the approximation is a crude superset.
2. re-write this context-free grammar as network grammar rules, or at

least, keep it in the back of your mind when you write the network grammar.

- make local (in some sense) modifications to the network grammar in order to improve efficiency. Some permissible operations on a grammar are described in (Woods, 1969), but many others are possible. Often the modifications serve to merge several would-be productions into one composite structure. Thus modularity is preserved although with a greater granularity.
- insert appropriate checks which relate proposed parses to various semantic information, e.g. in the data base. This serves the double purpose of increasing efficiency and fitting the grammar more closely to the desired input language.
- insert operations which construct the desired output from the parse (e.g. using the ATN primitive BUILD), and if desired, operations which effect the data base.

This method of organizing ATN grammars, which we shall call the standard one, encourages a reasonably good grammar structure. This will be further discussed in a later section.

Woods's system or very similar ones have been used by a number of other reasearchers for writing grammars. (Winograd's system has also been used, although apparently to a smaller extent). During the past few years, there has also been some new trends. One important idea is to incorporate non-determinism as a feature into the "host" programming system, usually the LISP system. This approach has some obvious advantages over the ATN parser, which is an extra level above the programming system. It has been used in a parser at SRI (Paxton and Robinson, 1973).

Wilks (1973) has described a language analyzer whose design is new in several ways. The input text is first cut up into fragments, which are often only a part of a sentence. Each fragment is analyzed independently and converted to a coherent structure, and then these structures are integrated. To analyze a fragment, each word is exchanged for its semantic definition (or a choice of several such); a pattern-matching operation is performed whereby the fragment is classified and assigned a structure, and links between the semantic definitions of the words in the fragment are found. The parse is considered more successful if there are many such links.

It should be clear that Wilks' approach is radically different from both the classical approach and the programmed grammar approach. From the structuring viewpoint, it seems à priori that it should be able to do as well as a programmed grammar. However, since the programmed grammar approach is in relatively common use, and since Wilks' system has not yet been much tested, we shall restrict our continued discussion in this paper to the programmed grammar approach, and in particular, to ATN grammars. Some alternative solutions within the latter approach will be discussed toward the end of the paper.

Program structure in the classical approach

The classical approach allowed well structured programs and grammars. The strict division between parsing and assimilation was probably an advantage from the structuring viewpoint, at least in systems with a limited ambition.

Its chief deficiency was in efficiency, although of course a grammar-writer of programmer might be tempted to sacrifice grammar structure in attempts to regain some of that efficiency. The parser was typically driven by a grammar, which consisted of a number of productions and/or transformations, each of which could have a reasonable intuitive interpretation.(Some of the large transformational grammars would seem to be counter examples of this, at least in the eyes of an outsider, but this might be because the grammar was overloeaded with tasks which could otherwise have been handled by the assimilation step). Finally, the assimilation of a parse tree can be described as a set of transformations associated with the symbols which mark non-terminal nodes in the parse tree, where again the transformations typically have an intuitive interpretation.

The modularity of assimilation operators, and their place in the system, shall be exemplified with a simple procedure for assimilating noun phrases formed with the definite article. Using the approach in (Sandewall 1971, 1972), the parser is assumed to map the sentence into a tree with lexical items at the leaves and "function" words (within or outside the language, but chosen from a fixed and relatively small vocabulary) and the non-terminal nodes. For example, the sentence "The dogs that John brough, are barking," might map into the structure shown in figure 1.

- - - - - - - - - - - - - - - - - - - -

insert figure 1 about here

- - - - - - - - - - - - - - - - - - - -

Assimilation is performed bottom-up as an evaluation, where terminal nodes evaluate to themselves, and non-terminal nodes apply the procedure named

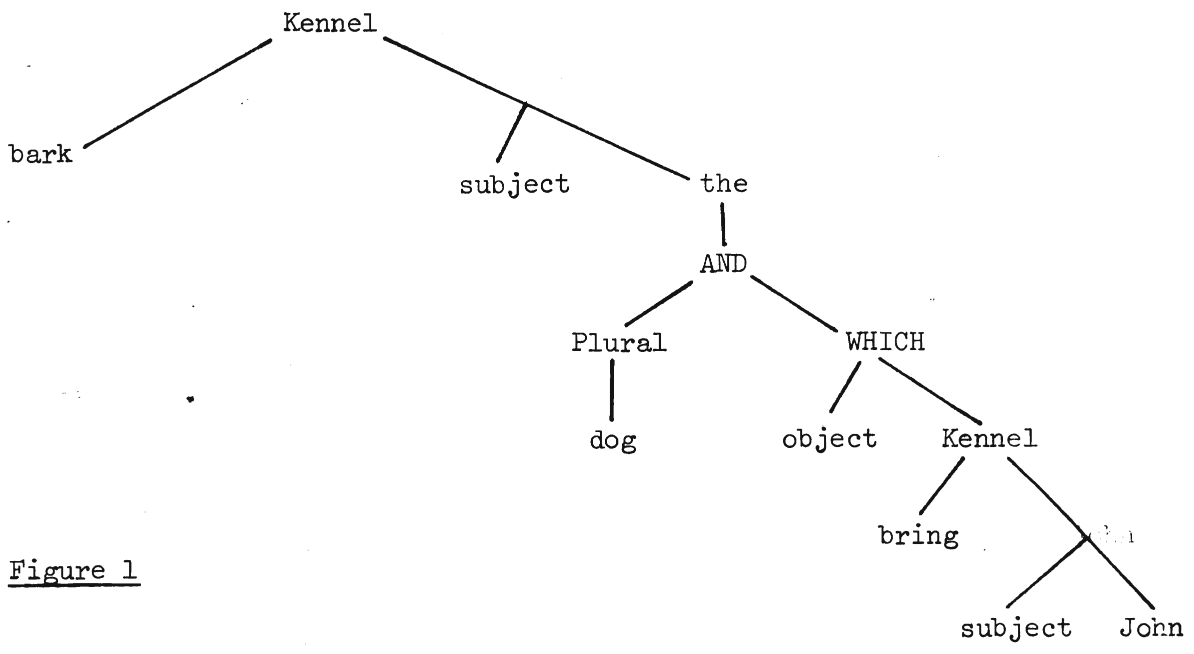

Figure 1

by the symbol in this node, to the evaluated sub-trees below the node. Thus in evaluating "the father of my mother", represented by the structure in Figure 2.

- - - - - - - - - - - - - - - - - - -

insert figure 2 about here

- - - - - - - - - - - - - - - - - - -

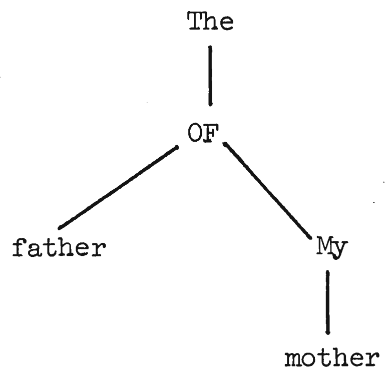

Figure 2

we do the following evaluations:

1. father evaluates to itself
2. mother evaluates to itself
3. My(mother) evaluates to some node n which names my mother
4. OF(father,n) evaluates to some structure p which characterizes the property of being n's father.
5. The(p) evaluates to some object which has or obtains the property p.

Assimilation of the full sentence results in some structure a, and if this sentence was given as an assertion, the operative step of the program might choose to store a in the data base, i.e. store a link between a and the (node for the) present context, indicating that a holds in the context. The storage operation might also trigger some forward deduction.

The definite article is associated with a procedure the(p,cx), where the argument p is a property, and the second argument cx is optional, and specifies the context in which the denoted object has the property p. In practice, the procedure must represent an idealized version of the English definite article in the sense that there are some very difficult cases which can not be handled. The procedure is non-deterministic, and tries a number of cases, of which the following are the most important ones:

1. (Main case). A list of lately mentioned objects is scanned to find some which has (have) the property p. The exact choice of which objects to consider depends on the position of the phrase The(p) in the sentence's parse tree. The default choice of context in which the object has the property also depends on the position of the phrase.
2. (Default case). If not other case applies, a new object o is created,

assigned the property p in a suitable context, and returned.

3. (Example: "The father of John" as in Fig. 3). If the argument p is an expression headed by the operator OF, which corresponds to some

- - - - - - - - - - - - - - - - - -

insert Figure 3 about here

- - - - - - - - - - - - - - - - - -

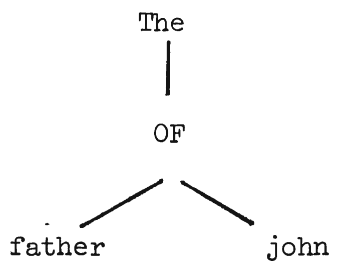

Figure 3

usages of the English word of, then case (1) is permitted to consider the whole data base, rather than just a list of lately mentioned objects. If an object with the property p still can not be found, case (2) applies.

4. (Example: "This car is no good. The brakes don't work, and ...").
If some object o is in what we call "focus" from the preceding context, then it is checked whether OF(p,o) has a plausible interpretation. If so, The(OF(p,o)) is evaluated and may return a value, or values. Rumelhart and Norman (1973) consider this case along with our preceding cases, and call it "foregrounding" rather than "focus".

Ideally, exactly one value should be returned from one of these cases, but in practice this will not always be so. Particularly when case (4) applies, the non-determinism should be used in such a fashion that the process returns one value but remains dormant, so that it can later be recalled and asked to return some other value. This is accomplished e.g. in Conniver with the AU-REVOIR primitive (McDermott and Sussman, 1972).

The four cases that have so far been specified are particularly convenient to work with since they are "local", i.e. they only require access to the "arguments" = the evaluated subtree below the node labelled The. The finer points in case (1) are an exception to this. There also remain some other cases which are not local, such as:

5. (Example: "John is the author of this book"). We admit structures such as in figure 4, where o is an object and p a property. The

- - - - - - - - - - - - - - - -

insert Figure 4 about here

- - - - - - - - - - - - - - - -

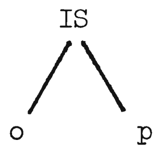

Figure 4

given example results in a parse tree whose structure is that of Figure 5. The operator the should then be the identity function and return its evaluated argument. It should also have the side-effect of noticing

- - - - - - - - - - - - - - - -

insert Figure 5 about here

- - - - - - - - - - - - - - - -

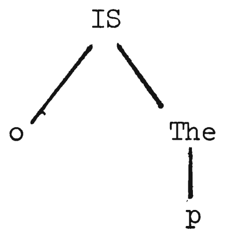

Figure 5

that only one object has the property that its argument denotes.

6. (Example: "He thought that the author of the book was dead"). In the referentially opaque case in modal constructions, we do not want <u>The(p)</u> to evaluate to an object which has the property <u>p</u> in the main context of the text, but instead to an object which "has" this property in a certain other context (e.g. some person's beliefs).

The following case may have to be handled separately, or can be treated as a special case of (6), depending on which system of representation is chosen for the data base:

7. (Example: "If a hungry cat meets a mouse, then the cat is likely to eat the mouse"). In expressions whose logical counterpart obviously require variables, English often introduces each variable using the indefinite article, and then uses the definite article or some equivalent (e.g. a pronoun) for further references to the same variable.

In case (5) and in some possible solutions to case (7), the procedure associated with the name <u>the</u> must look "upwards" and perhaps "sideways" in the parse tree, and can not be restricted to look "down" at the arguments. The natural way to implement case (6) is to let the procedure for <u>the</u> have access to a free context variable that can be bound higher up in the evaluation (assuming a LISP-type evaluation scheme). One might also imagine cases where an operator wants to look at its unevaluated arguments. We shall refer to all of these as <u>non-standard evaluation schemes</u>.

It is well known how to implement such non-standard schemes; the implementation problems will only arise in the programmed grammar approach. Also, non-standard schemes do not in themselves disturb the structure of the assimilation "program", and do not offer any particular temptation use "bad" programming practices. The assimilation program can remain a collection of procedures, associated with symbols in the parse tree, and both symbols and procedures can hopefully be assigned their own, intuitively satisfactory interpretations.

The definite article is used in this paper to illustrate some basic concepts with regard to assimilation operators. Quite naturally, the cases that have been given here are not sufficient as a linguistic description of the use of the definite article in English. Our claim is instead that assimilation of sentences can be structured with advantage as an evaluation,that the definite article could correspond to one assimilation operator, and that problems because of non-determinism and the need for non-standard evaluation schemes are characteristic for assimilation operators.

Modularity in the programmed grammar approach

Parsers for programmed grammars can in general be viewed as interpreters for non-deterministic programming languages, so they can in principle be used to implement any approach one chooses to use. Specific approaches therefore correspond to specific ways of organizing the grammar/program.

In particular, one could use a programmed grammar to implement the parsing step of the "classical" approach. The sentence is then scanned in at least two of the "passes": parsing and assimilation. From the point of view of modularity, this solution is not much different fbom the classical approach considered in the preceding section; the only difference being (perhaps) that network grammars tend to have larger modules then production grammars.

At the other extreme, there is the case of "one-pass" systems, where the syntactic analysis is integrated with all assimilation operations. There is finally an intermediate case, namely a two-pass system where some operations that would have been relegated to assimilation in the classical approach, are performed in the first, grammar pass. Such operations could include the use of semantic information about individual words, word meanings, and phrases. Other operations, such as determining referents, are saved to the second pass.

While the classical approach always assumed two passes through the sentence, the programmed grammar approach can therefore be used with either one or two passes. (In counting passes, we now do not consider the operative step, where e.g. questions are answered, and the possibility that it would scan the internalized sentence). In choosing between the one-pass and the two-pass approach, the issue is clear: If it is very useful to assimilate phrases early in the sentence, and use the resulting information when parsing other parts of the sentence, then one prefers the one-pass system. If it is more useful to parse the latter parts of the sentence, and use the information derived therefrom in assimilating earlier parts, then one chooses the two-pass approach.

The usefulness of assimilated information for the continued parsing is well recognized. We shall therefore discuss one-pass parsers with some detail, in order to find out whether and how information from continued parsing can be useful for assimilation.

A reasonable method for writing a one-pass grammar is the following: Design an ATN-grammar according to the standard method given earlier in the paper, excluding assimilation operations such as finding referents. (The resulting grammar would certainly be useful for the two-pass approach). Then modify the grammar by inserting calls to assimilation procedures in appropriate places. As a rule of thumb, whenever a two-pass grammar would create a sub-expression, often using a BUILD command, the one-pass grammar should evaluate that sub-expression immediately.

In one-pass grammar, it is important that the non-determinism required by assimilation, and the [redacted] non-determinism of the parser interact correctly. This might be easy or very difficult, depending on the parser system and its host programming language. Apart from this, the method is straight-forward as long as the assimilation operators do not need any non-standard evaluation schemes. Nothing is then changed compared to the two-pass approach except the order of execution. However, as even the simple examples above made clear, non-standard schemes are needed even for moderately difficult sentences. The question is then whether they can be provided within the grammar/program system.

In ATN terms, the following is required. Each level in the parsing process (a new level is created by each PUSH operation) must have access to stack functions, whereby it can find out where it is being called from, and access registers on higher levels. Furthermore, to account for situations as in case (6) of the <u>the</u> procedure ("he thought that the author of the book was dead"), one level must be able to re-set a register e.g. for context, and lower levels must be able to access the value.

Such facilities exist, or can easily be implemented in current systems. However, they will only work as long as information is created on the higher (or side-ordered) level before it is accessed. For a concrete example, suppose the "sentence" level of a grammar is able to recognize when the main verb implies a modality, and sets a register to indicate this, and suppose further that the definite article procedure, called from a lower level which analyzes a noun phrase, accesses this register. Things then go well in sentences like "he thought that the author of the books was dead": "He though that" sets the register, which then may be used by the process that takes care of "the author of the books". But it is easy to change the example so that the information is provided only <u>after</u> it is needed: "The author of the books is dead, I think".

The other two examples of sentences which require non-standard evaluation schemes can be modified similarly: "The author of the book is not known" is again a statement about the property of being author of the book, and therefore the noun phrase should not evaluate to its referent. In parsing "A hungry cat will try to catch any mouse it sees", as long as the grammar has only parsed "a hungry cat", it does not know whether the general or the specific sense is intended.

From a program structuring viewpoint, this is the kind of problem for which one must find an appropriate programming technique or language feature, and where (as with recursion) the programmer is otherwise likely to write a badly structured program in trying to deal with the problem. One candidate method is to let the lower level split into several computational branches, each anticipating one possible continuation on the higher level, and to arrange that the higher level suppresses all branches except the right one. This method is however not fully satisfactory, partly because the same lower-level "subroutine" might sometimes be used when the higher-level information is already accessible, and therefore only needs to be checked, and sometimes when it is not yet accessible. Another reason is that the same higher-level information might be needed several times on lower levels.

The systematic solution is instead to define the parser/interpreter so that a variable may be accessed regardless of whether it has been assigned a value or not, and if it does not yet have a value, the system takes the responsibility for splitting the computation into several branches, some of which are suppressed when the value is finally set. The primary need is for the ability to inspect future values of variables (or "registers", to use ATN terminology) in this fashion. In some cases, one might also wish to access other information that will become available in the future computation, such as the fact that the process executes a certain statement in the program (or passes through some node in the ATN grammar network). It is therefore appropriate to use the term <u>the process lookahead feature</u>, which indicates that the executing process uses information that will become available in its own future operation.

Process lookahead would fit well into very-high-level languages, but does not exist in present systems. In QLISP (Reboh and Sacerdoti, 1974) for example, an attempt to access an undefined variable may be considered as a "fail" (causing a backtrace) or may obtain a quantity "undefined" which satisfies f(undefined) = undefined for most functions.

The implementation of process lookahead raises some interesting problems. First, it is not obvious that exactly one of the branches will eventually survive. After a logical test on future information, which causes the process to branch, each branch may behave differently, and it is possible that each branch becomes satisfied. Or else, each branch might develop so that the entrance criterium is not satisfied - i.e. no branch survives.

Another interesting question is the execution order for the branches. We still consider the case where a higher level has called a lower one, and where the lower level tests for information which will become available later, on the higher level. We then have two or more alternative ("OR-connected") branches on the lower level, and this group is "AND-connected" with the continued computation on the higher level. One strategy is to immediately continue the higher-level computation: it might sometimes be possible to do this without awaiting the result from the lower level. Ideally, one could continue on the higher level until the accessed information becomes available. This is however only possible in situations where one would not lose anything by a two-pass approach (i.e. the one-pass approach does not gain anything).

In other cases, one of the lower-level branches may be the normal one, while the other one(s) are exceptional cases. It is then natural to proceed with the normal branch on the lower level, return to the higher level and proceed there, and backtrace only if the normal choice was not satisfied. The sentence "The author of the book is dead, I think" is likely to cause such a backtrace. Furthermore, if the posteriori check fails, it provides heuristic information for the decision where to backtrace to, namely one other of the branches set up in the process lookahead.

A concrete example

Process look-ahead has been motivated by general considerations. In order to check the usefullness of this language feature for structuring

[illegible]

[illegible] an ATN grammar, an experiment was performed where Cedvall's ATN compiler (Cedvall, 1974) was imcremented with a provisional implementation of process lookahead. An ATN grammar for a fragment of English was then written for this system.

Process lookahead for register contents was implemented as follows:
For each register which might be accessed before it had been set, a special PREDICT arc was added early in the network. The PREDICT arc stated the possible values ~~that the register could take, and was~~
that could be assigned to the register, and was implemented to branch into several processes, one for each value. Also, the later would-be assignments to such a register were changed into DECIDE operations which checked that the register contained the right value, and fail-ed if this

condition was not satisfied.

This implementation was clearly quite inefficient, but it did serve the intended ~~purpose of enabling us to write a grammar using the intended~~ purpose of checking out a grammar ~~written with process~~ which assumed process lookahead.

Some other minor features also had to be added to the parser; a GETRR primitive which accesses a given register on any super-ordinated level (which otherwise would have had to be done with SEND operations), and a facility whereby a lower level could find out where it is called from on the higher level. The latter feature was also implemented in a makeshift fashion without changing the parser, simply by adding operations to arcs in the network which set a dedicated register FROM with the name of the node that the arc proceeds from.

The grammar was written according to the method suggested above, i.e. assimilation operations were added after the grammar proper had been specified. The grammar is described in Figues 6-7, using the following notation:

Full lines mean TO transfers (normal case), dotted lines mean
  Jump transfers.

Double lines mean PUSH arcs, i.e. call to sub-levels. In this case the
  distinction between To and JUMP is irrelevant.

WRD and MEM arcs (i.e. checks for a specific word or choice of words)
  are indicated by quoting the words. Other arcs are CAT arcs.

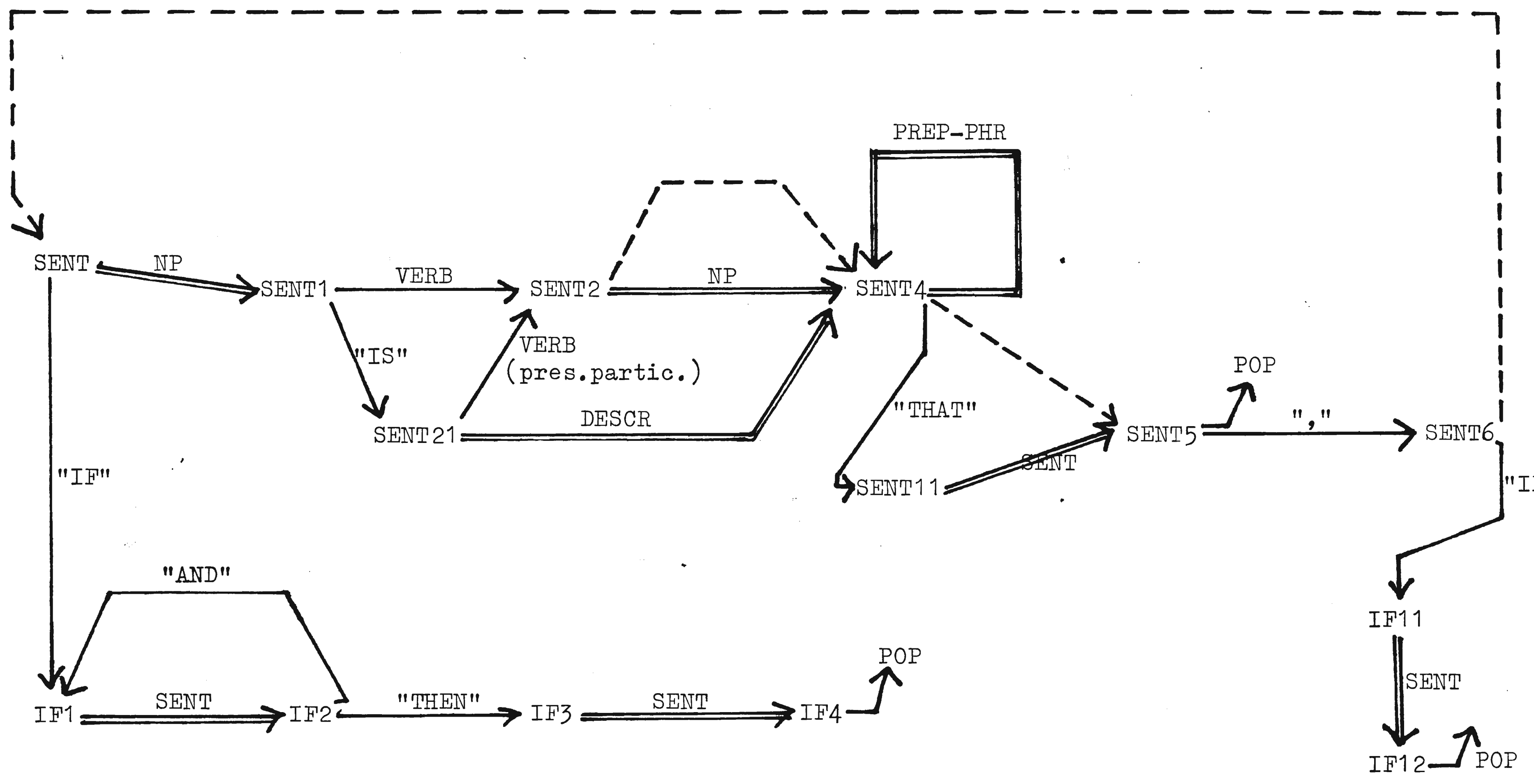
SENT
NP
SENT1
VERB
SENT2
NP
SENT4
PREP-PHR
"IS"
SENT21
VERB
(pres.partic.)
DESCR
"THAT"
SENT11
SENT
SENT5
POP
","
SENT6
"IF"
IF11
SENT
IF12
POP
"IF"
"AND"
IF1
SENT
IF2
"THEN"
IF3
SENT
IF4
POP

Figure 6

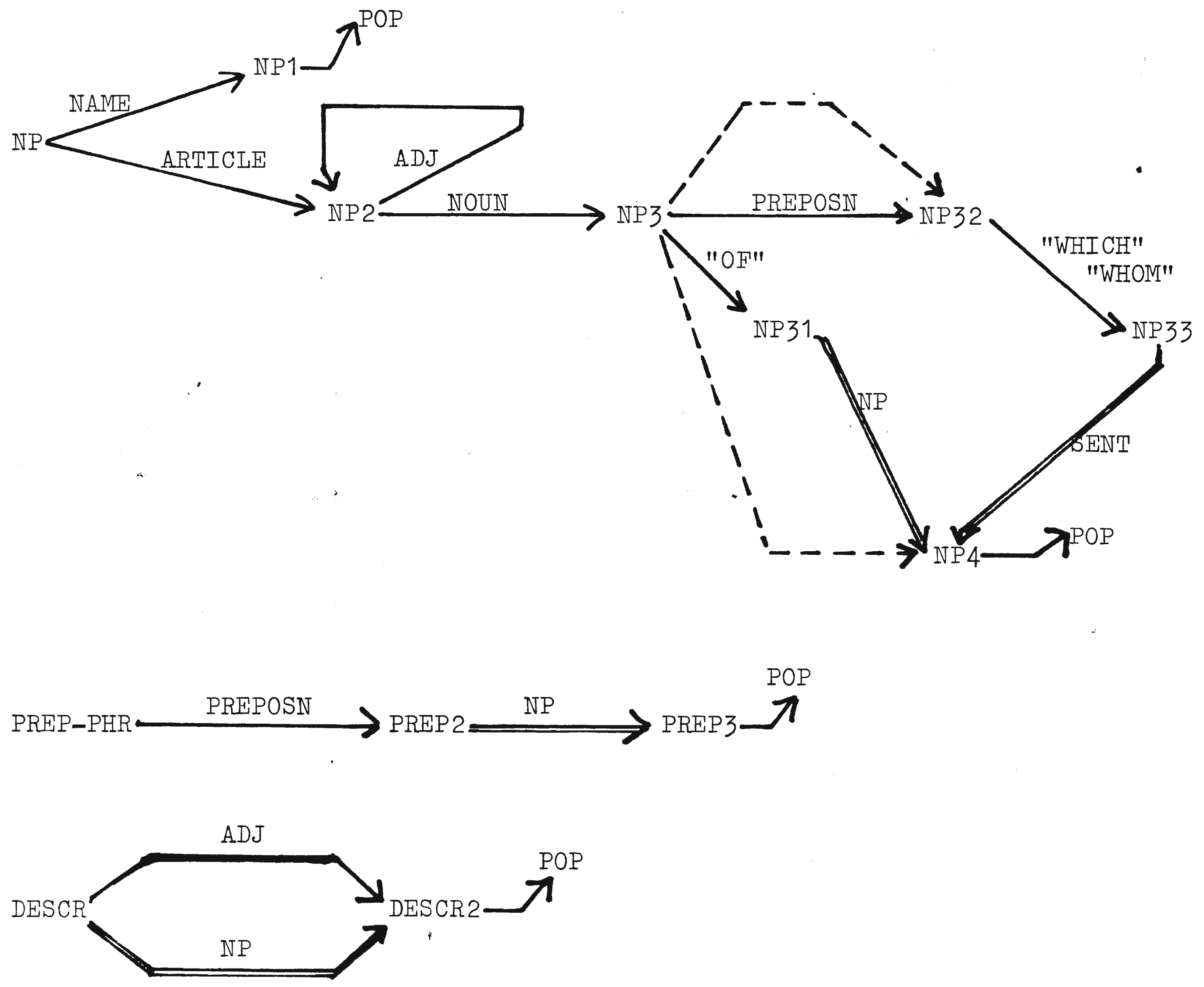
NP
NAME
NP1
POP
ARTICLE
ADJ
NP2
NOUN
NP3
PREPOSN
NP32
"WHICH"
"WHOM"
NP33
"OF"
NP31
NP
SENT
NP4
POP
PREP-PHR
PREPOSN
PREP2
NP
PREP3
POP
DESCR
ADJ
NP
DESCR2
POP

Figure 7

Conditions tested for, and assignments to registers, are not indicated in the figures, but are not essential to the understanding of the grammar. They are used in the obvious way, i.e. successive components in the sentence are accumulated on each level, and then used to compose a resulting expression in the POP (=RETURN) operation.

- - - - - - - - - - - - - - - - - -

insert Figures 6 and 7 about here

- - - - - - - - - - - - - - - - - -

After this, operators for the definite and indefinite article in one-pass usage were implemented along the lines suggested in the preceding section. The following observations were made:

The assimilation operators needed two pieces of information from the higher level. One was the "context" of the immediately super-ordinated sentence. Teh main sentence was considered as <u>regular</u>; the subordinated sentence in e.g. "I think that ..." or "..., I think" was considered as <u>modal</u>, and the clauses in "if... then...." was considered as <u>conditional</u>. Since the implementation of lookahead required a small and fixed number of cases, only these three were used. In a practical system one would of course wish to structure them, and use "in situation s" for "regular"; "in the mind of person A" for "modal", etc. This would however not essentially affect these assimilation operators.

The other information needed from the higher level was the position and usage of the noun phrase. This required access to which higher-level arc the lower level was being called from, and also to a register that the higher level had to set, to distinguish e.g. "the author of the book is angry" from "the author of the book is not known". This was called the subjmode register.

The context register was sometimes used after, sometimes before it had been logically set. The subjmode register was always used before it was logically set.

There was no need for any essential changes of the grammar network in order to implement the assimilation operators. Assignments to the context and subjmode registers were associated with arcs late in the network, where the proper value had been determined. The operators for the articles could then be programmed in a clean and self-contained way. Only, because of the makeshift implementation, it was necessary to add two nodes at the beginning of the sentence-level network in order to make the tentative assignments to the registers that could be accessed in process lookahead.

Thus the hypothesis that a pure grammar/program structure could be retained in the one-pass system, in spite of the need for non-standard evaluation schemes, and that the lookahead feature would be of use, was confirmed for this example.

Two somewhat unexpected things occurred. First, for subordinated sentences it was often possible to set the context register from outside, in the call (using a SENDR operation). In parsing "I think that he is angry", the top level sentence can tell the lower level sentence that it is in a "modal" context. In such cases, the branching that the implementation does at the entrance to the sentence level, shall be suppressed. Also, on exit from the network after parsing "he is angry" (or in general a "standard" sentence) the grammar normally sets the context register to regular, thus informing noun phrases in the sentence who made a prediction about that. But again, it is clear that this late assignment (i.e. check of the provisional value) should not be executed if the context had been imposed from outside.

Another observation was that it was sometimes desirable to "unassign" a register which had already been assigned a value. This happened for sentences such as "He is dead, I think", where the analysis procedes through the beginnings of the sentence, and then realizes (at the comma) that the sentence that has been analyzed so far is a subordinate one. It should then retroactively PUSH the process that has so far been carried out, loop back to the beginning of the present level, and start over with logically unassigned context and subjmode registers.

An alternative solution would have been to make instead a PUSH operation in order to handle the phrase "I think" (in the same example) on a lower level, and then assemble the right structure for the whole sentence when control returns to the upper level. This solution is however impure, and causes trouble at least for (admittedly somewhat wicked) sentences such as "This is a good place, he said, I think".

It is not entirely clear how a general ability to unassign registers should be realized in a definite implementation. In the experiment, it was only set up for a special case.

Modularity in other approaches

In the two-pass programmed grammar approach, where assimilation is delayed to the second pass, the particular difficulty for program structure that has been the concern of this paper does not arise. The major heuristics for obtaining a well-structured grammar is then the normal method described earlier, i.e. to structure the grammar/program at least partly according to a production grammar. Conversly, the structure may deteriorate if the programmer over-uses the possibility of passing information between registers instead of backtracking.

Some of the more recent programmed-grammar-type understanders are of particular interest from the structuring viewpoint. Riesbeck (1973) has written a "conceptual analyzer", i.e. a parser which transforms antural-language sentences to Schankian conceptual dependency graphs. This program dovetails with Rieger's (1974) program for "processing the meaning content of natural language utterances", which internalizes the dependancy graphs into a "memory structure". (But whose main purpose of course is to do more advanced things:) Thus the two-pass approach is quite pronounced in this work.

The BBN speech project (Woods, 1973) uses a grammar which can be simply characterized as an ATN grammar with a facility to start the analysis at any point in the grammar graph, and move both forward and backwards from there. In such a system, it becomes even more important to have a facility where tests using the value of a variable can be handled even if the variable has not yet been assigned. The problem domain of the program (the moonrock query system) presumably does not raise the need for handling articles as discussed in this paper. It is not clear whether it should be classified as a "one-pass" or a "two-pass" system.

The PINTLE parser (Walker, 1974) used in the SRI speech project, is predictive because of the special problems of speech understanding. When a part of a sentence has been analyzed, it is used to generate plausible candidates for the next words. The candidates are then checked against the acoustic information. When this approach is used for non-trivial subsets of language, it seems that one will need as much information as possible from the already parsed parts, and in particular, that it becomes very desirable to use an one-pass system for parsing and assimilation.

Rumelhart and Norman (1973) describe ~~are working with~~ a system for modelling human memory, which includes parsing and assimilation ~~(Rumelhart and Norman, 1973)~~. The parser is implemented as an ATN network, and seems to be an one-pass system. The procedure associated with the definite article is discussed to some length in the given reference, and is treated in a manner very similar to ours. However, only those cases which manage with local information are considered, so the problem discussed in the present paper are automatically excluded.

References

Cedvall, M., Dokumentation av en kompilator och tillhörande runtime-system f,r nätverksgrammatiken. Memorandum DLU 74/14. Computer Sciences Department, Uppsala University, 1974.

IJCAI 1973 = Proceedings of the Third International Joint Conference on Artificial Intelligence. Stanford Research Institute, 1973.

McDermott, D.V. and Sussman, G.J., The Conniver Reference Manual. Memo No. 259, MIT Artificial Intelligence Laboratory, 1972.

Palme, J., Making Computers Understand Natural Language. In Findler, N.V. and Meltzer, B. (Eds.) Artificial Intelligence and Automatic Programming. Edinburgh University Press, 1971.

Reboh, R. and Sacerdoti, E., QLISP, ... (full title to follow). Stanford Research Institute, 1974.

Paxton, W.H. and Robinson, A.E. A Parser for a Speech Understanding System. In IJCAI 1973.

Rieger, Ch.J. Conceptual Memory. Ph.D. thesis, Computer Science Dept., Stanford University, 1974.

Riesbeck, c. Computer Analysis of Natural Language in Context. Ph.D. thesis, Computer Science Dept., Stanford University, 1973.

Rumelhart, D.E. and Norman, D.A. Active Semantic Networks as a Model of Human Memory. In IJCAI 1973.

Sandewall, E.J. Representing Natural Language Information in Predicate Calculus. In Meltzer, B. and Michie, D. (Eds.), Machine Intelligence 6. Edinburgh University Press, 1971.

Sandewall, E.J. PCF-2, a First-Order Calculus for Expressing Conceptual Information. Computer Sciences Dept., Uppsala University (Sweden), 1972.

Sussman, G.J. Why Conniving is Better than Planning. Internal memorandum, MIT Artificial Intelligence Laboratory, 1972.

Walker, D.E. Speech Understanding Through Syntactic and Semantic Analysis. In IJCAI 1973.

Wilks, Y. Understanding without Proofs. In IJCAI 1973.

Winograd, T. Understanding Natural Language. Edinburgh University Press, 1972.

Woods, W.A. Augmented Transitional Networks for Natural Language Analysis. Report No. Cs-1, Aiken Computation Laboratory, Harvard University, December 1969.

Woods, W.A. and Makhoul, J. Mechanical Inference Problems in Continuous Speech Understanding. In IJCAI 1973.