NATURAL LANGUAGE GENERATION FROM PLANS

Chris Mellish

jnitive Science Research Paper

·ial no: CSRP 031

* University of Sussex
piitive Studies Programme
liool of Social Sciences
lmer
ighton BN1 9QN

NATURAL LANGUAGE GENERATION FROM PLANS

(Preliminary Progress Report)

Chris Mellish
Cognitive Studies Programme
University of Sussex

1.0 INTRODUCTION AND RATIONALE

As computers become entrusted with increasingly complex tasks and the necessary software for this becomes increasingly "intelligent", so it becomes increasingly frequent that decisions and actions taken by machines cannot be fully comprehended by single human beings. Hence in parallel with the development of intelligent software there is a need for a corresponding development of facilities by means of which a machine may explain its actual anc planned actions in a way comprehensible to a person. One mode in which suet explanations might be expressed is natural language. However, in spite of the urgent practical need, as well as the fascinating theoretical questions raisec by the subject, relatively little work in AI on natural language processing ha« looked at natural language generation rather than understanding, althougt perhaps this is changing.

This work aims to produce a system that produces natural language "explanations[11] of plans. Although opinions differ on exactly what a plan is, the notion of a plan is crucial to problem-solving in a number of areas of AI, foi instance game playing CWilkins 823 and natural language understanding CWilensk>, 833. It is hoped that plans will provide a rich, yet formally delimited, inpul for a natural language generator. It is also hoped that there are domain-independent strategies for explaining plans and that therefore the resulting program can be transferred between domains with relatively little trouble.

A program that can explain plans in natural language could be used in varioui types of practical systems. First of all, it could be used together with ar intelligent program to provide explanations of that program's planned actions, Secondly, it could be used, in conjunction with a planning program, to helj another agent (probably a human being) to make plans in a complex domain.

As an initial exploration into the problems involved, a first-pass computei program has been written, which in some sense generates natural language explanations of plans. The program has been designed with the second kind 01 application in mind. It is currently set up to work with the domain of (a small subpart of) the POPLOG programming environment CSloman 833. Such a program coulc eventually form a component of an intelligent "help" system for the POPLOt system. This paper describes the current program and some of its limitations.

2.0 APPROACHES TO NATURAL LANGUAGE GENERATION

Before the program is described in detail, it is worth summarising some oi the approaches that have been used in natural language generation systems anc stating the basic philosophy of this project.

McDonald CMcDonald 833 distinguishes between systems that are grammar controlled and those that utilise direct production. The distinction here is whether the program is under the direction of a linguistic component (hypothesising things that can be said and then checking against the "message")

or whether it is under the direction of the "message" itself (this being "evaluated" by a special "text generation" evaluator). Systems in the first category include [Simmons and Slocum 72] and [Goldman 75], whereas the system of [Swartout 83] would be in the second category. As McDonald points out, direct production systems are generally ignorant of grammar, and this limits the smoothness of the output (such systems cannot incorporate general rules about context and embedding); however, they are probably more directed and efficient than the grammar-controlled models. We follow McDonald (and [Davey 78]) in adopting something nearer to the direct production approach but introducing an explicit intermediate grammatical representation.

Given a direct production approach to natural language generation, there is a decision to be made as to whether the generator should work basically in a "top down" or a "bottom up" fashion. The system of [Mann and Moore 81] adopts a bottom-up strategy called "fragment and compose", which involves splitting the message into small fragments and then finding ways of combining these together. [McKeown 82], on the other hand works from high level plans ("schemas") of whole paragraphs, finding fragments that can fill the expectations generated by them. We would like to investigate a top-down approach to language generation, where a whole paragraph (or text...) is planned in advance. In this way, we hope to be able to make use of rhetorical strategies such as found by [Weiner 79].

A final question that should be asked about a natural language generator is to what extent it is supposed to represent a model of human performance. McDonald has heavily constrained the operations performed by his generator with psychological constraints in mind. Our object is, instead of striving for a psychological model, to strive for the generation of the most "readable" text, by whatever means are most appropriate. That is, we are interested in producing a description of how it is possible at all to generate good quality text - what knowledge is needed for this and what processes are involved.

3.0 PLANS, STATES AND ACTIONS

This section briefly outlines the representation of plans, states and actions assumed by the current program. One problem that immediately arises is that, although they are agreed on the general nature of what a "plan" is, AI researchers disagree a lot on the details, and a universally accepted formalism for plans is therefore not available. Perhaps all that is agreed on is something like the following rather vague definition (from [Schank and Abelson 77]):

> "A plan is made up of general information about how actors achieve goals. A plan explains how a given state or event was prerequisite for, or derivative from, another state or event"

If planning programs have no uniform organisation, then a general natural language explainer of plans must make as few assumptions as possible about their output. We have therefore assumed that a planning program generates a desired sequence of actions and also provides information about the goal of the plan and the initial state of the world before the plan is enacted. This is not a plan in the sense of the above definition, and in order to make sense of it, the generator must hypothesise the information that lacks. Thus the program must have available general information about actions and their preconditions and effects. We have chosen to model the program's knowledge of actions on that used by the STRIPS program [Fikes and Nilsson 71]. For example, the following might be the Prolog clause representing the action of "an agent going to a place" (such an action does not exist in the POPLOG domain). (In order to understand this paper, it is not necessary to have a full understanding of Prolog. All the

Prolog clauses presented can be viewed simply as datastructures with various components. Note, however, that names which start with uppercase letters denote variables).

```
operator(go(Agent,Place),
     [at(Agent,PlaceNow),route(PlaceNow,Place)],
     [at(Agent,PlaceNow)],
     [at(Agent,Place)]).
```

The first line of this clause provides the name of the action. The second gives the list of preconditions (the agent must be at some place "PlaceNow" and there must be a route from there to the desired goal). The third gives the propositions to be "deleted" from the world when the action is performed, and the fourth gives the propositions to be added. This example operator illustrates the fact that actions and states in the domain are represented in a simple logic notation (no quantification being allowed).

Our program at present distinguishes between two kinds of effects that an action can produce (whether they be propositions to be added or deleted). These could be called the primary and secondary effects. With the above operator, the usual reason for going somewhere is to get to that place, not to get away from the place one is currently at. Hence the added fact is of primary importance. In an explanation, one is likely to mention secondary effects only when they are relevant to the plan; primary effects, on the other hand, are likely to be mentioned anyway. Primary effects (or "objects") are represented by the program by clauses such as the following:

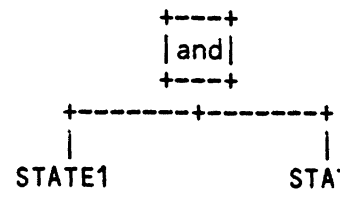
```
object(go(Agent,Place),at(Agent,Place)).
```

With this general representation of operators (actions), from a given initial state of the world and list of desired actions, the program can construct a triangle table for the plan (see [Fikes Hart and Nilsson 72]). Indeed this, rather than the simple sequence of actions, is the representation of the input that it works with.

Occasionally there may arise in the domain of application complex states and actions, or the program may need to reason in terms of such for the purposes of planning the text. Here is the simple grammar for states and actions that is used for this purpose:

```
STATE ::=   [not,STATE]              | STATE is not the case
            [and,STATE1,STATE2]      | STATE1 and STATE2 are both true
            [enabled,ACTION]         | the preconditions of ACTION are true
            [done,ACTION]            | ACTION has just been performed
            [doing,ACTION]           | ACTION is currently being performed
            PRIMITIVE STATE

ACTION ::=  [then,ACTION1,ACTION2]   | doing ACTION1, ACTION2 sequentially
            [achieve,STATE]          | making STATE true
            PRIMITIVE ACTION
```

The square brackets notation used here may not be familiar to all readers. An item like "[and,STATE1,STATE2]" is used to denote a tree structure, where "and" is the label at the top node and "STATE1" and "STATE2" denote the (only) two subtrees. Graphically, such a structure might be displayed as:



Although this grammar of states and actions has not been fully investigated as a formal calculus, there are various equivalences between formulae in this scheme. For instance:

```
[achieve,[done,ACTION]]  ===  ACTION
```

4.0 THE DOMAIN-DEPENDENT DICTIONARY

Since the program is supposed to be able to "explain" plans in any domain and to any person, it must have an extra input which specifies the special features of the domain under consideration and the person who is receiving the explanation. This section briefly explains what form that extra input takes, with some examples. Much of this is currently of an ad-hoc nature.

4.1 Introduction of verbs

It is necessary to specify how primitive states and actions in the domain can be described in English. This normally involves the introduction of verbs. The following Prolog clauses illustrate the specification of verbs for an action and a state in the POPLOG domain, the action of deleting a file and the state of having free space on the disc:

```
lex(delete(F),delete,[agent:user,obj:F]).
lex(disc_space,have,[agent:user,obj:NP]) :-
   NP matches
    [corenp = [headnoun = $$space,
               postadjs =+ [prep = $$on,
                            head = ......]
              ]
    ].
```

The first component of such a "lex" structure is the name of the state o action involved. The second is the verb that can be used to describe it, and th third is an indication of how surface cases of that verb are to be filled in This third component can specify that a case (eg "agent", "obj") can be fille by a noun phrase denoting a particular individual (eg "user" - the hearer), noun phrase denoting an individual mentioned in the state or action (eg "F" i the "delete" case), or a "canned" noun phrase. The specification after the ":- in the entry for "disc_space" is a way of specifying the structure of th special phrase "space on the disc" ("....." indicates more detail that has nc been included here). For more information about this notation see 6.4 below.

4.2 Normal Forms

Sometimes a particular action or state in the domain cannot be expressed in English as a simple (present tense, active) use of a verb. In this case, it can be convenient to consider the action or state as a complex object formed from a hypothesised primitive object that can be expressed as a simple use of a verb (An example of where this is advantageous is given below). Here are some such "rewrite rules" from the POPLOG domain:

```
rewrite(deleted(F),Ldone,delete(F)3).
rewrite(viewing(F),[doing,lookat(F)3).
```

Thus, for example, the state "deleted(F)" is most readily expressed as a passive sentence - "F has been deleted". It is related to the primitive action "delete(F)" which can be expressed by a simple use of a verb.

4.3 Noun Phrase Introduction

The problem of how to choose a noun phrase to mention a particular object is complex, and has not been addressed in this project (so far). Instead, rules are provided that map every possible referent onto a fixed noun phrase. Here are some examples:

```
fill_np(user,NP)  :- NP matches Ccorenp = Cpron = $$youD3.
fill_np(temp,NP)  :- NP matches
                        Ccorenp = Cdet = $$the,
                                  adjs =+ $$temporary,
                                  noun = $$file3
                        3.
```

Here the referent "user" has been mapped onto the pronoun "you" and the referent "temp" has been mapped onto the phrase "the temporary file". More details about the notation used for these phrases is given below in 6.4.

4.4 User Model

One of the prime advantages of having a general natural language generator, rather than a system relying on fixed templates, should be that an explanation generated can be finely tailored to the needs and abilities of the reader. It is not a prime aim of this work to investigate how a person's knowledge of a complex domain can be represented, but obviously the generator needs to be sensitive to this issue. At present, the program's model of the user consists of fixed knowledge about what things the user finds "obvious" - which actions are suggested obviously by certain goals, and which results obviously follow from certain actions. Here is one of the "obviousness" clauses used in the POPLOG domain:

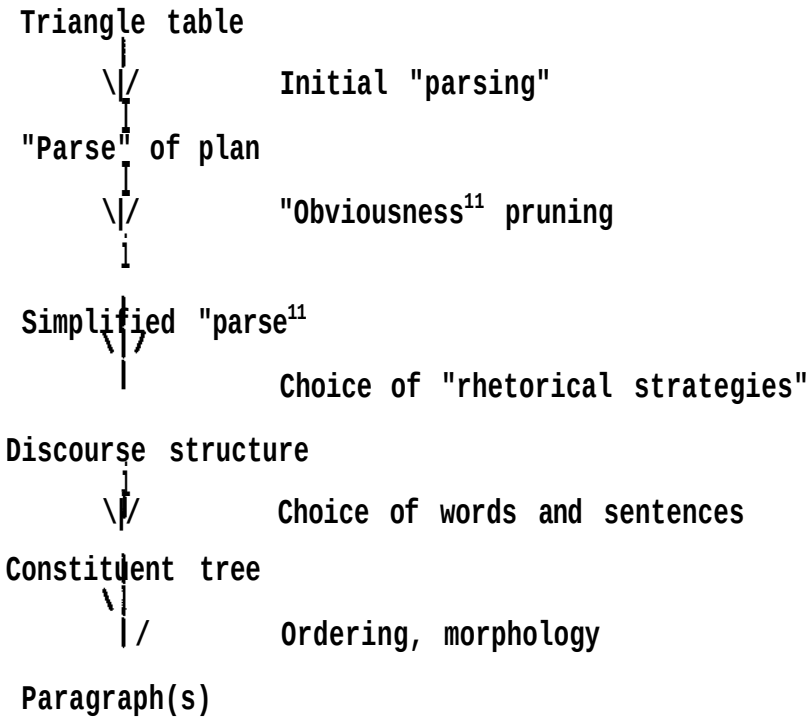
```
obviousjresult(select star(X),Cdoing,display(system,star(X))3).
```

This expresses the fact that it is obvious that if the user marks the name of a file displayed on his VDU screen with an asterisk, then the system will be displaying an asterisk in front of the file name. One advantage of having a natural language generator available is that "obviousness" information can

actually be obtained in advance by presenting the user with English sentences, with which (s)he must agree or disagree.

5.0 OVERALL STRUCTURE OF THE PROGRAM

The stages through which the current program works can be conveniently shown by the following diagram:



Initially, the triangle table must be "parsed" into a representation when the program can see the rationale for each particular action. This is a form o constrained "plan recognition": constrained because the overall goal is given The idea here is that the hypothetical planner may work in unknown ways, bu that only certain kinds of rationalisations for actions can be explained by th generator. It does not matter particularly whether the rationalisation tha arises is like the one used by the original planner - it suffices that i provide a complete justification of the actions.

Having obtained a representation of the important structure of the plan which has involved extracting from the full representation given by the triangl table, the program still may have a relatively large structure to work with. On of the central problems of natural language generation is deciding what to leav out. As Mann and colleagues point out [Mann and Moore 81], "the demands c smooth text production are incompatible with expression of all of the availabl information". The second stage of the program involves pruning parts of the pU that are "obvious" to the reader (in terms of the primitive user model describe in the last section).

The third stage involves looking at the complexity of the parts of tl remaining plan structure and making decisions about strategies for explainir them. For instance, an action that has many complex justifications » preconditions to be achieved may need to be explained by a whole paragraph, wi* other explanations nested within it; in simpler cases, whole strings of actioi can be described by a single sentence. The result of this stage is an outline the structure of the whole discourse to be produced.

The discourse structure is now mapped onto a detailed syntactic ^presentation of the text to be generated. This involves recursively working though the levels of structure, generating constituent trees for the paragraphs and sentences involved. At the lowest levels, actual sentence forms and lexical items are selected. However, at this point the constituent trees contain no ordering information - they are solely a representation of constituent structure.

The final stage involves producing a linear output from this constituent tree. This involves introducing ordering and morphology. At present, ordering rules are stated explicitly for every constituent type; in future versions it would be advantageous to investigate general ordering principles. For instance, it might be useful to use the ID/LP framework of GPSG CGazdar and Pullum 82].

6.0 AN EXAMPLE

In order to provide a non-trivial example of the program working, it is unfortunately necessary to give a brief summary of some relevant features of the POPLOG domain. The example is concerned with the screen editor, VED. It assumes that, although it is possible to be editing several files simultaneously, only one edit "buffer*[1] may be displayed on the user's terminal at a given time (the real VED does not have this restriction). The scenario is as follows:

> The user has been editing a file, and wishes to write it (to disc). Unfortunately, his quota of space on the disc is exhausted, and therefore he must delete some existing files first. There is a VED command which causes the user to be presented with an editor buffer showing the names of his files. Given that, the user can mark the files he wishes to delete and invoke another command to convert this buffer into a set of operating system commands to delete the file(s). These commands can then be executed. At this point, the user is still looking at the (now transformed) directory listing file. In order to write the original, he must quit this new file and resume editing the old one.

A relatively naive user of POPLOG might wonder how to react when (s)he receives an error message on attempting to write the first file. An intelligent HELf system might produce a plan (essentially as above) for getting the file written, It could then be explained by a program of the kind presented here.

In terms of our representation of the POPLOG domain, the goal of the plan is:

> written(file)

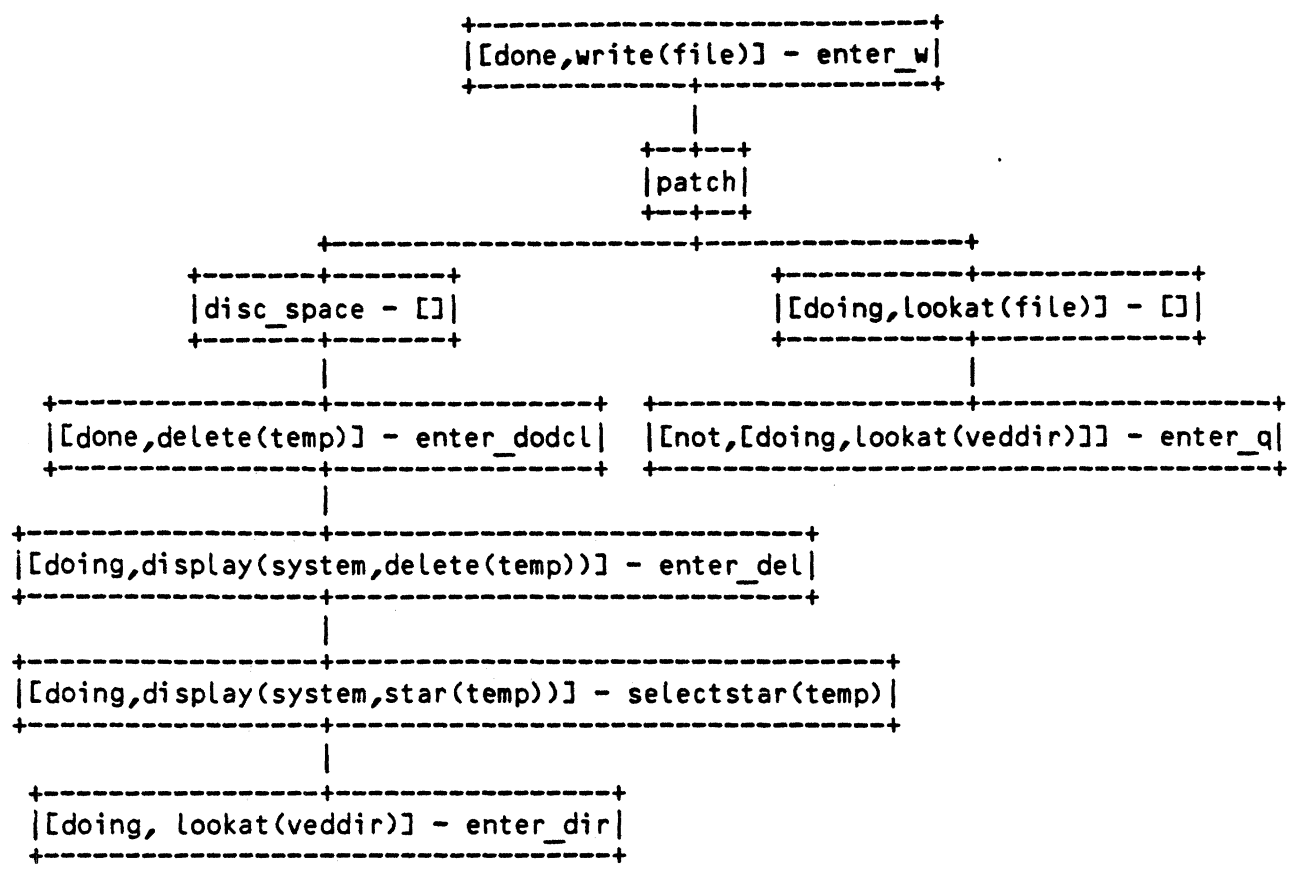
and the triangle table looks as follows. The actions are shown in sequence witl brief explanatory comments. Before each action is a sequence of lists (delimiter by 'C[1] and '3[1], with items separated by commas). These lists represent the fact: that are true just before the action is performed. The last "row" indicate: those that were true initially and have not changed; the second from last roi indicates those that became true after the first action, and so on.

> [in_ved, viewing(file), vedbuffer(file), lastved(file), file(temp)]
>
> enter_clir ;;; get directory listing
>
> Lviewing(veddir), vedbuffer(veddir), buff_has(veddir,temp),

```
      lastved(veddir)]
    [in_ved, vedbuffer(file), file(temp)]

selectstar(temp)    ;;; mark files to be deleted

    [buff_has(veddir,star(temp))]
    [viewing(veddir), vedbuffer(veddir), lastved(veddir)]
    [in_ved, vedbuffer(file), file(temp)]

enter_del           ;;; convert to os commands

    [buff_has(veddir,delete(temp))]
    []
    [viewing(veddir), vedbuffer(veddir), lastved(veddir)]
    [in_ved, vedbuffer(file), file(temp)]

enter_dodcl         ;;; execute os commands

    [deleted(temp), disc_space]
    [buff_has(veddir,delete(temp))]
    []
    [viewing(veddir), vedbuffer(veddir), lastved(veddir)]
    [in_ved, vedbuffer(file)]

enter_q             ;;; quit directory buffer

    [viewing(file), lastved(file)]
    [deleted(temp), disc_space]
    [buff_has(veddir,delete(temp))]
    []
    []
    [in_ved, vedbuffer(file)]

enter_w             ;;; write original file

    [written(file)]
    [viewing(file), lastved(file)]
    [deleted(temp), disc_space]
    [buff_has(veddir,delete(temp))]
    []
    []
    [in_ved, vedbuffer(file)]
```

.1 Initial "parsing"

nformation is now extracted from this representation to form a "parsed' epresentation of the plan, in which justifications for actions are mad xplicit. This looks as follows:



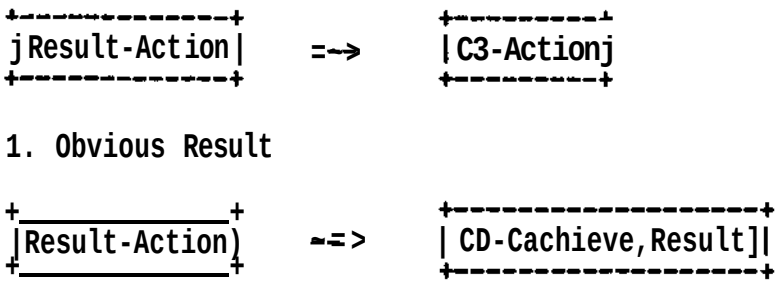
Each node of this tree structure (except the "patch" annotation) has two parts (shown separated by a '-'). The first is a state and the second is an action. The meaning of the node is that at some point this action needs to be performed in order that the state be achieved. For the action to be enabled, however, the states associated with the child nodes must have been achieved (in the left-right order shown). Sometimes the action is marked empty ("[]"). This indicates that the resulting state is made true automatically by the performing of the actions below it in the tree. In the terminology of section 3.0, the state is a secondary effect of the actions. This plan has a fairly simple structure: the main action has two preconditions, and the other actions are parts of "chains" of actions leading up to the attaining of these subgoals.

The "patch" node in the tree is not really a proper node. It serves as an annotation, recording the fact that the second subgoal (that of looking at the file to be written) was already satisfied originally, but has to be resatisfied because the achieving of the first subgoal (that of getting space in the disc) has messed it up. This is Sussman's notion of "prerequisite clobbers brother goal" [Sussman 75].

Note that the topmost goal "written(file)" has been rewritten in normal form "[done,write(file)]" (as have all states and actions).

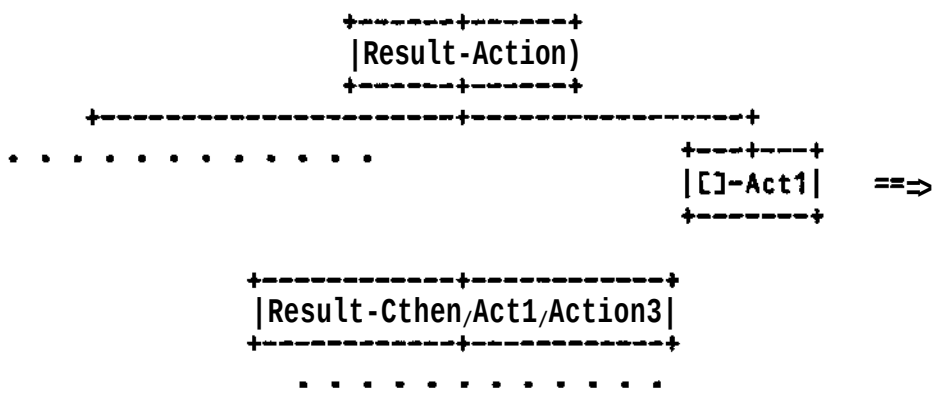
6.2 "Obviousness" Pruning

"Pruning" rules are now applied to this tree, to remove parts that are "obvious" (in the sense of our crude user model). There are three rules used at present; here is a schematic representation of them:



1. Obvious Result

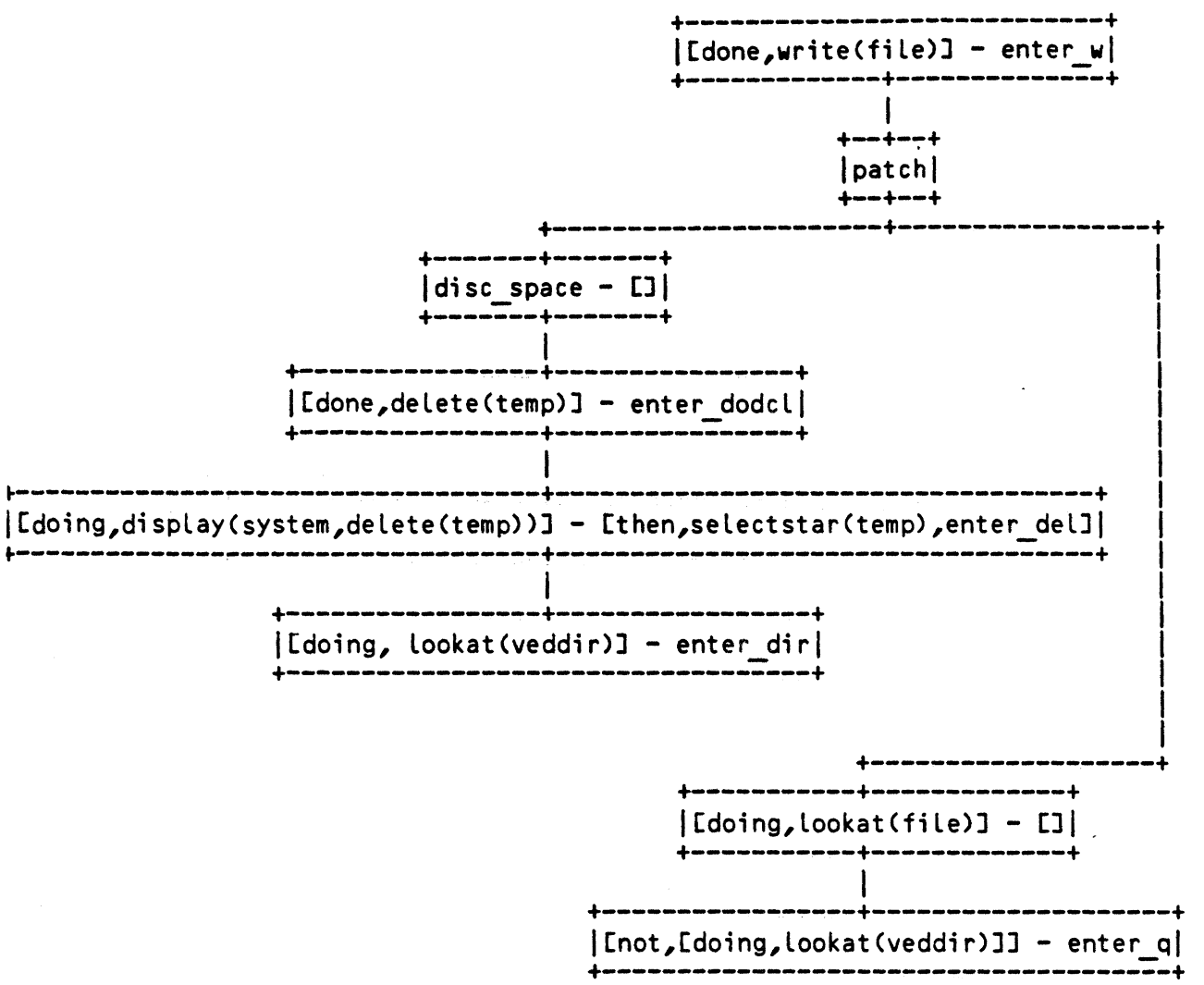
2. Obvious Action



3. Collapsing of Actions

The "obvious result" rule applies if a certain state is an obvious result of an action. It then simply deletes the state from the description, with the implication that it need not be mentioned in the justification. The "obvious action" rule applies when there is an obvious action to achieve a given goal. Ir this case, the action is not deleted, but rather is replaced by the (in general vaguer) action to "achieve the state"; the state need not now be mentioned a; the result. The program originally simply deleted the action in this case However, although it is quite possible to leave out descriptions of intermediate states in an explanation of what to do, leaving out actions results in a strangi kind of story where states change without any apparent cause. Finally th» "collapsing of actions" rule collects together sequences of actions that are t< be performed without mention of intermediate results.

After this "pruning" stage, the plan looks as follows:



All that has happened here is that the obvious result of

[doing,display(system,star(temp))] - selectstar(temp)

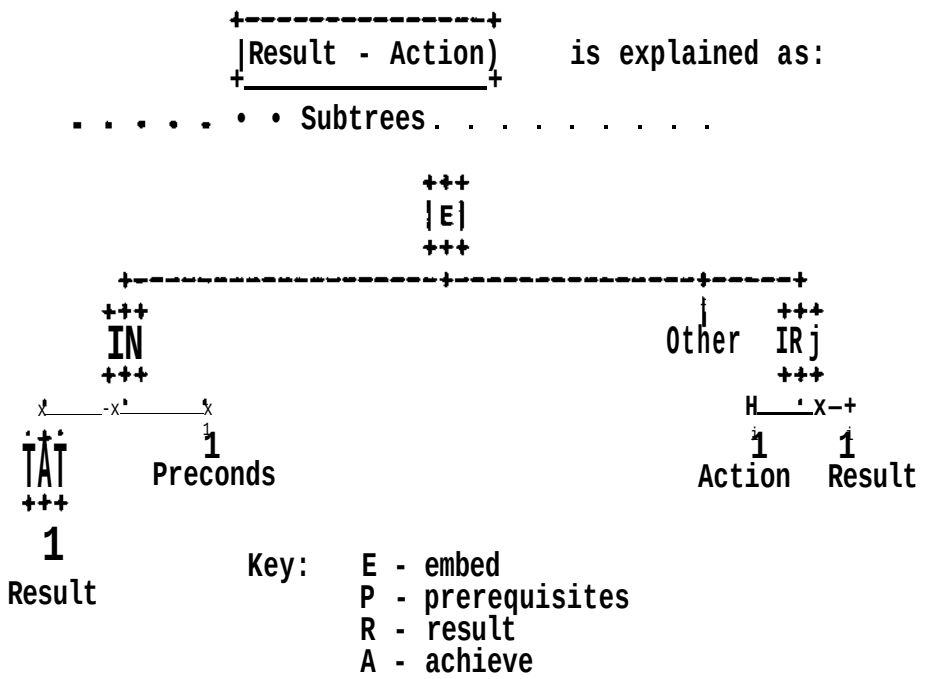
has been deleted, and the action has been combined with the next action up the tree (this is the "obviousness" case discussed in section 4.4). Of course, given a different representation of what was obvious to the reader, different operations might have been performed.

6.3 Choice of Rhetorical Strategies

Having "pruned" surplus information from its representation of the plan, the program now plans the overall "shape" of the discourse to be generated. For this it makes use of a simple "grammar" for discourses, as follows:

```
DISCOURSE ::= [embed,DISCOURSE1,DISCOURSE2,DISCOURSE3]
              [contrast_sequence,DISCOURSE1,DISCOURSE2]
              [sequence,DISCOURSE1,DISCOURSE2]
              [do,ACTION]
              [result,ACTION,STATE]
              [prerequisites,ACTION,STATE]
```

At this stage, the program must Look at patterns in the plan representation and decide whether they are best expressed as single sentences, sequences of sentences or perhaps as complex paragraphs, with sub-explanations embedded within explanations. A state-action pair that has more than one precondition to be established will generally give rise to an "embed[11] structure, where the main goal is introduced ("DISCOURSE1"), then the subgoals are described ("DISCOURSE?[1]), then a concluding sentence ("DISC0URSE3") recapitulates on the achievement of the main goal; a sequence ("chain") of actions, each feeding the next in a simple way, will generally give rise to a "sequence" structure, where the first action and its result are described ("DI^COURSEI"), followed by the others in sequence ("DISC0URSE2"); if there is a "patch" annotation, this might be changed to a "contrast sequence", where some kind of contrastive conjunction is needed between the sentences. Here is a simplified representation of one of the rules used;



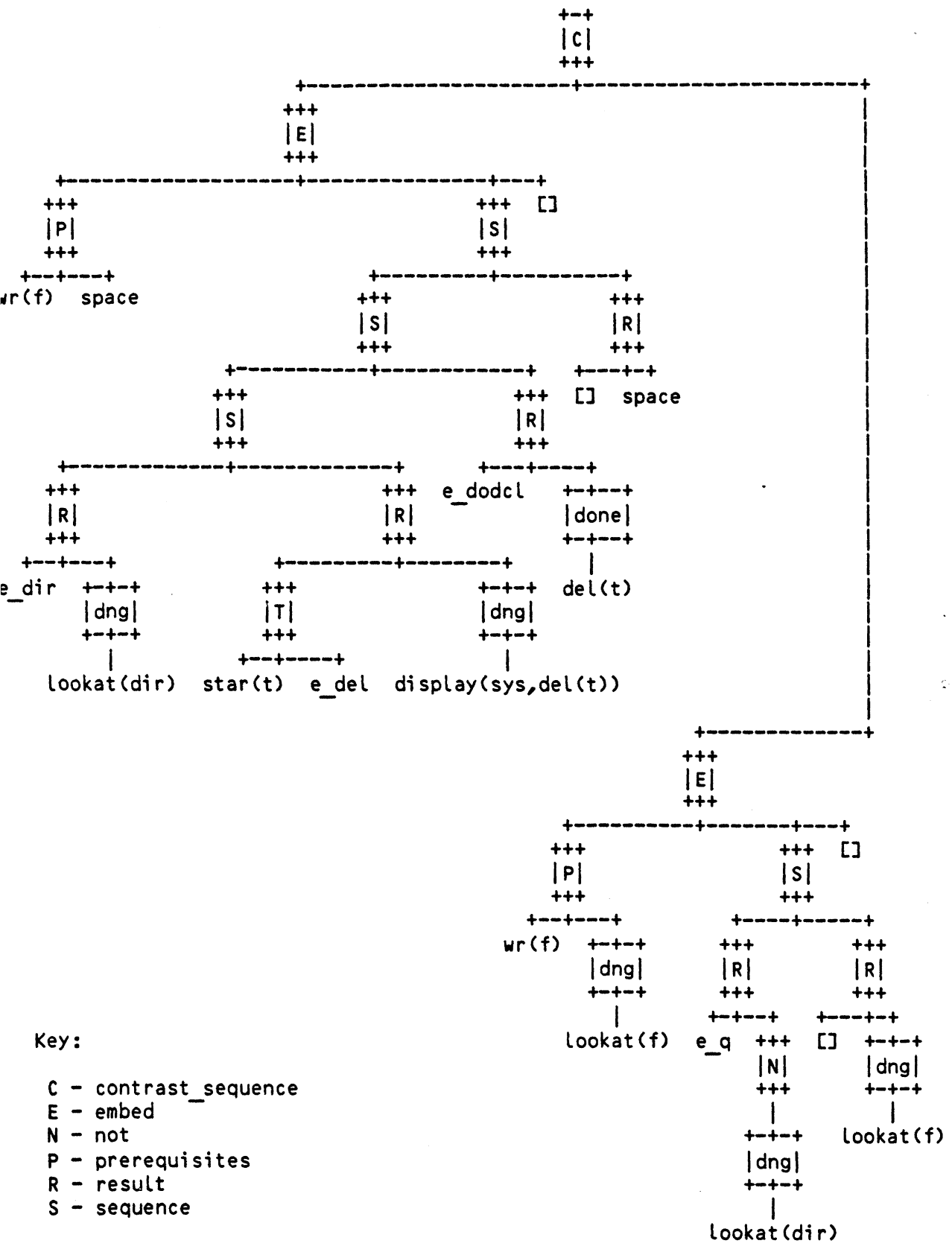
Example discourse planning rule

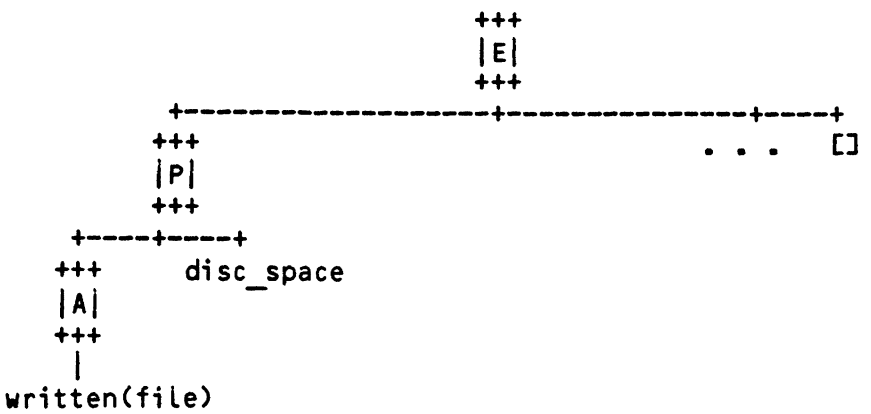
Not mentioned in this diagram are the extra conditions: there must be mor than one subtree, there must not be a "patch" annotation, "Preconds" is th conjunction of states achieved by the subtrees, and "Other" is the explanatio generated for the subtrees.

The structures at the base level of the discourse structure generated (abov the level of (possibly complex) states and actions) are of three possible forms "Cdo,ACTIOND" involves saying that some ACTION must simply be performec "[result,ACTION,STATE:r involves a statement that doing a particular ACTIC causes STATE to result, and "[prerequisites,ACTION,STATED" involves stating th prerequisites (STATE) for some given ACTION.

The top level of the plan in our example is treated roughly as specified I the above rule; that is, a structure like the following is generated for th* part of the plan:

NATURAL LANGUAGE GENERATION FROM PLANS





A straightforward rendering of the "Prerequisites" part of this structure (if written(file)" was defined in the domain-dependent dictionary to give rise to the file has been written") would be something like:

> In order to get the file to have been written, you must have space on the disc.

ortunately, by expressing "written(file)" in the normal form, in terms of an ction that can be rendered by a simple use of a verb, this expression can be implified. That is, since

> written(file) === [done,write(file)],
>
> [achieve,written(file)] === write(file)

y the identity discussed in section 3.0. The sentence can therefore come out s:

In order to write the file, you must have space on the disc.

The result of planning the complete discourse to explain the plan is the ollowing large structure, where for lack of space the states and actions have ad to be abbreviated:

6.4 Choice of Words and Sentences

Our natural language generator works in a strongly "top down" mode, planning the structure of the whole explanation and only filling in individual sentences at the last minute. This has implications for how we organise the process of choosing actual words and sentences. In general, the global planning of the discourse might impose quite vague constraints on a sentence. For instance (in a more advanced system than this), it might be that in a particular sentence the focus must change from one object to another; this in itself is not enough to pin down the exact syntactic structure tSidner 793, but it does impose constraints on what structures can be chosen. On the other hand, it might be decided (again, in a more advanced system than this) that the sentence must be in the passive voice in order to have parallel structure with some previous sentence. In general, these constraints can interact in subtle ways, and one needs a framework where one can:

(1) record vague information about the syntactic and semantic structure

(2) have immediate consequences of such constraints worked out

(3) have inconsistent constraints detected quickly

Two grammar formalisms have been especially popular with researchers into computer natural language generation, largely because they allow for the independent specification of different features of a sentence/ discourse and provide simple mechanisms for investigating the consequences of such decisions. They are Systemic Grammar and (more recently) Unification Grammar (this is Appelt's term CAppelt 823; it is essentially Martin Kay's [Kay 793 Functional Grammar).

It is very hard to weigh up the advantages and disadvantages of these two frameworks. Possibly one should adopt Systemic Grammar, as grammars within this framework are more developed (eg. CWinograd 723, CMann??3). The approach taken here is inspired largely by Unification Grammar, however, partly because certain aspects of this were straightforward to implement (much of Kay's notion of unification is similar enough to logical unification that it is easy to implement in Prolog).

In Unification Grammar, one represents information about a particular sentence by a functional description, which is basically a statement about attributes of the sentence and their values (the information about a filler will itself in general be a functional description). The grammar can also be expressed as a functional description, which may also specify alternative structures and the necessary sharing of values between different attributes (possibly in different phrases). Given a partial functional description for a sentence, the action of matching it against the grammar functional description ("unification") has the effect of validating the "grammaticality" of the sentence and filling in any attributes whose values follow from those already given. It is important to note that attributes in a unification grammar may be of a semantic, as well as a syntactic nature.

McKeown's CMcKeown 823 use of a unification grammar is essentially as described above. That is, a partial functional description for a whole sentence is built and then "unify"ed with the grammar. This leads to a rather expensive,

non-deterministic "unification[11] process at the end. Our approach is to check each piece of functional description as it is generated. This may possibly involve non-determinism, if a specified description is very vague, but in general the kinds of descriptions that we want to specify seem not to introduce choices (especially if the grammar is organised with this in mind). As yet, we have only a very primitive grammar and can only handle sharing of attributes between a phrase and its parent, however, and so we do not have a solid basis for comparison.

Here is a simplified example of a rule for converting from part of the discourse plan (as above) into a functional description for the text. It illustrates the notation that we use for functional descriptions and gives an example of where the global structure dictates certain aspects of a sentence in advance.

```
filUCprerequisites,Action,State],Paragraph) :-
   Paragraph matches
      tconjn = Croot = 'in order[1]],
       first s [mode = non_finite,
                aux = pres-to~inf],
       rest  = Cmode = finite,
                aux = pres-must~inf]
```

This rule states that a "prerequisites" type structure gives rise to a paragraph with a "first" sentence, a conjunction and a "rest" sentence. The conjunction is "in order", the first sentence must be non-finite with "to" and an infinitive, and the other sentence must be finite with a "must" auxilliary. An example "paragraph" of this form would be:

In order to make an omellette you must break eggs

6.5 Ordering and Morphology

The functional description (or constituent tree) generated in the last stage contains no ordering information. Kay provides a special notation for expressing ordering constraints in functional grammar, but we have not adopted this Rather, there are specific rules for each kind of constituent dictating the order in which the parts should appear. The program therefore recursively walks round the functional description, generating words according to these strict rules. It is at this stage that morphological processing is done (although the program is grossly defective in this respect). One could imagine noticing parallel structures and introducing conjunction and ellipsis at this point as well.

Here, at Last, is the "English" text generated for our example:

> in order to write the first file you must have space on the disc, type ENTER_DIR and you yill be looking at a directory listing, then mark the temporary file with a asterisk then type ENTERJ&EL and the system will be displaying delete(temp). then type ENTERJ>ODCL and you will have deleteen the temporary file, then you will have space on the disc"
>
> however in order to write the first file you must be looking at the first file, type ENTERJ1 and you will not be looking at a directory listing, then you will be looking at the first file

There is obviously scope for improving the appearance of this text cosmetically - for starting sentences with capital letters, having word endings generated properly and for replacing various "pseudo-words" with English expressions. Overall, the text is fairly comprehensible (to somebody fairly familiar with the pOPLOG domain), although certain features of it suggest that more radical improvements might be necessary.

7.0 PROBLEMS AND LIMITATIONS

Now that we can examine the output of the program, certain limitations of the current approach become apparent. The program can actually generate several alternative "explanations" of a plan (roughly in order of merit), and some of the problems are only apparent if we look at the "second best" solutions; nevertheless, some of these features are things that should not be present in any output of the program. Here is a list of some of the current problems with the program:

(1) No use of pronouns. Natural discourse makes great use of pronouns and other anaphoric devices. Because it cannot introduce pronouns, the program can generate sentences like "in order to write the first file you must be looking at the first file" which is at best clumsy, at worst misleading. We plan to introduce focus annotations at the first stage of planning of the overall discourse structure, and hope to capitalise on the work of Grosz CGrosz 77] for the design of this planning stage. For how to interpret annotations about the desired focus of given sentences, we hope to capitalise on the work of Sidner CSidner 793.

(2) Repetition of constructions. It is quite possible for the program to generate deeply nested explanations using the same format at each level. For instance, consider the following: "In order to write the first file you must have space on the disc. In order to have space on the disc you must have deleted the temporary file. In order to delete the temporary file ...". Such nested explanations can actually be excluded for this example plan, by suitable tightening of the discourse planning rules. However, sometimes explanations do need to be nested. In such cases, the different levels should be marked by the use of different constructions and suitable paragraph layout to make the structure clearer. Weiner CWeiner 79] notes the function of various small words like "anyway" in making clear the structure of an explanation.

(3) Lack of general information. The reader is told how to solve his particular problem, but is not given general rules that will help him solve similar

problems in future. The insertion of statements of general rules (according to the known knowledge of the user) plays an important part in a good explanation. The program needs to be able to explain general rules, and make use of such strategies as explaining by example (and analogy?). There is no obvious reason why this should not be possible within the current framework.

(4) Poor generation of NPs. As has been noted, noun phrases are all "canned" in the current system. Sometimes they can be replaced by pronouns, as noted above, but there remain situations where a full noun phrase is needed to refer to an object. The question arises as to what information should be used in this description. The descriptions of some objects can be taken from the goal, for instance "the file to be written"; for other objects (such as the temporary file used in our example) it is necessary to have some indication of why they appear in the plan. Perhaps the planner should be expected to provide more information about this.

(5) Absence of negative commands and warnings. The only actions described are those that lead to the overall goal being achieved - there is no inclusion of information about sequences that don't work or sequences that can have dangerous consequences (like accidentally marking the wrong file for deletion). Again, one requires a richer interface with the planner in order for this information to be available. If the inclusion of information of this sort is required, then the idea of integrating the natural language generator with the planner must be taken very seriously.

8.0 REFERENCES

Appelt, D. E., "Planning Natural Language Utterances", Proceedings of the US National Conference on Artificial Intelligence, 1982.

Davey, A., Discourse Production: A computer model of some aspects of a speaker, Edinburgh University Press, 1978.

Fikes, R. E., Hart, P. E. and Nilsson, N. J., "Learning and Executing Generalised Robot Plans", Artificial Intelligence 3, 1972.

Fikes, R. E. and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence 2, 1971.

Gazdar, G. and Pullum, G. K., "Generalised Phrase Structure Grammar: A Theoretical Synopsis", Cognitive Studies Research Paper CSRP 7, University of Sussex, 1982.

Goldman, N. M., "Conceptual Generation", in Schank, R. C. (Ed), Conceptual Information Processing, North Holland, 1975.

Grosz, B., "The Representation and Use of Focus in Dialogue Understanding", SRI Technical Note 151, Menlo Park, California, 1977.

Kay, M., "Functional Grammar", in Procs of the 5th Annual Meeting of the Berkeley Linguistics Society, Berkeley, 1979.

McDonald, D. D., "Natural Language Generation as a Computational Problem: An

Introduction", in Brady, M. (Ed) Computational Theories of Discourse, MIT Press, 1983.

Mann, W. C. and Moore, J. A., "Computer Generation of Multiparagraph English Text", AJCL Vol 7, No 1, 1981.

McKeown, K. R., "Generating Natural Language Text in Response to Questions about Database Structure", PhD Thesis, Computer and Information Science, University of Pennsylvania, 1982.

Schank, R. C. and Abelson, R. P., Scripts, Plans, Goals and Understanding, Erlbaum, 1977.

Sidner, C. L., "Towards a Theory of Definite Anaphora Comprehension in English Discourse", PhD Thesis, Artificial Intelligence Laboratory, MIT, 1979.

Simmons, R. F. and Slocum, J. "Generating English Discourse from Semantic Networks", CACM 15(1972), 891-905.

Sloman, A., "POPLOG: A Multi-Purpose, Multi-Language Program Development Environment", Cognitive Studies Programme, University of Sussex, 1983.

Sussman, G. J. A Computer Model of Skill Acquisition, Elsevier, 1975.

Swartout, W. R., "XPLAIN: A System for Creating and Explaining Expert Consulting Programs", Artificial Intelligence 21, 1983.

Weiner, J. L., "The Structure of Natural Explanation: Theory and Application", Report SP-4035, System Development Corp, 1979.

Wilensky, R., Planning and Understanding: A Computational Approach to Human Reasoning, Addison Wesley, 1983.

Wilkins, D. E., "Using Knowledge to Control Tree Searching", Artificial Intelligence 18, 1982.

Winograd, T., Understanding Natural Language, Academic Press, 1972.