American Journal of Computational Linguistics Microfiche 33

PROCEEDINGS

13TH ANNUAL MEETING

ASSOCIATION FOR COMPUTATIONAL LINGUISTICS

2: LANGUAGE GENERATION SYSTEMS

Timothy C. Diller, Editor

Sperry-Univac
St. Paul, Minnesota 55101

PREFACE

The papers comprising this microfiche (the second of five) present in expanded form (as submitted by their authors) the six talks given in Session 2: Language Generation Systems. Various aspects of generation are considered, among them: relationships between parsing and generation (Knaus), planning modules and data structures basic to story development (Meehan), semantic networks and linguistic generators (Shapiro and Slocum), message structures and translation strategies (McDonald), and lexical processes in compound noun formation (Rhyne). Thanks to Martin Kay for chairing this session.

Timothy C. Diller

Program Committee Chairman

TABLE OF CONTENTS

American Journal of Computational Linguistics *Microfiche 33 : 4*

A Framework for Writing Generation Grammars for Interactive Computer Programs

DAVID MCDONALD

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

This paper outlines the stucture and operation of the linguistic component from a language generation system in an interactive program. The component receives messages describing what is to be said formulated in the representation of the main program and produces fluent English utterances appropriate to the current discourse situation. The component is data-directed and uses a procedural grammar, organized as a set of strategies.

Interactive, specialist programs presently under development will need to produce fluent, intentional English utterances in responce to particular, complex situations. This creates a requirement for language generating facilities that is not faced in transformational grammar, mechanical translation programs, or paraphrase generating programs. As a component of an interactive, specialist program, the production of the English must be driven <u>directly</u> by the communicative intentions of the program and by the discourse situation.

We can imagine that the overall program consists of a number of cooperating modules - for parsing and interpreting what is said to it, for solving problems in its domain, for managing its memory, and, in particular, for generating utterances to communicate with its users. This generation component can be profitably viewed as having three aspects or "sub-components".

1) Situation/domain specialists that are activated when the program recognizes what situation it is in. They then decide what message will be produced. They will decide what effect on the listener is desired, and exactly what objects and relations are to be mentioned.

   For example, an appointment scheduling program might be told to "schedule a group meeting for Friday" and then find that a critical member of the group is unavailable. The situation specialists in the scheduling program are the ones to decide whether it is more appropriate to simply say "I can't", or whether to volunter information - "I can't; Mitch won't be back until Monday".

2) Models of the audience and the discourse situation to use in constructing utterances. There must be a record of the past conversation to guide in the selection of pronouns. Also, the program must have models of, and heuristics about what the audience already knows and therefore doesn't have to be told. This information may be very specific and domain dependent. For example, in chess, one can say "the white queen could take a knight". There is no need to say "a black knight", because this information is supplied by inferences from what one knows about chess - inferences that the speaker assumes the listener shares.

3) Linguistic knowledge about how to construct understandable utterances in the English language. Obviously, this information will include a lexicon associating objects and relations from the main program with strategies for realizing them in English (particular words, phrases,

syntactic constructions, etc.). There is also a tremendous amount of information which describes the characteristics of the English language and the conditions of its use. It specifies the allowable arrangements of strategies and what modifications or alternatives to them may be appropriate in particular circumstances.

Of the three aspects just described, my work has concentrated on the third. What follows is drawn from my thesis (McDonald '75) and from ongoing research.

The Linguistic Component

The linguistic knowledge required for generating utterances is put into one component whose job is to take a message from the situation specialists and construct a translation of that message in English. The messages are in the representation used by the main program and the situation specialists. The translation is done by a data-directed process wherein the elements and structure of the message itself provide the control.

The design of the linguistics component was arrived at independent of any particular main program, for the simple reason that no programs of adequate complexity were available at the time. However, at the present time a grammar and lexicon is being developed to use with at least two programs being developed by other people at MIT. They are an appointment scheduling program (Goldstein '75) and an advisor to aid users of MACSYMA (Genesereth '75). The short dialog below is an example of the degree of fluency we are hoping to eventually achieve. The dialog is between a scheduling program acting as an appointment secretary (P), and a student (S).

(S) I want to see Professor Winston sometime in the next few days.
(P) He's pretty busy all week. Can it wait?
(S) No, it can't. All I need is his signature on a form.
(P) Well, maybe he can squeeze you in tommorrow morning. Give me your name and check back in an hour.

Messages

Using the current message format and ignoring the details of the scheduler's representation, the phrase "maybe he can squeeze you in tommorrow" could have come from a message like this one, put together by one of the situation specialists.

```
Message-1      features= ( prediction )
     event      (event  actor <Winston>
                        action <fit person into full schedule>
                        time <31-10-75, 9am-12am>)
     hedge      <is possible>
     aim-at-audience  hedge
```

Messages have features describing the program's communicative intentions - what sort of utterance is this to be; what effect is it to have. Messages list the objects to be described (the right hand column) along with annotations for each object (left hand column) to show how they relate to the rest of the message. The phrases on the right in angle brackets represent actual structures from the scheduler with those meanings.

The Lexicon

Translation from the internal representaiton of a computer program to natural language has the same sort of problems as translating between two natural languages. The same concepts may not be available as primitives in both representations, and the conventions of the target language may require additional information that was not in the source. Generally speaking translation cannot be one for one.

What English phrase is best for a particular element in a program's message will depend on what is in the rest of the message and of what the external context is. In such circumstances, translation by table-lookup is inadequate. In this component, in order to allow all factors to be considered, the translation of each element is done by individualized procedures called "composers".

For each main program that the linguistic component becomes associated with, a lexicon must be created which will list the elements of the main program's representation that could appear in a message (i.e. "prediction","event","<Winston>", etc.). With each element is recorded the composer that will be run when the time comes to produce an English description for it (examples will be given shortly). Some composers may be applicable for a whole class of elements, such as "events". They would know the structure that all events have in common (e.g. actor, action, time) and would know how to interpret the idiosyncratic details of each event by using data in the lexicon associated with them.

The Grammar - strategies

The bulk of the grammar consists of "strategies". Strategies are associated with particular languages rather than with particular main programs as composers are. A given strategy may be used for several different purposes. A typical case is the strategy use-simple-present-tense: a clause in the simple present ("prices rise") may be understood as future, conditional, or timeless, according to what other phrases are present.

Each composer may know of several strategies, or combinations of strategies which it could use in describing an element from the message. It will choose between them according to the context - usually details of the element or syntactic constraints imposed by previously selected strategies. The strategies themselves do no reasoning; they are implemented as functions which the composers call to do all the actual construction of the utterance.

The Translation Process

At this point, the outline of the data-driven translation process can be summarized. A message is given for translation. The elements of the message are associated in a lexicon with procedures to describe them. The procedures are run; they call grammatical strategies; and the strategies construct the English utterance.

Of course, if this were all there was to it, the process would never run, because all of the subprocesses must be throughly coordinated if they are not to "trip over their own feet", or, for that matter, if ordinary human beings are to be able to design them. In a system where the knowledge of what to do is distributed over a large number of separate procedures, control structure assumes central importance.

Plans

Before describing the control structure, I must lay out some additional aspects of the design of the linguistics component. Messages are translated directly into English surface structure form. There is no interlingua or intermediate level of structure comparable to the deep structures of Transformational Grammar, or the semantic nets of Simmons (73) or Goldman (74).

Determining the appropriate surface structure, however, requires planning, if for no other reason than that the message can only be examined one piece at a time. The entire utterance must be organized before a detailed analysis and translation can get underway. As this is done, the "proto-utterance" is represented in terms of a sort of scaffolding - a representation of the ultimate surface structure tree insofar as its details are known with extensive annotation, explicit and implicit, to point out where elements that are not yet described may be positioned, and to implement the grammatical restrictions on possible future details as dictated by what has already been done.

The scaffolding that is constructed in the translation of each message is called its "plan". Plans are made up of syntactic nodes of the usual sort - clauses, noun groups, etc. - and nodes may have features in the manner of systemic grammar (Winograd '72). Nodes have subplans consisting of a list of named slots marking the possible positions for sub-constituents, given in the order of the eventual surface structure. Possible slots would be "subject", "main verb", "noun head", "pre-verb-adverb", and so on. The syntactic node types will each have a number of possible plans, corresponding to the different possible arrangements or sub-constituents that may occur with the different combinations of features that the node may have. Depending on the stage of the translation process, a slot may be "filled" with a pointer to an internal object from the message, a syntactic node, a word or idiom, or nothing.

The translation process

The translation is done in two phases. The second phase does not begin until the first is completely finished. During the first phase, a plan is selected and the elements of the message are transferred, largely untouched; to the slots of the plan and features added to its nodes. During the second phase, the plan is "walked" topdown and from left to right. Composers for message elements in the plan's slots are activated to produce English descriptions for the elements as they are reached in turn. Both processes are data-directed, the first by the particular contents of the message and the second by the structure of the plan and the contents of its slots.

There are sound linguistic reasons for this two stage processing. Most parts of a message may be translated in terms of very modular syntactic and lexical units. But other parts are translated in terms of relations <u>between</u> such units, expressed usually by ordering or clause-level syntactic mechanisms. The exact form of the smaller units cannot be determined until their larger scale relations have been fixed. Accordingly, the objective of the first phase is to determine what global relationships are required and to choose the plan, features, and positions of message elements within the plan's slots that will realize those relationships. Once this has been done, English descriptions for the elements can be made independent of each other and will not need to be changed after they are initially created.

One of the most important features of natural language is the ability to omit, pronominalize, or otherwise abbreviate elements in certain contexts. The only known rules and huristics for using this feature are phrased in terms of surface structure configurations and temporal ordering. Because the second phase works directly in these terms, stating and using the available heuristics becomes a straightforward, tractable problem.

"Maybe he can squeeze you in tommorow morning"

The rest of this paper will try to put some flesh on your picture of how this linguistics component works by following the translation of the message given in the beginning to the sentence above. The message was this.

```
Message-1     features= ( prediction )
     event     (event actor <Winston>
                      action <fit person into full schedule>
                      time <31-10-75,9am-12am>)
     hedge     <is possible>
     aim-at-audience  hedge
```

The intentional features of a message tend to require the most global representation in the final utterance, because that is where indicators for questions, special emphasis, special formats (e.g. comparison), and the like will be found. By convention then, the composers associated with the intentions are given the job of arranging for the disposition of all of the message elements. The total operation of phase one consists of executing the composer associated with each feature, one after the other.

This message has only one feature, so its composer will assume all the work. The linguistics component is implemented in MACLISP, features (and annotations and slots and nodes) are atoms, and composers are functions on their property lists.

```
Prediction
     composer-with  (lambda ... )
```

Making a prediction is a speech act, and we may expect there to be particular forms in a language for expressing them, for example, the use of the explicit "will" for the future tense. Knowledge of these would be part of the composer. Inside the main program, or the situation specialist, the concept of a prediction may always include certain parts: what is predicted, the time, any hedges, and so on. These part are directly reflected in the makeup of the elements present in the message, and their annotations mark what internal roles they have. There does not need to be a direct correspondence between these and the parts in the linguistic forms used, the actual correspondence is part of the knowledge of the prediction composer.

Typically, for any feature, one particular annotated element will be of greatest importance in seting the character of the whole utterance. For predictions, this is the "event". The prediction composer chooses a plan for the utterance to fit the requirements of the event-element. The realization of any other elements will be restricted to be compatible with it.

The prediction composer does not need to know the element's linguistic correlates itself, it can delegate the work to the composer for the element itself. The element look like this.

```
(event actor <Winston>
       action <fit person into full schedule>
       time <31-10-75,9am-12am>)
```

The first word points to the name of the composer, and the pairs give particular details. There is nothing special about the words used here (actor, action, time), just as long as the composer is designed to expect the information in those places that the message-assembler wants to put it. The event composer's strategy is to use a clause, and the choice of plan is determined by the character of the event's "action".

The action is "<fit person into full schedule>", and it will have two relevant properties in the lexicon: "plan", and "mapping". Plan is either the name of a standard plan to be used, or an actual plan, partially filled with words (i.e. it can be a phrase). "Mapping" is an association list showing how the subelements of the message are to be transferred to the plan.

```
<fit person into full schedule>
   PLAN
        node-i   (clause transl particle)
         slots frontings nil
               subject nil
               vg node-j (verb-group particle)
                slots modal nil
                      pre-vb-adv nil
                      mvb "squeeze"
                      prt "in"
               object1 <person being talked about>
               post-modifiers nil
   MAPPING
        (( actor  subject )
         ( time post-modifiers))
```

The event composer proceeds to instanticte the nodes in the phrase and make the transfers; the prediction composer then takes the resulting plan, and makes it the plan of the whole utterance.

Two message elements remain, but actually there is only one, because "aim-at-audience" is supplying additional information about the hedge. The annotation means that the contents of the hedge (<is possible>) are more something that we want to tell the audience than a detail of the prediction. This will affect how the element is positioned in the plan.

The prediction composer looks in the lexicon to see what grammatical unit will be used to realize <is possible>, and sees, let us say, two possibilities involving different configurations of the adverb "maybe" and the modal "can be able to", with the differences hinging on the placement of the adverb. Theoretically, adverbs can be positioned in a number of places in a clause, depending on their characteristics. In this instance, the choice is forced because of a heuristic written into the grammar of adverbs and accessible to the composer, that says that when the intent of an adverb is directed to the audience, it should be in the first position (the "frontings" slot). This choice implies putting "can" in the modal slot directly. The alternative with "maybe" in the pre-vb-adv slot would have necessitated a different form of the modal, yielding "may be able to". These details would have been taken care of by syntactic routines associated with the verb group node.

All the message elements have been placed and the first phase is over. The plan is now as below.

```
node-1   (clause transl particle)
  slots frontings  "maybe"
        subject  <winston>
        vg  node-2  (verb-group particle)
          slots modal  "can"
                pre-vb-adv  nil
                mvb  "squeeze"
                prt  "in"
        object1  <person being talked about>
        post-modifiers  nil
```

The second phase controller is a simple dispaching function that moves from slot to slot. "Frontings" contains a word, so the word is printed directly (there is a trap for morphological adjustments when necessary). "Subject" contains an internal object, so the controller should go to the lexicon for its composer and then come back to handle whatever the composer replaced the element with.

However, there is always an intervening step to check for the possibility of pronominalizing. This check is made with the element still in its internal form. The record of the discourse is given directly in terms of the internal representation and test for prior occurence can be as simple as identity checks against a reference list, avoiding potentially intricate string matching operations with words.

In the dialog that this message came from, there is clear reference to <winston>, so it can be pronominalized and "he" is printed.

Any slot, or any node type may have procedures associated with it that are executed when the slot or node is reached during the second phase. These procedures will handle syntactic processes like agreement, rearangement of slots to realize features, add function words, watch scope relationships, and in particular, position the particle in verb-particle pairs.

Generally, particle position ("squeeze John in" vs. "squeeze in John") is not specified by the grammar - except when the object is a pronoun and the particle <u>must</u> be displaced. This, of course, will not be known untill after the verb group has been passed. To deal with this, a subroutine in the "when-entered" procedure of the verb group is activated by the "particle" procedure. First, it records the particle and removes it from the VG plan so it will not be generated automatically. A "hook" is available on any slot for a procedure which can be run after pronominalization is checked and before the composer is called (if it is to be called). The subroutine incorporates the particle into a standard procedure and places it on that hook for the object1 slot. The procedure will check if the object has been printed as a pronoun, and if so, prints out the particle (which is now in the proper displaced position). If the object wasn't pronominalized, then it does nothing, nothing has yet been printed beyond the verb group, and other heuristics will be free to apply to choose the proper position. Since <person being talked about> is here equal to the student, the person the program is talking with, it is realized as the pronoun "you" and the particle is displaced.

Going from <31-10-75,9am-12am> to "tomorrow morning" may be little more than table lookup by a "time" composer that has been designed to know the formats of the time expressions inside the scheduler.

This presentation has had to be unfortunately short for the amount of new material involved. A large number of interesting details and questions about the processing have had to be omitted. At the moment (September, 1975), the data and control structures mentioned have been fully implemented and tests are underway on gedanken data. Hopefully, by the end of 1975 the component will have a reasonable grammar and will be working with messages and lexicons form the two programs mentioned before. A MIT A.I. lab technical report describing this work in depth should be ready in the spring of next year.

David McDonald

David McDonald
Cambridge, Mass.

References cited in the text:

Genesereth, M. (1975) A MACSYMA Advisor. Project MAC, MIT, Cambridge, Mass.

Goldman, N. (1974) "Computer Generation of Natural Language from a Deep Conceptual Base". memo AIM-247, Stanford Artificial Intelligence Lab., Stanford, Calif.

Goldstein, I. (1975) "Barganing Between Goals". in the proceedings of IJCAI-4, available from the MIT AI lab.

McDonald, D. (1975) The Design of a Program for Generating Natural Language. unpublished Master's Thesis, MIT Dept. of Electical Engineering.

Simmons, R. (1973) "Semantic Networks: Their Computation and Use for Understanding English Sentences". in Schank and Colby eds. Computer Models of Thought and Language.

Winograd, T. (1972) Understanding Natural Language. Academic Press, New York, NY.

American Journal of Computational Linguistics *Microfiche 33 : 18*

INCREMENTAL SENTENCE PROCESSING

RODGER KNAUS

Systems Software Division
Social and Economic Statistics Administration
Bureau of the Census
Washington, D. C. 20233

A human who learns a language can both parse and generate sentences in the language. In contrast most artificial language processors operate in one direction only or require separate grammars for parsing and generation. This paper describes a model for human language processing which uses a single language description for parsing and generation.

1. Choice of Parsing Strategy

A number of constraints limit the processors suitable as models of human language processing. Because short term memory is limited, the listener must absorb incoming words into larger chunks as the sentence is heard. Also because he is expected to reply within a couple seconds after the speaker finishes, regardless of length of the speaker's utterance, the listener must do much of the semantic processing of a sentence as he hears it.

Bever and Watt point out that the difficulty in understanding a sentence S is not predicted by the number of transformations used to generate S. Furthermore the process of detransformation appears too time-consuming (Petrick) for the approximately two seconds before a listener is expected to reply.

A depth first transition network parser (Woods, Kaplan), in which parsing difficulty is measured by the number of arcs traversed, correctly predicts the relative difficulty of active and passive sentences progressive and adjectival present participle sentences and the extreme difficulty of multiple center embeddings. However a syntactically directed depth first parser does not explain why syntactically similar sentences such as

(5A) The horse sold at the fair escaped.

(5B) The horse raced past the barn fell.

vary in difficulty, nor does it explain experiments on the completion and verification of ambiguous sentences (MacKay, Olsen and MacKay) which suggest that a pruned breadth first strategy is used to parse sentences. Sentences with two equally plausible alternatives took longer to process than sentences with only one likely interpretation. This extra processing time may be attributed to the construction of two alternate interpretations over a longer portion of the sentence when more than one interpretation is plausible.

In addition subjects sometimes become confused by the two interpretations of an ambiguous sentence. Finally in experiments in which subjects hear an ambiguous sentence in one ear and a disimbiguating sentence simultaneously in the other ear (Garrett) the interpretation of the ambiguity actually perceived by the subject may be switched between the possibilities by changing the disambiguating sentences.

```
Step 3 (a):  (S (NP (N mail) (N Boxes))
                (V like) (NP) (PP*))
       (b):  (S (NP (NP (N mail) (N Boxes))
                    (PP (PREP like) NP) (PP*))
                V(NP) (PP*))
       (c):  (S (NP (N mail)) (V Boxes)
                (PP (PREP like) NP) (PP*))
       (d):  (S (V mail) (NP (N Boxes))
                (PP (PREP like) NP) (PP*))
       (e):  (S (V mail)
                (NP (NP (N Boxes))
                    (PP (PREP like) NP) (PP*))
                (PP*))
```

After completing the sentence after Step 4, the parser produces phrase markers from a, c, d and e by adding the last word and deleting unfilled optional nodes. The phrase marker obtained from 4B is rejected because it contains an unfilled obligatory V node.

The incremental parser adds each successive sentence word to the partially completed phrase markers built from the earlier part of the sentence. The new word is added at the leftmost oblig unfilled node of each partial phrase marker and at all optional nodes to the left of this node.

Three different operations are used to add a new word to a partial parse. The word may be directly added to an unexpanded node, as in Step 3a above. Alternatively, a new word may be attached to an unfilled node with a left branching acyclic tree built from the grammar such as (PP PREP NP) or (S (NP N (N*)) V (NP) (PP*)). Attaching occurs in steps 1 and 3c.

Finally a subtree of an existing partial phrase marker may be left embedded in a larger structure of the same grammatical category, as in steps 3b and 3e above. The embedding operation uses at most two left branching trees built from the grammar: a tree T1 with a single cycle on the left branch is used to replace the existing subtree E being embedded. In step 3e, for example, the structure (S (V mail) (NP NP (PP*)) (PP*)) would be obtained. The E is used to expand the leftmost unexpanded node of T1; for 3 b this results in:

3e. (S (V mail) (NP (NP (N Boxes) (N*)) PP*) (PP*)).

Finally to the resulting structure the new sentence word is added through direct node expansion or attaching with an acyclic left branching tree; in the example above this produces 3e from 3e.'

Using direct expansion attaching and embedding, the incremental parser finds all the phrase markers of sentences in context free or regular expression language; a formal definition of the parser and a proof of its correctness appear in [10].

Sometimes, as at steps 3b and 3e, the same structure (a prepositional phrase in step 2) is used in more than one partial parse. Following Earley's Algorithm, the incremental parser builds a single copy of the shared substructure SØ and maintains pointers linking SØ to nodes in larger structures which SØ expands.

For all its tree building operations the incremental parser uses a finite set of trees. i.e., the trees with only left subnodes expanded and at most one cycle on the leftmost branch. These trees may be computed directly from the grammar and referenced by root and leftmost unexpanded node during the parse.

Using these preconstructed trees, the incremental parser requires only a fixed number of operations to add a new word to a partial parse: a retrieval on a doubly indexed set, copying the left branching tree, and at most four structure changing operations to paste words and trees together.

Like Earley's Algorithm, IP processes each word proportionally to sentence length. However on sentences satisfying a <u>depth difference</u> <u>bound</u>, the parsing time per word is constant. Because humans can't remember large numbers of sentence words but must process speech at an approximately constant rate, a constant parsing time per word is a necessary property of any algorithm modeling human language processing.

Let the depth of constituent C in phrase marker P be defined as the length of the path from the root of C to the root of P. If T1 and T2 are two adjacent terminals with T1 preceding T2, the depth difference from T1 to T2 is defined as the difference in depth between T1 and the root of the smallest tree containing T1 and T2. For example in the phrase marker

```
(9)  (S (NP (NP (DET the) (N telephone))
            (PP (PREP IN) (NP (DET the) (N room)))
            (V rang) (ADV loudly))
```

the depth difference between "the" and "telephone" is 1 and between "room" and "rang" is 3.

The depth difference between T1 and T2 is the number of nodes from T1 to the node expanded when adding T2 on a postorder traversal from T1 in the partial phrase marker containing T1 but not T2. The depth difference between T1 and T2 also represents the number of constituents of which T1 is the rightmost word.

A proof (requiring a formal definition of the incremental parse) that parsing time per word is constant in depth difference bounded sentences appears in [10]. Informally the depth difference bound places a bound both on the number of next nodes to expand which may follow a given terminal and on the amount of tree traversal which the parser must perform to find each next unexpanded node. Since each modification requires only a fixed number of operations, each of which is bounded on the finite set of at most once cyclic left branching trees, the computation adding a new word to existing partial parses is bounded independently of sentence length.

Natural language sentences tend to have small depth differences. Both right branching sentences and left branching sentences (found in Japanese for example) have an average depth difference over each three or four word segment of two or less. On the other hand sentences are difficult to understand when they have two consecutive large depth differences, such as the multiple center embedding

(10) The rat the cat the dog bit chased died.

or the complex noun phrase in

The pad on a clarinet in the last row which I fixed earlier for Eb fell out.

Furthermore in ambiguous sentences such as

(11) Joe figured that it was time to take the cat out.

Kimball observes that subjects prefer the reading with the smaller depth difference. Finally, Blumenthal found that subjects tended to understand a multiple center embedded sentence as a conjunctive sentence. The conjunctive sentence contains a re-arrangement with lower depth differences of the constituents of the center embedded sentence.

3. Sentence Generation

The syntactic form given to a sentence depends on the information being communicated in a sentence and on the cultural context in which the sentence appears. Clark and Haviland show that a speaker uses various syntactic devices sentences to place the "given" information known to the listener before the information "new" to the listener. Particular syntactic structures are also used to emphasize or suppress particular kinds of information; for example newspaper traffic accident reports usually begin with a passive sentence such as

(12) An elderly Lakewood man was injured when...,
presumably to emphasize the result of the accident.

To capture the dependence of syntax on semantic content and social context, the sentence generator uses function-like grammar rules of the form

(Rulename Cat Variables Predicate Forms).

Rulename is the name of the rule and cat is the grammatical category of the constituent generated by the rule.

Variables is a list of formal parameters. Usually the variable list contains a variable bound during rule execution to a node in a semantic network and another variable bound to a <u>control association list</u> containing information about the context in which the generated constituent will appear and possibly the syntactic form the constituent should have.

Predicate is a Boolean-valued form on the parameters in Variables. A rule is used only when Predicate is true.

Forms is a list of forms depending on Variables which generate terminals or calls to the grammar for subconstituents of CAT.

An example of a generation rule is

```
(SPI S.(X Y) (Equal (Voice Y) (Quote Passive))
             (NP (Object X) Y)
             (Beverb X)
             (Pap (Action X))
             (M* X Y))
```

which generates simple passive sentences. The variable X is bound to a node in a semantic network and Y to a control association list. The rule is applied only if the control alist contains a passive flag and if the semantic node has an object and action; in general a rule is applied only if the semantic subnodes called in the rule body appear in the semantic net. The form (NP (Obj X) Y) generates a form (NP XØ YØ), where XØ is the semantic node on the object indicator from X, and YØ is the value of Y. Beverb and Pap are procedures which generate respectively a form of the verb "to be" and a past participle form of the verb Action(X). M* is a procedure which generates a list depending on X and Y such as (PP<Value of Time(X)> <Value of Y>) for generating optional prepositional phrases or relative clauses.

As each rule is applied, the list of terminals and calls to grammar rules generated by the rule is added to a phrase marker representing the structure of the sentence being generated.

Grammar calls in the phrase marker are expanded top down and left to right, in a preorder traversal of the growing phrase marker. As terminals are generated they are printed out.

As an example, illustrating the effect of semantic and social contest on sentence generation, an initial sentence of a traffic accident report,

(13). A man was killed when a car hit him in Irvine.

was generated from the semantic nodes

```
A1: Agent    AØ        A2: Agent          AØ:  Class man
    Object   VØ            Action hit
    Action Kill            Object VØ
    Place Irvine           Instrument Car
    Cause AZ
```

and the control alist.

Purpose: Introduction; cases: object, cause, place

using a grammar built for generating traffic accident report sentences. To summarize a trace of the generation, a call to the sentence rule with purpose = introduction generates a sentence call with voice = passive. The passive rule applies and a noun phrase on AØ is called for. Because Purpose = Introduction a NP rule applies which calls for a NP to be generated on the semantic class to which AØ belongs. Because CASES contains TIME and CAUSE, the passive rule generated calls for modifying structures of these CASEs. Because the cause semantic node A2 has an action, the modifier rule M => Relative conjunction S generates the cause while the time is described by a prepositional phrase. The pronoun "him" is generated by a noun phrase rule NP-1 which generates a pronoun when the first semantic argument to the left of the NP-1 call in the generation phrase marker which is described by the same pronoun as the semantic argument A of NP-1 is in fact equal to A.

4. Finding Semantic Preimages

While the generator described in section 3 produces sentences from semantic and contextual information, the incremental parser described in section 2 recovers merely the syntactic structure of a sentence. To obtain the semantic arguments from which a sentence might have been generated a procedure to invert the generation rule forms must be added to the incremented parser.

While the incremental parser begins the construction of constituents top down, it completes them syntactically in a bottom up direction. In fact IP executes postorder traversals on all the syntactic parse trees it builds; of course if a particular partial phrase marker can not be finished, the traversal is not completed. However each node not a tree terminal of a syntactic phrase marker visited by the incremental parser is a syntactically complete constituent.

When the parser visits a syntactically complete constituent C, it applies a function INVERT to find the semantic preimages of C. In finding the semantic structure of C, INVERT has available not only the syntactic structure of C, but also the semantic preimages which it found for subconstituents of C. INVERT finds the set of generation rules which might produce a constituent having the same syntactic form as C. For each such rule R., INVERT constructs all the possible parings between each output-generating form F of R and the constituents of C which F might produce. For example if C is

```
    (S (NP Man) (Beverb is) (PAP Injured))
```

the pairing established for the passive sentence rule would be

```
    (NP (Object X) Y)        (NP the man)
    (Beverb X)               (Beverb is)
    (Pap (Action X))         (Pap Injured)
    (M* X Y)                 NIL
```

The pair ((Equal (Voice Y) PASSIVE) T) is also created, since the rule predicate is true whenever a rule applies.

Each indicidual pair P in such a pairing of a rule form and rule form outputs is processed by a function FIND which returns an association list containing possible values of the rule parameters (X and Y in the example above) which would produce the output appearing in P. For the example above FIND would produce

```
    (( X ((Object Man))) (Y NIL))
    (( X ((Time Past))) (Y NIL))
    (( X NIL) (Y (( Cases Nil)))).
    (( X NIL) (Y (( Voice Passive))))
```

Using an extension to association lists of the computational logic Unification Algorithm, these association lists are unified into a single association list, which for the example is

```
    (( X ((Agent man) (Time Past) (Action Injure))
    (( Y ((Cases Nil) (Voice Passive))))
```

Finally INVERT creates a grammar rule call,

```
    (S ((Agent man)(Time Past)(Action Injure))
       ((Cases Nil)(Voice Passive))))
```

from the association list and stores the result in the inverse image of C.

In finding a semantic preimage, the INVERT function must know which grammar rules might produce a particular grammatical constituent. This information is computed by symbolically evaluating the grammar rules to produce the strings of regular expression grammar nonterminals (as opposed to grammar calls) representing the possible output of each rule. The resulting relation from rules to strings is inverted into a table giving possible rules generating each string.

The heart of this symbolic evaluator is a function ETERM on the output generating forms of a rule which returns a list all lists of regular expression nonterminals representing the output of a form. ETERM takes advantage of the similar syntax of most grammar rule forms, and is defined in simplified form (with comments in angle brackets) as

```
Eterm (form) =
    if atom (form) then NIL
    <terminates recursion>
    else if car (form) is a grammatical category
      then list (list (car (form)))
      <these forms generate a single grammar call>
    else if car (form) = FUNCTION or LAMBDA
      then ETERM (cadr (form))
    else if car (form) = LAMBDA
      then ETERM (caddr (form))
    else if car (form) = LIST
      if form is not properly contained in a LIST
      expression
      then Mapcar((Function Concatenate)
                  (Cartesian
                 ((Mapcar (Function ETERM)
                          cdr (form))))
     <outer LISTS are used to create lists of grammar calls>
    else if form is inside a LIST expression
      ETERM (cadr (form))
      <inner lists are used to create grammatically>
    else if car (form) = MAPCONC then make optional
         and repeatable all the nonterminals returned
         in ETERM ([function argument of MAPCONC])
```

```
else if car (form) = COND
     then MAPCONC((LAMBDA(X) ETERM ([last form in X])
                  (cdr form)
     <returns alternatives from each branch of the COND>
else if car (form) is a user-defined function
     then ETERM ([definition of function])
else if there is a stored value for ETERM (form)
     then that value
else ask the the user for help
```

The function FIND which returns possible bindings for rule variables when given a rule form and its output is defined below. The variable ALIST holds the value of the association list being hypothesized by FIND; this variable is NIL when FIND is called from INVERT.

Like ETERM, the definition of FIND is based on the rules for evaluating recursive functions.

```
FIND (Alist form value)=
     if eval(form alist)=value then list (Alist)
     else if recursion depth exceeded, then NIL
     else if atom (form) then list (Merge (list (cons
                                      (form Value)) Alist)
     else if car (form)= COND
          let L = clauses which might be entered by
                  evaluating form
          then Mapconc (FM l)   where
          FM (clause) = list (Merge Find (Alist Car (clause)T)
                                    Find (Alist last (clause)))
     else if car (form) = Quote then if cadr (form) = value
                                         then Alist else NIL
     else if car (form) is a defined function
          then FIND (Alist (Substitute cdr (form) for
                            formal parameters in definition
                            of car (form))
                       Value)
     else if car (form) = MAPCONC (fn lst)
          then Merge (Find (Alist lst value)
                     For each X in lst, Merge (Alist for X))
     <this clause makes the assumption, which works in
       practice, that fn generates either one-element
       or empty lists>
     else NIL
```

With a definition of FIND similar to the one above, the parser found the preimage

```
(S (((place ((class (park))))
     (agent ((class (man))))
     (action (walked]
```

[the extra parentheses denote lists of alternatives] for the sentence

(13) The man walked in the park.

generated by the grammar

```
[SØ S (X) T (NP (Agent X)) (V(Action X))
            (Optional ((PP (Place X) ((Case Place]
[NPØ NP (X) T (Det X) (N (Class X]
[PPØ PP (XY) T (Prep XY) (NPX]
```

and the preposition function

```
Prep (XY) = Selectq (Assoc CASE Y)
                    (Place IN)
                    (Instrument WITH)
                    (Source FROM]
```

5. Implementation

The processors described in this paper have been programmed in University of California, Irvine, LISP and run in about 45K on a PDP-10 computer.

References

1. Bever, Thomas G. 1970. In [7] and [5].
2. Clark, Herbert H. and Haviland, Susan E. 1975 Social Sciences Working Paper, 67. U.C. Irvine.
3. Colby, Benjamin N. 1973. American Anthropologist 75, 645-62.
4. Florres d'Arcaio and Levalt, eds. 1970 Advances in Psycholinguistics, North Holland, Amsterdam.
5. Garrett, Merrill, F. 1970. in [5].
6. Haynes, John R. 1970. Cognition and the Development of Language. John Wiley.
7. Kaplan, Ronald M. 1972. A.I. 3, 77-100
8. Kimball, John 1974. Cognition 2,1,15-47.
9. Knaus, Rodger. 1975. Ph.D Thesis. U.C. Irvine.
10. MacKay, Donald G. 1966. Perception and Psychophysics. 426-36.
11. Olson, James N. and MacKay, Donald G. JVLVB 13, 45770.
12. Petrick, S. R. In [14].
13. Rustin, Randall. 1973. Natural Language Processing.
14. Watt, Wm. 1970. In [7].
15. Woods, Wm. 1973. In [14].

American Journal of Computational Linguistics *Microfiche 33 : 33*

A Lexical Process Model of Nominal Compounding in English

JAMES R. RHYNE

Department of Computer Science
University of Houston
Houston, Texas 77004

ABSTRACT

A theoretical model for nominal compound formation in English is presented in which the rules are representations of lexical processes. It is argued that such rules can be generalized to account for many nominal compounds with similar structure and to enable new compounds to be produced and understood. It is shown that nominal compounding depends crucially on the existence of a "characteristic" relationship between a nominal and the verb which occurs in a relative clause paraphrase of a compound which contains the nominal. A computer implementation of the model is presented and the problems of binding and rule selection are discussed.

Linguistic Issues.

Nominal compounds are sequences of two or more nominals which have the semantic effect of noun phrases with attached relative clauses. The rightmost nominal is generally the primary referent of the compound the other nominals restrict the reference of the rightmost nominal in much the same fashion that a relative clause does. There are, of course, exceptions in which the rightmost nominal is figurative or euphemistic (e.g. family jewels). Compounds occur frequently in English and Germanic languages, but infrequently in the Romance languages where their function is largely performed by nominal-preposition-nominal sequences (e.g. chemin de fer, agent de change).

The syntactic structure of nominal compounds is quite simple --the three variants are N-N, N-participle-N, and N-gerund-N. In the N-N form, either of the two nominals may in fact be yet another nominal compound, giving a structure like (N-N)-N or N-(N-N); the first of these forms seems to occur much more often than the second (examples of each type are: typewriter mechanic, liquid roach poison).

I assume that the process of nominal compounding is syntactically a process in which a relative clause is reduced by deleting all elements of the relative clause but one and preposing the single remaining element in front of the antecedent nominal. In addition, the clause verb may be nominalized and preposed. Other linguists have proposed different derivations for nominal compounds; Lees [3], for example, derives nominal compounds from nominal-preposition-nominal sequences. There are two reasons why I feel that Lees approach is wrong: (1) there are English compounds for which no reasonable equivalent nominal-preposition-nominal paraphrase can be given (e.g. windmill), and (2) there are subtle meaning differences between the nominal compounds and their nominal-preposition-nominal counterparts (county clerk vs. clerk for the county). If nominal compounds and nominal-preposition-nominal sequences are derived from forms like relative clauses, then the differences in meaning can be accounted for by deriving each form from a distinct relative clause; the relative clauses may, of course, be quite closely related to each other.

I have spoken rather loosely about deriving nominal compounds from relative clauses; I am not proposing a derivation system which operates on surface forms of the language, and what I intend that the reader should understand is that an underlying form for a nominal compound is derived from an underlying form for a relative clause by a language process which I term a lexical rule because, as we shall see, the operation of such rules depends crucially on the specific lexical items which are present in the underlying structures. Linguists have identified a number of lexical processes in English; some examples of such processes may be found in [1] and [2].

The underlying forms associated with relative clauses and nominal compounds in the model of nominal compounding being presented here are networks (trees for the most part) defined in terms of a case grammar which is closely related to that used by Simmons [5]. The cases which appear in this system fall into two general categories: (1) cases of the clause verb, which are the following -- Performer, Object, Goal, Source, Location, Means, Cause, and Enabler -- and (2) structural cases, which are RELCL (relative clause) and COMP (compound). I will not explain these cases in detail, as that is the subject of a forthcoming paper. But the following observations will illuminate the case system for verb cases. The case system distinguishes the immediate performer of an act from a remote cause or agent of the act. The reason for this distinction lies in an intimate connection between verbs and the assumed or habitual performer of the act which is the reference of the verb. The case system also distinguishes an active causative agent of an act from an agent which merely permits the act to occur; this distinction in the case system permits two classes of verbs to be distinguished according to whether the surface subject commonly causes the act or permits the act to occur.

The case system used in the present model of nominal compounding is not a deep case system; on the contrary, it seems that nominal compounding is a lexical process which occurs rather near the surface in a derivational grammar model. An example which can be given to support this is the compound _ignition key_; this is a key which turns a switch which enables a complex sequence of events to take place that ultimately result in the ignition of a fuel/air mixture in an engine, or one may describe it equivalently as a key which causes ignition. The first description corresponds to a deep case level of description while the second corresponds to the level at which the compound ignition key is formed. I would argue that if one takes the deep case approach, then one is forced to include a great deal of structure in the rules for nominal compounding; in particular, the rule for _ignition key_ must remove all of the links in the causal chain leading to the ignition act. The deletion of this intermediate information must be done to obtain the description given in the second case, and to include the deletion procedure in both a compounding rule and in the rule process which leads to the shorter description means unnecessarily duplicating the procedure. Moreover, if one derives compounds from paradigm relative clauses of the second sort, e.g. key which causes an action to occur, then it is possible to generalize compound forming rules so that a single rule may produce several compounds. It will not be possible to do this if deep cases are used as the deep case structure of _firing key_ will be quite different from that of _ignition key_.

In order to understand the model of compounding which is being presented here, it is essential to consider the function of compounding in language. In my view, compounding is a process which allows a speaker to systematically delete information from an utterance just when the speaker has reason to expect that the hearer can reconstruct that information. In effect, I consider compounding (and a great many other linguistic processes) to be examples of linguistic encoding which are used to speed up communication, and the grammar shared by the speaker and hearer must include the encoding and decoding functions.

Consider the nominal compound steam distillation, which refers to the distillation of some substance with steam; the hearer of the compound steam distillation knows that distillation is the derived nominal form of distill. The hearer also knows what the common or characteristic cases of the verb distill are: the agent is invariably a person or machine (this would be the occupant of the Cause case slot in my system), the instrument (or Means) may be an apparatus or a heated medium such as steam and the Goal is a liquid which is missing some of the constituents that it entered the distillation process with.

It happens that in English, whenever a derived nominal of an act is the right element in a compound, then the left element is almost always an occupant of one of the case slots of the verb. In order to recreate the underlying relative clause structure, it is only necessary for the hearer to properly choose the case for the nominal steam. A great deal of lexical information can be brought to bear on this question; for example, steam is not a liquid, it is water vapor and thus it cannot be the starting substance or the end product of a distillation process. Steam might be the Cause of the act of distillation except that there do not seem to be any compounds in English which have distillation as the right element and a Cause as the left element. Thus the hearer can assign steam to the Means case with some assurance.

In another example, shrimp boat, the hearer can ascertain by lexical relations involving the word boat, that boats are characteristically used to catch marine life. One choice for the main verb in a synonymous relative clause is catch, which will have boat as an element of the Means case. The Cause for catch is commonly a person or perhaps a sophisticated machine designed to catch things (i.e. a trap). The Object is characteristically an animal. There is a strong characteristic relation between the animal being caught and the means used to catch it, for example mink is trapped, calves are roped, birds are netted, and fish are caught with a boat. This relation exists as a rule in the lexicon of both the speaker and the hearer and it enables the speaker to produce the nominal compound and the hearer to understand it.

Furthermore, shrimp boat is one member of a class of closely related nominal compounds which includes lobster boat, whale boat, tuna boat and many others. It would be most interesting if a single rule could be formulated which would generate all of these compounds. A lobster boat is a boat which is used to catch lobster, a tuna boat is a boat which is used to catch tuna, and so forth. All of these examples are identical except for the particular marine animal being caught. The logical next step is the creation of a rule which generalizes the individual marine animals to the common category of marine animal. This rule will state that a marine animal boat is a boat which is used to catch marine animals.

In making this generalization, I have given the rule the power to help interpret novel compounds and to generate them. With this power comes a difficulty, which is constraining the rule so that it does not generate bad compounds or produce incorrect interpretations. The key to this constraint lies in what I will term the characteristic or habitual aspect of nominal compounds. In the case of the boat compounds, a boat will only be a shrimp boat if it is characteristically, usually, habitually or invariably used to catch shrimp. So the operation of a compounding rule is enabled only if a characteristic aspect is associated with the verb; in English, this is usually indicated by an adverb or an adverbial phrase. If the speaker is willing to assert that a boat is characteristically used to catch turtles, then the nominal compound turtle boat may be used. The hearer will use the general rule to place turtle and boat in the proper case slots, and because a compound was used by the speaker, the hearer will infer that the boat is one which is characteristically used to catch turtles.

There are other problems which arise with the generalization of rules; for example, compounding never produces a compound in which the left element is a proper noun, unless the proper noun is the name of a process (e.g. Markov process) or is a Source, Performer, or Goal of an act of giving. It also seems to be true that compounds are not generally formed when a lexical item is several levels below the genéral term which appears in the rule (e.g. repairmidget) or when a cross-classificatory term is used (e.g. automobile Indian as an Indian who repairs automobiles). With all of the preceding discussion in mind, I would now like to turn to the model of nominal compounding which I have presently implemented and running.

The Computer Model

The computer model of compounding accepts relative clause structures as input and produces nominal compound structures as output when the input is appropriate. It is written in a language with many parentheses the language was chosen for its program development facilities, i.e. built-in editor, rather than for its interpretive capabilities. The program which produces nominal compounds is a pattern matching interpreter; it applies a rule of compound formation by matching one side of the rule with the input structure, and if certain criteria are satisfied by the match, items from the input structure are bound into the rule, transferred to the other side of the rule, and a copy is then made of the other side of the rule. The result is a nominal compound structure.

The model has two components: a rule interpreter and a lexicon of rules for compounding. There is nothing tricky about rule application. Consider the nominal compound flower market and its associated relative clause paraphrase market where flowers are characteristically sold. These phrases have in my system the underlying structures shown in Figure 1. The notation in square braces means that the verb sell has the characteristic aspect in this instance.

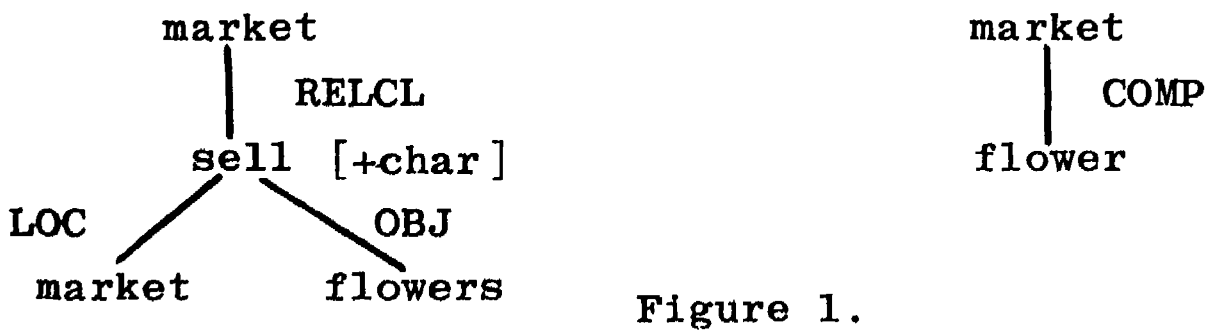


Figure 1.

These two structures can be made into a rule by linking them together. Whenever a relative clause structure identical to that in Figure 1 is received, the rule applies and a copy is created of the nominal compound *flower market*. The matching procedure is a relatively straightforward, top down, recursive process which has backtracking capability in the event that a structure or case occurs more than once at any given level of the structure. There are two problems which arise; however: if the rule is generalized to account for compounds other than *flower market*, then the lexical items in the rule will behave as variables and some provisions must be made for binding of values to these variables; also, the rule interpreter must have some heuristics for selecting appropriate rules if the time required to produce a compound is not to increase exponentially with the size of the lexicon.

The present version of the model only partly solves the binding problem. Consider the rule given in Figure 2 which is a generalization of that given in Figure 1.

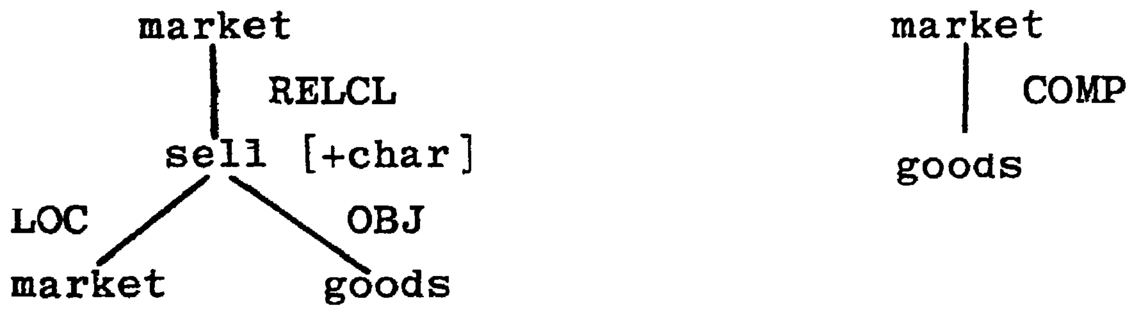


Figure 2.

If this rule is to apply to the relative clause structure given in Figure 1 and generate the compound flower market, then the rule interpreter must recognize that the relative clause in Figure 1 is an instance of that given in Figure 2. The matching procedure does this by determining that the reference set of the nominal *flowers* is a subset of the reference set of the nominal *goods*.

In addition, the nominal *flowers* must be carried across to the other side of the rule and substituted there for *goods* before the other side of the rule is copied. Thus *market* and *goods* must be bound across the rule so that whatever lexical item matches either of these nominals becomes the value associated with these nominals on the other side of the rule.

In the initial version of the model, this binding was established explicitly when the rule was entered into the lexicon, but this seemed unsatisfactorily *ad hoc*. In a subsequent version, the identity of the lexical items on both sides of the rule was the relation used to establish binding relationships. Consider, however, the structure shown in Figure 3.

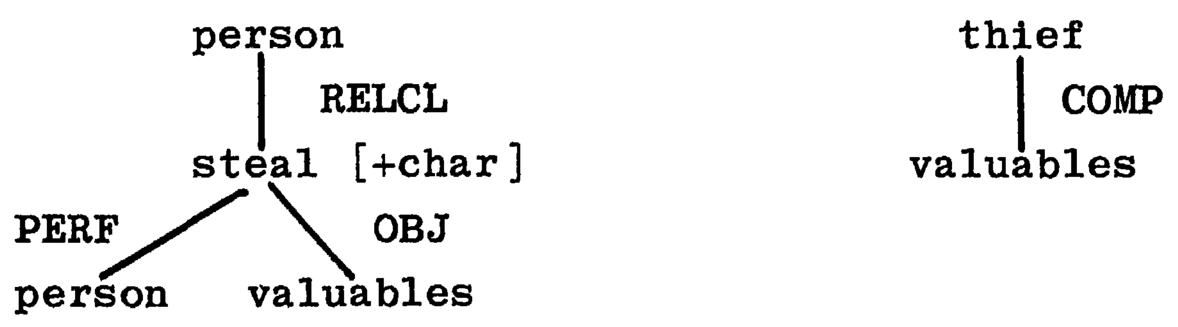


Figure 3

Here *person* should be bound to *thief* but the previous technique is not able to establish this binding. The reason that we know that *person* and *thief* should be bound is because we know that a thief is a person who steals characteristically. In the most recent version of the model, this information is used to find the binding relationships when the rule of identity does not work. The lexicon is searched for a rule which can be used to establish this binding. The rule which is used in the example shown in Figure 3 is displayed below in Figure 4.

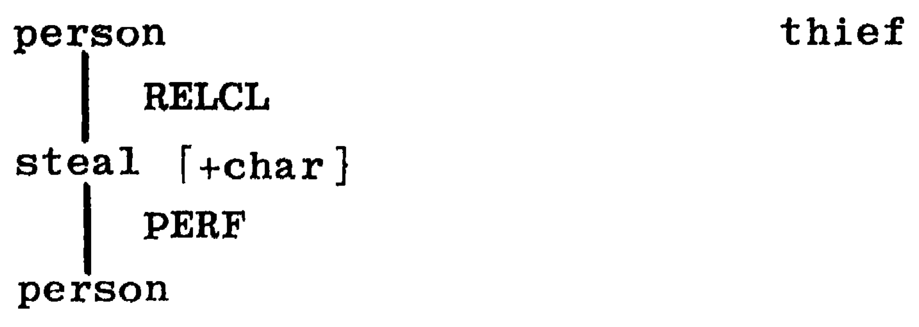


Figure 4

From the structures given in Figure 4, one can see that *person* should be bound to *thief* because the rule states that the reference set of *thief* is the same as the reference set of *person* as restricted by the relative clause.

The technique of using lexical rules to establish bindings works in virtually every instance, but it has the defect of requiring that the information that a thief is a person who steals things be represented in the lexicon twice at least. A new model is under construction which attempts to reduce this redundancy by allowing the rules to have multiple left and right parts.

The problem of selecting appropriate rules is rather easier to solve. In most compounds in English, there is a characteristic association between the right element of the nominal compound and the main verb of the associated relative clause paraphrase. These two elements which occur on opposite sides of the compounding rule supply a great deal of information about the possibilities for application of the rule. So, in the model, each rule in the lexicon is indexed by the main verb of the relative clause and by the right element of the nominal compound. This index actually contains some environmental information as well; for the clause verb, this environmental information is the case frame of the verb and the fact that it is the main verb of the relative clause -- for the compound nominal, the environmental information is just the fact that the nominal is the rightmost one in a nominal compound.

The basic model has been tested with a set of several hundred nominal compounds and is very successful in coping with a wide variety of compound types. The productivity of the rules varies greatly; some rules may produce hundreds of compounds while other rules may only result in one or two compounds. Frozen forms such as <u>keel</u> <u>boat</u> are handled by a rule which generates only one compound; there is a rule for each frozen form. The rule structures contain exclusion lists associated with each lexical item in the rule, and these exclusion lists prevent the rule from operating whenever a lexical item matches one of the items on an exclusion list if the items occur at corresponding locations in the structures.

The model is quite quick in operation; on a high speed display console, it will generally produce compounds much faster than a person sitting at the console can conveniently read them. This is mainly due to the rule selection heuristic, but the match procedure has been carefully optimized as well.

Conclusions

The model program is an excellent demonstration of the appropriateness of the basic theory; moreover, the rules themselves can be generalized to deal with syntactic processes, so there is no discontinuity in the grammar model between the lexical processes and the syntactic processes. It seems clear that the rules could also be used to represent other lexical processes in language and this is currently being pursued.

There is no reason why the rules could not be used for recognition as well as for the production of nominal compounds. The bindings are not one-way, and the matching procedure will work equally well for compound structures. The reasons why the computer model is a production model are: (1) that the computer model assumes the semantic correctness of the input relative clause structures, and (2) that compounds are often ambiguous and may be paraphrased by two or more relative clauses, while the converse of this is almost never true. A recognition model would have to generate underlying relative clause structures for each ambiguity and a semantic component would have to screen the relative clauses for semantic errors.

I hope that the reader has noticed the avoidance of rule procedures in this model. When I began working on the design of the computer programs, I had in mind the creation of a model which once implemented in LISP could be extended merely by adding new rules without having to construct any additional LISP programs. I ultimately wanted to have a model which could "learn" new rules by systematic generalization and restriction of existing rules. I feel that this would be relatively easy with rule structures and extremely difficult with rule procedures written in a programming language. Furthermore, I subscribe to Karl Popper's ideas of scientific endeavour, and rule structures appealed because it would be more difficult to bury flaws or ill understood aspects of compounding and rule processes in structures than in procedures where the computational power of the programming language permits and even encourages _ad hoc_ solutions to be found to problems.

Acknowledgements

I would like to here acknowledge the suggestions made by Robert F. Simmons, Carlota Smith, Mary Ross T. Rhyne, Laurent Siklossy, and Stanley Peters which have helped improve my understanding of nominal compounding.

1. Chomsky, N. "Remarks on Nominalization," in *Readings in English Transformational Grammar*, Jacobs, R. and Rosenbaum, P. eds. Ginn, Waltham, Massachusetts, 1970.
2. Gruber, J. "Studies in Lexical Relations." Ph. D. thesis, MIT, 1965.
3. Lees, R. *The Grammar of English Nominalizations*. Mouton, The Hague, 1968.
4. Rhyne, J. "Lexical Rules and Structures in a Computer Model of Nominal Compounding in English." Ph. D. thesis, The University of Texas at Austin, 1975.
5. Simmons, R. "Semantic Networks: Their Computation and Use for Understanding English Sentences," in *Computer Models of Thought and Language*, Schank, R. and Colby, K. eds. W. H. Freeman, San Francisco, 1973.

American Journal of Computational Linguistics *Microfiche 33 : 45*

GENERATION AS PARSING FROM A NETWORK INTO A LINEAR STRING

STUART C. SHAPIRO

Computer Science Department
Indiana University
Bloomington 47401

ABSTRACT

Generation of English surface strings from a semantic network is viewed as the creation of a linear surface string that describes a node of the semantic network. The form of the surface string is controlled by a recursive augmented transition network grammar, which is capable of examining the form and content of the semantic network connected to the semantic node being described. A single node of the grammar network may result in different forms of surface strings depending on the semantic node it is given, and a single semantic node may be described by different surface strings depending on the grammar node it is given to. Since generation from a semantic network rather than from disconnected phrase markers, the surface string may be generated directly, left to right.

Introduction

In this paper, we discuss the approach being taken in the English generation subsystem of a natural language understanding system presently under development at Indiana University. The core of the understander is a semantic network processing system, SNePS (Shapiro, 1975), which is a descendant of the MENTAL semantic subsystem (Shapiro, 1971a, 1971b) of the MIND system (Kay, 1973). The role of the generator is to describe, in English, any of the nodes in the semantic network, all of which represent concepts of the understanding system.

and other computations are required in the process of pasting these trees tog ther in appropriate places until a single phrase marker is attained which will lead to the surface string. Since we are generating from a semantic network, all the pasting together is already done. Grabbing the network by the node of interest and letting the network dangle from it gives a structure which may be searched appropriately in order to generate the surface string directly in left to right fashion.

Our system bears a superficial resemblance to that described in Simmons and Slocum, 1972 and in Simmons, 1973. That system, however, stores surface information such as tense and voice in its semantic network and its ATN takes as input a linear list containing the semantic node and a generation pattern consisting of a "series of constraints on the modality" (Simmons et al., 1973, p. 92

The generator described in Schank et al., 1973, translates from a "conceptual structure" into a network of the form of Simmons' network which is then given to a version of Simmons generation program. The two stages use different mechanisms. Our system amounts to a unificatio of these two stages.

The generator, as described in this paper, as well as SNePS, a parser and an inference mechanism have been written in LISP 1.6 and are running interactively on a DEC system-10 on the Indiana University Computing Network.

Representation in the Semantic Network

Conceptual information derived from parsed sentences or deduced from other information (or input directly via the SNePS user's language) is stored in a semantic network. The nodes in the network represent concepts which may be discussed and reasoned about. The edges represent semantic but non-conceptual binary relations between nodes. There are also auxiliary nodes which SNePS can use or which the user can use as SNePS variables. (For a more complete discussion of SNePS and the network see Shapiro, 1975.)

The semantic network representation being used does not include information considered to be features of the surface string such as tense, voice or main vs. relative clause. Instead of tense, temporal information is stored relative to a growing time line in a manner similar to that of Bruce, 1972. From this information a tense can be generated for an output sentence, but it may be a different tense than that of the original input sentence if time has progressed in the interim. The voice of a generated sentence is usually determined by the top level call to the generator function. However, sometimes it is determined by the generator grammar. For example, when generating a relative clause, voice is determined by whether the noun being modified is the agent or object of the action described by the relative clause. The main clause of a generated sentence depends on which semantic node is given to the generator in the top level call. Other nodes connected to it may result in relative clauses being generated. These roles may be reversed in other top level calls to the generator.

The generator is driven by two sets of data: the semantic network and a grammar in the form of a recursive augmented transition network (ATN) similar to that of Woods, 1973. The edges on our ATN are somewhat different from those of Woods since our view is that the generator is a tranducer from a network into a linear string, whereas a parser is a transducer from a linear string into a tree or network. The changes this entails are discussed below. During any point in generation, the generator is working on some particular semantic node. Functions on the edges of the ATN can examine the network connected to this node and fail or succeed accordingly. In this way, nodes of the ATN can "decide" what surface form is most appropriate for describing a semantic node, while different ATN nodes may generate different surface forms to describe the same semantic node.

A common assumption among linguists is that generation begins with a set of disconnected deep phrase markers. Transformations

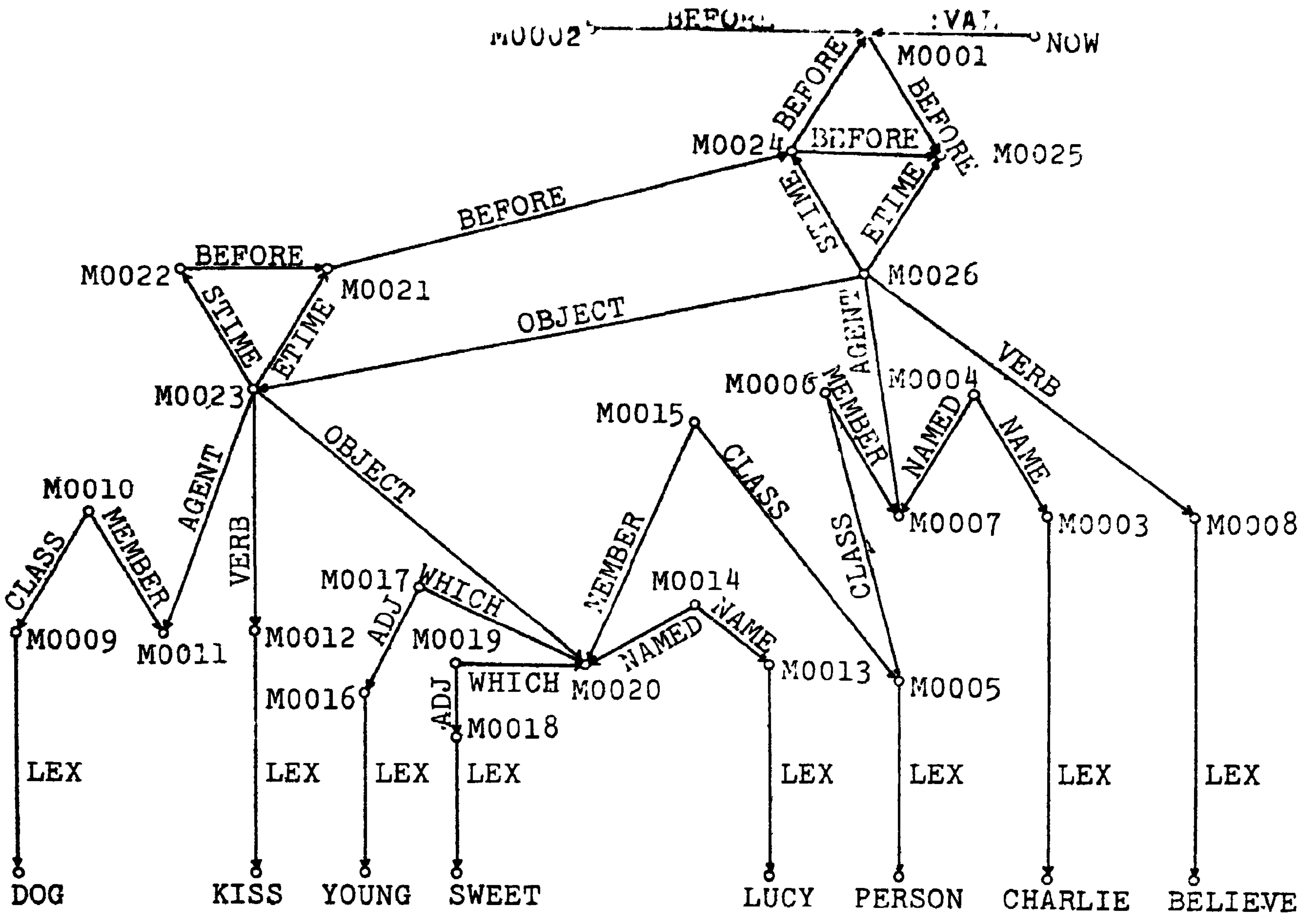


Figure 1: Semantic Network Representation for "Charlie believes that a dog kissed sweet young Lucy," "Charlie is a person," and "Lucy is a person."

...formation considered to be features of surface strings are not stored in the semantic network, but are used by the parser in constructing the network from the input sentence and by the generator for generating a surface string from the network. For example, tense is mapped into and from temporal relations between a node representing that some action has, is, or will occur and a growing time line. Restrictive relative clauses are used by the parser to identify a node being discussed, while non-restrictive relative clauses may result in new information being added to the network.

The example used in this paper is designed to illustrate the generation issues being discussed. Although it also illustrates our general approach to representational issues, some details will

```
*(SNEG M0026)
(CHARLIE IS BELIEVING THAT A DOG KISSED SWEET YOUNG LUCY)
*(SNEG M0023)
(A DOG KISSED SWEET YOUNG LUCY)
*(SNEG M0007)
(CHARLIE WHO IS BELIEVING THAT A DOG KISSED SWEET YOUNG LUCY)
*(SNEG M0004)
(CHARLIE IS A PERSON WHO IS BELIEVING THAT A DOG KISSED SWEET YOUNG LUCY)
*(SNEG M0006)
(CHARLIE WHO IS BELIEVING THAT A DOG KISSED SWEET YOUNG LUCY IS A PERSON)
*(SNEG M0008)
(THE BELIEVING THAT A DOG KISSED SWEET YOUNG LUCY BY CHARLIE)
*(SNEG M0011)
(A DOG WHICH KISSED SWEET YOUNG LUCY)
*(SNEG M0010)
(THAT WHICH KISSED SWEET YOUNG LUCY IS A DOG)
*(SNEG M0012)
(THE KISSING OF SWEET YOUNG LUCY BY A DOG)
*(SNEG M0020)
(SWEET YOUNG LUCY WHO WAS KISSED BY A DOG)
*(SNEG M0014)
(LUCY IS A SWEET YOUNG PERSON WHO WAS KISSED BY A DOG)
*(SNEG M0015)
(SWEET YOUNG LUCY WHO WAS KISSED BY A DOG IS A PERSON)
*(SNEG M0017)
(SWEET LUCY WHO WAS KISSED BY A DOG IS YOUNG)
*(SNEG M0019)
(YOUNG LUCY WHO WAS KISSED BY A DOG IS SWEET)
```

<u>Figure 2</u>: Results of calls to the generator with nodes from Figure 1. User input is on lines beginning with *.

certainly change as work progresses. Figure 1 shows the semantic network representation for the information in the sentences, "Charlie believes that a dog kissed sweet young Lucy," "Charlie is a person," and "Lucy is a person." Converse edges are not shown, but in all cases the label of a converse edge is the label of the forward edge with '*' appended except for BEFORE, whose converse edge is labelled AFTER. LEX pointers point to nodes containing lexical entries. STIME points to the starting time of an action and ETIME to its ending time. Nodes representing instants of time are related to each other by the BEFORE/AFTER edges. The auxiliary node NOW has a :VAL pointer to the current instant of time.

Figure 2 shows the generator's output for many of the nodes of Figure 1. Figure 3 shows the lexicon used in the example.

```
(BELIEVE((CTGY.V)(INF.BELIEVE)
    (PRES.BELIEVES)(PAST.BELIEVED)(PASTP.BELIEVED)(PRESP.BELIEVING)))
(CHARLIE((CTGY.NPR)(PI.CHARLIE)))
(DOG((CTGY.N)(SING.DOG)(PLUR.DOGS)))
(KISS((CTGY.V)(INF.KISS)
    (PRES.KISSES)(PAST.KISSED)(PASTP.KISSED)(PRESP.KISSING)))
(LUCY((CTGY.NPR)(PI.LUCY)))
(PERSON((CTGY.N)(SING.PERSON)(PLUR.PEOPLE)))
(SWEET((CTGY.ADJ)(PI.SWEET)))
(YOUNG((CTGY.ADJ)(PI.YOUNG)))
```

Figure 3: The lexicon used in the example of Figures 1 and 2.

Generation as Parsing

Normal parsing involves taking input from a linear string and producing a tree or network structure as output. Viewing this in terms of an ATN grammar as described in Woods, 1973, there is a well-defined next input function which simply places successive words into the * register. The output function, however, is more complicated, using BUILDQ to build pieces of trees, or, as in our parser, a BUILD function to build pieces of network.

If we now consider generating in these terms, we see that there is no simple next input function. The generator will focus on some semantic node for a while, recursively shifting its attention to adjacent nodes and back. Since there are several adjacent nodes, connected by variously labelled edges, the grammar author must specify which edge to follow when the generator is to move to another semantic node. For these reasons, the same focal semantic node is used when traversing edges of the grammar network and a new semantic node is specified by giving a path from the current semantic node when pushing to a new grammar node. The register SNODE is used to hold the current semantic node.

The output function of generation is straightforward, simply being concatenation onto a growing string. Since the output string is analogous to the parser's input string, we store it in the reg-

```
garc ::= (TEST test [action]*(TO gnode))
         (JUMP [action]*(TO gnode))
         (MEM wform (word*) test [action]*(TO gnode))
         (NOTMEM wform (word*) test [action]*(TO gnode))
         (TRANSR ([regname] regname regname) test [action]*(TO gnode))
         (GEN gnode sform [action]*regname [action]*(TO gnode))

sform ::= wform
          SNODE

wform ::= (CONCAT form form*)
          (GETF sarc [sform])
          (GETR regname)
          (LEXLOOK lfeat [sform])
          sexp

form ::= wform
         sform

action ::= (SETR regname form
           (ADDTO regname form*)
           (ADDON regname form*)
           sexp

test ::= (MEMS form form)
         (PATH sform sarc* sform)
         form
         sexp

gnode ::= <any LISP atom which represents a grammar node>

word ::= <any LISP atom>

regname ::= <any non-numeric LISP atom used as a register name>

sarc ::= <any LISP atom used as a semantic arc label>

lfeat ::= <any LISP atom used as a lexical feature>

sexp ::= <any LISP s-expression>
```

Figure 4: Syntax of edges of generator ATN grammars

ister *. When a pop occurs, it is always the current value of * that is returned.

Figure 4 shows the syntax of the generator ATN grammar. Object language symbols are), (, and elements in capital letters. Meta-language symbols are in lower case, Square brackets enclose optional elements. Elements followed by * may be repeated one or more times. Angle brackets enclose informal English descriptions.

Semantics of Edge Functions

In this section, the semantics of the grammar arcs, forms and tests are presented and compared to those of Woods' ATNs.† The

† All comparisons are with Woods, 1973.

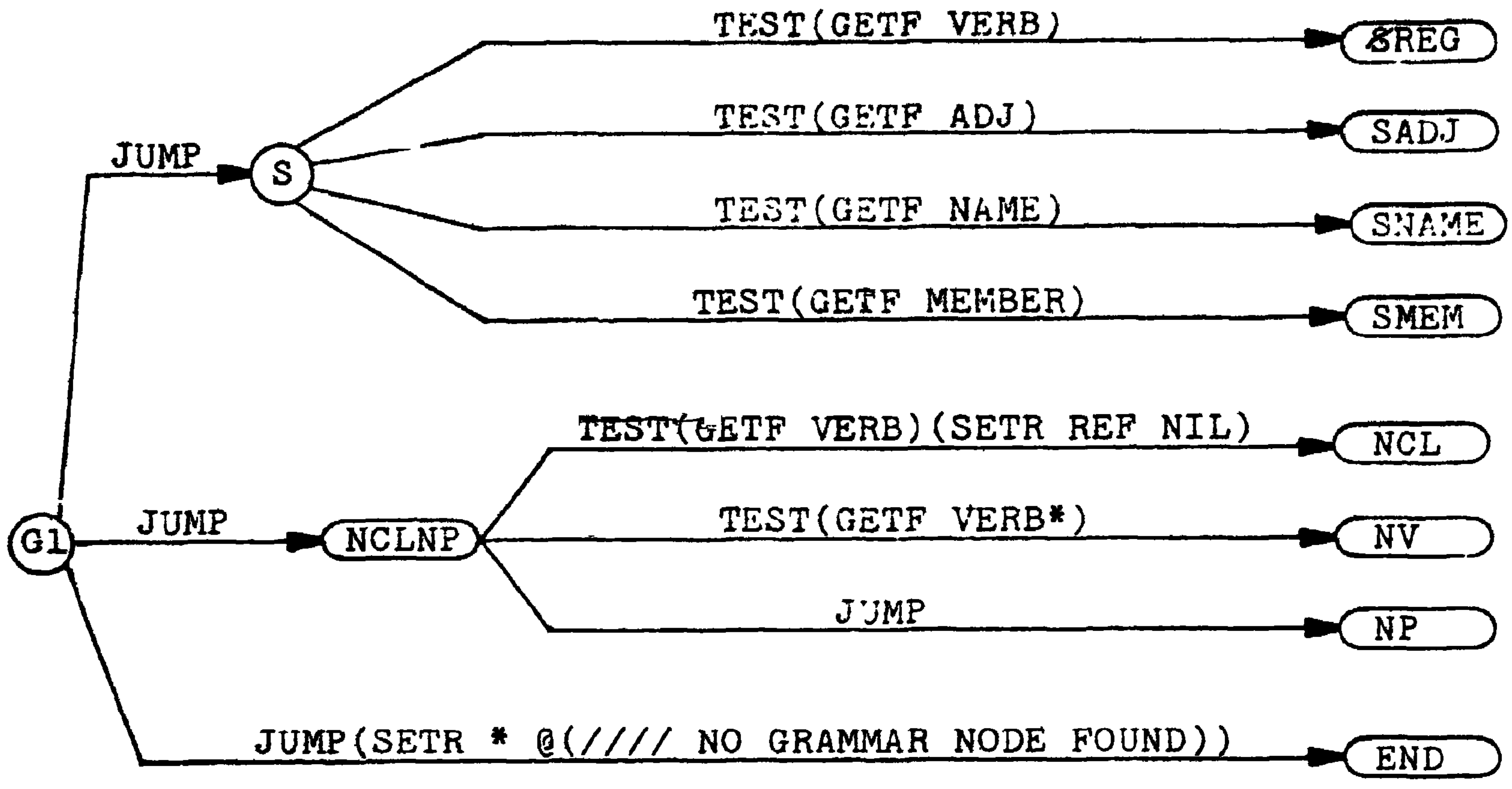


Figure 5: The default entry into the grammar network.

essential differences are those required by the differences between generating and parsing as discussed in the previous section.

(TEST test [action]*(TO gnode))

If the test is successful (evaluates to non-NIL), the actions are performed and generation continues at gnode. If the test fails, this edge is not taken. TEST is the same as Woods' TST, while TEST(GETF sarc) is analogous to Woods' CAT.

(JUMP [action]*(TO gnode))

Equivalent to (TEST T [action]*(TO gnode)). JUMP is similar in use to Woods' JUMP, but the difference from TEST T disappears since no edge "consumes" anything.

(MEM wform (word*) test [action]*(TO gnode))

If the value of wform has a non-null intersection with the list of words, the test is performed. If the test is also successful the actions are performed and generation continues at gnode. If either the intersection is null or the test fails, the edge

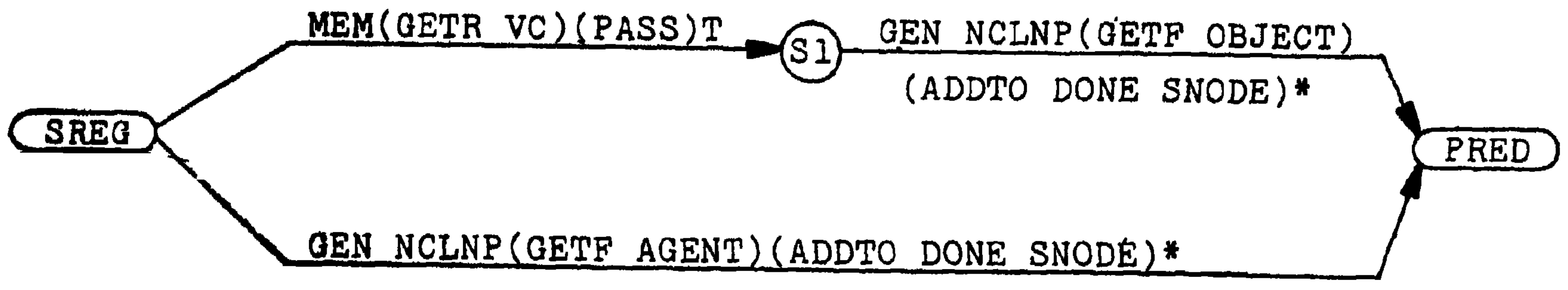


Figure 6: Generation of subject of subject-verb-object sentence.

is not taken. This is similar in form to Woods' MEM, but mainly used for testing registers.

(NOTMEM wform (word*) test [action]*(TO gnode))

This is exactly like MEM except the intersection must be null.

(TRANSR ([regname$_1$] regname$_2$ regname$_3$) test [action]*(TO gnode))

If regname$_1$ is present, the contents of regname$_2$ are added on the end of regname$_1$. If regname$_3$ is empty, the edge is not taken. Otherwise, the first element in regname$_3$ is removed and placed in regname$_2$ and the test is performed. If the test fails, the edge is not taken, but if it succeeds, the actions are performed and generation continues at gnode. TRANSR is used to iterate through several nodes all in the same semantic relation with the main semantic node.

(GEN gnode$_1$ sform [action]*regname [action]*(TO gnode$_2$))

The first set of actions are performed and the generation is called recursively with the semantic node that is the value of sform and at the grammar node gnode$_1$. If this generation is successful (returns non-NIL), the result is placed in the register regname, the second set of actions are performed and generation continues at gnode$_2$. If the generation fails, the edge is not taken. This is the same as Woods' PUSH but requires a semantic node to be specified and allows any register to be used to hold the result. Instead of having a POP edge, a return automatically occurs when

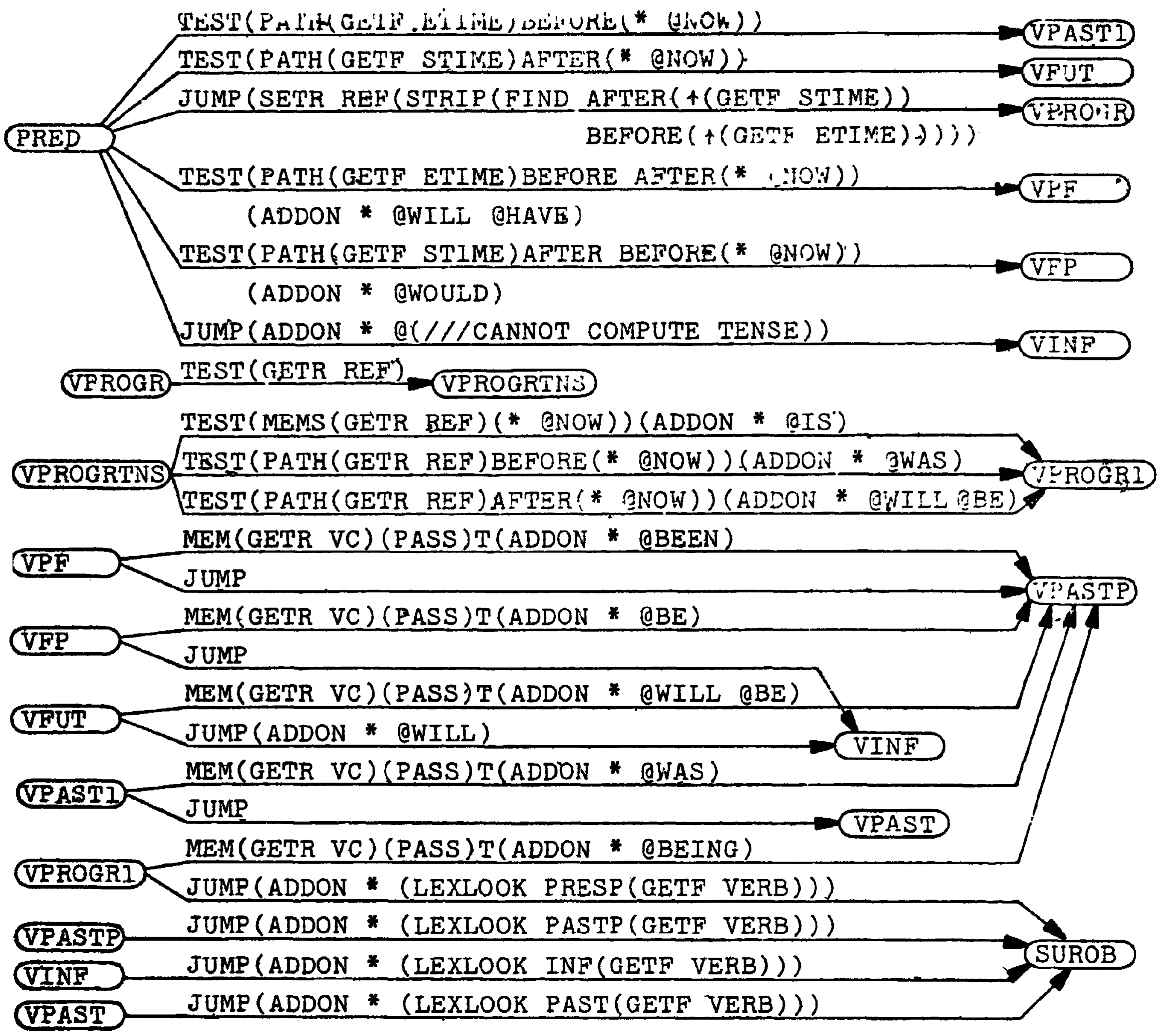

PRED
TEST(PATH(GETF ETIME)BEFORE(* @NOW))
VPAST1
TEST(PATH(GETF STIME)AFTER(* @NOW))
VFUT
JUMP(SETR REF(STRIP(FIND AFTER(↑(GETF STIME)) BEFORE(↑(GETF ETIME)))))
VPROGR
TEST(PATH(GETF ETIME)BEFORE AFTER(* @NOW)) (ADDON * @WILL @HAVE)
VPF
TEST(PATH(GETF STIME)AFTER BEFORE(* @NOW)) (ADDON * @WOULD)
VFP
JUMP(ADDON * @(///CANNOT COMPUTE TENSE))
VINF
VPROGR
TEST(GETR REF)
VPROGRTNS
VPROGRTNS
TEST(MEMS(GETR REF)(* @NOW))(ADDON * @IS)
TEST(PATH(GETR REF)BEFORE(* @NOW))(ADDON * @WAS)
TEST(PATH(GETR REF)AFTER(* @NOW))(ADDON * @WILL @BE)
VPROGR1
VPF
MEM(GETR VC)(PASS)T(ADDON * @BEEN)
JUMP
VPASTP
VFP
MEM(GETR VC)(PASS)T(ADDON * @BE)
JUMP
VFUT
MEM(GETR VC)(PASS)T(ADDON * @WILL @BE)
JUMP(ADDON * @WILL)
VINF
VPAST1
MEM(GETR VC)(PASS)T(ADDON * @WAS)
JUMP
VPAST
VPROGR1
MEM(GETR VC)(PASS)T(ADDON * @BEING)
JUMP(ADDON * (LEXLOOK PRESP(GETF VERB)))
VPASTP
JUMP(ADDON * (LEXLOOK PASTP(GETF VERB)))
VINF
JUMP(ADDON * (LEXLOOK INF(GETF VERB)))
VPAST
JUMP(ADDON * (LEXLOOK PAST(GETF VERB)))
SUROB

Figure 7: Tense generation network.

transfer is made to the node END. At that point, the contents of the register named * are returned.

(CONCAT form form*)

The forms are evaluated and concatenated in the order given. Performs a role analogous to that of Woods' BUILDQ.

(GETF sarc [sform])

Returns a list of all semantic nodes at the end of the semantic arcs labelled sarc from the semantic node which is the value

Tense	Active	Passive
past	broke	was broken
future	will break	will be broken
present progressive	is breaking	is being broken
past progressive	was breaking	was being broken
future progréssive	will be breaking	will be being broken
past in future	will have broken	will have been broken
future in past	would break	would be broken

Figure 8: The tenses of "break" which the network of Figure 7 can generate.

of sform. If sform is missing, SNODE is assumed. Returns NIL if there are no such semantic nodes. It is similar in the semantic domain to Woods' GETF in the lexical domain.

(GETR regname)

Returns the contents of register regname. It is essentially the same as Woods' GETR.

(LEXLOOK lfeat [sform])

Returns the value of the lexical feature, lfeat, of the lexical entry associated with the semantic node which is the value of sform. If sform is missing, SNODE is assumed. If no lexical entry is associated with the semantic node, NIL is returned. LEXLOOK is similar to Woods' GETR and as also in the lexical domain.

(SETR regname form)

The value of form is placed in the register regname. It is the same as Woods' SETR.

(ADDTO regname form*)

Equivalent to (SETR regname (CONCAT (GETR regname) form*)).

(ADDON regname form*)

Equivalent to (SETR regname (CONCAT form* (GETR regname))).

(MEMS form form)

Returns T if the values of the two forms have a non-null intersection, NIL otherwise.

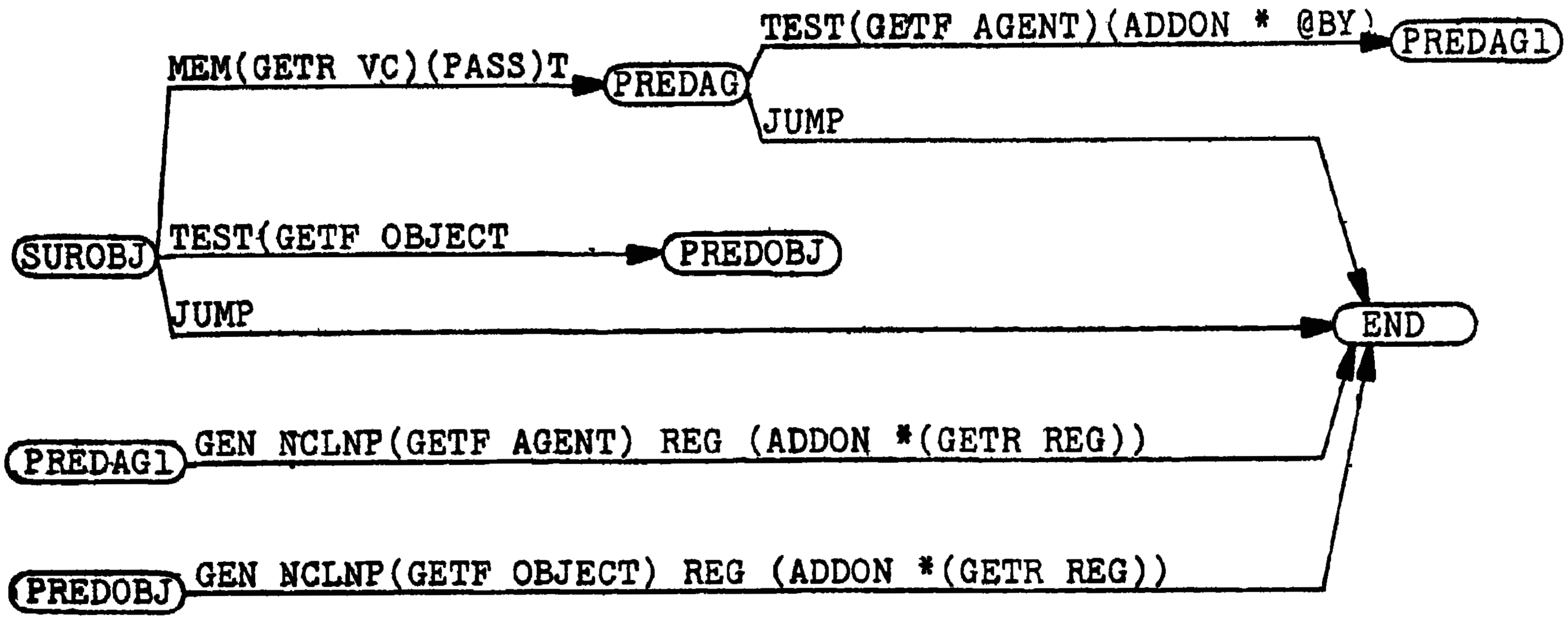

MEM(GETR VC)(PASS)T
PREDAG
TEST(GETF AGENT)(ADDON * @BY)
PREDAG1
JUMP
SUROBJ
TEST(GETF OBJECT
PREDOBJ
JUMP
END
PREDAG1
GEN NCLNP(GETF AGENT) REG (ADDON *(GETR REG))
PREDOBJ
GEN NCLNP(GETF OBJECT) REG (ADDON *(GETR REG))

Figure 9: Generating the surface object.

(PATH $sform_1$ sarc* $sform_2$)

Returns T if a path described by the sequence of semantic arcs exists between the value of $sform_1$ and $sform_2$. If the sequence is $sarc_1$ $sarc_2$ ... $sarc_n$, the path described is the same as that indicated by $sarc_1$* $sarc_2$* ... $sarc_n$*. If no such path exists, NIL is returned. (Remember, * means repeat one or more times.)

Discussion of an Example Grammar Network

The top level generator function, SNEG, is given as arguments a semantic node and, optionally, a grammar node. If the grammar node is not given, generation begins at the node G1 which should be a small discrimination net to choose the preferred description for the given semantic node. This part of the example grammar is shown in Figure 5. In it we see that the preferred description for any semantic node is a sentence. If no sentence can be formed a noun phrase will be tried. Those are the only presently available options.

Semantic nodes with an outgoing VERB edge can be described by a normal SUBJECT-VERB-OBJECT sentence. (For this example, we have not used additional cases.) First the subject is generated,

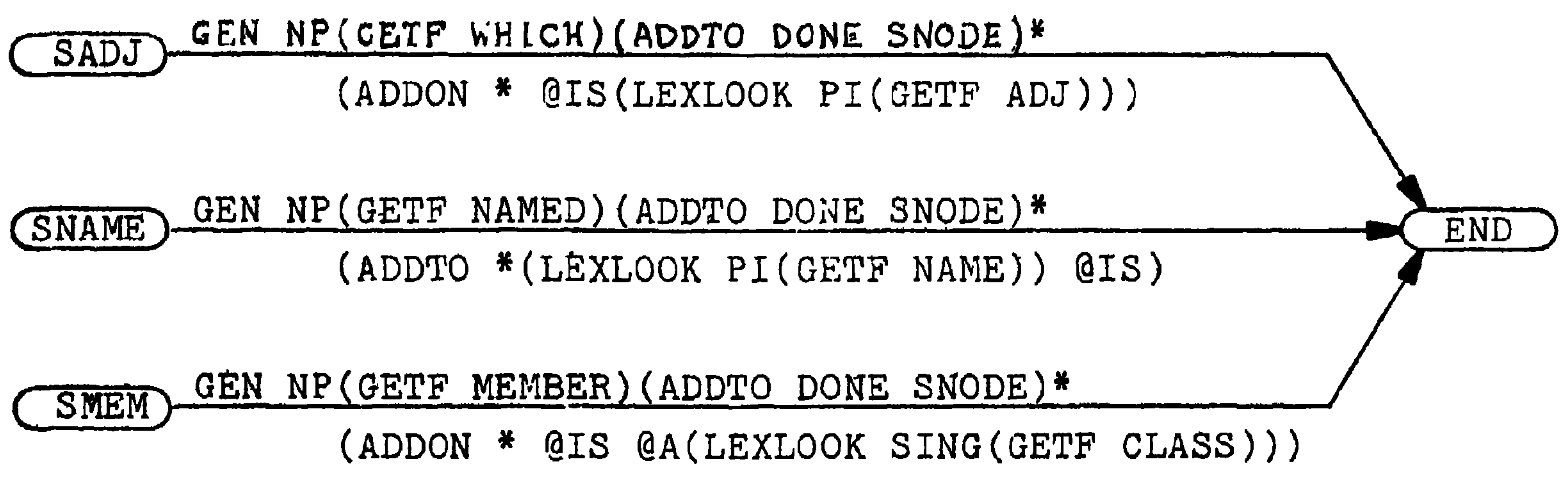

SADJ
GEN NP(GETF WHICH)(ADDTO DONE SNODE)*
(ADDON * @IS(LEXLOOK PI(GETF ADJ)))
SNAME
GEN NP(GETF NAMED)(ADDTO DONE SNODE)*
(ADDTO *(LEXLOOK PI(GETF NAME)) @IS)
SMEM
GEN NP(GETF MEMBER)(ADDTO DONE SNODE)*
(ADDON * @IS @A(LEXLOOK SING(GETF CLASS)))
END

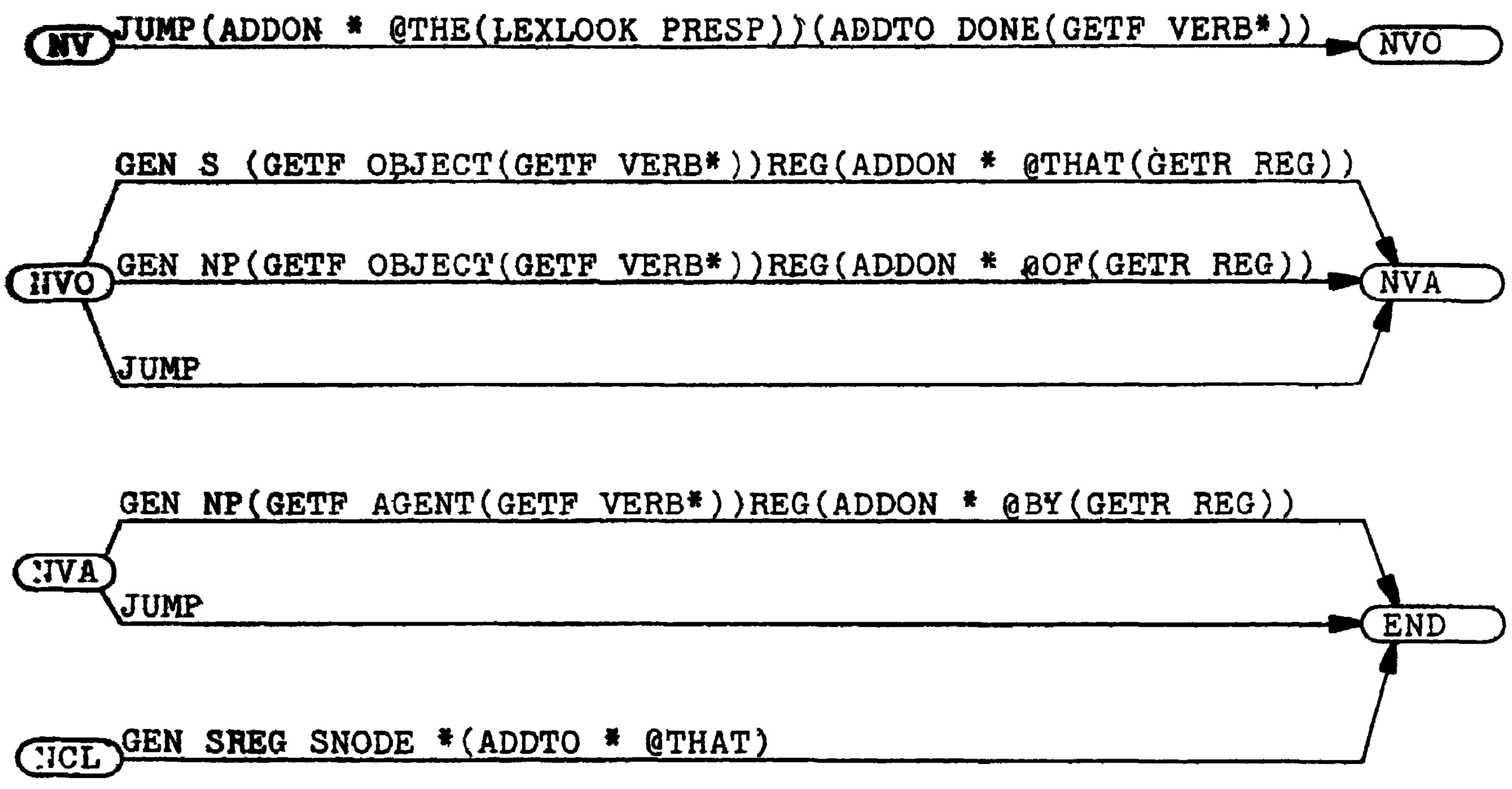
Figure 10: Generating the three "non-regular" sentences.

which depends on whether the sentence is to be in active or passive voice. Alternatively, the choice could be expressed in terms of whether the agent or object is to be the topic as suggested by Kay, 1975. Figure 6 shows the network that generates the subject. The register DONE holds semantic nodes for which sentences are being generated for later checking to prevent infinite recursion. Without it, node M0023 of Figure 1 would be described as, "A dog which kissed young sweet Lucy who was kissed by a dog which kissed..."

The initial part of the PRED network is concerned with generating the tense. This depends on the BEFORE/AFTER path between the starting and/or ending time of the action and the current value of NOW, which is given by the form (* @NOW). Figure 7 shows the tense generation network. Figure 8 shows the tenses this network is able to generate.

After the verb group is generated, the surface object is generated by describing either the semantic agent or object. Figure 9 shows this part of the network

The other three kinds of sentences are for describing nodes representing: (1) that something has a particular adjective attributable to it, (2) that something has a name, (3) that something is a member of some class. The networks for these are shown in Figure 10. Again, the DONE register is used to prevent such sentences as "Sweet young Lucy is sweet," "Charlie is Charlie," and "A dog is a dog."



NV
JUMP(ADDON * @THE(LEXLOOK PRESP))(ADDTO DONE(GETF VERB*))
NVO
GEN S (GETF OBJECT(GETF VERB*))REG(ADDON * @THAT(GETR REG))
NVO
GEN NP(GETF OBJECT(GETF VERB*))REG(ADDON * @OF(GETR REG))
NVA
JUMP
GEN NP(GETF AGENT(GETF VERB*))REG(ADDON * @BY(GETR REG))
NVA
JUMP
END
NCL
GEN SREG SNODE *(ADDTO * @THAT)

Figure 11: Generating nominalized verbs and sentences.

Fugure 5 showed three basic kinds of noun phrases that can be generated: the noun clause or nominalized sentence, such as "that a dog kissed sweet young Lucy"; the nominalized verb, such as "the kissing of sweet young Lucy by a dog"; the regular noun phrase. The first two of these are generated by the network shown in Figure 11. Here DONE is used to prevent, for example, "the kissing of sweet young Lucy who was kissed by a dog by a dog."

The regular noun phrase network begins with another descrimination net which has the following priorities: use a name of the object; use a class the object belongs to; use something else known about the object. A lower priority description will be used if all higher priority descriptions are already in DONE. Figure 12 shows the beginning of the noun phrase network. Adjectives are added before the name or before the class name and a relative clause is added after.

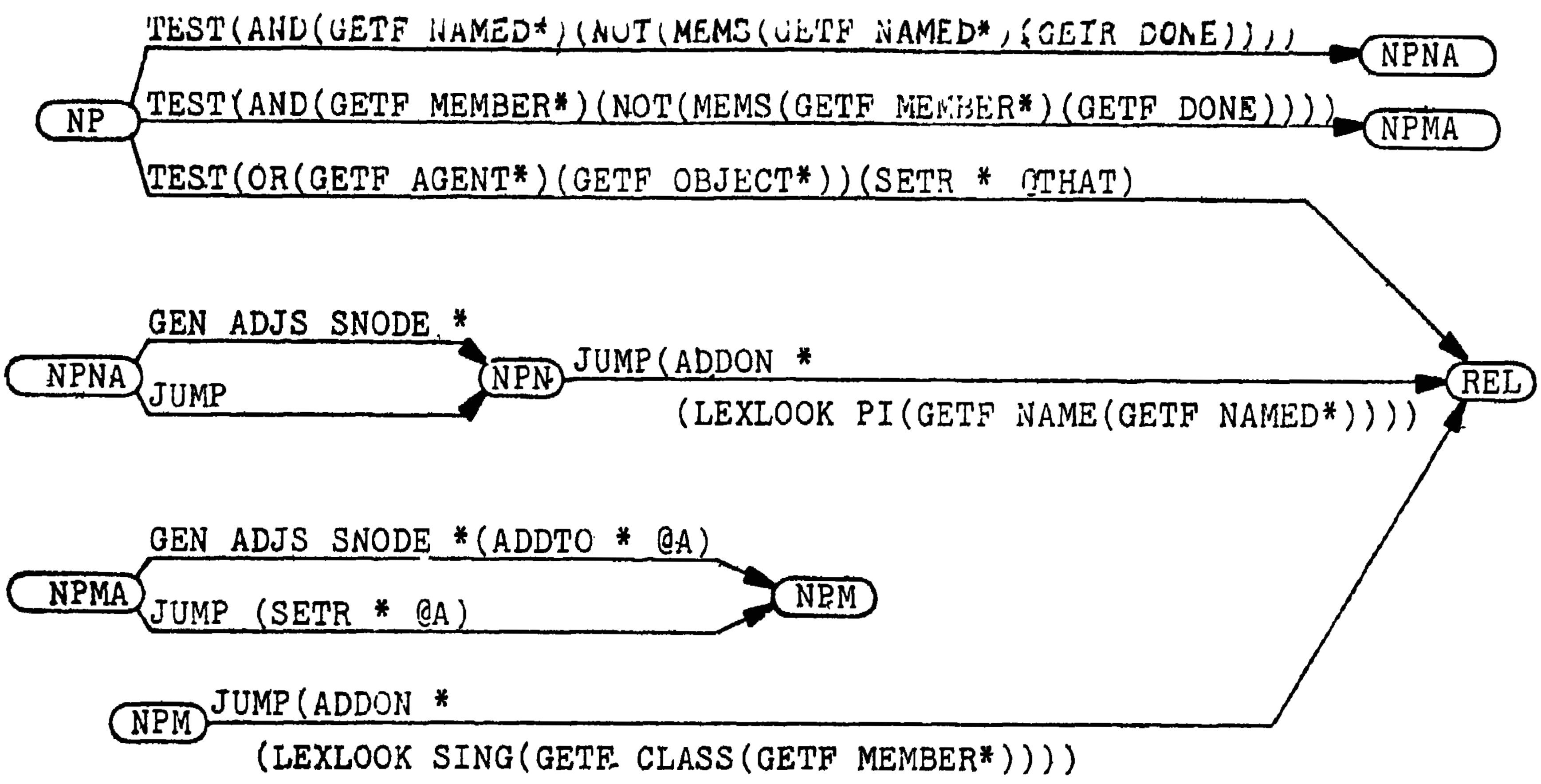
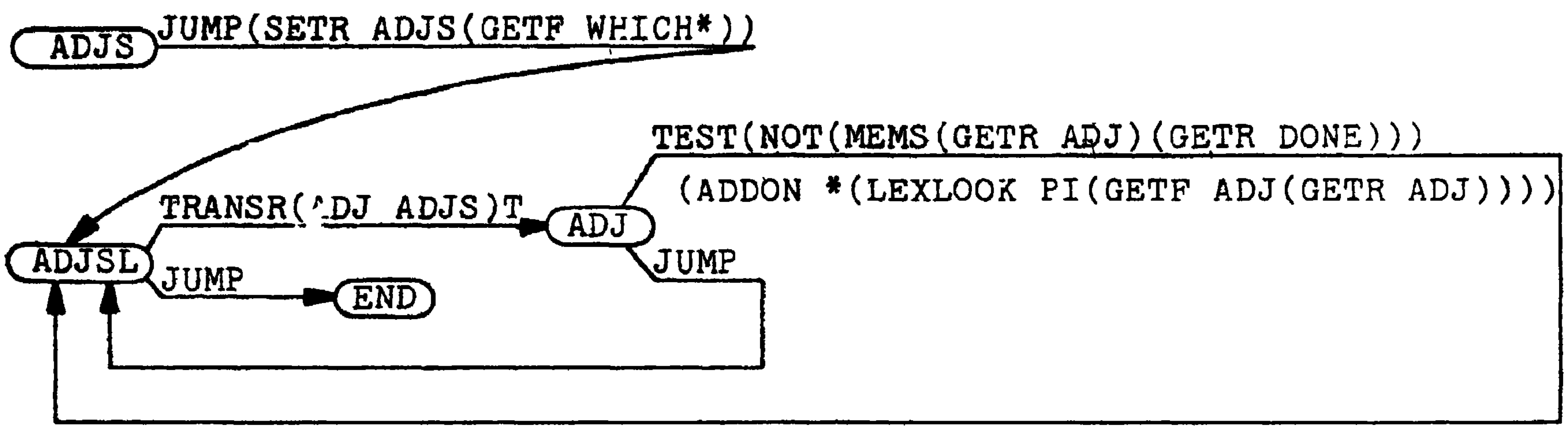

NP
TEST(AND(GETF NAMED*)(NOT(MEMS(GETF NAMED*)(GETR DONE)))
NPNA
TEST(AND(GETF MEMBER*)(NOT(MEMS(GETF MEMBER*)(GETF DONE))))
NPMA
TEST(OR(GETF AGENT*)(GETF OBJECT*))(SETR * @THAT)
NPNA
GEN ADJS SNODE *
JUMP
NPN
JUMP(ADDON *
(LEXLOOK PI(GETF NAME(GETF NAMED*))))
REL
NPMA
GEN ADJS SNODE *(ADDTO * @A)
JUMP (SETR * @A)
NPM
NPM
JUMP(ADDON *
(LEXLOOK SING(GETF CLASS(GETF MEMBER*))))

Figure 12: The beginning of the noun phrase network.

Figure 13 shows the adjective string generator and Figure 14 shows the relative clause generator. Notice the use of the TRANSR edges for iterating. At this time, we have no theory for determining the number or which adjectives and relative clauses to generate, so arbitrarily we generate all adjectives not already on DONE but only one relative clause. We have not yet implemented any ordering of adjectives. It is merely fortuitous that "sweet young Lucy" is generated rather than "young sweet Lucy". The network is written so that a relative clause for which the noun is the deep agent is preferred over one in which the noun is the deep object. Notice that this choice determines the voice of the embedded clause. The form (STRIP(FIND MEMBER (↑ SNODE) CLASS (FIND LEX PERSON))) is a call to a SNePS function that determines if the object is known to be a person, in which case "WHO" is used rather than "WHICH". This determination is made by referring to the semantic network rather than by including a HUMAN feature on the lexical entries for LUCY and CHARLIE.



ADJS
JUMP(SETR ADJS(GETF WHICH*))
TEST(NOT(MEMS(GETR ADJ)(GETR DONE)))
(ADDON *(LEXLOOK PI(GETF ADJ(GETR ADJ))))
TRANSR(ADJ ADJS)T
ADJ
ADJSL
JUMP
END
JUMP

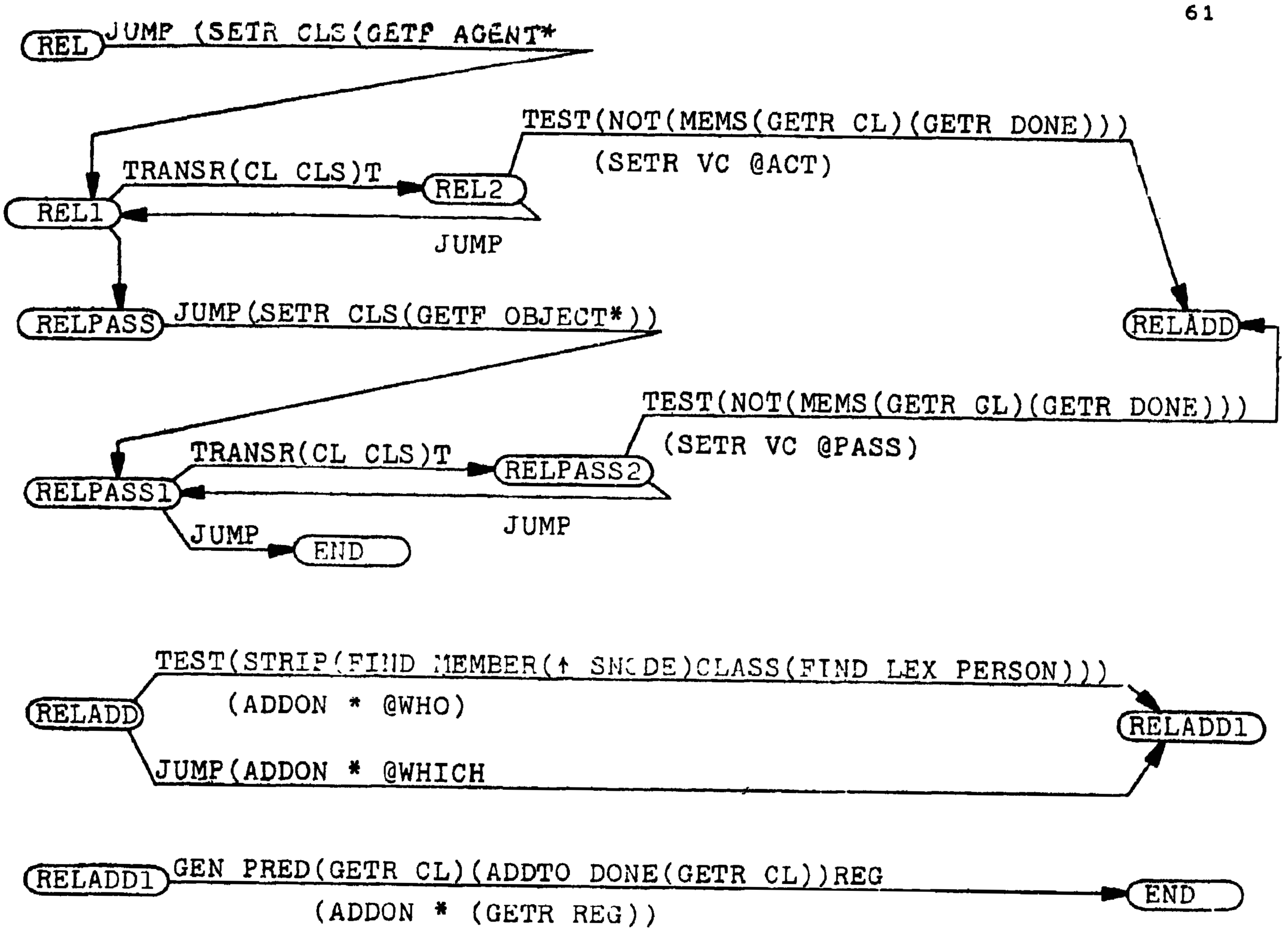
Figure 13: The network for generating a string of adjectives.

Notice that any information about the object being described by a noun phrase may be used to construct a relative clause even if that information derived from some main clause. Also, while the generator is examining a semantic node all the information about that node is reachable from it and may be used directly. There is no need to examine disjoint deep phrase markers to discover where they can be attached to each other so that a complex sentence can be derived.

Future Work

Additional work needs to be done in developing the style of generation described in this paper. Experience with larger and richer networks will lead to the following issues: describing a node by a pronoun when that node has been described earlier in the string; regulating verbosity and complexity, possibly by the use of resource bounds simulating the limitations of short term memory; keeping subordinate clauses and descriptions to the point of the conversation possibly by the use of a TO-DO register holding the nodes that are to be included in the string.

In this paper, only indefinite descriptions were generated. We are working on a routine that will identify the proper subnet of the semantic network to justify a definite description. This must be such that it uniquely identifies the node being described.



REL
JUMP (SETR CLS(GETF AGENT*
REL1
TRANSR(CL CLS)T
REL2
JUMP
TEST(NOT(MEMS(GETR CL)(GETR DONE)))
(SETR VC @ACT)
RELPASS
JUMP(SETR CLS(GETF OBJECT*))
RELADD
RELPASS1
TRANSR(CL CLS)T
RELPASS2
JUMP
TEST(NOT(MEMS(GETR CL)(GETR DONE)))
(SETR VC @PASS)
JUMP
END
RELADD
TEST(STRIP(FIND MEMBER(↑ SNODE)CLASS(FIND LEX PERSON)))
(ADDON * @WHO)
JUMP(ADDON * @WHICH
RELADD1
RELADD1
GEN PRED(GETR CL)(ADDTO DONE(GETR CL))REG
(ADDON * (GETR REG))
END

Figure 14: The relative clause generator.

Acknowledgements

The author is indebted to John Lowrance, who implemented the generator, Stan Kwasny, who implemented the parser, Bob Bechtel, who worked out the temporal representation, Nich Vitulli and Nick Eastridge, who implemented versions of SNePS, and Jim McKew for general software support. Computer service was provided by the IUPUI Computing Facilities. Typing and graphics were done by Christopher Charles.

References

Bruce, B.C. 1972. A model for temporal references and its application in a question answering program. Artificial Intelligence 3, 1, 1-25.

Kay, M. 1973. The MIND system. Natural Language Processing, R. Rustin (Ed.), Algorithmics Press, New York, 155-188.

Kay, M. 1975. Syntactic processing and functional sentence perspective. Theoretical Issues in Natural Language Processing R. Schank and B.L. Nash-Webber (Eds.), Bolt Beranek,& Newman, Inc., Cambridge, Massachusetts.

Schank, R.C.; Goldman, N.; Rieger, C.J., III; and Riesbeck, C. 1973. MARGIE: memory, analysis, response generation, and inference on English. Proc. Third International Joint Conference on Artificial Intelligence, Stanford University, August 20-23, 255-261.

Shapiro, S.C. 1971a. The MIND system: a data structure for semantic information processing. R-837-PR. The Rand Corp., Santa Monica, California.

Shapiro, S.C. 1971b. A net structure for semantic information storage, deduction and retrieval. 2nd International Joint Conference on Artificial Intelligence: Advance Papers of the Conference, British Computer Society, London, 512-523.

Shapiro, S.C. 1975. An introduction to SNePS. Technical Report No. 31, Computer Science Department, Indiana University, Bloomington,

Simmons, R.F. 1973. Semantic networks: their computation and use for understanding English sentences. Computer Models of Thought and Language, R.C. Schank and K.M. Colby (Eds.), W.H. Freeman and Co., San Francisco, 63-113.

Simmons, R.F., and Slocum, J. 1972. Generating English discourse from semantic nets. Comm. ACM 15, 10, 891-905.

Woods, W.A. 1973. An experimental parsing system for transition network grammars. Natural Language Processing, R. Rustin (Ed.), Algorithmics Press, New York, 111-154.

American Journal of Computational Linguistics *Microfiche 33 : 63*

Speech Generation from Semantic Nets

JONATHAN SLOCUM

Artificial Intelligence Center
Stanford Research Institute
Menlo Park, California 94025

ABSTRACT

Natural language output can be generated from semantic nets by processing templates associated with concepts in the net. A set of verb templates is being derived from a study of the surface syntax of some 3000 English verbs: the active forms of the verbs have been classified according to subject, object(s), and complement(s); these syntactic patterns, augmented with case names, are used as a grammar to control the generation of text. This text in turn is passed through a speech synthesis program and output by a VOTRAX speech synthesizer. This analysis should ultimately benefit systems attempting to understand English input by providing surface structure to deep case structure maps using the same templates as employed by the generator.

Acknowledgment

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by the U. S. Army Research Office under Contract No. DAHC04-75-C-0006.

INTRODUCTION

If computers are to communicate effectively with people, they must speak, or at least write, the user's natural language. The bulk of the work in computational linguistics has been devoted to computer understanding of natural language input, but relatively little effort has been expended in developing natural language output. Most English output systems have been along the line of "fill in the blank" with perhaps some semantic constraints imposed; there have been few attempts at language generation from what one could call "semantic net" structures (Simmons and Slocum, 1972; Slocum, 1973; Goldman, 1974).

Perhaps generation is considered a much easier problem. The success of understanding efforts is generally believed to depend on some workable theory of "discourse organization" which would account for effects of context and would show how anaphoric expressions (pronouns and noun phrases) are resolved and how sentences are ordered in the output. As it happens, these mechanisms are precisely those that a "response generator" must incorporate if it is to appear intelligent. The study of generation will play an important role in solving the problem of understanding if it can demonstrate a mapping from deep semantic structures to surface strings.

Let us briefly outline some relevant processes in the speech understanding system being developed by SRI and SDC (Walker et al., 1975, and Ritea, 1975). The user initiates a session by establishing communication with the system; all subsequent dialog (input and output) is monitored by a "discourse module" (Deutsch, 1975) to maintain an accurate conversational context. An executive coordinates various knowledge sources -- acoustic, prosodic, syntactic, semantic, pragmatic, and discourse -- to "understand" successive utterances.

The analyzed utterance is then passed to the "responder" -- another component of the discourse module. The responder may call the question-answerer if the input is a question; it may call a data base update program if the input is a statement of fact; or it may decide on some other appropriate reply. The content of the response is passed to the generator, perhaps with some indication of how it is to be formulated. The reply may be a stereotyped response ("yes", "no", "I see"), a noun phrase (node), a sentence (verb node), or, eventually, a paragraph.

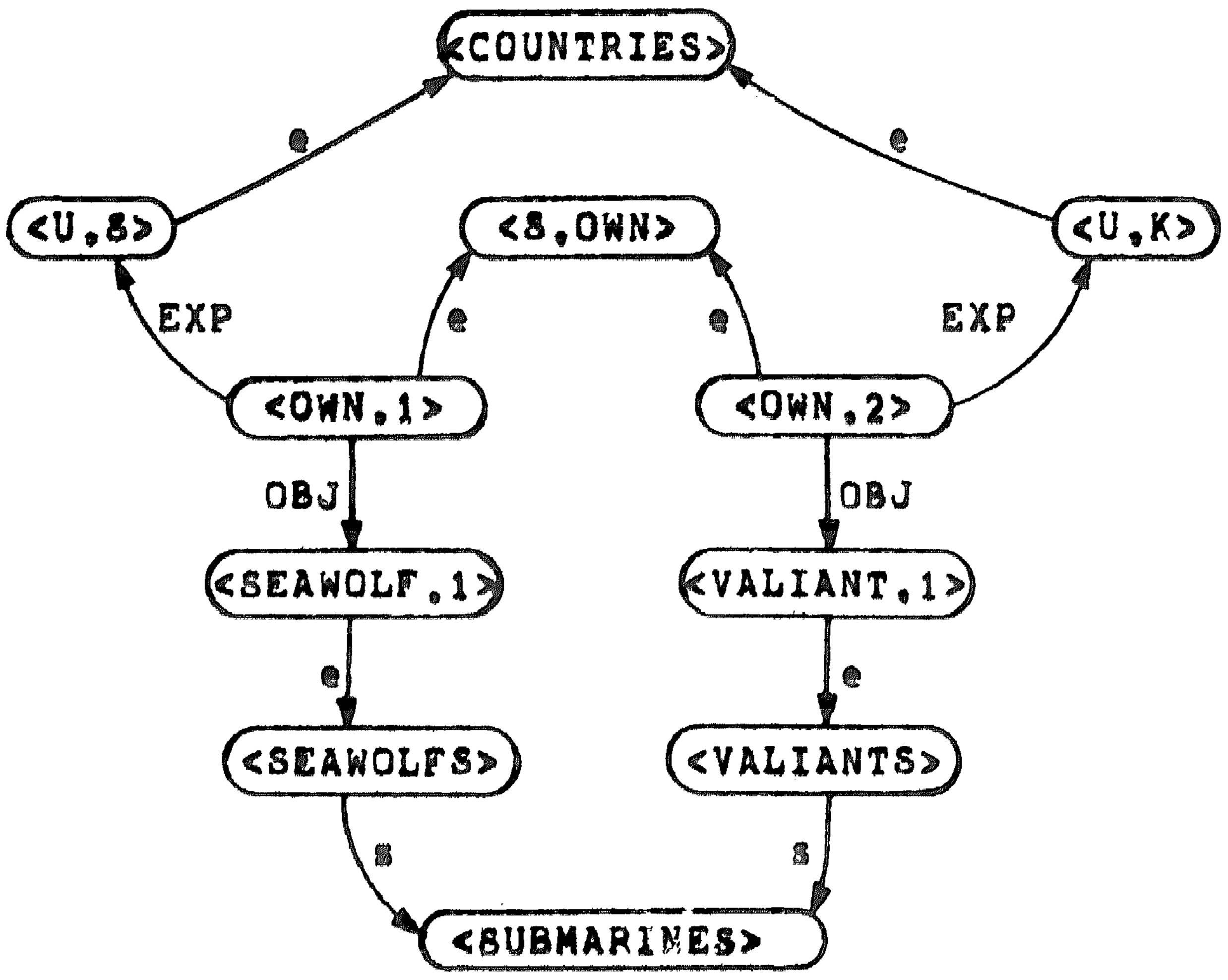
The generator outputs stereotyped responses immediately; if the response is more complicated (a "noun" node, "verb" node, or eventually a network), a more detailed program is required. This program will determine exactly how the response is to be formulated -- as an NP, S, or sequence of Ss; it may be required to choose verbs and nouns with which to express the deep case net structures, as well as a syntactic frame for the generation. The generator produces the response in "text" form; this in turn is passed to a speech synthesis program for transformation and output by a commercial VOTRAX speech synthesizer. Currently no sentence intonation or stress contouring is being performed. Since the major interest of this paper is in "text" generation, no further reference to the synthesis step will be made.

CONSTRAINTS ON RESPONDING

There are several considerations involved in responding appropriately to an utterance. First, there are "conversational postulates" (Gordon and Lakoff, 1975) shared by the users of a language; these postulates serve to constrain the content and form of communications from the speaker to the hearer. For instance, the speaker should not tell the hearer something the hearer already knows, lest he be bored; yet the speaker cannot tell the hearer something the hearer knows absolutely nothing about, or the hearer will not comprehend. The speaker should relate the news in his message to the prior knowledge of the hearer; this requires the speaker to have a model of the hearer These heuristics must operate in conjunction with a "response producer" to constrain what may be output by a "sentence" generator. We are only beginning to understand how to incorporate these postulates in a language processing system.

Then there is the matter of constructing the basic sentence Normal English syntax requires at least one verb in the sentence; choosing a main verb constrains the surface structure. For instance, in the absence of compounds any verbs other than the main verb will have to appear in another form: nominal, infinitive, gerund, participle, or subordinate clause. How does the relevant information contained in a semantic net indicate the appropriate form? The traditional answer is "by means of the lexicon." We will explore the relationship between net and lexicon and advance a methodology for representing a map from deep case structure to surface structure.

This paper focuses on a philosophy of single-sentence formatting: choosing a main verb, choosing the gross structure of the output sentence, and deciding how to generate appropriate noun phrases. Our examples will employ simplified semantic net structures, somewhat like those in the actual SRI "partitioned semantic net" system (Hendrix, 1975). Nodes in the net may represent physical objects, relationships, events, sets, rules, or utterances, as in the example below. Directed labelled arcs connect nodes and represent certain "primitive" time-invariant relationships.



In the net fragment above, the U.S. and the U.K. are elements (e) of the set of countries. As EXPeriencers they each participate in OWNing situations involving as OBJects particular submarines; each submarine is an element of some class of submarines, and these classes are subsets (s) of the set of all submarines.

GENERATION TEMPLATES

The first requirement for generation is to derive some templates for English sentences. We choose a simple verb for demonstration -- OWN. We note that our verb has several "synonyms": HAVE, POSSESS, and BELONG (TO). Since each of these verbs (including OWN) has other sense meanings, we posit a node <S.OWN> in the net that corresponds to the abstract "ownership" sense they have in common; this node will be the "prototypical" OWN, in that it will incorporate the "meaning" of the situation of owning (including any semantic constraints on its arguments), and in that all instances of owning situations will be related to it. With this node we will associate the appropriate verbs (OWN, POSSESS, HAVE, BELONG) and templates. Note that one template will not suffice for all four verbs; for instance, the subject of BELONG is the OBJect entity, while in the other (active) verbs the subject is the EXPeriencer:

```
EXP owns OBJ ; OBJ is owned by EXP
EXP possesses OBJ ; OBJ is possessed by EXP
EXP has OBJ ; OBJ belongs to EXP
```

So we propose the corresponding templates:

```
[OWN (EXP Vact OBJ) (OBJ Vpas BY EXP)]
[POSSESS (EXP Vact OBJ) (OBJ Vpas BY EXP)]
[HAVE (EXP Vact OBJ)] [BELONG (OBJ Vact TO EXP)]
```

Now, in order to speak about a particular owning situation, we pursue the hierarchy to find the "canonical" S.OWN, choose a verb (say, BELONG) and an associated template (OBJ Vact TO EXP), and generate the constituents consecutively.

But we have a problem; there is no indication of how the EXP and OBJ arguments are to be generated. NP will not always suffice; note for instance that the predicate argument of "hope" in "John hoped to go home" must be an infinitive phrase (rather than the gerund phrase that NP might produce). Even a cursory study of a few hundred verbs in the language shows that they have very definite (and regular) constraints on the syntactic form of their constituents. These constraints appear to be matters for the lexicon rather than the grammar. Therefore, we associate verbs and templates with word senses (prototypical nodes in the net) rather than implement them via grammar rules, and we explicitly incorporate the constituent types in the templates:

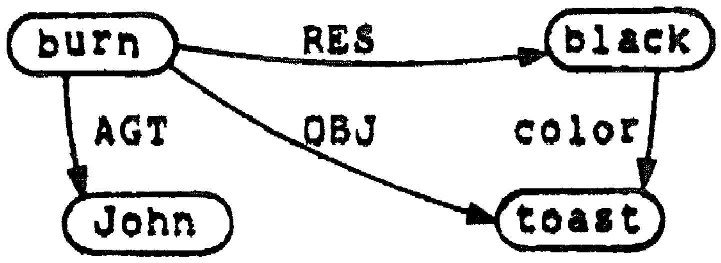
```
[OWN ((NP EXP) Vact (NP OBJ)) ((NP OBJ) Vpas BY (NP EXP))]
[POSSESS ((NP EXP) Vact (NP OBJ)) ((NP OBJ) Vpas BY (NP EXP))]
[HAVE ((NP EXP) Vact (NP OBJ))]
[BELONG ((NP OBJ) Vact TO (NP EXP))]
```

A set of patterns like these is associated with every "prototype verb" node in the knowledge base. It would seem that all we need is an interpreter that, given any "verb instance" node in the knowledge base, looks up the patterns for that type of node, chooses a verb, a corresponding template for the verb, and then proceeds to "evaluate" the pattern:

```
verb [OWN.1-->S.OWN] --> belong
temp --> [(NP OBJ) Vact TO (NP EXP)]

(NP OBJ) --> the Seawolf
Vact --> belongs
TO --> to
(NP EXP) --> the U.S.
```

But we still run into trouble with our simple scheme. Consider the sentence, "John burned the toast black."



By using the simple pattern ((NP AGT) Vact (NP OBJ)) we could easily generate the "incorrect" sentence, "John burned the black toast," since (NP OBJ) might include the color of the toast. We need a pattern more like ((NP AGT) Vact (NP OBJ) (Mod RES)), in which the RESult of the action will be directly related to the verb. However, this is not quite enough -- at least, not without a very complicated interpreter -- because the interpreter must know that (NP OBJ) cannot include the verb's RES argument (black). Thus, by convention, we may indicate an extra argument to be passed to a constituent generator (such as the function NP) to denote the item(s) not to appear in the resultant constituent:

((NP AGT) Vact (NP OBJ RES) (Mod RES))

The pattern (NP OBJ RES) means "generate an NP using the OBJect of the verb, but do not include the RESult of the verb in the NP." This convention actually prevents enormous proliferation of patterns (i.e., a pattern copy for every possible "missing" constituent). This level of detail would be unreasonable if few other verbs could use this template; however, there are more than a hundred verbs that share this same pattern. Since there are relatively few templates, each shared by several tens or hundreds of verbs, the use of templates proves to be quite helpful.

There are other sources of potential pattern proliferation, an important one being the combinatorial arrangements of case arguments of time, manner, and other such adverbials, as well as other (possibly non-adverbial) case arguments such as source, goal, instrument, etc. Some of these arguments are rather constrained in their positions in the sentence, but others may appear almost anywhere:

> "Yesterday the ship sailed from the lighthouse to the dock."
> "The ship sailed from the lighthouse to the dock yesterday."
> "Yesterday the ship sailed to the dock from the lighthouse."

It is of course unreasonable to try to maintain all the possible patterns; instead we leave insertion of these adverbial arguments to a single heuristic routine (described below). There are several justifications for this, among them: (1) the particular form of the verb cannot be generated until the subject object(s) and complement(s) have been generated, (2) these adverbials are so universal as to appear in almost any of the patterns and in several possible places, and (3) there are some heuristic constraints involved in the placement of arguments.

One may question whether passive templates should be stored; certainly, they could be derived. On the other hand, neglecting to store them would force us to indicate with each verb (sense), whether it can (or, sometimes, must) be passivized. Indicating "transitive" is not enough since there are transitive verbs (i.e., verbs that take an object) that cannot be passivized. Since we have to store the information anyway, we can save some code and computing time by storing the passive template.

There are several reasons for generating the verb after the major arguments. First the subject must be generated so that the verb can be made to agree in number. Second, certain word senses are true of verb-particle combinations while not of the isolated verb. Since, in addition, particles must appear after objects that are short (like pronouns) but before objects that are long (like noun phrases), the particle must be positioned after the object is generated. Finally, insertion of some adverbials (e.g. "not") requires an auxiliary verb -- thus verb generation must follow adverbial generation.

VERB PATTERNS

This study started with the 25 "verb patterns" presented by Hornby (1954). These in turn came from a dictionary by Hornby et al., (1948). Verbs in the dictionary are classified according to their gross syntactic patterns of subject, object(s), and complement(s); most of the patterns are sub-divided. The authors claim that these patterns account for all constructions involving all the verbs in their dictionary -- and, by extension, in the language. This classification is not immediately useful to computational linguists since it does not address underlying semantics. Nevertheless, it is clear that it can serve as the basis for a derivation of underlying case structures and, particularly, as a basis for "generation templates."

These patterns are being converted into templates much like those derived earlier; the analysis is being performed with respect to about 3000 verbs drawn from the dictionary (Slocum, to appear). These templates serve as the major portion of a modular "generation grammar," with the remainder in the form of heuristic functions for constructing syntactic constituents.

NOUN PHRASES

What to include in a noun phrase should be another matter for the discourse module to judge. There are no well-formulated rules accounting for anaphora in English; indeed, there are few well-established parameters other than that the hearer must be able to resolve the (pro)nouns to their referents. The speaker should employ anaphora in order to avoid repetition, but only if his model of the hearer indicates that the hearer can resolve the ambiguity. There are some low-power pronominalization rules that could be directly incorporated in a generator -- reflexivization, for example. Nevertheless, it is important to realize that when a generator is unaware of the conversational context, it should not independently decide how to generate noun phrases; it can only decide when to do so. This situation has not been universally recognized, but it is becoming increasingly clear that a discourse module must be consulted during the generation phase. The discourse module will not know ahead of time what NPs are to be produced unless it performs many of the same operations that the generator would do anyway. Yet the context-sensitive decision strategy may have to resort to such measures as disambiguating the proposed output using the model of the hearer in order to determine what anaphora is resolvable. It is unreasonable to incorporate this strategy in the generator, since for many reasons it must be part of the discourse module.

Therefore the generator should pass any "noun" constituent to the discourse module (perhaps with its recommendation about how to produce the constituent); the module must determine if a pronoun or bare noun is ambiguous to the hearer, and, if so, what to add to the noun in order to make the desired referent clear. In the current SRI system, noun patterns (Slocum, to appear) are used to control noun phrase generation. Much like verb patterns, noun patterns order the constituents in the phrase and indicate how each constituent is to be generated by naming a function to be called with the network constituent:

[(DET) (Adj QUAL) (Adj SIZE) (Adj SHAPE) (Adj COLOR) (N)]

Patterns like this are distributed about the network hierarchy; in the future, the discourse module will decide for each pattern constituent whether it is to appear in the phrase.

HEURISTIC RULES

Hornby describes three basic positions for adverbs in the clause: "front" position, "mid" position, and "end" position. Front position adverbs occur before the subject: "Yesterday he went home; from there he took a taxi." The interrogative adverbs (e.g. how, when) are typically constrained to front position; others may appear there for purposes of emphasis or contrast.

Mid position adverbs occur with the verb (string); if there are modal or auxiliary verbs, the adverb occurs after the first one. Otherwise the adverb will appear before the verb, except for "unstressed" finites of be, have, and do: "We often go there"; "she is typically busy"; "he is still waiting."

End position adverbs occur after the verb and after any direct or indirect object present. While relatively few clauses have more than one adverb in front position or more than one in mid position, it is common for several adverbs to appear in end position in the same clause: "they play the piano poorly together".

Adverbials of time (answering the question, "when?") usually occur in end position, but may appear in front position for emphasis or contrast. Adverbials of frequency (answering the question, "how often?") can be split into two groups. The first group is composed of single-word adverbs that typically occur in mid position but also may be in end position; the second is composed of multiple-word phrases that appear in end position or, less frequently, in front position. Adverbs of duration ("[for] how long?") usually have end position, with front position for emphasis or contrast. Adverbs of place and direction normally have end position. Adverbs of degree and manner have mid or end position, depending on the adverb.

Along with such rules concerning the positions of various types of adverbs, there must be a mechanism to order the adverbs that are to occur in the "same" position. There are some heuristics: among adverbials of time (or place) the smaller unit is usually placed first, unless it is added as an afterthought: "the army attacked the village in force on a hot August afternoon, just after siesta". Adverbials of place and direction usually precede those of frequency, which in turn precede those of time.

These rules are implemented in the same routine that produces the verb; when a template is first interpreted -- much as a sequence of function calls -- the "Vact" or "Vpas" keys are ignored. Once the subject, object(s) and complement(s) indicated by the template are generated, this "clean up" routine is called. It employs the heuristics described above to add the adverbial constituents and verb, then concatenates the constituents to produce a complete clause.

DISCUSSION

In theory, the set of possible English sentences is infinite. The obvious question then arises, "If one tries to account for them with templates, won t there be an infinite number of templates?" The simple answer is, "No, for some of the same reasons that allow a finite grammar to generate an infinite number of strings." One can produce sentences of arbitrary length by (1) arbitrary embedding, and (2) arbitrary conjunction. One does not do so by including arbitrary numbers of distinct case arguments. Even so the number of basic patterns could be extremely large. Evidence, however, is to the contrary: the eventual number of templates would appear to be several times the number of patterns, owing to the substitution of particular prepositions for "prep" in the syntactic patterns, and the assignment of different case names to a particular constituent depending on the particular verb used.

REFERENCES

Deutsch, Barbara G. Establishing Context in Task-Oriented Dialogs. Presented at the Thirteenth Annual Meeting of the Association for Computational Linguistics, Boston, Massachusetts, 30 October - 1 November 1975.

Goldman, Neil M. Computer Generation of Natural Language from a Deep Conceptual Base. AI Memo 247, Artificial Intelligence Laboratory, Stanford University, Stanford, California, 1974.

Gordon, David, and Lakoff, George. Conversational Postulates. Syntax and Semantics, Volume 3: Speech Acts. Edited by Peter Cole and Jerry L. Morgan. Academic Press, New York, 1975.

Hendrix, Gary G. Expanding the Utility of Semantic Networks through Partitioning. Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, 3-8 September 1975, 115-121.

Hornby, A. S., Gatenby, E. V., and Wakefield, H. The Advanced Learner's Dictionary of Current English. Oxford Press, London, 1948.

Hornby, A. S. A Guide to Patterns and Usage in English. Oxford Press, London, 1954.

Ritea, H. Barry. Automatic Speech Understanding Systems. Proceedings, Eleventh Annual IEEE Computer Society Conference, Washington, D. C., 9-11 September 1975.

Simmons, Robert F., and Slocum, Jonathan. Generating English Discourse from Semantic Networks. Communications of the ACM, 1972, 15, 891-905.

Slocum, Jonathan. Question Answering via Canonical Verbs and Semantic Models: Generating English from the Model. Technical Report NL-13, Department of Computer Sciences, University of Texas, Austin, Texas, January 1973.

Slocum, Jonathan. Verb Patterns and Noun Patterns in English: A Case Analysis. Artificial Intelligence Center, SRI, Menlo Park, California, (in preparation).

Walker, Donald, E., et al. Speech Understanding Research. Annual Report, Project 3804, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California, June 1975.

American Journal of Computational Linguistics *Microfiche 33 : 78*

Using Planning Structures to Generate Stories

JAMES R. MEEHAN

Yale University
New Haven, Connectiuct 06511

ABSTRACT

TALE-SPIN is a program which makes up stories by using planning structures as part of its world knowledge. Planning structures represent goals and the methods of achieving those goals. Requirements for a particular method depend on static and dynamic facts about the world. TALE-SPIN changes the state of the world by creating new characters and presenting obstacles to goals. The reader / listener makes certain plot decisions during the telling of the story. The story is generated using the notation of Conceptual Dependency and is fed to another program which translates it into English.

<u>INTRODUCTION</u> TALE-SPIN is a computer program which makes up stories about characters who plan how to solve certain problems

---

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract N00014-75-C-1111.

and then carry out their plans. The planning procedures interact with a data base of knowledge about other characters and objects in the world, memory, and the personal relationships which exist between characters. The stories are represented in Conceptual Dependency and are passed to a program which expresses them in English. The reader is asked to make certain decisions about the story during the process of generation. Here is an example.

```
     JOE BEAR WAS FAMISHED.  HE DECIDED HE WOULD BE FULL IF HE
ATE  SOME  HONEY.  HE WANTED TO FIND OUT WHERE THE HONEY WAS.
HE THOUGHT THAT IRVING BIRD WOULD TELL  HIM  WHERE  THE  HONEY
WAS.
     JOE BEAR WALKED TO THE TREE WHERE IRVING  BIRD  WAS.   HE
ASKED IRVING BIRD IF HE WOULD TELL HIM WHERE THE HONEY WAS.

>> DECIDE:  DOES *IRVINGBIRD* AGREE?  *NO

     IRVING BIRD REFUSED TO TELL JOE BEAR WHERE THE HONEY WAS.
JOE  BEAR  DECIDED IRVING BIRD WOULD LIKE HIM TO GIVE THE WORM
TO HIM.  JOE BEAR ASKED IRVING BIRD IF HE WOULD TELL HIM WHERE
THE HONEY WAS IF HE GAVE THE WORM TO HIM.

>> DECIDE:  DOES *IRVINGBIRD* AGREE?  *YES

     HE THOUGHT THAT HE WOULD LIKE JOE BEAR TO GIVE IT TO HIM.
HE  WANTED  TO  FIND  OUT WHERE THE WORM WAS.  HE THOUGHT THAT
IRVING BIRD WOULD TELL HIM WHERE THE WORM WAS.  JOE BEAR ASKED
IRVING BIRD IF HE WOULD TELL HIM WHERE THE WORM WAS.

>> DECIDE:  DOES *IRVINGBIRD* AGREE?  *YES

     IRVING BIRD DECIDED HE WOULD TELL JOE BEAR WHERE THE WORM
WAS.  IRVING BIRD TOLD HIM IT WAS AT A PATCH OF GROUND.
     HE WALKED TO THE PATCH OF GROUND WHERE WORM WAS.  HE TOOK
THE  WORM   HE  WALKED TO THE TREE WHERE IRVING BIRD WAS.  HE
GAVE THE WORM TO IRVING BIRD.

>> DECIDE:  DOES *IRVINGBIRD* KEEP HIS PROMISE?  *NO

     HE REFUSED TO TELL JOE BEAR WHERE THE  HONEY  WAS.   JOE
BEAR TOLD IRVING BIRD HE IS GOING TO STRIKE HIM IF HE DOES NOT
TELL HIM WHERE THE HONEY WAS.

>> DECIDE:  DOES *IRVINGBIRD* IGNORE THE THREAT?  *NO

     IRVING BIRD DECIDED HE WOULD  TELL  JOE  BEAR  WHERE  THE
HONEY WAS.  IRVING BIRD TOLD HIM IT WAS AT THE BEEHIVE.
```

```
     JOE BEAR THOUGHT THAT HENRY BEE WOULD GIVE THE  HONEY  TO
HIM.  JOE BEAR WALKED TO THE BEEHIVE WHERE HENRY BEE WAS.  HE
ASKED HENRY BEE IF HE WOULD GIVE THE HONEY TO HIM.

>> DECIDE:  DOES *HENRYBEE* AGREE?  *YES

     HENRY BEE DECIDED HE WOULD GIVE IT TO  JOE  BEAR.   HENRY
BEE GAVE IT TO JOE BEAR.  HE ATE IT.  HE WAS FULL.  THE END.
```

Here is a story which TALE-SPIN generates which the translator is not yet capable of producing in English:

```
     JOE BEAR WAS HUNGRY.  HE THOUGHT THAT IRVING  BIRD  WUOLD
TELL  HIM  WHERE  SOME HONEY WAS.  HE WALKED TO THE TREE WHERE
IRVING BIRD WAS.  HE ASKED IRVING BIRD TO TELL HIM  WHERE  THE
HONEY  WAS.  IRVING BIRD TOLD HIM THE HONEY WAS IN A [certain]
BEEHIVE.
     JOE BEAR WALKED TO THE BEEHIVE WHERE THE HONEY  WAS.   HE
ASKED  HENRY  BEE  TO  GIVE HIM THE HONEY.  HENRY BEE REFUSED.
JOE BEAR TOLD HIM WHERE SOME FLOWERS  WERE.   HENRY  BEE  FLEW
FROM THE BEEHIVE TO THE FLOWERBED WHERE THE FLOWERS WERE.  JOE
BEAR ATE THE HONEY.
     HE WAS VERY TIRED.  HE WALKED TO  HIS  CAVE.   HE  SLEPT.
THE END.
```

TALE-SPIN starts with a small set of characters and various facts about them. It also has a set of problem-solving procedures which generate the events in the story. Many decisions have to be made as the story is being told. Some are made at random (names of characters, for example); others depend on the relationships between characters (whom one asks for information, for example); others are made by the reader (whether a character keeps a promise, for example).

TALE-SPIN generates sentences using the representation system of Conceptual Dependency (Schank 1975). Some of the Conceptual Dependency (CD) structures are passed on to a program which expresses them in English. (The original version of that program was written by Neil Goldman for the MARGIE system. The present version has been modified by Walter Stutzman and Gerald

De Jong.) The sentences which are not passed to the translator are those which represent easily inferred ideas. Neither program yet worries about the style of expression; that is, we worry about whether to say a newly generated piece of the story, but not much about how to say it.

A TALE-SPIN story involves a single main character who solves some problem. To make the process interesting, obstacles are introduced, some by the reader if he chooses, and some at random. For instance, the reader's decision that Irving Bird is not going to tell Joe Bear what he wants to know produces an obstacle to Joe Bear's plan to find something out. Some obstacles are created when certain scenes are included in the story. For instance, the initial world state has no bees in it, but when it comes time in the story to conjure up some actual honey, we do so by creating a whole scene which includes some honey in a beehive in a tree and a bee who owns that honey. The bee may or may not be at home. If he is, Joe Bear is going to have another obstacle in his plan when he gets to the beehive.

The story is the narration of some of the events which occur during the solution (or non-solution) of the problem. (That is, more things happen in the solution of a problem than a storyteller says or needs to say.) TALE-SPIN differs from other problem-solving systems in several ways: (1) the problems it solves are those requiring interaction with other, unpredictable characters rather than with a data base of theorems or blocks or circuits; (2) the world inside TALE-SPIN grows: new characters are created with unpredictable effects on the story; (3)

obstacles are _deliberately_ introduced; (4) an "unsuccessful" story, one in which the problem is not solved, can be just as interesting as a "successful" one.

PLANNING STRUCTURES Planning structures are what we use to organize knowledge about planful activity, which is represented in CD by a chain of causes and effects. The planning structures include delta-acts, planboxes, packages, scripts, sigma-states, rho-states, and pi-states.

A _delta-act_ is used to achieve a particular goalstate. Delta-prox (written here as dPROX) is the procedure for becoming proximate to some location. A delta-act is defined as a goal, a set of planboxes, and a decision algorithm for choosing between planboxes.

A _planbox_ is a particular method for achieving a goalstate. All the planboxes under a delta-act achieve the same goalstate. Each planbox has a set of preconditions (some of which may be delta-acts), and a set of actions to perform. "Unconscious" preconditions are attached to planboxes which would never occur to you to use. If you're trying to become proximate to X, you don't even think about persuading X to come to you when X is an inanimate object. "Uncontrollable" preconditions cannot be made true if they're not already true. (The assumption is that they are sometimes true.) "Plantime" preconditions are the things you worry about when you're making up the plan. You don't worry about "runtime" preconditions until you're executing the plan. ("Planning" is a mental activity. PLAN is, in fact, one of the primitive ACTs of CD. "Executing a plan" is performing a logically structured sequence of actions to achieve the goal of the plan.) If I'm planning to get a Coke out of the machine upstairs, I worry about having enough money, but I don't worry about walking up the stairs until I'm at the stairs. That the machine actually has some Coke is an uncontrollable runtime precondition: I don't worry about it until I get there, and there's nothing I can do if it is empty when I get there.

A package is a set of planboxes which lead to a goal act rather than state. The PERSUADE package, for instance, contains planboxes for X to persuade Y to do some act Z. The planboxes include asking, giving legitimate reasons, offering favors in return, threatening, and so on.

Goalstates come in various flavors. There are the goals which are associated with the delta-acts: the goal of dPROX is to be somewhere, the goal of dKNOW is to find out the answer to some question, the goal of dCONTROL is to possess something. But there are also goals of satiation, called sigma-states. For example, sHUNGER organizes the knowledge about satisfying hunger (invoking dCONTROL of some food, eating). TALE-SPIN also uses sigma-state knowledge in the bargaining process; offering someone some food in return for a favor is legitimate since it will satisfy a precondition for sHUNGER. There are also goals of preservation, called pi-states, which are most interesting when they are in danger of being violated. The logic of the THREATEN planbox in the PERSUADE package, for example, derives from the fact that physical violence conflicts with pHEALTH.

A SAMPLE DELTA-ACT: dPROX TALE-SPIN does not include all nine delta-acts described by Abelson (1975). It contains the three which closely correspond to primitive acts: dPROX (PTRANS), dCONTROL (ATRANS), dKNOW (MTRANS).

Here is an outline of dPROX:

```
              dPROX(X,Y) -- X wishes to be near Y
Planbox Ø: if X is already near Y, succeed.
Planbox 1: X goes to Y
  uncontrollable precondition: can X move himself?
  plantime precondition: dKNOW(location of Y)
  runtime precondition: dLINK(location of Y)
  action: PTRANS to location of Y
  runtime precondition: is Y really there? (We may have
     gotten false information during the dKNOW.)
Planbox 2: Y comes to X
  unconscious precondition: is Y animate?
  uncontrollable precondition: is Y movable?
  action: PERSUADE Y to PTRANS himself to X    (PERSUADE package)
Planbox 3: Agent A brings X to Y
  uncontrollable precondition: is X movable?
  action: X gets AGENT to bring X to Y    (AGENCY package)
Planbox 4: Agent A brings Y to X
  unconscious precondition: is Y animate?
  uncontrollable precondition: is Y movable?
  action: X gets AGENT to bring Y to X    (AGENCY package)
Planbox 5: X and Y meet at location Z
  unconscious precondition: is Y animate?
  uncontrollable precondition: is Y movable?
```

actions: PERSUADE Y to PTRANS himself to Z and dPROX(X,Z)

THE DATA BASE Planning structures are essentially procedural. The non-procedural data base used by the planning structures is divided into five classes.

1. Data about individual PPs (Picture Producers, nouns) where applicable: height; weight; where their home is; who their acquaintances are.

2. Data common to classes of PPs (e.g., data common to all birds) where applicable: what they eat; what their goals (sigma-states) are; whether they are animate (capable of MBUILDing), movable, self-movable; how they move around.

3. Sigma-state knowledge indicating how to achieve a sigma-state and what the plantime preconditions are that someone other than the planner can achieve. This is used in the bargaining process. Joe Bear offers to bring Irving Bird a worm because dCONTROL(FOOD) is a plantime precondition for sHUNGER which Joe Bear can achieve for Irving Bird. There are no plantime preconditions for sREST that he can achieve for Irving Bird (except maybe to leave him alone).

4. Memory: what everybody knows (thinks, believes); what Joe Bear knows; what Joe Bear thinks Irving Bird knows; etc. Planbox Ø of dKNOW, for example, accesses Memory to test whether Joe Bear already knows the answer to the question being asked, or whether it is public knowledge. Since both the question and the facts in Memory are represented in CD, the pattern match is very simple, taking advantage of CD's canonical representation of meaning.

5. Personal relationships. The relationship of one character to another is descibed by a point on each of three scales: COMPETITION, DOMINANCE, and FAMILIARITY. Scale values range from -10 to +10. The relation "is a friend of" is represented by a certain range on each of the three scales. The relation "would act as an agent for" is represented by a different range. The sentence "Joe Bear thought that Irving Bird would tell him where the honey was" comes from the "Ask a Friend" planbox of dKNOW. There is a procedure which goes through a list of Joe Bear's acquaintances and produces a list of those who qualify as "friends", i.e., those who fit somewhere within the "friend" range.

Relations are not symmetric: Joe Bear may think of Irving Bird as his friend, so he might ask him where the honey is, but Irving Bird may not think of Joe Bear as his friend at all, in which case he might refuse to answer Joe Bear.

Relationships can change. If Joe Bear becomes sufficiently aggravated at his "friend" Irving Bird and has to threaten to bash him in the beak in order to get him to tell him where the honey is, then the relationship between them deteriorates.

We plan to extend this feature to describe a character's "default" relationship: how he relates to total strangers. This would not necessarily be the point (0,0,0) but rather some point which would be used to give a rough indication of the character's "personality". Big bad Joe Bear might rate at (+6,+9,+4), where small meek Bill Worm might rate at (-6,-10,-4).

Changing a relationship is a type of goal we haven't yet considered in much detail, although goals of relationships (<u>rho-states</u>) clearly exist. The procedure for getting someone to like you (rLIKE) might contain planboxes for ATRANSing gifts, MTRANSing sweet nothings, etc., in addition to changing your own feelings toward that person so that if he (she) asks you to do something, you don't refuse.

Information gets into the data base in several ways. Memory data gets produced directly by the planning structures. Changes in relations are side-effects of the present set of planning structures. But things have to start somewhere. There is a function CREATE(X) which invents a new item of type X (e.g., bear, flower, berry). Associated with each type of item is a small procedure called a <u>picture</u> which invents the desired item and others as required. For example, when we create some honey, we also create a beehive, a tree, and a bee. The honey is "owned" by the bee and is inside the beehive which is in the tree. The bee may or not be at home. Randomly chosen names, heights, weights, etc., are attached. All this data is then added to Memory.

The CREATE function is called when needed; remember that TALE-SPIN models the process of <u>making up</u> a story as you go along. We will now follow, in detail, the production of the second sample story.

CREATE a bear, which invokes a picture procedure which invents a bear. Assume the bear is named Joe; although since the name is chosen at random from a list of first names, it is just as often Irving. A cave is also invented, and has Joe in it.

Joe´s location becomes public knowledge.

CREATE a bird, named Irving, and a tree which is his home. Irving´s location is also now public knowledge.

Assert that Joe is hungry. This fact enters Joe´s Memory. We also "say" this; that is, we pass it to the English translator which then produces the sentence "JOE BEAR WAS HUNGRY".

Invoke sHUNGER.

Choose at random a food that bears eat: honey. Assert that Joe is now planning to achieve the goal (sigma-state) of satisfying his hunger. Assert that he has decided that eating the food can lead to the achievement of his goal.

sHUNGER calls dCONTROL(honey). This forms a new goal, namely, that Joe have some honey. dCONTROL´s "Planbox 0" asks Memory if the goal is already true: does Joe already have some honey? The answer comes back: no. A plantime precondition is to know the location of some honey, so dCONTROL calls dKNOW(where is honey?). (The question is represented in CD, not English.)

dKNOW forms the new goal. dKNOW´s "Planbox 0" asks Memory whether Joe knows the location of any honey. Memory says no. Planbox 1 tests whether the question can be answered by consulting a standard reference (e.g., "What time is it?"). That fails. Planbox 2 tests whether the question requires expertise: no. Planbox 3 tests whether this is a "general information" question. It is, so we assert that Joe is planning to answer this question using Planbox 3 ("Ask a Friend").

Planbox 3 starts. Choose a friend: Irving. dKNOW calls the PERSUADE package to try to get Irving to answer Joe's question.

PERSUADE asks Memory whether Joe thinks that Irving cannot answer the question. Answer: no. Irving is a "friend", so we try the ASK planbox. Assert that Joe thinks that Irving will tell him where the honey is. PERSUADE calls dPROX(Irving), since Joe needs to speak to Irving.

dPROX asks Memory whether Joe is already near Irving. Memory says no. Planbox 1: is Joe self-movable? Yes. Assert that Joe is planning to be near Irving by going there himself. dPROX calls dKNOW(where is Irving?).

dKNOW's "Planbox Ø" asks Memory whether Joe already knows where Irving is. The answer comes back: yes, Irving is in a certain tree. dKNOW returns this to dPROX. (We will omit future references to "Planbox Ø".)

dPROX asserts that Joe walks to the tree where Irving is. We ask Memory whether Irving is actually there. He is, so dPROX has achieved its desired goal; his change in location is added to Memory. dPROX returns to PERSUADE.

Joe asks Irving where some honey is. The reader now gets to decide whether Irving agrees to do so. Assume the reader says yes. We ask Memory whether Irving actually knows where any honey is. If he did, we would have Irving tell him, but he doesn't, so we CREATE some honey: a storyteller can create solutions to problems as well as obstacles! Some honey is invented, along with a beehive, a tree, and a bee (Henry) who is at home. Irving tells Joe that the honey is in the beehive. ASK succeeds, so

PERSUADE succeeds, so dKNOW succeeds: Joe knows where some honey is.

Back in dCONTROL, we ask Memory whether [Joe thinks that] anyone owns the honey. Memory says that Henry does, so dCONTROL's Planbox 1 ("Free for the taking") fails. Planbox 2 is to PERSUADE Henry to give the honey to Joe.

Given no relation between Joe and Henry (they don't know each other), the only planboxes in PERSUADE which can be used are ASK and INFORM REASON.

We try ASK first. This calls dPROX(Henry) which succeeds since Joe knows where Henry is; we omit the details here. Joe asks Henry to give him the honey, and the reader decides that Henry refuses.

We try INFORM REASON next. We choose a goal of Henry's and build a causal chain backwards from the goal. For example, one of Henry s goals is to "eat" flowers. (TALE-SPIN thinks that what bees do to flowers is equivalent to eating.) In order to eat a flower, you have to "control" a flower, which results from someone (possibly you yourself) ATRANSing the flower to you. We test whether what Joe is trying to PERSUADE Henry to do matches ATRANSing a flower. It doesn't. (Joe is trying to PERSUADE Henry to ATRANS the honey to him.) We then consider that in order to ATRANS a flower, you have to be near the flower, which results from someone PTRANSing you to the flower. Does this match? No. We repeat this process a few times, trying to construct a short inference chain which connects what Joe is trying to persuade Henry to do with one of Henry's goals. INFORM

REASON fails, and we return to dCONTROL.

The next Planbox is called "Steal". We ask Memory whether Henry is home; if he weren't, Joe would simply take the honey. But Memory tells us that Henry is home, so STEAL calls PERSUADE to get Henry to leave home; that is, Joe is now going to try to persuade Henry to PTRANS himself from the hive.

In the context of STEAL, the ASK planbox is not used. Joe tries INFORM REASON again and succeeds in producing the following chain: we get to the idea of someone PTRANSing himself to a flower again as we did before, but we notice that this does match what we are trying to persuade Henry to do: the connection is that Henry will PTRANS himself from the beehive to the flower. Joe now considers the precondition for Henry's PTRANSing himself to the flower, namely, that Henry has to know where the flower is. Memory does not indicate that Joe thinks that Henry knows where a flower is, nor does Joe know where a flower is, but rather than invoke dKNOW(where is a flower?), we CREATE a flower: this is legitimate in a plan to steal something. Joe now tells Henry that there is a flower in a certain flowerbed, and then asks Henry if he would like to fly to that flower. Henry agrees and flies away. PERSUADE succeeds, and returns to dCONTROL.

Joe now takes the honey from the hive, so dCONTROL succeeds and returns to sHUNGER. Memory is modified to indicate that Joe knows that he has the honey, but that Henry does not.

Joe now eats the honey, and has achieved the sigma-state of not being hungry. But, when bears eat, they become tired, so sREST is invoked.

sREST is very short. It requires a dPROX(cave), which is easily achieved, and then Joe goes to sleep.

Since the main goal has been achieved, and the goal produced as a consequence of that goal has also been achieved, the story ends.

What distinguishes stories from simple sequences of events? Coherency is important: there has to be a logical flow from one event to the next. This is represented in CD as a chain of acts which result in states which enable further acts and so on. Interest is important: something interesting or unusual has to happen or else the reader will begin to wonder what the point of the story is. TALE-SPIN creates impediments to goals, on the assumption that the overcoming of obstacles can make an interesting story. "One day Joe Bear was hungry. There was a jar of honey right next to him. He ate it. The end" is not a story. It shouldn't be that easy.

On the other hand, it shouldn't be too hard either. In theory at least, there is a cost-effectiveness calculus which people employ when deciding how much energy to expend on a subgoal, based on how much the goal is worth to them. This process prevents the plans from being too complicated.

As the story is generated, various plot decisions have to be made. Some decisions are made at random, others are made by the reader. When Joe Bear threatens Irving Bird because Irving Bird won't tell him where the honey is, the reader gets to decide whether Irving Bird is going to ignore the threat.

We use planning structures because any program which reads or writes a story, whether of the folktale variety or the New York Times variety, must have a model of the logic of human activity. It might be easier to simulate the generation of a highly stylized form of story, as Klein (1974) has done using Propp's analysis of a class of Russian fairy tales, but there is little generality there. One could use any of the well-known problem-solving systems like MICRO-PLANNER, but the story is the proof procedure, and the procedure used there does not correspond to my conception of how people solve problems. That's not a criticism of MICRO-PLANNER as a problem-solver, but only as a model of <u>human</u> problem-solving.

User interaction was included for two reasons. First, the interactive feature now serves as a heuristic for placing bounds on the complexity of the story. Beyond, some number of obstacles to the goal, a story becomes a kind of joke. Second and more important, extensions to TALE-SPIN will include more sophisticated responses than the present yes/no variety.

<u>THE FUTURE OF TALE-SPIN</u>. There are a lot of things that TALE-SPIN doesn't do yet that would improve it as a storyteller. Here are some of the theoretical problems we will be working on in the immediate future. (1) Bargaining, as it exists now in TALE-SPIN, is a pretty one-sided affair, with the main character making all the proposals. Irving Bird is just as likely to suggest that Joe Bear go get him a worm as Joe is to offer to do so. Counter-proposals are certainly common enough. (2) Future stories should include planning on the part of more than one character. The present stories are all "about" the bear, and

only incidentally involve the bird and other characters. The stories are more concerned with reaction than interaction. (3) For every plan, there may be a counter-plan, a plan to block the achievement of a goal: a plan for keeping away from something or someone; a plan not to find out something, or to be convinced that it isn't true; a plan to get rid of something you own. (4) How much of a plan do people consider in advance? We have made some efforts in this area by making the distinctions between kinds of preconditions. Certainly the most important improvement here will be the cost-effectiveness reasoning. (5) The theory of telling stories (what to say) now implemented in TALE-SPIN is to express violations of sigma-states ("Joe Bear was hungry"), physical acts, and those mental acts which provide motivation or justification for later events. The reader is assumed to be able to infer the rest. This seems to work reasonably well for the present simple stories, but may have to be modified to suit longer, more complicated stories.

REFERENCES

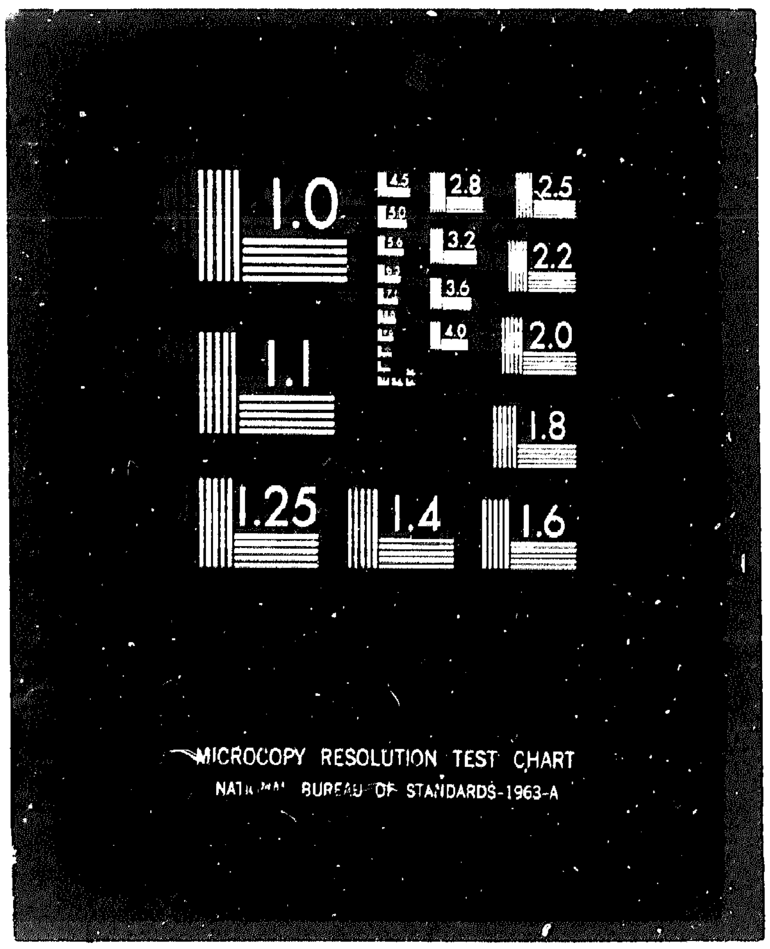
Abelson, R. P. (1975). Concepts for representing mundane reality in plans. In D. Bobrow and A. Collins, eds. Representation and understanding: Studies in cognitive science. Academic Press, New York.

Klein, S. et al (1974). Modelling Propp and Levi-Strauss in a meta-symbolic simulation system. Technical Report 226, University of Wisconsin at Madison.

Schank, R. C. (1975). Conceptual Information Processing. American Elsevier, New York. This includes contributions by Neil M. Goldman, Charles J. Rieger III, and Christopher K. Riesbeck.

Schank, R. C. and Abelson, R. P. (1975). Scripts, plans and knowledge. In Proceedings of the 4th International Joint Conference on Artificial Intelligence.

END


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A