AUGMENTED PHRASE STRUCTURE GRAMMARS

George E. Heidorn
Computer Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, NY

ABSTRACT

Augmented phrase structure grammars consist of phrase structure rules with embedded conditions and structure-building actions written in a specially developed language. An attribute-value, record-oriented information structure is an integral part of the theory.

I. INTRODUCTION

An augmented phrase structure grammar (APSG) consists of a collection of phrase structure rules which are augmented by arbitrary conditions and structure building actions. This basic idea is not new, having been used in syntax-directed compiling [e.g. 1] as well as in natural language processing [e.g. 2], but what is new are the particular language in which these rules are written and the algorithms that apply them.

This brief paper is intended to serve as an introduction to augmented phrase structure grammars. First, the form of data structure used is discussed, followed by discussions of the analysis and synthesis of text, i.e. decoding and encoding. (Although this session of the workshop is devoted to natural language input, this brief discussion of synthesis is included because one of the important features of APSG is the consistent manner in which both decoding and encoding are specified.) Then there is a section on implementations and applications, followed by concluding remarks.

II. DATA STRUCTURE

The data structure used by APSG is a form of semantic network, consisting of "records" which are collections of attribute-value pairs. Records represent entities, either physical or abstract, such diverse things as vehicles, actions, words and verb phrases. There are three different kinds of attributes: relations, which have as their values pointers to other records; properties, which have as their values either numbers or character strings; and indicators, which have bit string values and usually serve in a role similar to features in linguistic terminology.

A record that has a NAME attribute is called a "named record" and can be referred to by using the value of the NAME attribute in single quotes. Named records are used to hold information that is relatively permanent, such as information about relevant words and concepts, and are defined in the following manner (where the parentheses enclose structure-building information):

```
SERVIC ('ACTIVITY',E,ES,ING,ED,
        TRANS,PS='VERB',XYZ=3)
```

It is convenient to picture a record as a box enclosing a column of relation and property names on the left and a column of corresponding values on the right. Indicators which are present in the record (i.e. have a non-zero value) are listed at the bottom of the box. The named record 'SERVIC' defined above could be drawn as:

NAME	"SERVIC"
SUP	'ACTIVITY'
PS	'VERB'
XYZ	3
E,ES,ING,ED,TRANS	

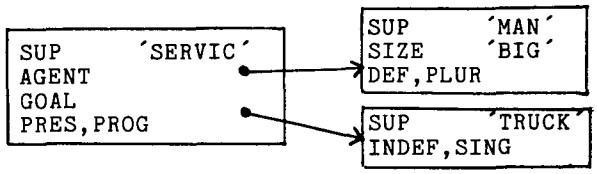
Double quotes enclose a character string, single quotes enclose the name of a named record. The values of the SUPerset and PS (part-of-speech) attributes are really pointers to the records 'ACTIVITY' and 'VERB' and could be drawn as directed lines to those other records if they were included in the diagram.

The named record 'SERVIC' given here could be considered to be a dictionary entry stating that the VERB stem SERVIC can take endings E, ES, ING and ED, the VERB SERVIC is TRANSitive, and the concept SERVIC is an ACTIVITY. (When a named record name appears without the explicit mention of an attribute name, the SUPerset attribute is assumed.) The XYZ attribute was included just to illustrate a numerically-valued property. Of course, the true meaning of any of this information depends completely upon the way it is used by the APSG rules.

During decoding and encoding, records called "segment records" are employed to hold information about segments of text. For example, the segment "are servicing" could be described by the record:

SUP	'SERVIC'
PRES,P3,PLUR,PROG	

which could be interpreted as saying that "are servicing" is the present, third person, plural, progressive form of "service". Similarly, the sentence "The big men are servicing a truck." could be described by:



where the indicators DEF and INDEF mean definite and indefinite, respectively. The sentence "A truck is being serviced by the big men." could be described by exactly the same record structure but with the addition of a PASSIVE indicator in the record on the left.

During a dialogue some records that begin as segment records may be kept to become part of longer term memory to represent the entities (in the broadest sense of the term) that are being discussed. Segment records then might have pointers into this longer term memory to show referrents. So, for example, the sentence "They are servicing a truck." might be described by the same record structure shown above if the referrent of "they" was known to be a certain group of men who are big.

III. ANALYSIS OF TEXT (DECODING)

Decoding is the process by which record structures of the sort just shown are constructed from strings of text. The manner in which these records are to be built is specified by APSG decoding rules. A decoding rule consists of a list of one or more "segment types" (meta-symbols) on the left of an arrow to indicate which types of contiguous segments must be present in order for a segment of the type on the right of the arrow to be formed. Conditions which must be satisfied in order for the rule to be applicable may be stated in parentheses on the left side of the rule, and structure-building operations to be performed when a new segment record is created are stated in parentheses on the right side.

For illustrative purposes, some of the rules which would be required to produce the segment records shown in the previous section will be discussed here. Complete examples are given in Reference 3.

If the string "servicing" appeared in the input, and the substring "servic" were described by the VERBSTEM segment record

```
SUP      'SERVIC'
```

then the rule

```
VERBSTEM(ING#)   I   N   G -->
  VERB(SUP(VERBSTEM,PRESTART)
```

would form the VERB segment record

```
SUP      'SERVIC'
PRESPART
```

to describe the string "servicing", identifying it as the present participle form of service. This rule says that if a segment of the string being decoded is described as a VERBSTEM, and the associated segment record has a SUP attribute which points to a named record which has an ING indicator (as the named record for 'SERVIC' defined in the previous section would), and this segment is followed immediately by the characters "i", "n" and "g", then create a VERB segment record with the same SUP as the VERBSTEM and with a PRESPART indicator, to describe the entire segment ("servicing" in this case).

Then the rule

```
VERB  --> VERBPH(¢VERB)
```

would create aVERBPHrase segment record which is a copy (¢) of the VERB segment record just shown.

If the string "are" appearing in the input were described by the VERB segment record

```
SUP      'BE'
PRES,P3,PLUR
```

then the rule

```
VERB('BE')   VERBPH(PRESPART)  -->
      VERBPH(PROG,FORM=FORM(VERB))
```

would produce the new VERBPH segment record

```
SUP      'SERVIC'
PRES,P3,PLUR,PROG
```

from the twojust shown, to describe the string "are servicing". This rule says that if a segment of the string being decoded is described as a VERB with a SUP of 'BE', and it is followed by a segment described as a VERBPH with a PRESPART indicator, then create a new VERBPH segment record which is a copy (automatically, because the segment type is the same) of the VERBPH segment record referred to on the left of the rule, but which as a PROGressive indicator and the FORM information from the VERB. FORM would have previously been defined as the name of a group of indicators (i.e. those having to do with tense, person and number). Similar rules can be used to recognize passives, perfects and modal constructions.

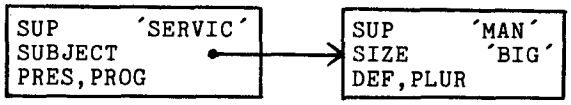
Continuing with the example, if the string "the big men" were decoded to the NOUNPH segment record

```
SUP      'MAN'
SIZE     'BIG'
DEF,PLUR
```

then the rule

```
NOUNPH  VERBPH(NUMB.EQ.NUMB(NOUNPH), SUBJECT)
   -->  VERBPH(SUBJECT=NOUNPH,-NUMB,-PERS)
```

would produce the new VERBPH segment record (the one on the left in this diagram)



from the previous VERBPH record, to describe the string "the big men are servicing". It is important to realize that the record on the left in the above diagram is a segment record that "covers" the entire string and that the record shown on the right (which is the same one from the previous diagram) just serves as the value of its SUBJECT attribute. The rule above says that if a NOUNPH is followed by a VERBPH, and the NUMBer indicators of the VERBPH are the same as the NUMBer indicators of the NOUNPH, and the VERBPH does not already have a SUBJECT attribute, then create a new VERBPH segment record which is a copy of the old one, give it a SUBJECT attribute pointing to the NOUNPH record, and delete the NUMBer and

PERson indicators. Considering the subject to be part of the verb phrase in this manner can simplify the handling of some constructions involving inverted word order.

If the string being decoded were "the big men are servicing a truck.", a rule similar to the last one shown above could be used to pick up the direct object. Then the rule

```
./VERBPH(SUBJECT,OBJECT| ¬TRANS*|PASSIVE).
          --> SENT(*VERBPH)
```

could be applied, which says if a VERBPH extending between two periods has a SUBJECT attribute and also either has an OBJECT attribute or does not need one because there is no TRANSitive indicator in the named record pointed to by the SUP (i.e. the verb is intransitive) or because there is a PASSIVE indicator, then call it a SENTence.

To get the record structure describing this string into the form shown near the end of the previous section, one more rule would be needed:

```
SENT($'ACTION',¬PASSIVE,SUBJECT)  -->
      SENT(AGENT=SUBJECT,GOAL=OBJECT,
        -SUBJECT,-OBJECT)
```

This says that for a non-PASSIVE ACTION SENTence that still has a SUBJECT attribute, set the AGENT and GOAL attributes to the values of the SUBJECT and OBJECT attributes, respectively, and then delete the SUBJECT and OBJECT attributes from the record. The notation $'ACTION' is read "in the set 'ACTION'" and means that the named record 'ACTION' must appear somewhere in the SUPerset chain of the current record. In the previous section the named record 'SERVIC' was defined to have a SUP of 'ACTIVITY'. If the named record 'ACTIVITY' were similarly defined to have a SUP of 'ACTION', the segment record under discussion here would satisfy the condition $'ACTION'.

From the above examples it can be seen that the condition specifications take the form of logical expressions involving the values of attributes. Each element in a condition specification is basicaly of the form value.relation.value, but this is not obvious because there are several notational shortcuts available in the rule language. For example, 'BE' is short for SUP.EG.'BE',PRESPART is short for PRESPART.NE.0, and ¬SUBJECT is short for SUBJECT.EQ.0. The elements are combined by and's (commas) and or's (vertical bars).

In most cases the attribute whose value is being tested is to be found in the segment record associated with the constituent, but that is not always the case. For example, ING* tests the value of the ING indicator in the named record pointed to by the SUP of the segment record, and could be written ING(SUP) or ING(SUP).NE.0. Another example is NUMB(NOUNPH) which was used to refer to the value of the NUMB indicators in the NOUNPH segment in one of the rules above.

From the examples it can also be seen that creation specifications take the form of short procedures consisting of statements for setting the values of attributes. Each element in a creation specification is basically of the form attribute=value (where "=" means replacement), but again this is not obvious becuase of the notational shortcuts used. For example, SUP(VERBSTEM) is short for SUP=SUP(VERBSTEM), PRESPART is short for PRESPART=1 (note that this form has a different meaning when it is used in a condition specification), and -SUBJECT is short for SUBJECT=0.

In all of the examples here, the attribute whose value is set would be in the segment record being built, but that need not always be the case. If, for example, there were some reason to want to give the AGENT record of an action an ABC attribute equal to one more than the XYZ attribute of the concept record associated with that action (i.e. the named record pointed to by its SUP), the following could be included in the last rule shown:

```
ABC(AGENT)=XYZ(SUP)+1
```

which can be read as "set the ABC attribute of the AGENT of this record to the value of the XYZ attribute of the SUP of this record plus 1." There is no limit to the nesting of attribute names used in this manner.

Although in the example rules given here the conditions are primarily syntactic, semantic constraints can be stated in exactly the same manner. Much of the record building shown here can be considered semantic (and somewhat case oriented). The important point, however, is that the kind of condition testing and structure building done is at the discretion of the person who writes the rules. Complete specifications for the APSG rule language are given in Reference 3.

The decoding algorithm used with APSG is basically that of a bottom-up, left-to-right, parallel-processing, syntax-directed compiler. An important and novel feature of this algorithm is something called a "rule instance record", which primarily maintains information abut the potential applicability of a rule. A rule instance record is initially created for a rule whenever a segment which can be the first constitutent of that rule becomes available. (A terminal segment becomes available by being obtained from the input stream, and a non-terminal segment becomes available whenever a rule is applied.) Then the rule instance record "waits" for a segment which can be the next constituent of the associated rule to become available. When such a segment becomes available, the rule instance record is "extended". When a rule instance record becomes complete (i.e. all of its constituents are available), the associated rule is applied (i.e. the segment record specified on the right is built and made available). There may be many rule instance records in existence for a particular rule at any point in time.

Because of the parallel processing nature of the decoding algorithm, when a segment record is created to describe a portion of the input text it does not result in the destruction of other records describing the same portion or parts of it. Local ambiguities caused by multiple word senses, idioms and the like may result in more than one segment record being created to describe a particular portion of the text, but usually only one of them is able to combine with its neighbors to become part of the analysis for an entire sentence.

IV. SYNTHESIS OF TEXT (ENCODING)

Encoding is the process by which strings of text are produced from record structures of the sort already shown. The manner in which this processing is to be done is specified by APSG encoding rules. The right side of an encoding rule specifies what segments a segment of the type on the left side is to be expanded into. Conditions and structure-building actions are included in exactly the same manner as in decoding rules.

The encoding algorithm begins with a single segment record and its associated type side-by-side on a stack. At each cycle through the algorithm, the top pair is removed from the stack and examined. If there is a rule that can be applied, it results in new pairs being put on the top of the stack, according to its right hand side. Otherwise, either the character string value of the NAME attribute of the SUP of the segment record (e.g. "servic") is put out, or the name of the segment type itself (e.g. "I") is put out. Eventually the stack becomes empty and the algorithm terminates, having produced the desired output string.

For example, if at some point the following pair were to come off the top of the stack:

VERBPH	SUP 'SERVIC' PRES,P3,PLUR,PROG

the following encoding rule could be applied:

```
VERBPH(PROG)  -->
    VERB('BE',FORM=FORM(VERBPH))
    VERB(-PROG,-FORM,PRESPART)
```

resulting in the following two pairs being put on the top of the stack:

VERB	SUP 'BE' PRES,P3,PLUR
VERBPH	SUP 'SERVIC' PRESPART

The above rule says that a VERBPH segment with a PROGressive indicator should be expanded into a VERB segment with a SUP of 'BE' and the same FORM indicators as the VERBPH, followed by a new VERBPH segment which begins as a copy (automatically) of the old one and then is modified by deleting the PROG and FORM indicators and setting the PRESPART indicator.

When the VERB segment shown above comes off the stack, a rule would be applied to put the string "are" into the output. Then, after application of a couple more rules, the top of the stack would have the four pairs

VERBSTEM	SUP 'SERVIC'
I	null
N	null
G	null

which would result in the string "servicing" being produced after four cycles of the algorithm. Complete encoding examples may be found in Reference 3.

V. IMPLEMENTATIONS AND APPLICATIONS

As part of the original work on APSG a computer system called NLP (Natural Language Processor) was developed in 1968. This is a FORTRAN program for the IBM 360/370 computers which will accept as input named record definitions and decoding and encoding rules in exactly the form shown in this paper and then perform decoding and encoding of text [3]. A set of about 300 named record definitions and 800 rules was written for NLP to implement a specific system (called NLPQ) which is capable of carrying on a dialogue in English about a simple queuing problem and then producing a program in the GPSS simulation language to solve the problem [3,4].

More recently a LISP implementation of NLP has been done, which accepts exactly the same input and does the same processing as the FORTRAN version. An interesting feature of this new version is that the compiler part, whose primary task is to translate condition and creation specifications (i.e. the information in parentheses) into lambda expressions, is itself written as a set of APSG rules. This work is part of a project at IBM Research to develop a system which will produce appropriate accounting application programs after carrying on a natural language dialogue with a businessman about his requirements. APSG is also being used in the development of a natural alaaguage query system for relational data bases and is being considered for use in other projects at IBM. None of this recent work has been documented yet.

VI. CONCLUDING REMARKS

APSG clearly has much in common with other current computational linguistic theories, with the ideas of procedural specification and arbitrary conditions and strucutre-building actions being very popular at this time. It would seem to be most similar to Woods' augmented transition networks (ATN) [5], especially as used by Simmons [6]. Registers in the ATN model correspond closely to attributes of segment records in APSG, and the semantic network structures of Simmons are very close to the record structures of APSG.

Context-free phrase structure grammars have been known to be inadequate for describing natural languages for many years, and context-sensitive phrase structure grammars have not been found to be very useful, either. Augmented phrase structure grammars, however, appear to be able to express the facts of a natural language in a very concise and convenient manner. they have the power of computer programs, while maintaining the appearance of grammars.

Although APSG was used successfully to implement one fairly large system (NLPQ), it is too early to do a thorough appraisal of its capabilities. Through the extensive use anticipated in the next year however, its strengths and weaknesses should become more apparent.

ACKNOWLEDGEMENTS

I am indebted to my former students at the Naval Postgraudate School for their efforts on the original implementation and application, my colleagues at IBM Research -- Martin Mikelsons, Peter Sheridan, Irving Wladawsky and Ted Codd -- for their interest, ideas and work on the current implementatons and applications, and my wife, Beryl, for her typing assistance and general helpfulness.

REFERENCES

1. Balzer, R.M., and Farber, D.J., "APAREL - a parse-request language," COMM. ACM 12, 11 (Nov. 1969), 624-631.

2. Thompson, F.B., Lockemann, P.C., Dostert, B., Deverill, R.S., "REL: a rapidly extensible language system," In PROC. 24th NAT'L CONF., ACM, NY, 1969, 399-417.

3. Heidorn, G.E., "Natural language inputs to a simulation programming system," Technical Report NPS-55HD72101A, Naval Postgraduate School, Monterey, California, Oct. 1972.

4. Heidorn, G.E., "English as a very high level language for simulation programming," Proc. Symp. on Very High Level Languages, SIGPLAN NOTICES 9,4 (April 1974), 91-100.

5. Woods, W.A., "Transition network grammars for natural language analysis," COMM. ACM 13, 10 (Oct. 1970), 591-606.

6. Simmons, R.F., "Semantic networks: their computation and use for understanding English sentences," in COMPUTER MODELS OF THOUGHT AND LANGUAGE, R.C. Schank and K.M. Colby (Eds.), W.H. Freeman and Co., San Francisco, Calif., 1973, 63-113.

DIAGNOSIS AS A NOTION OF GRAMMAR

Mitchell Marcus
Artificial Intelligence Laboratory
M.I.T.

This paper will sketch an approach to natural language parsing based on a new conception of what makes up a recognition grammar for syntactic analysis and how such a grammar should be structured. This theory of syntactic analysis formalizes a notion very much like the psychologist's notion of "perceptual strategies" [Bever '70] and makes this formalized notion - which will be called the notion of wait-and-see diagnostics - a central and integral part of a theory of what one knows about the structure of language. By recognition grammar, we mean here what a speaker of a language knows about that language that allows him to assign grammatical structure to the word strings that make up utterances in that language.

This theory of grammar is based on the hypothesis that every language user knows as part of his recognition grammar a set of highly specific diagnostics that he uses to decide deterministically what structure to build next at each point in the process of parsing an utterance. By deterministically I mean that once grammatical structure is built, it cannot be discarded in the normal course of the parsing process, i.e. that no "backtracking" can take place unless the sentence is consciously perceived as being a "garden path". This notion of grammar puts knowledge about controlling the parsing process on an equal footing with knowledge about its possible outputs.

To test this theory of grammar, a parser has been implemented that provides a language for writing grammars of this sort, and a grammar for English is currently being written that attempts to capture the wait-and-see diagnostics needed to parse English within the constraints of the theory. The control structure of the parser strongly reflects the assumptions the theory makes about the structure of language, and the discussion below will use the structure of the parser as an example of the implications of this theory for the parsing process. The current grammar of English is deep but not yet broad; this has allowed investigation of the sorts of wait-and-see diagnostics needed to handle complex English constructions without a need to wait until a grammar for the entire range of English constructions could be written. To give some idea of the scope of the grammar, the parser is capable of handling sentences like:

Do all the boys the librarian gave books to want to read them?
The men John wanted to be believed by shot him yesterday.

It should be mentioned that certain grammatical phenomena are not handled at all by the present grammar, chief among them conjunction and certain important sorts of lexical ambiguity. There is every intention, however, of expanding the grammar to deal with them.

Two Paradigms

To explain exactly what the details of this wait-and-see (W&S) paradigm are, it is useful to compare this notion with the current prevailing parsing paradigm, which I will call the guess-and-then-backup (G&B) paradigm. This paradigm is central to the parsers of both Terry Winograd's SHRDLU [Winograd '72] and Bill Woods' LUNAR [Woods '72] systems.

In a parser based on the G&B paradigm, various options are enumerated in the parser's grammar for the next possible constituent at any given point in the parse and these options are tested one at a time against the input. The parser assumes tentatively that one of these options is correct and then proceeds with this option until either the parse is completed or the option fails, at which point the parser simply backs up and tries the next option enumerated in the parser's grammar. This is the paradigm of G&B: enumerate all options, pick one, and then (if it fails) backup and pick another. While attempts have been made to make this backup process clever, especially in Winograd's SHRDLU, it seems that it is very difficult, if not impossible in general, to tell from the nature of the cul de sac exactly where the parser has gone astray. In order to parse a sentence of even moderate complexity, there are not one but many points at which a G&B parser must make guesses about what sort of structure to expect next and at all of these points the correct hypothesis must be found before the parse can be successfully completed. Furthermore, the parser may proceed arbitrarily far ahead on any of these hypotheses before discovering that the hypothesis was incorret, perhaps invalidating several other hypotheses contingent upon the first. In essence, the G&B paradigm considers the grammar of a natural language to be a tree-structured space through which the parser must blindly, though perhaps cleverly, search to find a correct parse.

The W&S paradigm rejects the notion of backup as a standard control mechanism for parsing. At each point in the parsing process, a W&S parser will only build grammatical structure it is sure it can use. The parser does this by determining, by a two part process, which of the hypotheses possible at any given point of the parse is correct before attempting any of them. The parser first recognizes the specific situation it is in, determined both on the basis of global expectations resulting from whatever structure it has parsed and absorbed, and from features of lower level substructures from a little ahead in the input to which internal structure can be assigned with certainty but whose function is as yet undetermined. Each such situation can be so defined that it restrains the set of possible hypotheses to at most two or three. If only one hypothesis is possible, a W&S parser will take it as given, otherwise it will proceed to the second step

of the determination process, to do a differential diagnosis to decide between the competing hypotheses. For each different situation, a W&S grammar includes a series of easily computed tests that decides between the competing hypotheses. The key assumption of the W&S paradigm, then, is that the structure of natural language provides enough and the right information to determine exactly what to do next at each point of a parse. There is not sufficient room here to discuss this assumption; the reader is invited to read [Marcus '74], which discusses this assumption at length.

The Parser Itself

To firm up this talk of "expectations", "situations", and the like, it it useful to see how these notions are realized in the existing W&S parsing system. Before we can do this, it will be necessary to get an overview of the structure and operation of the parser itself.

A grammar in this system is made up of packets of pattern-invoked demons, which will be called modules. (The notion of packet here derives from work by Scott Fahlman [Fahlman '73].) The parser itself consists of two levels, a group level and a clause level, and any packet of modules is intended to function at one level or the other. Modules at group level are intended to work on a buffer of words and word level structures and to eventually build group level structures, such as Noun Groups (i.e. Noun Phrases up to the head noun) and Verb Groups (i.e. the verb cluster up to the main verb), which are then put onto the end of a buffer of group level structures not yet absorbed by higher level processes. Modules at clause level are intended to work on these substructures and to assemble them into clauses. The group buffer and the word buffer can both grow up to some predetermined length, on the order of 3, 4, or 5 structures. Thus the modules at the level above needn't immediately use each structure as it comes into the buffer, but rather can let a small number of structures "pile up" and then examine these structures before deciding how to use the first of them. In this sense the modules at each level have a limited, sharply constrained look-ahead ability; they can wait and see what sort of environment surrounds a substructure in the buffer below before deciding what the higher level function of that substructure is. (It should be noted that the amount of look-ahead is constrained not only by maximum buffer length but also by the restriction that a module may access only the two substructures immediately following the one it is currently trying to utilize. This constraint is necessary because the substructure about to be utilized at any moment may not be the first in the buffer, for various reasons.)

Every module consists of a pattern, a pretest procedure, and a body to be executed if the pattern matches and the pretest succeeds. Each pattern consists of an ordered list of sets of features. As structures are built up by the parser, they are labelled with features, where a feature is any property of a structure that the grammar wants to be visible at a glance to any module looking even casually at that structure. (Structures can also have registers attached to them, carrying more specialized sorts of information; the contents of a register are privileged in that a module can access the contents of a register only if it knows the name of that register.) A module's pattern matches if the feature sets of the pattern are subsumed by the feature sets of consecutive structures in the appropriate buffer, with the match starting at the effective beginning of the buffer.

Very few modules in any W&S grammar ae always active, waiting to be triggered when their patterns match; a module is active only when a packet it is in has been activated, i.e. added to the set of presently active packets. Packets are activated or deactivated by the parser at the specific order of individual modules; any module can add or remove packets from the set of active packets if it has reason to do so.

A priority ordering of modules provides still further control. Every module is assigned a numerical priority, creating a partial ordering on the active modules. At any time, only the highest-prioritied module of those whose patterns match will be allowed to run. Thus, a special purpose module can edge out a general purpose module both of whose patterns match in a given environment, or a module to handle some last-resort case can lurk low in a pool of active modules, to serve as default only if no higher-prioritied module responds to a situation.

Firming Up The Notion Of Situation

This, in brief, is the structure of the W&S parser; now we can turn to a discussion of how this structure reflects the theoretical framework discussed above. Let us begin by recasting a statement made above: In deciding what unique course of action to take at any point in the parse, the parser first recognizes a specific well-defined situation on the basis of a combination of global expectations and the specific features of lower level substructures which are as yet unabsorbed.

It should now become clear that what it means to have a global expectation is that the appropriate packet is active in the parser, and that each module is itself the specialist for the situation that its packet, pattern and pretest define. The grammar activates and deactivates packets to reflect its global expectations about syntactic structures that may be encountered as a result of what it has seen so far. (The parser might also activate packets on the basis of what some higher level process in a natural language understanding system tells it to expect by way of discourse phenomena.) These packets often reflect rather large scale grammatical expectations; for example, the following are some packets