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 as 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 referents. So, for example, the sentence "They are servicing a truck." might be described by the same record structure shown above if the referent 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 a VERBPHrase 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 two just 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)

SUP	'SERVIC'		SUP	'MAN'
SUBJECT		→	SIZE	'BIG'
PRES, PROG			DEF, PLUR	

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 basically 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 because 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 about the potential applicability of a rule. A rule instance record is initially created for a rule whenever a segment which can be the first constituent 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	<table border="1"> <tr> <td>SUP</td> <td>'SERVIC'</td> </tr> <tr> <td>PRES,P3,PLUR,PROG</td> <td></td> </tr> </table>	SUP	'SERVIC'	PRES,P3,PLUR,PROG	
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	<table border="1"> <tr> <td>SUP</td> <td>'BE'</td> </tr> <tr> <td>PRES,P3,PLUR</td> <td></td> </tr> </table>	SUP	'BE'	PRES,P3,PLUR	
SUP	'BE'				
PRES,P3,PLUR					
VERBPH	<table border="1"> <tr> <td>SUP</td> <td>'SERVIC'</td> </tr> <tr> <td>PRESPART</td> <td></td> </tr> </table>	SUP	'SERVIC'	PRESPART	
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	<table border="1"> <tr> <td>SUP</td> <td>'SERVIC'</td> </tr> <tr> <td>I</td> <td>null</td> </tr> <tr> <td>N</td> <td>null</td> </tr> <tr> <td>G</td> <td>null</td> </tr> </table>	SUP	'SERVIC'	I	null	N	null	G	null
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 language 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 structure-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 Postgraduate 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 implementations 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.