PARALLELIZATION OF AUGMENTED PHRASE-STRUCTURE GRAMMARS FOR NATURAL-LANGUAGE PROCESSING

D. Terence Langendoen
Clinton Jeffery

Department of Linguistics
The University of Arizona
Tucson, AZ 85721 USA

Version of 4:53 p.m. January 21st, 1990

Abstract

Note: This is the abstract we submitted for the competition. We will have to revise it somewhat.

The theory of augmented phrase-structure grammar (APSG) is widely used in natural-language processing, both for parsing of natural-language input and generating natural language output by machine. APSGs consist essentially of context-free phrase-structure productions (or their inverses), augmented by tests for the applicability of the productions and by actions for constructing internal representations of the interpretation of the input and potential output. APSGs are capable of modeling any theory of grammar of natural language currently in use and can be designed to provide both elegant processing models of particular linguistic theories and also efficient mechanisms for parsing, generating and translating natural-language materials.

The most comprehensive implementation of an APGS is the PLNLP English Grammar (PEG), developed primarily by Karen Jensen of IBM Research. PEG is written in PLNLP, a programming language designed by George Heidorn of IBM Reearch. Instructions in PLNLP consist of APSG productions. For the past several years, the senior author has worked with Heidorn and Jensen, for one year as a Visiting Scientist at IBM T.J. Watson Research Center, and also under contract with IBM at the City University of New York and the University of Arizona. His work to date has primarily involved the writing of a "user's guide" to PLNLP for linguists and developing test models of various grammatical constructions in English and other natural languages.

In this work, we intend to accomplish two tasks. First, we will determine the possibility of parallelizing the parsing algorithm in PLNLP, by implementing a compiler for PLNLP using the IBM 3090 vector facility. Our implementation will use vector instructions to apply in parallel those productions in the parser which are candidates for application at each stage of the parse. We expect thereby to achieve a significant performance improvement over previous PLNLP implementations, particularly for ambiguous input. Second, we will explore the use of the architecture of the IBM 3090 to generate in parallel alternative responses to or analyses of ambiguous natural-language input (e.g., *I need the report on my desk*). Given a parser which assigns more than one analysis to such an input, we will submit the different interpretations to different engines to generate in parallel appropriate responses or analyses in parallel.

Table of Contents

The Nature of Parsing

Parsing (or decoding) may be defined as the process of determining the structure and interpretation of textual material, and a parser (or decoder) as the system that carries out the process. Broadly speaking, parsing may be done in two different ways: from the top down, by breaking up the text into successively smaller parts, as in traditional sentence diagramming; and from the bottom up, by combining the elements of the text into successively larger parts. Both approaches are are widely used in the parsing of computer programs (Aho, Sethi and Ullman 1986). and in the parsing of natural-language text by machine (Grishman 1986, Allen 1988). Designing parsers for natural-language text is a much harder problem than designing parsers for programming-language text. Consequently, most parsers that have been built for natural-language text provide only for a tiny subpart of the language. A notable exception is the PLNLP English Grammar (PEG) system, which is designed to parse ordinary English written text (Jensen 1986, ...).

Parsing in PLNLP

In this paper, we consider the problem of bottom-up parsing of natural-language text from left to right, making use of the PLNLP programming language (Heidorn 1972, ...). Parsing in PLNLP, as in many systems for natural-language processing by machine, makes use of augmented phrase-structure rules for manipulating records, defined as a collection of attributes, each of which has a value (boolean, integer, string, pointer to another record, or a list of pointers). Each such rule analyzes a sequence of records associated with the corresponding sequence of textual segments as a single record associated with the segment that encompasses that sequence of textual segments. In PLNLP, the simplest parsing rules are expressed schematically as in Example (1).

(1) $\mathrm{SYM}_1(\mathrm{tests}) \ldots \mathrm{SYM}_n(\mathrm{tests}) \rightarrow \mathrm{SYM}_{n+1}(\mathrm{actions})$

In Example (1), SYM_1 through SYM_n refer to the records that are being combined, and SYM_{n+1} refers to the record that is created for the textual segment made up of the concatenation of the segments associated with SYM_1 through SYM_n. Each SYM_i in Example (1) is itself the value of the designated attribute SEGTYPE for the corresponding record, and can be thought of as either a terminal or a nonterminal symbol in an augmented phrase-structure grammar. It is a terminal symbol if it corresponds to a single character (alphabetic, numeric or other single-character symbol, such as a punctuation mark) in the input string; otherwise it is a nonterminal symbol. Accompanying each symbol in the left side of the rule is a set (possibly empty) of tests of attribute values for that record or other records in the rule. The actions that are performed in connection with the record referred to in the right side of the rule create attribute values for that record. The ability to perform essentially arbitrary tests and actions in the course of executing a parsing rule of the form in Example (1) gives the system of augmented phrase structure rules greater generative power than the class of context-free phrase-structure grammars. For example, the PLNLP program in Figure 1 is capable of parsing all and only all of the strings of the artificial language in Example (2), which is known not to be a context-free language (Chomsky 1956).

(2) $\{xx. \mid x$ is any nonnull string of letters$\}$

Some PLNLP Conventions

Before we can describe how PLNLP parsing rules apply, we must first explain a few conventions regarding how PLNLP works.[1]

1. The designated records SNTBEG and SNTEND are inserted at the very beginning and the very end of the input. The input must end in one of the designated punctuation marks {. ! ?}. A blank (represented in PLNLP rules as "#") is inserted before the first character in the input and before the final punctuation mark. Sequences of two or more blanks in the input are reduced to a single blank.
2. For each alphabetic character in the input, a record is created whose SEGTYPE attribute has the value 'LETTER';[2] for simplicity, we say that a LETTER record is created in this circumstance. Similarly, for each numeric character in the input, a DIGIT record is created; and for each other nonblank character in the input, a SPECIAL record is created. Each of these re-

[1] For further details, see (Heidorn 1972, Langendoen 1989).

[2] The SEGTYPE attribute is actually a pointer to a certain kind of record, known as a "named record", whose properties are described in 4. on page 4

```
/* XX RULES */
/* Program for parsing strings of the form XX., where X is any string */
/* of letters. */
DECODE:

ROUTINES
          . "NLP-TIE", TOP "NLP-TOP", REST "NLP-REST"

DECODING

/* Create a LEFTSTR segment, made up of initial substrings of the */
/* input. Store the sequence of letters that make it up in the */
/* LETTERS attribute. */
(100)     # LETTER            --> LEFTSTR(LETTERS=LETTER)
(110)     LEFTSTR LETTER      --> LEFTSTR(LETTERS=LETTERS...LETTER)

/* If any input letter matches the first letter in LETTERS of LEFTSTR, */
/* begin a RIGHTSTR segment, made up of LEFTSTR plus the input letter. */
/* Store in its LLETTERS attribute the rest of the LETTERS of LEFTSTR. */
/* Store in its LETTERS attribute the matching input letter. */
(120)     LEFTSTR LETTER(STR.EQUAL.STR(TOP<LETTERS(LEFTSTR)>))
            --> RIGHTSTR(LLETTERS=REST<LETTERS(LEFTSTR)>), LETTERS=LETTER)

/* Continue to build up RIGHTSTR as long as the input letter matches */
/* the first letter in its LLETTERS attribute. */
/* Add the input letter to its LETTERS attribute, and remove the first */
/* letter from the LLETTERS attribute. */
(130)     RIGHTSTR LETTER(STR.EQUAL.STR(TOP<LLETTERS(RIGHTSTR)>))
            --> RIGHTSTR(LLETTERS=REST<LLETTERS(RIGHTSTR)>),
                         LETTERS=LETTERS...LETTER)

/* Analyze the final period and the blank preceding it as an instance */
/* of the segtype PUNC. */
(140)     # .                 --> PUNC

/* Parsing is successful if the entire input except for the final */
/* punctuation has been read, and the LLETTERS attribute of RIGHTSTR */
/* is empty. Create a SENT record, with attributes identifying its */
/* leftpart and its rightpart as the same as the LETTERS attribute of */
/* RIGHTSTR. */
(150)     RIGHTSTR(¬LLETTERS) PUNC
            --> SENT(LEFTPRT=LETTERS(RIGHTSTR), RIGHTPRT=LETTERS(RIGHTSTR))

EOF
:END-OF-FILE:
```

Figure 1. PLNLP program for parsing the language in Example (2).

cords, as well as the blank record, also has a STR attribute, whose value is the string made up of that character, and an FW and an LW attribute whose values are respectively the number of the position immediately preceding the beginning of the segment associated with the record and the number of the position immediately following the end of the segment associated with the record.

3. For each string flanked on each end by a blank and not containing an internal blank, a STEM record is created, with appropriate STR, FW and LW attribute values.

4. Records with particular attributes may be declared in a special "RECORDS" section of the program. These records automatically receive a NAME attribute, whose value is the string consisting of the name by which it was declared; such records are called "named records". Named records are referred to by putting their names in single quotes.
5. If a STEM record is created whose STR attribute matches the NAME attribute of a named record, then another STEM record is created which includes all of the attributes of the named record. Thus the RECORDS section can be thought of as providing a lexicon or dictionary for the program, and this convention as providing for lexical lookup.
6. A PLNLP program can invoke any Lisp/VM function. Certain functions which have been especially written for PLNLP programming must, however, be declared in a "ROUTINES" section and given new names. Among the most common of these are NLP-TIE, which is similar to Lisp/VM APPEND; NLP-TOP, which is similar to Lisp/VM FIRST or CAR; and NLP-REST, which is similar to Lisp/VM CDR. PLNLP functions can also be written, and put into a "PROCEDURES" section.
7. A PLNLP program must begin with the header line "DECODE:", and end with the two footer lines "EOF" and ":END-OF-FILE:".
8. The body of parsing rules are contained in a section labeled "DECODING".
9. If the program also includes text-generating rules, these must appear in a section labeled "ENCODING". Generating rules are written like ordinary augmented phrase-structure rules (i.e., the single symbol appears to the left of the arrow).
10. A comment line may appear anywhere in a PLNLP program. It is indicated by a slash ("/") as the last character in the line.

The Parsing Algorithm in PLNLP

The method of applying parsing rules to the input stream is described in detail in Heidorn (1972: 238ff.). Here we consider its most important aspect, the the parsing algorithm for PLNLP, which Heidorn (1972: 239) describes as follows:

1. Get the next character from the input stream and consider it to be a segment of that type (e.g., an "e" is a segment of type E).

2. Create a rule instance record for each rule for which this segment can be the first constituent, if there are any such rules, making note of this continuent in each record.
3. Make a copy of each rule instance record for which this segment can be the next constituent, if there are any such records, making note of this constituent in each new record.
4. If there is a rule instance record which is complete (i.e. has all of its constituents), create a segment record according to the right side of the associated rule (i.e. apply the rule), erase the rule instance record, and go to step 2.
5. If the input stream is not empty, go to step 1.
6. Halt.

We show how this algorithm works in the processing of the input *abab.* by the PLNLP program in Figure 1 on page 3.[3] For simplicity, we represent a rule instance record (RIR) as a list made up of its number, the number of the rule, a list of the records that have already been processed, and a list of records remaining to be processed.

Step	*Action*
1	Create a # record, which is associated with the first character in the input stream, by means of appropriate FW, LW and STR attributes.
2	Search the 6 parsing (decoding) rules for # as the first constituent. Two are found. Create the following RIRs: (1 100 (#) (LETTER)) (2 140 (#) (.))
3	Search the RIRs other than the ones just created for # as the next constituent. This step is vacuous in this case.
4	Search the 2 RIRs for completeness. None are found.
5	Go to step 1.
1	Create A and LETTER records for the next character in the input stream.
2	Search the 6 parsing rules for A, LETTER as first constituent. None are found.
3	Search the 2 RIRs for A, LETTER as next constituent. One is found. Copy and modify RIR#1.

[3] This input is modified to *#abab#.*, where # is the blank symbol, as described in 1. on page 2

(1 100 (# LETTER) NIL)

4 Search the 3 RIRs for completeness. One is found. Create LEFTSTR record by rule 100. Erase all instances of RIR#1. Go to step 2.

2 Search the 6 parsing rules for LEFTSTR as first constituent. Two are found. Create the following RIRs.

(3 110 (LEFTSTR) (LETTER))

(4 120 (LEFTSTR) (LETTER))

3 Search the 1 previously created RIR for LEFTSTR as next constituent. None is found.

4 Search the 3 RIRs for completeness. None are found.

5 Go to step 1.

1 Create B, LETTER records for the next character in the input.

2 Search the 6 parsing rules for B, LETTER as first constituent. None are found.

3 Search the 3 RIRs for B, LETTER as next constituent. One is found. Copy and modify RIR#3.[4]

(3 110 (LEFTSTR LETTER) NIL)

4 Search the 4 RIRs for completeness. One is found. Create LEFTSTR record by rule 110; erase all instances of RIR#3; go to step 2.

2 Search the 6 parsing rules for LEFTSTR as first constituent. Create:

(5 110 (LEFTSTR) (LETTER))

(6 120 (LEFTSTR) (LETTER))

3 Search the 2 previously created RIRs (RIR#2, RIR#4) for LEFTSTR as next constituent. None are found.

4 Search the 4 RIRs for completeness. None are found.

5, 1 Create A, LETTER records the the next input character.

2 Search the 6 parsing rules for A, LETTER as first constituent. None are found.

3 Search the 4 RIRs for A, LETTER as next constituent. Two are found. Copy and modify RIR#5, RIR#6.[5]

(5 110 (LEFTSTR LETTER) NIL)

[4] Although LETTER is the next constituent in RIR#4, the test associated with the LETTER record in rule 120 is not satisfied. Hence it is not copied.

[5] RIR#4 is not copied because its LEFTSTR and LETTER records are not continguous in the input.

(6 120 (LEFTSTR LETTER) NIL)

4 Search the 6 RIRs for completeness. Two are found. Create the records LEFTSTR by 110 and RIGHTSTR by 120. Erase all instances of RIR#5, RIR#6; 4 in all. Go to step 2.

2 Search the 6 parsing rules for LEFTSTR, RIGHTSTR as first constituent. Two found.[6] Create the RIRs:

(7 110 (LEFTSTR) (LETTER))

(8 130 (RIGHTSTR) (LETTER))

3 Search the 2 previously created RIRs for LEFTSTR, RIGHTSTR as next constituent. None are found.

4 Search 4 RIRs for completeness. None are found.

5, 1 Create B, LETTER records for the next input character; create STEM record for the input string ABAB.

2 Search the 6 parsing rules for B, LETTER, STEM as first constituent. None are found.

3 Search the 4 RIRs for B, LETTER, STEM as next constituent. Two are found. Copy and modify RIR#7, RIR#8.

(7 110 (LEFTSTR LETTER) NIL)

(8 130 (RIGHTSTR LETTER) NIL)

4 Search the 6 RIRs for completeness. Two are found. Create LEFTSTR record by 110; RIGHTSTR record by 130. Erase all instances of RIR#7, RIR#8, 4 in all. Go to step 2.

2 Search the 6 parsing rules for LEFTSTR, RIGHTSTR as first constituent. Three found. Create the RIRs:

(9 110 (LEFTSTR) (LETTER))

(10 130 (RIGHTSTR) (LETTER))

(11 150 (RIGHTSTR) (PUNC))

3 Search the 2 previously created RIRs for LEFTSTR, RIGHTSTR as next constituent. None are found.

4 Search the 5 RIRs for completeness. None are found.

[6] Rule 150 is not found, since the test associated with the RIGHTSTR segment is not satisfied at this point.

5, 1 Create # record for the next input character.

2 Search the 6 parsing rules for # as first constituent. Two are found. Create:

(12 100 (#) (LETTER))

(13 140 (#) (.))

3 Search the 5 previously created RIRs for # as next constituent. None are found.

4 Search the 7 RIRs for completeness. None are found.

5, 1 Create ., SPECIAL records for the next input character.

2 Search the 6 parsing rules for ., SPECIAL as first constituent. None are found.

3 Search the 7 RIRs for ., SPECIAL as next constituent. One is found. Copy and modify RIR#12.

(12 140 (# .) NIL)

4 Search the 8 RIRs for completeness. One is found. Create PUNC record by 140. Erase all instances of RIR#12. Go to step 2.

2 Search the 6 parsing rules for PUNC as first constituent. None are found.

3 Search the 6 RIRs for PUNC as next constituent. One is found. Copy and modify RIR#11.

(11 150 (RIGHTSTR PUNC) NIL)

4 Search the 7 RIRs for completeness. One is found. Create SENT record by 150. Erase all instances of RIR#11.

5 No action.

6 Halt.

In Figure 2 on page 9, a simple trace of the parse of this input is provided, leaving out the creation of the individual character, LETTER and STEM records (i.e., those records that are created by the system rather than by the parsing rules themselves), and in Figure 3 on page 10, the parse tree for this input is provided. In parsing this simple input, the set of six decoding rules is searched twenty times, while the RIRs are searched twenty-eight times, for an average of four RIRs per search.

```
INPUT:             #ABAB#.
FW OF CHARACTER:   2345678

FW SPAN  RULE
2 - 3    100:      # LETTER          -->  LEFTSTR
2 - 4    110:      LEFTSTR LETTER    -->  LEFTSTR
2 - 5    110:      LEFTSTR LETTER    -->  LEFTSTR
2 - 5    120:      LEFTSTR LETTER    -->  RIGHTSTR
2 - 6    110:      LEFTSTR LETTER    -->  LEFTSTR
2 - 6    130:      RIGHTSTR LETTER   -->  RIGHTSTR
7 - 8    140:      # .               -->  PUNC
2 - 8    150:      RIGHTSTR PUNC     -->  SENT
```

Figure 2. Simple trace of parse of the input string "abab."

Geometry of the Phrase Structure Grammar

The vectorization of PLNLP's parsing algorithm revolves around its core: the context-free phrase structure grammar (PSG). Since PSG's are widely used in both linguistics and computer science, our vectorization techniques are applicable to a wide variety of other PSG-based tools. Our approach to vectorization is relevant to phrase structure grammars in general (not just context-free ones).

As described above, every PSG consists of a set of terminal symbols, a set of nonterminal symbols, and a set rewrite rules used to parse (or in PLNLP parlance, decode) a linear sequence of terminal symbols into structures of terminal and nonterminal symbols and/or to encode structures of terminals and nonterminals into a linear sequence of terminal symbols.

On a non-vector machine, any nontrivial grammar must be represented in conventional main memory. The primary cost of the parsing algorithm employed is incurred in accessing main memory; simple algorithms require more memory accesses to the grammar rules themselves, while more sophisticated algorithms require more memory accesses for the instruction stream.

When mapping the PSG formalism onto a vector architecture, the chief parameters to be considered are the number of PSG rules, the maximum and average number of symbols in each rule, the number of vector registers available on the machine, and the maximum vector size supported by

```
                       SENT
                         |
                   ,-----+------,
                   |            |
               RIGHTSTR       PUNC
                   |            |
             ,-----+------,    ,+,
             |            |    | |
         RIGHTSTR         |    | |
             |            |    | |
       ,-----+-----,      |    | |
       |           |      |    | |
    LEFTSTR        |      |    | |
       |           |      |    | |
   ,---+----,      |      |    | |
   |        |      |      |    | |
LEFTSTR     |      |      |    | |
   |        |      |      |    | |
 ,-+-,      |      |      |    | |
 |   |      |      |      |    | |
 |LETTER LETTER LETTER LETTER  | |
 |   |      |      |      |    | |
 #   A      B      A      B    # .
```

Figure 3. Parse tree for the input "abab."

the machine. The number of PSG rules R, together with the maximum number of symbols found in any rule S, define a matrix of symbols with dimensions R x S. The number of vector registers V and the width (number of elements) W in each vector register define a matrix of vector register elements with dimensions V x W.

In representing the grammar within the vector machine, the matrix of symbols may or may not fit the vector registers, depending not only on the particular matrices defined by grammar and the machine in question, but also upon the mapping by which the matrix of symbols is stored in the vector registers and main memory of the machine. Below we present obvious mappings and their implications as a justification for our algorithm which follows. One might measure the efficacy of a given mapping by the percentage of symbols held within vector registers or the percentage of vector register elements utilized. A count of the actual number of main memory accesses required during the normal execution of the algorithm is a better measure of success.

PSG Rules as Vectors

The simplest mapping would be a one-to-one correspondence between PSG rules and vectors. Determining whether a rule matches the input string is very simple given such a layout: on the 3090 it can be performed in two vector instructions (details are ommitted):

VCER * compare the grammar rule vector against the input vector

VCZVM * count left zeros in the resulting VMR

Unfortunately, since the number of grammar rules is typically far greater than the number of vector registers available, the rules must in general be stored in main memory and loaded when needed by the algorithm. The degree of parallelism achieved by the mapping is limited to average length of the grammar rules. In the worst case this mapping achieves no parallelism at all.

Positions as Vectors

The obvious alternative is to lay out the rules "sideways". Placing the first element of all the rules in the first vector, and the second element of all rules in the second vector, and so on. By counting the number of symbols in each rule which match a prefix of the input, all the rules can be tested at once. At each step in the algorithm all the elements in the nth vector are compared to the nth symbol on the input. One vector is dedicated to holding the lengths of the rules; another vector holds the counts of matching prefixes as the input is examined. The innermost loop contains:

VCER * compare all the rules against the symbol (a scalar)

VAEQ * add a "1" into the counts of those rules which matched

Although this mapping requires slightly more pre- and post-processing in the algorithm, its central loop is comparable in complexity to the above. The degree of parallelism achieved by the mapping is limited by the number of rules in the grammar.

A Concrete Example

Although PLNLP accepts arbitrary PSG's, our interest is in maximizing performance for the kinds of grammars which come up in practice. As an example, IBM's PEG grammar has approximately 200 rules. PEG is nearly in Chomsky Normal Form, which is to say that most rules are binary; the average rule length is roughly 2.25, and the maximum rule length is 4. The IBM 3090 processor has 16 vector registers of up to 512 elements each. IBM's vector facility supports vectors of essentially arbitrary length using auxiliary index registers. On the other hand, the sixteen vector registers available impose a very specific structure on the organization of the optimal solution.

In the PEG example above, mapping each rule to a vector (and hence, a vector register) would result in roughly 0.5% register utilization. Only 8% of the rules could be held in the registers at one time; rules would general have to be loaded from main memory at each step of the input. The number of vector comparisons required at each step would be 200. At best, the vector implementation would run twice as fast as the scalar implementation.

Mapping each position to a vector, on the other hand, allows the entire PEG grammar to fit neatly within the 3090's vector registers. Only five vector comparisons are required at each stage of the input, four to count matching prefixes and one to check the counts against the length of the rules. Note that this technique not only improves performance in terms of memory access (as expected by the improved register utilization) but also reduces the number of instructions performed in comparisons. This latter result is a direct consequence of the increased parallelism employed in the mapping.

Elaborations

Although the PEG example motivates our selection of positions as vectors, it is highly dependent on the grammar submitted to PLNLP. Having dismissed mapping rules to vectors, we must now address the question of how to handle very long rules. For grammars whose longest rule is longer than twelve symbols, the entire grammar does not fit into the vector registers.

If most of the rules in the grammar are too large to fit the vector registers, it may be efficient to treat positions beyond the twelfth as vectors to be loaded and matched the same as the initial positions. If only a few rules have too many symbols it may be more efficient to adopt a mixed scalar/vector implementation. Allowing a vector register to hold the input and vector registers to hold the matching prefix counts and lengths for each rule as described above, the remaining vector registers can hold the first twelve symbols in each rule. This is enough to determine exact matches for rules with twelve symbols or less, and it is also enough to rule out longer rules where one of the first twelve rules does not match the input. Between these two cases, almost the entire match can be done purely in registers; the remainder can be done by a very few scalar operations.

Performance Measurements

Figure N compares our vectorizing PLNLP compiler against two alternatives: an implementation of PLNLP used internally by IBM, as well as our own compiler emitting scalar instructions instead of vector instructions. The purpose of the comparison against scalar version of our own compiler is to precisely isolate the effect of the vector facility upon parsing complexity. The purpose of comparison against the IBM implementation is to show the benefits to be had by switching to a modern compiler technology; our compiler emits code for IBM's C/370 product instead of the VM/LISP product.

Note that our compiler is considerably more portable than the previous implementation; the scalar code it generates is compatible with ANSI standard C and runs on machines as small as the IBM PC. Another significant improvement our compiler provides to the development process is improved compile-time; benchmarks compiling the same large grammar used to evaluate the performance of the parsing algorithm are provided in figure N + 1.

Conclusions

The degree of parallelization achieved by a vectorized implementation of a phrase structure grammar parser is dependent on both the grammar and the mapping of that grammar to machine vectors.

References

Aho, A.V., R. Sethi and J.D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley.

Allen, J. 1988. *Natural Language Understanding.* Menlo Park, CA: Benjamin.

Chomsky, N. 1956. Three models for the description of language. *IRE Transactions on Information Theory IT-2:3*, 113-124.

Grishman, R. 1986. *Computational Linguistics.* New York: Cambridge University Press.

Heidorn, G. 1972. *Natural Language Inputs to a Simulation Programming System.* Monterey, CA: Naval Postgraduate School.

Jensen, K. 1986. PEG 1986. Unpublished paper.

Langendoen, D.T. 1989. *A Linguist's Introduction to PLNLP.* Unpublished ms.