Toward Natural Language Computation[1]

Alan W. Biermann
Bruce W. Ballard[2]

Department of Computer Science
Duke University
Durham, North Carolina 27706

A computer programming system called the "Natural Language Computer" (NLC) is described which allows a user to type English commands while watching them executed on sample data appearing on a display screen. Direct visual feedback enables the user to detect most misinterpretation errors as they are made so that incorrect or ambiguous commands can be retyped or clarified immediately. A sequence of correctly executed commands may be given a name and used as a subroutine, thus extending the set of available operations and allowing larger English-language programs to be constructed hierarchically. In addition to discussing the transition network syntax and procedural semantics of the system, special attention is devoted to the following topics: the nature of imperative sentences in the matrix domain; the processing of non-trivial noun phrases; conjunction; pronominals; and programming constructs such as "if", "repeat", and procedure definition.

1. Introduction

Natural language programming has been proposed by many authors (Balzer[2], Green[13], Heidorn[17], Petrick[25], Sammet[27], Woods[38]) as the best way for humans to input their commands to computers. Humans have developed exquisitely efficient abilities for communicating with each other through natural language, and the possibility of similarly interacting with machines is worthy of investigation. The ability to program in natural language instead of traditional programming languages would enable people to use familiar constructs in expressing their requests, thus making machines accessible to a wider user group. Automatic speech recognition and synthesis devices could eventually smooth the communication even further.

On the other hand, many problems could arise when natural language programming is attempted (Dijkstra[11], Petrick[25], Simmons[32]), and any such research must deal with them. For example, it has been argued that current natural language technology is too primitive to handle a wide variety of syntactic and semantic constructs so that the user of such a system has the difficult task of learning what constitutes an acceptable input to the system. Instead of having to learn the relatively simple syntax of a clearly defined programming language, the user would be forced to learn a voluminous and very detailed set of rules giving what words and phrases can be used and how they can be combined. Thus the user would be taxed more heavily with a natural language system than with a traditional system. A second argument against natural language programming relates to its intrinsic vagueness and ambiguity. It is maintained that if one wishes to manipulate information precisely and reliably within a machine, a clearly defined and unambiguous language should be used. The programmer should not have to wonder about the meaning of a particular input to the system; he or she should know the meaning or be able to look it up easily in a manual. A third argument asserts that no one would use a natural language programming system, even if one existed, because it would be too verbose. Why should one be willing to input long and wordy descriptions of a desired computation when there exist simple, easy-to-learn, and concise notations for doing the same thing?

[1] This material is based upon work supported by the National Science Foundation under Grant Numbers MCS74-14445-A01 and MCS-7904120.

[2] Current address: Department of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210.

0362-613X/80/020071-16$01.00

1.1 A Natural Language Computer

Formidable as these criticisms may seem, this paper will attempt to show that some of them can be overcome with a careful system design, while others may be simply wrong. This paper describes a system, called the Natural Language Computer (NLC), which makes it possible to perform a limited amount of natural language programming. This system enables a person to sit at a computer display terminal, observe his or her data structures on the screen, and watch the computation proceed as the individual commands are typed. The current implementation is specifically designed for array and matrix computation. In the example interaction of Figure 1, the user is creating a subroutine to add up the rows and columns of an arbitrary matrix. Each item that is modified by a command is marked with an asterisk and other items used in the calculation are marked with an apostrophe. The first two commands cause two matrices to appear on the screen, one to be operated on and the other to receive the answer. The third command provides some sample data for the calculation and the fourth command indicates that the imperative verb "sumcolrow" is about to be defined. Subsequent inputs tell exactly what operations must be done to sumcolrow a matrix into another matrix. For example, if after this dialogue the system receives the input

"Sumcolrow matrix A1 into matrix Z."

where A1 and Z are matrices of appropriate dimensions, the procedure body following the define statement will be executed. Thus the dialogue causes the system to create a subroutine to be called by using the newly defined imperative verb "sumcolrow".

This process is explained in greater detail in later sections of the paper. The important point to be noticed here is that the user is able to watch the system respond to each command as it is entered. Whenever the system yields an undesired action, the user can back up and rephrase his or her command more clearly. This format for natural language programming enables users to examine system performance as each command is typed and to detect most errors immediately.

1.2 Concerning the Objections

Given this brief introduction to the NLC user interface, it is already possible to respond to the first criticism of natural language programming given above. Although we feel that present day natural language processors tend to have very limited capabilities, it is quite possible within the NLC environment to direct the user to restrict the inputs appropriately. For this reason, the user is asked to follow two simple rules which are easily understood from the user's point of view and which simultaneously ease the job of the system designers and implementers considerably.

```
"Display a 3 by 3 matrix. Call it testmat."
  Testmat
        1    2    3
   1    0    0    0
   2    0    0    0
   3    0    0    0
"Display a 3 by 2 matrix. Call it ans."
  Testmat                       ans
        1    2    3                   1    2
   1    0    0    0              1    0    0
   2    0    0    0              2    0    0
   3    0    0    0              3    0    0
"Fill testmat with random numbers."
  Testmat                        ans
        1    2    3                   1    2
   1    1*   4*  -7*              1   0    0
   2    2*   3*   1*              2   0    0
   3    4*   2*   2*              3   0    0
"Define a way to sumcolrow testmat into
 ans."
    "Add each column in testmat to
     column 1 in ans."
     testmat                     ans
        1'   2'    3'                  1*    2
     1  1'   4'   -7'              1 -2*    0
     2  2'   3'    1'              2  6*    0
     3  4'   2'    2'              3  8*    0
    "Add each row in testmat to
     column 2 of ans."
     testmat                       ans
         1     2     3                  1     2*
     1'  1'    4'   -7'             1 -2     7*
     2'  2'    3'    1'             2  6     9*
     3'  4'    2'    2'             3  8    -4*
"End the definition."
```

Figure 1. Defining the verb "sumcolrow".

The first rule concerns the *semantics* of inputs:

(1) The user may refer only to the data structures seen on the terminal screen and specify simple operations upon them.

That is, the user may refer to matrices, rows, columns, entries, labels, numbers, variables, etc., and specify simple operations such as add, subtract, move, exchange, delete, label, etc. The user may not use domain specific vocabulary or concepts such as airplane flights, seats, passengers, and reservations. This rule is easily explained to a user and makes it possible to build a system without getting into the peculiarities of any specific domain. Although it requires the user to translate his or her problem into the vocabulary of the system, it also makes it possible to experiment with the system in many different domains.

The second rule concerns the *syntax* of the inputs:

(2) The user must begin each sentence with an imperative verb.

This rule is also easy to explain to the user and it also greatly restricts the variety of sentences to be processed. If this rule is followed, the system can find out much about each clause from its first word, including what words or concepts may occur later in the clause.

In summary, the strategy for achieving person-to-machine language compatibility taken here is (1) to find a small number of simple rules which a person can easily follow to restrict the set of inputs; and then (2) to stretch the language processing technology to the point where it can reasonably cover that set. When this is done, the first criticism of natural language programming stated earlier is overcome.

The other major objections to natural language programming relate to its vagueness, ambiguity, and alleged verbosity. Perspectives on these issues can be achieved by examining some examples of natural language and the corresponding programs in traditional programming languages. Consider for example the command

"Square the sixth positive entry in matrix M."

Vagueness does not appear to be a problem with the English of this example. In fact, the sentence is probably shorter than most equivalent formulations written in traditional programming languages. The corresponding code in almost any programming language will require some declarations and a nesting of looping and branching constructs. As an additional example, the reader should examine the English language program and its corresponding PL/I counterpart which is included in the Appendix. Our experience so far with English language programming seems to indicate that the language is as precise as its user wants it to be. Concerning the length of English language programs, they seem to be comparable to the length of ordinary programs in the domains we have examined. Of course, one could write down a complicated arithmetic expression from some standard programming language and note that its English equivalent is relatively long, unreadable, and unwieldy. The solution to this problem is to include in the natural language processor the ability to handle such arithmetic expressions. Considering the complexity of any reasonable natural language processor, the cost of adding something like an arithmetic expression handler is modest. Other constructs from programming languages which are shown to be convenient could also be considered for inclusion.

1.3 Background

The NLC system has grown out of an earlier series of studies on the "autoprogrammer" (Biermann[6]) and bears much resemblance to it. Program synthesis in both the current and the previous systems is based upon example calculations done by the user on displayed data structures. In the current system, the example is done in restricted English with all its power, which is a dramatic departure from the earlier approach, which simply involved pointing with a light pen. However, it is expected that many of the features from the autoprogrammer, such as "continue" and "automatic indexing", will transfer quite naturally into NLC. This paper emphasizes the natural language aspects of the system, while other reports deal with some of the additional automatic programming features. The relationship of NLC to other research in natural language processing is discussed in a later section.

The next section presents an overview of NLC, after which subsequent sections discuss scanning, syntactic and semantic processing, and interpretation of commands in the "matrix computer". The next two sections discuss the processing of flow-of-control commands and the level of behavior achieved by the system. The final sections include a discussion of related research and conclusions.

2. System Overview

The NLC system is organized as shown in Figure 2, with the user input passing through the conventional sequence of stages: lexical, syntactic, and semantic processing. The scanner finds the tokens in the input sentence and looks them up in the dictionary. It performs some morphological processing and spelling correction for items not appearing in the dictionary. Additionally, abbreviations (such as "col" for "column") and spelled-out numbers and ordinals ("twenty-two", "seventh", etc.) are recognized. The identified words with their meanings are passed on to the parser, which is programmed with nondeterministic transition nets similar to the augmented transition networks of Woods[40]. The parser has the ability to screen out many syntactically correct but semantically meaningless structures so that the first parse it finds is usually correct. The parser output goes to the flow-of-control semantics routines which make decisions about the nature of the input command and then properly guide it through subsequent processing.

The input sentence may be a simple request for a system defined computation or it may be a flow-of-control command such as a user-defined subroutine call. An example of the first case is "Add row 1 to row 2." Here flow-of-control processing sends the sentence directly to the sentence semantics routines which resolve the noun groups and invoke the matrix

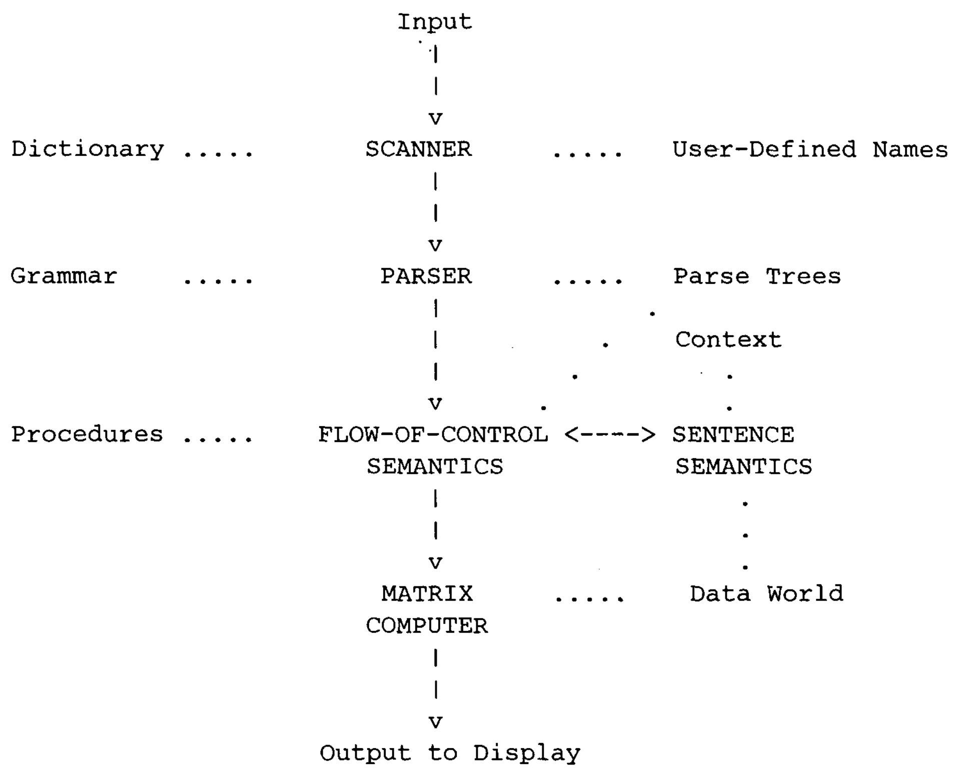

Figure 2. The NLC system modules (upper case) and their associated data structures (lower case).

computer to perform the indicated operation. An example of the second case is a command beginning with a user-defined verb such as "sumcolrow". Here flow-of-control processing brings in from a file the set of commands for the subroutine which defines the word "sumcolrow". Then those commands, with parameters properly instantiated, are sequentially transferred to sentence semantics for execution.

The major task of the sentence semantics routines is the processing of noun groups. They begin with the head noun in any particular noun group and build a representation for the meaning of the noun group by sequentially processing whatever modifying words and phrases there may be. These routines are concerned with qualifying relative clauses, prepositional phrases, adjectives, ordinals, pronouns, and numerous other constructions appearing in noun groups. The result of noun group processing is usually a designation of an item or set of items in the displayed data structures to be manipulated by the matrix computer.

Most imperative verbs such as "double" or "add" pass through the system without change until they reach the matrix computer. This routine then performs the indicated operation on the data specified by the processed noun groups. All changes in the data structures are immediately updated on the display screen, along with markers to show the user where the changes have been made. A few imperative verbs are not processed by the matrix computer. Some examples are "find" or "choose", which are processed by the sentence semantics module, and "repeat" or user-defined imperatives, which are processed by flow-of-control semantics.

Every effort has been made to modularize the system for understandability and easy modification. In addition, the design attempts to use limited computer resources economically. It is written in the C language and runs on a PDP-11/70 under the UNIX operating system.

3. The Scanner

The scanner collects the string of tokens from the input and identifies them as well as possible. These tokens may be numbers or ordinals in various forms, names known to the system, punctuation, or dictionary words which may be abbreviated or misspelled in a minor way. The scanner outputs a set of alternative definitions for each incoming token, and the syntax stage attempts to select the intended meaning for each one.

Each dictionary entry consists of a set of pairs of features. Two examples appear in Figure 3, the definitions of the word "zero" as an imperative verb and as an adjective. "Zero" as a verb takes one argument and no particle (type OPS1). The meaning of an imperative verb is built into the execution code of the matrix computer as explained in Section 6. As an adjective, the meaning of "zero" is embedded in the semantics code described in Section 5. That code will execute a routine associated with the name in the AMEANS field, zero.

```
1.  (QUOTE       zero)
    (PART        IMPERATIVE)
    (IMPERTYPE   OPS1)
2.  (QUOTE       zero)
    (PART        ADJ)
    (AMEANS      zero)
```

Figure 3. Two sample dictionary entries.

Figure 4 shows the output from the scanner for an example input sentence. Associated with each token is the set of alternate definitions proposed by the system and the syntax stage will attempt to make appropriate choices such that the sentence is meaningful. Most tokens are found in the dictionary, but the string "thee" is not. So dictionary entries are selected by the spelling corrector which are similar to the unknown. The token "y" is also not found in the dictionary but is recognized as the name of an existing matrix entity. The words "zero" and "to" appear in the dictionary with multiple meanings.

WORD	INTERPRETATION(S)		
Add	add	-	verb
y	y	-	propname
to	to	-	verbicle
	to	-	prep
thee	thee	-	propname
	the	-	art
	them	-	pron
	then	-	etc
	there	-	etc
	these	-	pron
	these	-	art
	three	-	num
zero	zero	-	verb
	zero	-	adj
	zero	-	num
entries	entries	-	noun
	.	-	punctuation

"Add y to thee zero entries."

Figure 4. Scanner output for a sample sentence giving alternate interpretations for each word.

4. Syntax

Most of the sentences processed by the system can be thought of as imperative verbs with their associated operands. For example, the sentence

> "Add the first and last positive entries in row 1 and the second to smallest entry in the matrix to each entry in the last row."

exhibits the overall form

> (add x to y)

where x is the noun group "the first and last ... in the matrix" and y is "each entry in the last row". The system separately processes constructions related to the imperative verbs and those related to noun groups. The following two sections discuss these types of constructions. Then, Section 4.3 describes a method for rejecting certain kinds of syntactically correct but semantically unacceptable parses, Section 4.4 describes our approach to handling syntactic ambiguity, and Section 4.5 gives the form of the output for the parser.

4.1 Imperatives And Their Operands

A transition net for processing the above imperative form for "add" is shown in Figure 5. The word PARSE means to call routines appropriate for parsing the indicated construct. IMPERATIVE refers to the imperative verb, and NG refers to the noun group. VERBICLE refers to a particular type of preposition which is often associated with an imperative verb to distinguish its operands. Thus in the sentences

> "Multiply x by y."
> "Store x in y."

the words "by" and "in" are verbicles. Of course, any given imperative will have only a few acceptable verbicles, so the parser checks that a suitable one is found.

```
        START
          |
   PARSE IMPERATIVE
          |
       PARSE NG
          |
    PARSE VERBICLE
          |
       PARSE NG
          |
      PARSE "."
          |
       SUCCEED
```

Figure 5. A top-level parser for sentences of the form "add X to Y".

4.1.1 Conjunction Handling

Although the routine of Figure 5 might be adequate for a large fraction of the sentences received by NLC, we decided to formulate a facility for handling a wide variety of conjunctions [33]. Toward this goal, a routine called MIX was designed as shown in Figure 6.

```
                START
                  |
                  v
----- PARSE A -----> PARSE ","
|                 |      A      |
|                 |      |      |
|                 |      ---- PARSE A
|                 |             |
|                 v             v
| --->PARSE "and" <--PARSE ","
| |               |
| |               v
| ---- PARSE A
|                 |
|                 v
-----> SUCCEED
```

Figure 6. A simplified transition network for MIX A.

Suppose A is a given construct and suppose x1, x2, and x3 are instances of that construct. Then MIX A will process forms such as

x1
x1 and x2
x1, x2, and x3
x1 and x2 and x3

and others. If, for example, A represents the imperative clause construct, then MIX A will process

"Add y1 to y2, add y3 to y4, and add y5 to y6."

If A is the unconjoined noun group, then MIX A will process

"row 1, row 2, and row 3."

Figure 7 shows how a series of calls of the MIX routine can be used to process reasonably complex nestings of conjunctions. For example, these routines will parse the sentence

"Add y1 to y2, to y3, and to y4
and y5 to y6
and add y7 to y8."

4.1.2 Other Sentence Forms

Of course, not all verbs take two operands and a verbicle as in the examples above. Indeed, verbs such as "call" have two operands without a verbicle:

"Call the matrix x." (Call y1 y2.)

There are also one-operand verbs which take a particle, such as "add up". Particles present a special problem since they can appear in various positions in the sentence; NLC handles most of the common placements. Many one-operand verbs appear without particles as in

"Double row 1." (Double y1.)

and there are verbs that take no operand: either with a particle, as in

"Back up."

or without a particle, as in

"Quit."

Most of the imperatives handled by NLC fall into one or more of the six categories listed above: zero, one, or two operands, with or without a verbicle/particle. The conjunction handling described above extends to all of these types of imperatives in a natural way. Although NLC has facilities for accepting imperatives with more than two operands or with formats other than those given here, a large proportion of all imperatives in our domain do fit into the simple scheme given here.

4.2 Noun Group Syntax

Four types of noun groups appear in the sentences processed by NLC. The most common type refers to the entities on the NLC display screen: numbers, entries, rows, matrices and so forth. These are the noun groups that appear as operands for the imperative verbs. Many examples appear in previous sections. The second type of noun group is the noun

```
        START                   START
          |                       |
        MIX S1             PARSE IMPERATIVE
          |                       |
       PARSE "."              MIX NGVNG
          |                       |
       SUCCEED                 SUCCEED

(a) Top level            (b) Clause level
    routine S                routine S1.

        START                   START
          |                       |
       PARSE NG             PARSE VERBICLE
          |                       |
       MIX VNG                 PARSE NG
          |                       |
       SUCCEED                 SUCCEED

(c) Noun-verbicle-       (d) Verbicle-noun
    noun level               level routine VNG
    routine NGVNG
```

Figure 7. A sentence parser allowing nested conjunctions.

result group, which refers to the result of a computation. Some examples are "the sum of rows 1 and 2" and "the absolute value of x" where in each case the object being referred to appears not on the screen but is found by manipulating displayed objects. The third type of noun group is the noun place group, as illustrated by "bottom" in

"Add the second from bottom row to row 3."

"Bottom" in this example is the place from which the ordinal processor begins counting. Some other words that can fit into this slot are "right", "left", "top", and "last". The fourth type of noun group is the noun procedure group, which refers to a procedure, a command, or a set of commands in the NLC input. This type is illustrated in

"Repeat the last three commands ten times."
"Double the entries the third command incremented."

Only the operand noun groups will be discussed in detail here.

Operand level noun groups follow a format similar to the one given by Winograd[37]. Let OPT be a routine which optionally calls a set of routines. As an illustration, OPT DETERMINER calls routines to parse a determiner. If those routines fail, however, OPT succeeds anyway, assuming that the noun group exists without a determiner. The basic format for the operand level noun group parser, given in Figure 8, is completely exercised by the noun group

"the first three positive matrix 1 entries which are odd"

DETERMINER:	the
ORDINAL:	first
NUMBER:	three
ADJECTIVE:	positive
CLASSIFIER:	matrix 1
NOUN:	entries
QUALIFIER:	which are odd

Since OPT is used to look for most of the constituents, the parser analyzes noun groups with those elements missing. (Examples: "the positive entries", "seven numbers greater than 10", "the smallest entry", etc.) Constructs of the form "row 1", "columns 2 and 3", or "the constant 4.5" require separate recognition.

The DETERMINER routine parses not only the simple determiners "the" and "a/an" but also a variety of quantifiers such as "all", "all of the", "both", "no more than six of the", "exactly two of the", and many others. The ORDINAL routine processes the common ordinals "first", "second", "next", and "last", which can also appear with superlatives ("second greatest") or with modifiers ("second from right", "second from last").

```
                  START
                    |
                    v
            OPT  DETERMINER
                    |
                    v
               OPT  ORDINAL
                    |
                    v
                OPT NUMBER
                    |
                    v
     ---->  OPT ADJECTIVE
     |                |
     ------------ v
     ----> OPT CLASSIFIER
     |                |
     ------------ v
               PARSE NOUN
                    |
                    v
     ----> OPT  QUALIFIER
     |                |
     ------------ v
                 SUCCEED
```

Figure 8. A Winograd-style noun phrase parser.

Six types of qualifiers are handled by NLC:

1. Preposition groups:
   "the rows IN MATRIX 2"
2. Adjective groups:
   "the numbers LARGER THAN 6"
3. Relative clauses:
   "the rows WHICH CONTAIN NEGATIVE NUMBERS"
4. ED groups:
   "the entries ADDED"
   "the entries ADDED TO"
   "the entries ADDED TO ROW 4"
   "the entries ADDED BY THE LAST COMMAND"
5. ING groups:
   "the columns CONTAINING 5.5"
6. Rank-shifted clauses:
   "the entries COLUMN 2 CONTAINS"

Many types of conjoined phrases are processed using the MIX routine as in "the first and last entries", "the first two and last three entries", "the first two and the last three entries", and others. Noun groups may be nested within other noun groups as illustrated in

```
"the largest entry
    in the first row
      of the matrix
        containing the column
          that was doubled by the second to last
                      command"
```

4.3 Semantic Checking During Syntactic Processing

If the parser is provided with some information about the types of nouns and the relationships they may have with each other, it can reject inappropriate parses. As an illustration, in the following phrase a possible parse of the qualifiers is as indicated by the parentheses.

the entry (in row 2 (in column 3))

That is, row 2 is "in" column 3 and the entry being referred to is in that row 2. However, in an ordinary matrix it is not possible for a row to be contained in a column and so it is desirable that this parse be rejected. The correct parse will be found if it is known that row-in-column is a disallowed pattern, forcing "row 2" to stand alone as a noun group:

the entry (in row 2) (in column 3)

Thus both the qualifiers "in row 2" and "in column 3" modify the noun "entry". Since entry-in-row and entry-in-column are semantically acceptable patterns, this parse can be passed to the semantics processor.

Observations of this type lead to the concept of semantically acceptable patterns and a mechanism for checking for them. A hash-coded table was added to NLC which contains the set of all semantically acceptable patterns for certain constructions. At various times during the processing, checks are made to see that a sensible parse is being assembled. Besides checking for compatibility in prepositional modifiers as indicated above, the system tests relationships given by relative clauses and adjective groups. It also checks that the operands of imperative verbs are legitimate.

4.4 Syntactic Ambiguity

The strategy for dealing with syntactic ambiguity is to attempt to anticipate the situations in which it is most likely to arise and to decide, whenever possible, which alternative is most reasonable. Having made such decisions, it is usually possible to order the grammar rules in such a way that the preferred parse is the one arrived at first, thus combining the efficiency of a blind search with the accuracy of a more extensive one. Perhaps surprisingly, the method has proven quite successful in meeting the stated objectives. (See [5].) This is due in part to the formulation of several general principles stemming from our observations of how natural language is employed in the NLC domain. The most important of these are:

1. Deep parses are generally preferred. Thus,
   "x in y in z"
   more often attaches the qualifier "in z" with y than with x when both readings are meaningful.
2. When ambiguity arises because of a conjunction, the intended conjuncts are likely to have similar type. This contrasts sharply with conventional programming languages, where operators rather than operands determine the "binding" in arithmetic expressions such as "a + b * c". The preference for conjoining similar units is automatically supplied by using the MIX routine described earlier.
3. Compatibility checks based on semantic relationships should be checked during the parse as described in Section 4.3. This offers the benefit of suspending parsing to obtain semantic information without incurring the inefficiency of such action.
4. Special cases exist and should be introduced as such, rather than erroneously generalized to the point of introducing the possibility for parses which users would find ungrammatical.

4.5 Syntactic Processor Output

The output of the syntax processor is a template for each clause giving the imperative verb and pointers to structures which represent the operands. Figure 9 gives an example of such an output.

```
Imperative Verb Template
OPERATOR  add
OPERAND   1
          ARTICLE   DETERMINED
          ARTSP     SING-PLUR
          NOUN      entry
          NOUNSP    PLUR
          Modifier 1
                    PREP    IN
                    NOUN    row
                    NOUNSP  SING
                    WHICH   1
          Modifier 2
                    COMP     GREATER
                    NOUN     CONSTANT
                    WHICH    6.0
VERBICLE to
OPERAND    2
           NOUN    PROPNAME
           QUOTE   X
```

Figure 9. Output of the syntax processor for the sentence "add the entries in row 1 greater than 6.0 to X.".

5. Sentence Semantics

The primary responsibility of the semantics module of NLC is the processing of noun groups to determine their referents. Input to semantics consists of the parse trees constructed by the syntactic processor. The imperative, along with its verbicle/particle, is saved for later context references, but not operated upon at this time. The principal role of semantics is to produce a precise internal representation that can be used by the matrix computer in carrying out the requested command.

A secondary role of semantics is to update context as a consequence of resolving noun phrases. In this way, one may refer to previous actions of the system. Thus:

"Clear the column that was added to column 2."
"Increment by 5 the row which the last command squared."

Context is also utilized in the location of referents for pronouns and other words requiring pronominal processing. Some examples are:

"Multiply the smallest entry by IT."
"Replace THAT ENTRY by ITS reciprocal."
"Subtract the NEXT 2 entries from each member of row 2."
"Sum up the OTHER entries in those rows."

The following sections describe briefly the representations used in the system, noun group resolution, and the processing of pronominal structures.

5.1 Internal Data Structures

For the matrix computer to carry out users' commands, physical addresses of the operands must be available. The resolved nouns must also be stored at a conceptual level so that later sentences may refer to the objects operated upon. For these reasons, the basic internal representation of the domain entities consists of a collection of intermediate structures from which hardware addresses are computed. Since the syntax parse trees are available, this intermediate notation does not refer to the natural language input.

Most of the internal structures, denoted "datareps", refer to a singular domain entity and have a fixed number of parameters. These "primitives" are: entry, row, column, matrix, domain, float constant, int constant, name, noun result, result, and command. As an example, the datarep for the noun group "row 2" is (ROW 1 2) which fills 5 bytes in memory and gives the name of the entity, the matrix number, and the item designation.

Plurals may arise in a variety of ways, some of which are presented here. In the simplest case, a plural datarep is the direct result of the resolution of a plural noun, as in

"rows 3, 4 and 5"
"the entries in rows 1 and 2"

Word-meaning routines such as adjective, ordinal, and superlative may produce a plural output, as in

"the positive entries in row 2"
"the last 3 entries that were doubled"
"the smallest 3 numbers in the last column"

In addition, plurals may result from the conjoining of singular datareps

"row 4 and column 5"
"row 2 and the last row"

or from conjunctions in which one or more of the conjuncts is itself plural

"row 3 and the rows containing positive entries"
"the first 3 and the last 2 rows in matrix 1"

An important feature common to all the types of conjunctions mentioned above is that the members of the "set" which represents the resulting plural datarep are themselves singular. Thus, for the noun phrase

"row 6 and the first 2 rows"

the resolution will be

```
SET of size 3:
    ROW 6
    ROW 1
    ROW 2
```

Because of the manner of manipulating the internal structures and passing them between modules of the NLC system, an array-like data structure was chosen for sets instead of a LISP-like representation. A detailed description of the precise mechanism for representing sets, beyond the scope of the present paper, may be found in Ballard[1].

5.2 Noun Group Resolution

The discovery of the meaning of a particular noun group begins with the head noun and any "in" qualifier which may be found. Thus in the phrase

"the smallest entry in row 2 greater than 10,"

the meaning of the words "entry in row 2" is initially represented as the set {(ENTRY 2 1), (ENTRY 2 2), . . . , (ENTRY 2 N)}. Then processing of other qualifiers and lastly prenominal modifiers has the effect of removing entries from the initial set. In this case, processing of "greater than 10" causes the system to reference the values of the listed entries and remove from the set those entries not meeting the specified criterion, "greater than 10". Processing of "smallest" results in all but the appropriate smallest entry being removed, and processing of "the" involves checking that the resulting set has only one member since the head noun is singular. The final meaning of the noun group is thus the set {(ENTRY 2 i)} for some i and this representation is passed to the matrix computer as an operand for some computation.

5.3 Pronominalization

The basic syntactic types of the pronouns within the matrix domain are the following:

1. pure pronoun
   "it" / "them"
   "itself" / "themselves"
2. pronominal determiner
   "THAT entry" / "THOSE columns"
3. possessive determiner
   "ITS rows" / "THEIR columns"
4. pronominal ordinal
   "NEXT row"
   "OTHER entries"

The fourth category is included among the listing of pronouns because the semantics involve most of the same principles. For instance, "the other entries" demands the semantics that would occur for "the entries other than ?", where "?" represents the most general possible pronoun, having no type or number constraints.

Pronoun reference is done by considering previous datareps rather than by traversing trees as described by Hobbs [22]. Specific guidelines for posing the eligible referents to pronouns in a reasonable order include, in order of importance:

1. In all cases, require type, number, and semantic constraints of the pronoun to agree with the datarep being examined.
2. Prefer more recently created datareps.
3. For case-level (operand) pronouns, try to match source with an old datarep source, destination with an old destination.
4. "Fuse", or conjoin, singular datareps to produce a plural referent if necessary. Thus

   "Add row 1 to row 2."
   "Double those rows."

   entails creating the set consisting of rows 1 and 2 at the time pronoun referent location occurs.
5. Consult more distant sentences only after trying all possibilities on an intervening sentence. Thus,

   "Clear rows 1 and 2."
   "Triple column 4."
   "Add row 3 to row 4."
   "Double those rows."

   will prefer the complicated but recent fusion in the immediately preceding command over the exact but less immediate plural three sentences earlier.

6. The Matrix Computer

The "matrix computer" of NLC is assigned two major tasks: (1) carrying out the computations which the user has requested and (2) displaying on the terminal the resulting data world. Since the latter function is conceptually simple (although tedious to code effectively) and since sample system outputs are provided in Figure 1, this section will concentrate only upon the techniques which the matrix computer uses to perform the desired computations.

As discussed in the previous section, essentially all processing of noun phrases is completed by the semantics module. What is made available then to the matrix computer is a collection of templates, similar to the ones generated as the parser output, as shown earlier in Figure 9, but with the noun arguments fully "resolved" into datareps as already described. As an example of the templates received by the matrix computer, consider the English input

"Add up the first row, double row 2, and subtract row 4 from row 5."

The semantics output for this input is

```
Template 1:
                Verb:                 "add"
                Verbicle/Particle:    "up"
                Operand:               (ROW 1)
Template 2:
                Verb:                 "double"
                Verbicle/Particle:     --
                Operand:               (ROW 2)
Template 3:
                Verb:                 "subtract"
                Verbicle/Particle:    "from"
                Operand 1:             (ROW 4)
                Operand 2:             (ROW 5)
```

The task of the matrix computer is to decide upon the appropriate operations and to apply them to the operands.

It was mentioned earlier that the imperatives, particles, and verbicles recognized at parse time pass through semantics without alteration. When the output from semantics becomes available to the matrix computer, the imperative verb and the associated verbicle/particle (if there is one) are looked up in a table to determine the appropriate action. In most cases, it has not been necessary to write a separate procedure for each imperative. Specifically, "double" is treated as a special case of "multiply". Thus the user input

"Double the first 2 entries in column 1."

entails the matrix computer operations

```
Arith-op(*, 2.0, (ENTRY 1 1))
Arith-op(*, 2.0, (ENTRY 2 1))
```

where "Arith-op" is the general coding capable of performing the basic arithmetic operations. In this way, the matrix computer has accommodated a large number of arithmetic commands by the mere addition of one table entry (8 bytes). Further instances of arithmetic verbs which make use of the arithmetic-op code are clear, copy, decrease, decrement, divide, halve, and many others.

In general, operating with a scalar (element, variable, or constant) upon an aggregate (row, column or matrix) means to operate independently on each member. The knowledge of how to perform the intended operations for all meaningful source-destination pairs must be coded into the system. This means specifying for each X-Y pair exactly what is required by

```
Arith-op(op, X, Y)
    for datareps 'X' and 'Y'
    and 'op' a member of { +, *, ... }
```

The remaining type of matrix computer operation deals with the noun result such as in

"Add the PRODUCT of the positive entries to row 1."
"Add the SUM of rows 3, 4 and 5 to row 6."

Notice that a noun result may yield a scalar as in the first example or a vector as in the second. The noun result is evaluated similarly to imperative verb operations, and the result of the calculation is inserted into the appropriate higher level structure for further processing.

7. Flow-of-Control Semantics

Thus far, this paper has discussed sentence by sentence processing, where system actions occur one at a time, determined directly from keyboard inputs. Most means of enhancing the usefulness of the system fall into one of two categories: (1) the introduction of programming-language type control structures and (2) the ability to define and execute self-contained portions of natural language "coding". These topics are addressed separately here.

7.1 Conditional Execution

To specify conditional execution of a command or a group of commands in English, one uses such words as "if", "when", "unless", "otherwise", etc. Following are some sentences typical of inputs that NLC-like systems will likely be called upon to process.

"IF row 3 contains a positive entry ..."
"IF the largest entry occurs in the last row ..."

Implementation of this facility is not complete on the NLC system. When it becomes totally operative, users will be told that they may begin sentences with the word "if" as well as with imperative verbs. The characteristic language feature appearing in each of the above sentences is the independent clause: declarative (rather than imperative) in nature, and requiring the evaluation of a Boolean (i.e., a condition whose truth or falsity is to be determined). Fortunately, the conjugated verbs are typically either "be" or one of the verbs which have already occurred in relative clauses, and so an appreciable degree of different syntactic processing is not required. Thus,

"if row 3 contains a positive entry ..."

relates directly to

"the rows which contain a positive entry"

The semantic routines originally written for qualifier verbs can, therefore, with a slight modification be used in handling these constructs.

7.2 Looping

NLC provides several ways of creating loops using the verb "repeat." In the typical situation, the user supplies and observes the execution of a sequence of commands on particular members of the data world. The system is then capable of abstracting, from the specific instructions, general code to operate on other entities. Frequently, an algorithm requires the application of a sequence of commands to several members of the domain. One way of accomplishing this is to make use of the following pattern.

"Choose an entry which ... and call it x."
"... to x."
"... x by ..."
"Repeat for the other entries."

When such a sequence is recognized, the "repeat" processor finds the set given to the most recent unmatched "choose" (or "pick", etc.) and thereby knows what "other" members are to have the intervening commands applied to them. In instances where there is no non-deterministic "choose"-type operation to delineate the statements to be repeated or to make explicit the set to which previous commands are to be applied, alternate versions of "repeat" are provided. Examples include

"Repeat the last 3 commands."
"Repeat the last command 5 times."
"Repeat the last 3 commands for all other odd entries."
"Repeat those commands twice on row 3."

7.3 Procedures

Another way of extending sentence by sentence processing is the facility for defining procedures, enabling the user to describe operations in terms of existing commands. Subsequent inputs may access the newly-created imperatives as though they had been previously defined, including their use in further procedure definitions. Programmers will recognize the following illustration as an instance of the "called-procedure" type of subroutine. There is no reason, however, for not providing the "function" procedure as well. Interestingly, the "noun result" discussed earlier corresponds to this value-returning subroutine. In addition, the NLC design includes the creation of new adjectives. The correspondences between natural language words and conventional programming language procedures are roughly as follows.

Natural Language	*Programming Language*
imperative verb	"called" subroutine
noun result *	"function" subroutine
adjective *	"predicate"

(* - not yet operative on NLC)

Both the noun result and the adjective routines require an explicit "return" command. Methods of incorporating them into the present system, as well as ways of relaxing the restrictions for the imperative verb procedures discussed below, are being developed.

In order to assure correct re-execution of the commands within a procedure, it is necessary to detect occurrences of the parameters among the nouns of the procedure body. To accomplish this, the system requires that the arguments at the time of procedure definition be names. When a user input indicates that a procedure is about to be defined, names are saved so that their re-occurrence can be recognized as denoting a parameter rather than simply the current argument. While the procedure definition is in progress, appropriate changes are made in the syntax trees, which are then saved on disc. As an example, suppose the user types

"Define a procedure to zap z into w."
"Double w."
"Add z to w."
"Negate z."
"End."

At this point, there will be four new files containing parse trees which can be informally represented as follows.

filename	*contents*
Zap.0001	double param-2
Zap.0002	add param-1 to param-2
Zap.0003	negate param-1
Zap.0004	end

This enables flow-of-control semantics processing, when an invocation of the new "zap...into" imperative is detected, to evaluate the arguments and substitute them appropriately into the procedure's syntax trees wherever param-i is present.

Syntax for user-created imperatives parallels that for the system-provided routines of corresponding type. For instance, the same type of verbicle/particle compatibility checking (where applicable) takes place. Thus some acceptable inputs are

"Zap row 3 into row 6."
"Zap into column 4 the second column."

and some intentionally rejected inputs are

"Zap row 5." [missing operand]
"Zap row 5 from row 6." [wrong verbicle]

8. System Behavior

The sentence processing capabilities of the system will be indicated in this section by demonstrating its ability to handle paraphrases and by describing an experiment in which it was used by paid subjects to solve problems.

8.1 Syntactic Breadth

To demonstrate the variety of the syntax handled by the system, fifty-five paraphrases are given below for the sentence

"Double the first row in matrix 1."

These paraphrases are not all exact in that some of them omit reference to matrix 1, assuming context makes it clear, others entail side effects, such as the creation of a label, etc. This set gives only a small fraction of all the possible paraphrases that can be processed, but it is representative. The typical time required for complete processing of each sentence is two seconds on the PDP-11/70.

The first set of paraphrases demonstrates some variations on the qualifier.

1. "Double the first row of matrix 1."
2. "Double the first row which is in matrix 1."
3. "Double the first row that appears in matrix 1."
4. "Double the first row that matrix 1 contains."
5. "Double the first row matrix 1 contains."

The matrix reference can also appear as a classifier

6. "Double the first matrix 1 row."

or if context indicates the matrix reference, it can be omitted.

7. "Double the first row."
8. "Double row 1."

A row may be referred to by the values it contains.

9. "Double the first row that contains a positive or a nonpositive number."
10. "Double the row that contains the first entry of matrix 1."
11. "Double the row in which the first entry of matrix 1 appears."
12. "Double the first row in which there is a positive or a nonpositive number."

13. "Double the row containing the first entry
of column 1."
14. "Consider column 2.
Consider the first entry in it.
Double the row which contains that entry."

Rows may be thought of as sets of entries.

15. "Double the entries of row 1."
16. "Double the elements in row 1."
17. "Double the row 1 entries."
18. "Double the row 1 numbers."

The next several sentences illustrate some quantifiers.

19. "Double all the entries of row 1."
20. "Double each entry in row 1."
21. "Double every entry in row 1."
22. "Double each one of the entries in row 1."

Assume the row has 5 members.

23. "Double the first five entries of matrix 1."

Some rows may be located positionally.

24. "Double the top row."

Suppose there are four rows in the matrix. The first row can be found by counting up from the bottom.

25. "Double the fourth row from the bottom."
26. "Double the fourth from the bottom row."
27. "Double the fourth from bottom row."
28. "Double the fourth from the last row."

Generality of ordinal processing allows for some rather strange sentences.

29. "Double the first one row."
30. "Double the first row from the top."

Row 1 can be located with respect to other rows.

31. "Double the row in matrix 1 corresponding to
row 1 in matrix 2."
32. "Double the row in matrix 1 which corresponds
to row 1 of matrix 2."

One can use multiple clauses by labelling or focusing attention in one clause and then using it in the second clause.

33. "Consider row 1 and double it."
34. "Consider row 1. Double it."
35. "Consider row 1. Double that row."
36. "Consider and double row 1."
37. "Consider row 1.
Double the row considered by the
last command."
38. "Consider row 1.
Double the row the last command considered."
39. "Consider matrix 1 and double its first row."
40. "Consider rows 2, 3 and 4.
Double the other row."
41. "Consider row 1 of matrix 2.
Double the row in matrix 1 corresponding to it."

Users may access entities by naming them.

42. "Call row 1 x. Double x."
43. "Call row 1 x. Double row x."
44. "Call row 1 x. Double it."
45. "Call row 1 x. Double the x row."
46. "Call the first entry x. Double the x row."

The "backup" command will undo the calculation of previous commands.

47. "Double row 1. Clear it. Backup."

Other imperatives can be used to achieve the result of "double".

48. "Add row 1 to itself."
49. "Add row 1 to row 1."
50. "Multiply row 1 by 2."
51. "Divide row 1 by 0.5."
52. "Divide 0.5 into the first row."
53. "Add the entries in row 1 to themselves."

Finally, noun result groups may be used.

54. "Put the product of 2 and row 1 into row 1."
55. "Subtract the negative of row 1 from
that row."

There are, of course, many paraphrases which are not currently recognized by NLC. Some examples include sentences with superfluous words or phrases:

1. "PLEASE double row 1."
2. "Double the VERY first row of matrix 1."
3. "Double the first BUT NOT THE
SECOND row."

certain unimplemented noun-result formats:

4. "Put twice row 1 into row 1."
5. "Put row 1 times 2 into row 1."

and verbs taking more than 2 operands:

6. "Add row 1 to itself, putting the result
into row 1."

8.2 Some Observations of Performance

In April of 1979, twenty-three students in a first course in programming at Duke were paid to be subjects in an experiment on the system. Each subject was left alone in a room with the display terminal and given a short tutorial to read, a few simple practice exercises to do, and a problem to solve. No verbal interactions were allowed between experimenter and subject except those related to the administration of the test. Typical times required to complete the tutorial, the exercises, and the problem were 35, 15, and 50 minutes, respectively. In the problem solving sessions, the subjects typed a total of 1581 sentences, 81 percent of which were processed immediately and correctly. Approximately half of the remaining 19 percent were rejected because of system inadequacies, and the other half were rejected because of errors in user inputs. This experiment is described in [5] by Biermann, Ballard, and Holler, with an analysis of the types of errors that were made. Also included in the experiment was a test of the subjects' ability to do the same problems in the programming language from their course, PL/C. These results are discussed in [5], too.

Some specific observations that have come out of the experiment and other usages of the system are as follows:

1. The vocabulary of over 300 words is nearly adequate for a reasonable class of problems. Only eight words were typed during the experiment which were not available in the system. However, any casual user who attempts to push the system capabilities significantly will quickly find many unimplemented words.
2. Some of the implemented words have inadequate definitions. For example, NLC will process "the entry corresponding to x" but not "the corresponding entry". The latter form is more difficult because the item to be corresponded to is not explicit.
3. The variety of the syntactic structures which are processed is approximately as good as indicated by the experiment: About 70 to 90 percent of a typical user's inputs will be handled by the parser.
4. The error messages for the system are inadequate.
5. The processor for quantification needs to be redesigned. We notice for example that NLC processes "Double EACH entry in the first column" but not "Double the first entry in EACH column." In the former case, the first column is found and the matrix computer doubles its entries in one operation. In the later case, the definitions of "entry", "first", "the", and "double" must be invoked in that order for every column.

9. Comparison With Other Work

A number of projects in automatic programming propose to have a user converse about a problem in a completely natural manner, using problem dependent vocabulary, and largely omitting discussion of data structures and coding details. Examples of such work have been described by Balzer[2,3], Biermann[4], Green[13,14], Heidorn[16,17,18,19], and Martin et al.[23]. Inputs to these systems typically include a collection of fragments about the program to be generated, in which case the system must perform considerable induction and synthesis to produce an output. While the long term goals of the NLC project are similar to those of these other projects, the method of research is somewhat different. Whereas many projects attempt to tackle problems associated with several levels of cognition at once, NLC attempts to begin with a reliable sentence-by-sentence processor and to add facilities slowly while reducing demands on the user.

Many of the research efforts in natural language processing have been associated with information retrieval from data base systems (Codd[9], Harris[15], Hendrix et al.[20,21], Petrick[25], Plath[26], Simmons[31], Thompson & Thompson[34], Waltz[35,36], and Woods[39]). Most of the inputs for these systems are questions or simple imperatives such as "list all of the ..." Top level sentence syntax for these systems may have more variety than NLC. At the noun group level, however, NLC appears to have more extensive capabilities. This is due to the need in the NLC environment to conveniently refer to objects or sets of objects on the basis of their properties, geometrical location, operations performed upon them, etc.

Concerning world modelling, a system which bears some resemblance to NLC is SOPHIE by Brown and Burton[8]. Their system allows natural language interaction with a simulator for an electric circuit.

In the artificial intelligence literature, there is much emphasis on (1) artificial cognitive abilities, (2) induction mechanisms, (3) problem solving facilities, and (4) mechanisms for dealing with context and sequence. Future work on NLC will move in the direction of adding such facilities, but in its current state the system works more like an interpreter for English in the style of programming language interpreters than like a "thinking" machine. Thus the mechanisms described in Bobrow & Collins[7], Cullingford[10], Minsky[24], Schank[28,29,30], Winograd[37], and others for various kinds of cognition and problem solving are, for the time being, largely without counterpart in NLC. The philosophy of this project has been to build from the bottom, attempting to solve the least difficult, though still challenging, problems first.

10. Conclusion

Natural language programming has seemed in recent years to be a rather remote possibility because of the slow progress in representation theory, inference theory, and computational linguistics. The NLC system is designed to compensate partially for the weakness of current technology in these areas by presenting the user with a good environment and with some well-designed linguistic facilities. All of the quoted phrases and sentences in this paper and the Appendix have been run on the system except for the "if" constructions in Section 7. Current efforts are aimed at the development of a number of flow-of-control semantics facilities for handling various types of control structures and definitions of new vocabulary items.

Appendix: A Natural Language Program and Its PL/I Equivalent

The following "pivot" routine uses a computational technique described in Gallie and Ramm[12] and gives an example of a nontrivial usage of the system.

```
"Display a 4 by 5 matrix and call it testmat."
"Fill the matrix with random values."
"Choose an entry and call it p."
"Define a method to pivot testmat about p."
  "Choose an entry not in the p row and not in
    the p column and call it q."
  "Compute the product of the entry which cor-
    responds to q in the p row and the entry
    which corresponds to q in the p column."
  "Divide the result by p and subtract this result
    from q."
  "Repeat for all other entries not in the p row
    and not in the p column."
  "Divide each entry except p in the p row by p
    and negate those entries."
  "Divide each entry except p in the p column
    by p."
  "Put the reciprocal of p into p."
"End the definition."
```

The PL/I equivalent program as given in [12] is as follows:

```
EXCHANGE:
  PROCEDURE(MATRIX,PIVROW,PIVCOL);
    DECLARE (MATRIX(*,*),PIVOT) FLOAT,
      (PIVROW,PIVCOL,ROWS,COLMNS,I,J)
       FIXED BINARY;
    /* DETERMINE NUMBER OF ROWS
        AND COLUMNS */
    ROWS = HBOUND(MATRIX,1);
    COLMNS = HBOUND(MATRIX,2);
    /* NAME THE PIVOT ELEMENT */
    PIVOT = MATRIX(PIVROW,PIVCOL);
    /* APPLY THE "RECTANGLE RULE" */
   DO I = 1 to PIVROW-1,
    PIVROW+1 TO ROWS;
      DO J = 1 TO PIVCOL-1,
       PIVCOL+1 TO COLMNS;
         MATRIX(I,J) = MATRIX(I,J)
           - MATRIX(I,PIVCOL) *
           MATRIX(PIVROW,J) / PIVOT;
      END;
    END;
    /* CHANGE THE OLD PIVOT ROW */
    DO J = 1 TO PIVCOL-1,
     PIVCOL+1 TO COLMNS;
       MATRIX(PIVROW,J) =
       - MATRIX(PIVROW,J) / PIVOT;
    END;
    /* CHANGE THE OLD PIVOT COLUMN */
    DO I = 1 TO PIVROW-1,
     PIVROW+1 TO ROWS;
       MATRIX(I,PIVCOL) =
       MATRIX(I,PIVCOL) / PIVOT;
    END;
    /* CHANGE THE PIVOT */
    MATRIX(PIVROW,PIVCOL) = 1 / PIVOT;
END EXCHANGE;
```

Acknowledgement

Miss Anne Holler has produced several sections of code in the NLC system, including the spelling corrector, the syntax networks for the "if" and "repeat" clauses, semantics and matrix computer routines for noun result groups, and all the terminal display routines. We are grateful to George Heidorn and the referees for extensive help in improving and shortening the original 87 page report.

References

1. Ballard, B.W. *Semantic Processing For A Natural Language Programming System* (Ph.D. Dissertation), Report CS-1979-5, Duke University, Durham, North Carolina, May, 1979.
2. Balzer, R.M. "A Global View Of Automatic Programming", Proc. 3rd Joint Conference On Artificial Intelligence, August, 1973, pp. 494-499.
3. Balzer, R.M. "Imprecise Program Specification", Proc. Consiglio Nazl. Ric. Ist. Elaborazione Inf., 1975.
4. Biermann, A. "Approaches To Automatic Programming", in *Advances in Computers,* Volume 15, Academic Press, New York, 1976, pp. 1-63.
5. Biermann A., Ballard, B., and Holler, A. "An Experimental Study Of Natural Language Programming," Report CS-1979-9, Duke University, Durham, North Carolina, July, 1979.
6. Biermann, A. and Krishnaswamy, R. "Constructing Programs From Example Computations", *IEEE Transactions on Software Engineering,* September, 1976, pp. 141-153.

7. Bobrow, D.G. and Collins, A. Ed., *Representation and Understanding,* Academic Press, New York, 1975.

8. Brown, J.S. and Burton, R.R. "Multiple Representations Of Knowledge For Tutorial Reasoning", in *Representation and Understanding* (Bobrow, D.G. and Collins, A., Eds.), Academic Press, New York, 1975, pp. 311 - 349.

9. Codd, E.F. "Seven Steps to RENDEZVOUS With The Casual User", IBM Report J1333 (#20842), January 17, 1974. Presented at the IFIP-TC2 Conference on Data Base Management, Cargese, Corsica, April 1-5, 1974.

10. Cullingford, R.E. *Script Application: Computer Understanding of Newspaper Stories* (Ph.D. Dissertation). Research Report #116, Yale University, 1978.

11. Dijkstra, E.W. "On The Foolishness Of 'Natural Language Programming'". Unpublished report, 1978.

12. Gallie, T. and Ramm, D. *Computer Science/I: An Introduction To Structured Programming.* Kendall/Hunt, Dubuque, Iowa, 1976.

13. Green, C. "A Summary Of The PSI Program Synthesis System", Proceedings Of 5th International Conference on Artificial Intelligence, Volume I, August 1977, pp. 380 - 381.

14. Green, C.C., Waldinger, R.J., Barstow, D.R., Elschlager, R., Lenat, D.B., McCune, B.P., Shaw, D.E., and Steinberg, L.I. "Progress Report On Program-Understanding Systems", Memo AIM-240, Stanford Artificial Intelligence Laboratory, Stanford, California.

15. Harris, L.R. "Status Report On ROBOT NL Query Processor", SIGART Newsletter, August, 1978, pp. 3-4

16. Heidorn, George E. "Augmented Phrase Structure Grammars", IBM Thomas J. Watson Research Center, Yorktown Heights, New York, December, 1975.

17. Heidorn, George E. "Automatic Programming Through Natural Language Dialogue: A Survey", IBM J. Res. Develop., July, 1976, pp. 302-313.

18. Heidorn, George E. *Natural Language Inputs To A Simulation Programming System.* Naval Postgraduate School, October, 1972.

19. Heidorn, George E. "Supporting A Computer-Directed Natural Language Dialogue For Automatic Business Programming", Research Report 26157, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, June, 1976.

20. Hendrix, Gary G., Sacerdoti, Earl D., Sagalowicz, Daniel, and Slocum, Jonathan "Developing A Natural Language Interface To Complex Data", ACM Trans. on Database Systems, June, 1978, pp. 105-147.

21. Hendrix, Gary G. "Human Engineering For Applied Natural Language Processing", Proceedings Of 5th International Conference on Artificial Intelligence, Volume I, August, 1977, pp. 183-191.

22. Hobbs, Jerry R. "Pronoun Resolution", Research Report 76-1, Department of Computer Sciences, City College of New York, August, 1976.

23. Martin, W.A., Ginzberg, M.J., Krumland, R., Mark, B., Morgenstern, M., Niamir, B., and Sunguroff, A. Internal Memos, Automatic Programming Group, MIT, Cambridge, Massachusetts, 1974.

24. Minsky, M. "A Framework For Representing Knowledge", in *The Psychology Of Computer Vision,* Winston, P.H. ed., McGraw-Hill, New York, 1975.

25. Petrick, S.R. "On Natural Language Based Computer Systems", IBM J. Res. Develop., July, 1976, pp. 314-325. Also appears in *Linguistic Structures Processing,* A. Zampolli, ed., North-Holland Publishing Company, Amsterdam, Holland, 1977.

26. Plath, W.J. "REQUEST: A Natural Language Question-Answering System", IBM J. Res. Develop., July, 1976, pp. 326-335.

27. Sammet, J.E. "The Use of English As A Programming Language", *Comm. ACM,* March, 1966, pp. 228-229.

28. Schank, R.C. "Identification of Conceptualizations Underlying Natural Language", in *Computer Models Of Thought And Language,* R.C. Schank, R.C. and K.M. Colby, Eds., W.H. Freeman and Company, San Francisco, 1973, pp. 187-247.

29. Schank, R. and Abelson, R. *Scripts, Plans, Goals, And Understanding.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

30. Schank, R.C. and Colby, K.M. *Computer Models of Thought And Language.* W.H. Freeman and Company, San Francisco, 1973.

31. Simmons, R.F. "Natural Language Question Answering Systems: 1969", *Comm. ACM,* January, 1970, pp. 15-30.

32. Simmons, R.F. Personal Communication at TINLAP-2 Conference, Univ. of Illinois, July, 1978.

33. Stockwell, R. Schachter, P., and Partee, B. *The Major Syntactic Structures Of English.* Holt, Rinehart and Winston, Inc., New York, 1973, pp. 294-418.

34. Thompson, Frederick B. and Thompson, Bozena H. "Practical Natural Language Processing: The REL System as Prototype", in *Advances In Computers,* Volume 13, M. Rubinoff and M.C. Yovits, Eds., Academic Press, New York, 1975, pp. 109-168.

35. Waltz, D.L. "An English Language Question Answering System For A Large Relational Database", *Comm. ACM,* July, 1978, pp. 526-539.

36. Waltz, D.L., ed. "Natural Language Interfaces", SIGART Newsletter, February, 1977.

37. Winograd, T. *Understanding Natural Language,* Academic Press, New York, 1972.

38. Woods, W.A. "A Personal View Of Natural Language Understanding", in "Natural Language Interfaces", *SIGART Newsletter,* February, 1977, pp. 17-20.

39. Woods, W.A., Kaplan, R.M., and Nash-Weber, B. *The Lunar Sciences Natural Language Information System: Final Report.* Report Number 2378, Bolt, Beranek and Newman, Inc., Cambridge, Massachusetts, 1972.

40. Woods, W.A. "Transition Network Grammars For Natural Language Analysis", *Comm. ACM,* October, 1970, pp. 591-606.

Alan W. Biermann is an associate professor in the Department of Computer Science at Duke University. He received the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley in 1968.

Bruce W. Ballard is an assistant professor in the Department of Computer and Information Science at The Ohio State University. He received the Ph.D. degree in computer science from Duke University in 1979.