On the nature of natural language programming: generating transformation rules

Shelia Guberman
piXlogic, LLC
PO Box 2411, Cupertino, CA, 95015
sguberman@pixlogic.com

Leonid Kuznetsov
Parascript, LLC
PO Box 0917, Mountain View, CA, 94043
lkuzn@parascript.com

Wita Wojtkowski
BSU
1910 University Dr.
wwojtkow@boisestate.edu

Abstract

In this paper we address the basic difference between the text written in a natural language and the text written in a programming language, the "natural programming language." We ponder the rules of transforming a human-directed text written in a natural programming language into a computer-directed text ready for the execution by a computer. We offer examples for the use of such rules and attempt to arrive at some generalizations.

Key words: natural language programming, transition rules, executable programs

Résumé

On examine les différences fondamentales entre texte en langage naturel et texte écrit dans un langage de programmation, un "langage de programmation naturel". On considère les règles de transformation permettant de passer d'un texte écrit dans un langage de ce dernier type, à un texte prêt à être exécuté par un ordinateur. On présente des exemples d'utilisation de ces règles, et on tente de les généraliser.

Mots clés : langage de programmation naturel, règles de transformation, programme exécutable.

Introduction

From the very beginning of the computer era the problem of creating a *natural* programming language arouse. The problem was and still is addressed by many: computational linguists, computer scientists as well as the language experts [Kronrod, 1968; Woods, 1970; Winograd, 1972; Aravind, 1974; Naur, 1975; Kaplan; 1978; Hooper, 1981; Knuth, 1984; Buckman and Edwards, 1999; Gildea, 2002; Popescu et al., 2003; Beuthe, 2003; Esik and Fulop, 2003). The problem stems from the difference between the natural language and programming languages developed so far (Beuthe, 2003). By a natural language we understand any human *spoken* and *written* language. By a programming language we understand any programming language developed to *program* computers.

Our purpose is to address the basic difference between the text written in a natural language and the text written in a programming language. The approach we take is essentially systemic: we treat both, the natural language and the programming language as systems. In keeping with systems thinking we ponder them in a holistic way. Thus we aim to gain the insights into the whole by understanding the linkages, interactions and processes between the elements that comprise the whole system. Since the essence of a given system is defined by parts and the structure of the system (the relations between the parts and the whole) we attempt to show that the chief difference between the text written in a natural language and the programming language is the difference in structure, that is, the difference in the rules, which express the interpretation of the parts depending on the whole. The ultimate goal is to transform the structure of a text written in a natural language to the structure of a text written in a programming language.

We agree with Beuthe (Beuthe, 2003) that, at the most basic level programming is the creation of specifications with zero ambiguities for how to perform a task. The requirement of no ambiguities is why the source code from the most useful programming languages resembles a mathematical proof more than specifications written in natural language. We need to build new layers of expression on top of the layers that already exist, so that we can work towards natural language programming (Beuthe, 2003).[i]. To that end, in this paper we introduce the notion of transformation rules, the rules of transforming a human-directed text written in a natural programming language into a computer-directed text intended for the execution by a computer. We offer examples for the use of such rules and attempt to arrive at some generalizations.

Brief on the literature review

As noted in the Introduction, there exists rich literature dedicated to this issue [Miller, 1981, Knuth, 1984, Esik, 2003; Baeten, 2003]. Majority of researchers writing on this subject emphasize the obvious: that the natural language has a much richer vocabulary, more complicated syntax, and is highly context dependent. In our opinion, the deepest analysis of that kind was done by Miller [Miller, 1981]. Most insightful treatment of the problem was also attempted by D. E. Knuth - the author of a number of outstanding books concerning programming. In 1984 Knuth published a paper entitled "Literature Programming" [Knuth, 1984]. In it he made clear that the main difficulties he experienced while programming was the "unnatural" sequence of operators he is forced to write when developing a program. We posit that to deal with such an issue, and to ultimately develop a "natural" programming language, one has to understand the difference between any natural and programming language.

The nature of the problem

Many programmers also noted this problem: the order of writing operators on paper (when developing an algorithm) is almost opposite to the order these operators may appear in the executable program. It turns out that experienced programmers know this fact [Esik et al, 2003]. As an example, let us examine compiling of the subroutine that adds two matrices. (When writing a computer program for this subroutine we use an abstract programming language, FORTRAN).

Let us first examine the natural sequence of instructions (building an algorithm) for such a calculation. This would be as follows:

   1. First add the first element of the first matrix A(1,1) to the first element of the second matrix B(1,1). Their sum is the first element of the matrix in question, C(1,1).

       C(1,1)=A(1,1)+B(1,1)

  2. Repeat this procedure for each of the N elements of the first row of the matrices.

```
      DO 10 I =1,N
      C(i,1)=A(i,1)+B(i,1)
  10  CONTINUE
```

  3. Do the same with each of M rows of the matrices.

```
       DO 20 j =1,M
        DO 10 i =1,N
        C(i,j)=A(i,j)+B(i,j)
   10   CONTINUE
   20   CONTINUE
```

   4. Next set dimensions of the matrices, i.e. to specify the operator

       DIMENSION A(N,M), B(N,M), C(N,M)

5. Now establish the data type for the matrices. Let us declare this to be integer data type. Thus we will write

       INTEGER A, B, C

  6. In this step name our subroutine and define the required information before the subroutine runs.

       SUBROUTINE MATR(A,B,C,N,M)

To execute our example on a computer, however, we need to rearrange the order of the operators. According to the rules of FORTRAN, for example, this has to be rearranged as follows: to keep the formal-syntactic rules of the language, we will place the RETURN operator at the end of the subroutine; the order of the appearance of each operator is now:

```
  8:        SUBROUTINE MATR(A,B,C,N,M)
  7:        INTEGER A,B,C
  6:        DIMENSION A(N,M), B(N,M), C(N,M)
```

```
4:       DO 20 J=1,M
2:       DO 10 I=1,N
1:         C(i,j)=A(i,j)+B(i,j)
3:    10  CONTINUE
5:    20  CONTINUE
9:       RETURN
```

The number placed on the left side of each line shows the order in which a given operator appears in the program. It should be noted that this order reveals the difficulty mentioned before. Moreover, it is also valuable to note the placement of the DO-CONTINUE repetitive statements. These operators appear after the statement's body (in our example they are declared after C=A+B operator). The situation is not really exceptional- it is common when writing (note: writing, but not formally and correctly putting together) programs. We thus continue the examination of the example to illustrate this problem.

We note that the subroutine described above is frequently included in some context, and called out from another program. Therefore, it may happen that parameters sent to it may be incorrect. For example N and M must be positive, i.e. a condition of usage should thus be stated.

```
10:       IF ((M.LE.0).OR.(N.LE.0)) GO TO 30
```

This command has to be inserted between 6-th and 4-th operators. Operators identifying an error after RETURN (at the end of the program) are:

```
11:    30  PRINT 31
12:    31  FORMAT ('INCORRECT PARAMETERS')
13:       STOP
```

Hence the conditional statement that checks for correctness is written at the end.

We point out that both versions of the instructions to add two matrices include exactly the same set of operators. The only difference is the *sequence* in which they are recorded. Whether one order or another would be chosen would depend on what was our objective when specifying these instructions. The instructions may be aimed at *understanding* them or at *executing* them.

On the structure: understanding versus execution

In our example, structure means the order of the statements. Our initial manner of compiling the subroutine was intended to explain to a human, who would read it, how to reach the goal (to obtain the sum of two matrices). Therefore, the required operators we placed in a sequence that allows a reader to comprehend the mathematical procedure (an algorithm) for adding matrices, and when understood, this reader would be able to use it. We note that although we applied FORTRAN's syntax, the manner with which we structured the procedure was aimed at a human. However, in order to make this algorithm appropriate for a run on a computer, we were compelled to rearrange it. Thus the structure intended for a computer processing is no longer human-aimed but computer-aimed. In this context, we can clearly imagine the situation in which any programmer may find himself or herself. If he or she is given a programming task in a precise form (that is, he or she already has the complete algorithm), this programmer needs only

to translate such an algorithm into a structure that a machine can accept and act upon. But if there is a need to work out an algorithm *and* create the structure for the machine, then this programmer will labor in two mental 'planes': understand and construct the algorithm for solving the problem under consideration *and* organize it appropriately (for both, human-aimed and computer-aimed approaches). We posit that, as a result, an increased psychological burden may be placed on the programmer, even though he or she may not recognize this fact. According to Knuth this is an important point. He writes [Knuth, 1984]:

For better understanding, the programmer has to indicate the order in which the program was developed (as opposed to the strict "top-down" or "bottom-up" order)

To deal with a problem of the psychological burden acknowledged above, Knuth created software environment where a programmer can write the program in, what he refers to, a 'natural' way – the way a programmer creates an algorithm for a solution of a given problem. Let us thus try to understand what is at the core of such an approach: the structure, that is, the order of statements.

In the natural language, the sequence of statements is predicated by the need to assure the understanding by the person for which a given statements are intended. Thus the main requirement in creating a text, for example, in a natural language, is to ensure the UNDERSTANDING of any statement at the time the statement appears. On the other hand, the sequence of statements in a programming language is predicated by the needs of the computer. Therefore, the core requirement, in any program development, is to ensure the proper EXECUTION of a given statement, at the time it appears. That is why it is so difficult to understand a program written by someone else. Essentially, the problem is this: computer programs are not created for understanding but for execution.
Here we note that Knuth [Knuth, 1984] wrote the following:

I am convinced that programmers, in order to understand the program, are trying to reconstruct the order in which different parts of the program were written…

We believe that this is a remarkable statement. It calls attention to a very general feature of the human intellect [Popper, 1977]. Let us also point out that a similar idea was formulated in 1970 by M. Bongard, in a book entitled "Recognition Problems" [Bongard, 1970]. Specifically, Bongard's "imitation principle" states that the best recognition of any objects is achieved when the recognizer understands how the objects in question were created. The imitation principle was further developed by Guberman [Guberman, 1975] and ultimately led to the creation and the commercial use of the application system for handwriting recognition. The main idea for a solution to the handwriting recognition problem stems from the understanding, that in writing pattern recognition it is necessary to build an algorithm that essentially restores the way writing pattern was formed with a pen (more details concerning this pattern recognition approach can be found in Guberman and Andreewsky [Guberman and Andreewsky, 1996].

Revisiting the nature of the problem

Thus we gained an insight and now understand that neither the complexity of the grammatical rules, nor the number of terms, nor the context dependency are the main features that distinguish the natural language from any programming language, but the logic of the text construction.

Although Knuth recognized the problem, he did not resolve it. Therefore this question arises: Is it possible to develop software that will automatically transfer the sequence of statements written with the logic of natural language to the correctly executable program? This paper represents our attempt at understanding the answer to this question.

The basic approach is this: create a programming language with a grammar indicative of human logic of thinking (as applied to programming tasks) not to a computer's execution logic. Ideally, such a language could enable a programmer to write down his or her solution algorithm any desired way, with an important restriction that another programmer could compile the program for a computer, using the original programmer's description. Thus we obtain both, UNDERSTANDING` and EXECUTABILITY.

To achieve this goal, we again posit that it is necessary to analyze how the human being transforms the algorithm's description into a working program. Let us use an example. Figure 1 shows a geometric figure on a two dimensional plane (x, y). The problem is to find the convex part of that figure. One can do it using a ruler and a pencil: start from any point on the borderline; substitute every concave part of the contour by a straight line.
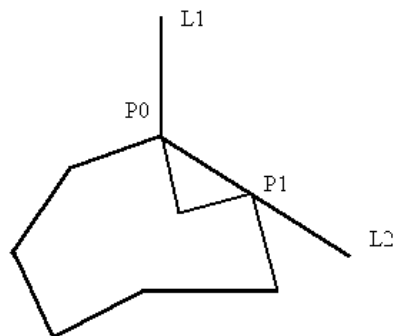

Figure 1 Geometric shape problem

For the computer that does not "see" the figure on the plane, steps taken to obtain the solution are a bit more complicated. These are:

1. Choose an upper point of the figure (point P0=(x0,y0) on the border). That point belongs to the concave closure.
2. Draw a ray L1 starting at this point (vertex) and going up. That ray will cross no points of the given figure.
3. Rotate the ray around the vertex (x0,y0) to the right until it will cross any other point on the border P1=(x1,y1). The part of the line (P0, P1) is a part of the concave closure.
4. Draw a ray from the point P1 in the same direction as the line (P0, P1) – L2. Obviously, it will be a part of the previous ray drawn from point P1 via point P2.
5. If capture initial point (P0) - that is the end, if not - go to 3 with a new ray L2 and new vertex P1.

Now let us describe the solution algorithm using programming language.

1. DIMENSION   XGR(N), YGR(N)- border of the given figure.
2. X0 =   , Y0 =   - starting point.
3. XOUT (0) = X0, YOUT (0) = Y0   - the starting
      point is the first point of the convex part.
4. DIMENSION XOUT(N), YOUT(N)   - keep the result.
5. CALL RAY (X0, Y0, FI , L)   - draw the ray L
                    from point (X0, Y0) with the slope FI
6. FI = 90
7. DO PSY = FI. 360   - rotate the ray clockwise
      CONTINUE.
8. CALL CROSS (L, XGR, YGR, TR, XCROSS, YCROSS)
                    -     does the ray L intersect the given curve {XGR, YGR)?
                    -     If "YES" then TR=TRUE, and point (XCROSS, YCROSS) is the intersection.
9. IF (TR) BREAK   - break the loop condition.
10. I = I + 1;   - loop-control variable
11. XOUT (I) = XCROSS,   YOUT (I) = YCROSS.
12. X0 = XCROSS,   Y0 = YCROSS   - new starting point.
13. FI=PSY
14. GO BACK TO 7   - go to find the next point.
15. IF (XOUT (I) . EQ. XOUT (0) . AND .
        YOUT (I) . EQ. YOUT (0))   RETURN
                          - break the loop condition - the end
16. X0 = XGR (J), Y0 = YGR (J)   - starting point
17. J = INDEX_MAX (YGR, N) – find the index of the upper point of the figure.
18. I = 0   - initialize the counter.

Next, the programmer has to develop a program that implements this algorithm on a computer. This programmer's activity involves a number of steps.

1. Collect all the data and variables, define their types and structures
2. Rearrange operators in order of execution by the computer
3. Include the operators neglected in the description of the algorithm (loop's frames, "if - then" operators, etc.) and add to the set of initial data the "hidden" variables (for example, loop counter)
4. Apply the standard subroutine frames: "subroutine xxx(yyy)" in the beginning and "end"
5. Check the developed program - not the algorithm - for bugs (wrong syntax, run out of array, lack of brackets, etc).
6. Test program using numerical examples in order to validate the correctness of the algorithm

We recall that the algorithm for calculating the convex space in our example was written as an explanation to another human being (more precisely - to a programmer) using formal constructs (operators) of a programming language. Is it possible to create a formal procedure that simulates programmer's activity described in steps 1 trough 6 above? That is, *is it possible to create a program that will understand an algorithm and successfully transform it into an executable program?*

First we have to apply Popper's principle [Popper, 1977] and construct a procedure that can prove (or disprove) that such a transformation is successful. Our problem now is this: how to prove that the procedure we develop really understands the algorithm? In our opinion, Turing's test [Turing, 1937] might be used here. For our situation the test might be constructed as follows: Let us consider two programmers, one performs a function of an analyst, the other of an engineer. The analyst writes the description of the algorithm in a style focused on understanding, utilizing operators of a programming language. The engineer receives this description, develops working

program, and sends it back to the analyst.[ii] [iii]Next, let us suppose that the appropriate program-translator is developed: the computer can transform algorithms into working computer programs.

If it is not possible to distinguish between the results produced by the human artifact (engineer's executable program, from the results produced by the computer artifact (algorithm's program-translator) Turing's test is completed. We also have confirmation that program -translator understood the algorithm (in the restricted sense of the ability to create an executable and correct working program).

In our formal analysis, if such an approach is possible, we can remove all comments communicating goals of the algorithm under consideration. Therefore, for our example, we now have a sequence of 18 operators (or groups of similar operators). The question we pose now is this: *Could we devise a set of rules that will transform these into a (correct) working program? Such transformation rules will comprise 'rules of understanding.'* Taking above into account let us now return to our two-dimensional convex curve example and consider each statement of the solution algorithm, one by one.

Statement # 1.   DIMENSION   XGR(N), YGR(N)   Q1

The dimension and the size of arrays are unknown. These might be defined later. To indicate that we are dealing with an unresolved question, we place Q1 symbol in the statement line.

Statement # 2.   X0 =   , Y0 =   Q2
The variables are not determined (Q2).

Statement # 3.   XOUT (0) = X0, YOUT (0) = Y0
The arrays are declared in the next statement.

Statement # 4.   DIMENSION XOUT(N), YOUT(N)   Q3
The size of the arrays has yet to be defined (Q3). Since this is a declaration, the operator has to be eventually shifted o the top lines of the solution algorithm.

Statement # 5.   CALL RAY (X0, Y0, FI , L)   Q4
The variables X0 and Y0 are defined in statement # 2. The variable L is the output. FI is not defined as yet (Q4).

Statement # 6.   FI=90
As a declaration it could be shifted to the top lines of the solution algorithm. Note that it also resolves the Q4.

Statement # 7.   DO PSY = FI, 360   Q5
               CONTINUE

Here the unresolved as yet question is this: which operators are included in the loop (Q5)? It should be noted that in contrast to the approach taken in programming languages, the loop's body is sited here, before loop declaration. Of course, operator RAY belongs to the body of the loop, because RAY depends on the loop's variable FI. If it turns out that X0 and Y0 depend on FI, then

statement # 2 is included in the loop's body as well. Thus, the rewritten the loop may be this:

      7. DO PSY = FI, 360   Q5
      5. CALL RAY (X0, Y0, FI , L)   Q4, Q6
      CONTINUE

Note that such a placement of the operator RAY creates unresolved as yet question (Q6). Q6 is this: why output L, calculated number of times, has no use as yet? We may encounter two possible answers:

      1) The loop's body is not complete; when complete it may include an operator that will use the value of L

      2) The loop has to execute iterative calculations (for example, summarizing series) and the useful result comes up after these iterative calculations are completed; it will be used in the operators that follow.

Statement # 8.   CALL CROSS (L, XGR, YGR, TR, XCROSS, YCROSS)   Q7

Because the operator CROSS uses the variable L, which is calculated inside the loop, the question Q7 arises. Q7 is this: Should CROSS operator be placed in the body of the loop, or should it use the final value of L, and thus do not be a part of the loop?

    Statement # 9.   IF(TR) BREAK
The use of the new variable TR defines the type of this variable –
The IF operator resolves both Q5 and Q6. It explicitly announces that it belongs to the loop (BREAK). Consequently this leads to the insertion the operator CROSS into the loop, since CROSS calculates the value of TR. That resolves Q7.

Now the loop is:

      7. DO PSY = FI, 360
      5. CALL RAY (X0, Y0, FI, L)   Q4,
      8. CALL CROSS (L, XGR, YGR, TR, XCROSS, YCROSS)
      9. IF(TR) BREAK
         CONTINUE

We note that the loop comprises three parts: loop's variable (one or more), body (executes calculations), exit (specifies condition for exiting the loop). In our example, loop's variable are FI, RAY; CALL is the body, IF the exit condition. (Note that another exit condition is FI=360).

    Statement #10.   I = I+1

This operator is a typical loop counter. That means that a loop with variable I exists. Because I was not initialized, Q7 arises.

    Statement # 11. No questions arise. All is OK.

Statement # 12. It is similar to the redefinition of the variables introduced in statement # 2.

Statement # 13. No questions arise. All is OK.

Statement # 14. Operator GO TO 7 defines a loop with variable I . Note that we have a loop with the variable I, loop's body (operators inside the loop), but no exit condition. Exit condition appears in next statement.

Statement # 15. This is the exit condition; the operator has to be moved to the loop. It has to be located after a variable Xout and Yout were calculated last, that is after the statement # 11.

Statement # 16.   X0 = XGR (J), Y0 = YGR (J)   Q8
It substitutes the operator from # 2. Thus it resolves question Q2, but generates question Q8: variable J is not as yet defined.

Statement # 17   J = INDEX_MAX (YGR, N)
Finds the index of the upper right point of the figure. If moved up, before statement #16 it will resolved Q8. Note that #17 and # 16 do not depend on any other variable and thus may be shifted to the top.

Statement # 18   I = 1
Shifted to the top it resolves Q7.

All transformations are completed and lines that can be shifted to the top are shifted. Transformed program will look as follows:

```
  1.   DIMENSION   XGR(N), YGR(N) - border of the given figure.
 4. DIMENSION XOUT(N), YOUT(N)
 17. J = INDEX_MAX (YGR, N)
  16. X0 = XGR ( J ), Y0 = YGR ( J )
   18. I = 0
 3. XOUT ( 0 ) = X0, YOUT ( 0 ) = Y0
 6. FI = 90
   3.   CALL RAY (X0, Y0, FI , L)
   4.   DO PSY = FI, 360
   5. CALL RAY (X0, Y0, FI, L)
 8. CALL CROSS (L, XGR, YGR, TR, XCROSS, YCROSS)
 9. IF(TR) BREAK
    CONTINUE
 10.  I = I + 1;
 11. XOUT ( I ) = XCROSS,    YOUT ( I ) = YCROSS.
 12. X0 = XCROSS,    Y0 = YCROSS
 13. FI=PSY
 15.  IF ( XOUT ( I ) . EQ. XOUT ( 0 ) . AND .
       YOUT ( I ) . EQ. YOUT ( 0 ) )        RETURN
 14.   GO BACK TO 7
```

Concerning the loops we note the following: the rules of loop building require that a given loop is build without interruptions. Specifically, in our example, after first loop is declared (statement # 4) next two statements (# 5 and # 6) are part of this particular loop as well. Statement # 7 positions a new loop, a sign that the previous one is completed. Statement # 15 belongs to this

loop, however, # 16 is a declaration, and is outside of this loop. This loop is completed. (In certain ambiguous situations such a rule helps in decision making.) We may also conform to the notation formalism of programming languages and do the following:

- substitute " = " by ".EQ." in IF operator,
- declare loop's variables: INTEGER I ,J ,N, FI ,
- create the formal frame of the subroutine: the name of the subroutine CLOSURE() (with a list of input and output variables) and the closing operator RETURN,
- find all variables, which are not initialized and place them as input parameters. In our example, these are XGR(N), YGR(N) – the border of the given figure, and N – the number of points in the border. The subroutine can be identified as CLOSURE(XGR, YGR, N).

At this point we are still not sure if the set of transformations we put to use so far is universal and may help resolve any problem. To that end let us attempt to extract from our example general rules of transformation. Thus we note the following: Each operator has to be a legal operator in the formal programming language, i.e. has to be locally legal. If it is not legal, this situation is made visible by generating a question Q_n. Such operator must be checked for being a globally legal as well. If it is globally illegal, it has to be marked by Q_n. Thus one operator can be marked by two or more questions. An example of locally illegal operator is the expression X0=

Following operators are locally legal:
$$A(I) = NOM$$
$$NOM = 1$$
However, first one is globally illegal since NOM is not defined. The problem can be resolved if we shift the second operator (NOM = 1) up in the sequence.

We can now generalize: after marking an operator with Q_n (unresolved as yet questions) we shift 'questioned' operator up. Such a shift continues until one of two things takes place: 1) an operator that may alter the value of any variable of the operator that is undergoing the shift is reached; or 2) the situation changes to the worst - a new and unresolved question arises.

We now can make another generalization: distinct operators as well as loops (blocks of operators) have to be analyzed and shifted according to the general rule identified above. We again note the following: To be legal, the loop has to contain the body, the variable of the loop, and the exit conditions. Thus when a legal loop is produced, it is open to modifications, i.e. certain operators that follow the loop can be included in the loop. The loop remains open until it is completed. However it continues to be open until requisite analysis reveals that operator that follows does not belong to this particular loop (as in the generalization described above). Under these conditions, the loop is decreed closed - no other operators can be included in the loop. We also note here that after a loop is closed it might be shifted as a block. Moreover, conditions for terminating the shift of a loop are the same as for the shift of a single operator. In summary, in Table 1, we put forth the purpose, operation and stop criteria for transformation rules identified and discussed above.

Purpose	Operation	Stop Criteria
'Questioned' operator	Shift up	a. Reaching an operator that may alter the value of any variable of the shifted operator b. Reaching location where new unresolved question arises
Loop	Build	Loop contains all components: loop's variable, exit condition and the body
Complete Loop	Shift up	Same as for the 'questioned' operator
Subroutine	Define input variables	All variables that are not initialized in the subroutine are listed as input parameters

Table 1

The purpose, operations, and the stop criteria for the transformation rules

Conclusion

In this paper we pondered the notion of the natural language programming and required transformations. We posit that the next step is to create a program that will automatically perform transformations we discussed, and to test this on a variety of *programs* and *programmers*. The objective of such a testing will be to discover the following: are the rules of transformation sufficient in all cases; can these be expanded; or is there no way to find a universal and limited set of transformation rules. Ultimately we would like to know if the notion of a natural language programming is just a dream and will remain only an unfulfilled dream.

References

- Aravind K. J, A Note on Partial Match of Descriptions: Can One Simultaneously Question (Retrieve) and Inform (Update)? *Proceedings of the Theoretical Issues in Natural Language Processing-2*, 184-186, 1978
- Beute B. Tools from the Garden Shack: Natural Programming Language http://www.artima.com/weblogs/viewpost.jsp?thread=4536.
- Baeten J., Lenstra J., Parrow J., Woeginger G. (Eds.), *Automata, Languages and Programming*, Lecture Notes in Computer Science 30th Int'l Colloquium, Springer, 2003
- Brooks, F.P., *The Mythical Man-Month: Essays in Software Engineering,* Addison-Wesley, Anniversary Edition, 1995
- Bruckman A. and Edwards E., Should We Leverage Natural Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language, *Proceedings of the Conference on Human Factors and Computing Systems*, Pittsburgh, Pennsylvania, 207-214, 1999
- Esik Z., Fulop Z. (Eds.), *Developments in Language Theory*, Lecture Notes in Computer Science, Springer, 2003
- Gildea D., Jurafsky, D., Automatic Labeling of Semantic Roles, *Computational Linguistics*, v. 28 No. 3, 245-288, 2002

- Grosz, B. J., Utterance and Objective: Issues in Natural Language Communication, *AI Magazine*, v. 1, No. 1, 11-20, 1980

- Grosz, B. J., Focusing and Description in Natural Language Dialogues, in *Elements of Discourse Understanding: Proceedings. Of a Workshop on Computational Aspects of Linguistic Structure and Discourse Setting,* (A.K. Joshi, I.A. Sag, and B.L. Webber, Eds.), Cambridge University Press, 1981

- Guberman S., Algorithm for the Recognition of the Handwritten Text, *Automation and Remote Control*, *MAIK* Nauka/Interperiodica Publishing, Moscow Kluwer Academic/Plenum Publishers*,* In English from *Automatika i Telemechanika*, No 5, 122-129, 1975

- Guberman S., and Andreewsky E., From Language Pathology to Automatic Language Processing, *Cybernetics and Human Knowing*, v. 3, No. 4, 41 – 53, 1996

- Hoopper, G.M., Keynote Address, in *History of Programming Languages*, (R.L. Wexelblat, Ed.), Academic Press, 7-20, 1981

- Kaplan S. J., Designing a Portable Natural Language Database Query System, *ACM Transactions on Database Systems* (TODS), v. 9 No.1, 1-19, 1984

- Keenan, E. L., Two Kinds of Presupposition in Natural Language, in *Studies in Linguistic Semantics* (C. J. Fillmore and D. T. Langendoen, Eds.), Holt, Rinehart, and Winston, 1971

- Knuth D.E., Literature Programming, *Computer Journal*, v. 27, No. 2, 97 – 111, 1984

- Kronrod A., *Etudes in Programming*, Nauka, Moscow, 1968

- Miller, L.A., Natural Language Programming: Styles, Strategies, and Constraints: *IBM Systems Journal*, v. 211, No.2, 184-215, 1981

- Naur, P., Programming Languages, Natural Languages, and Mathematics, *Proceedings of the 2nd Symposium on Principles of Programming Languages*, Palo Alto, California, 137-148, 1975

- Popescu A., Etzioni O., Kautz H., Towards a Theory of Natural Language Interfaces to Databases, Proceedings of the International Conference on Intelligent User Interfaces, Miami, Florida, 189-196, 2003

- Popper K. , The Logic of Scientific Discovery, Routledge, 1977

- Turing A M., On Computable Numbers, with an Application to the Entscheidungs problem, *Proceedings of the London Mathematical Society* v. 2, No. 42, 230-265, 1937

- Woods, W. A., Transition Network Grammars for Natural Language Analysis, *Communications of the ACM*, v. 13 No. 10, 591-606, 1970

- Wasserman A. I, Gutz, S., The Future of Programming, *Communications of the ACM,* v.25 No.3, 196-206, 1982

- Winograd, T., *Understanding Natural Language,* Academic Press, 1972

Endnotes

[i] Absence of needed layers of expression always causes difficulties when developing applications [Brooks, 1975] and when attempting to understand the application systems written by others [Miller, 1981, Knuth, 1984].

[ii] Let us also suppose that when repeating these tests, the analysts always works with a 'new' engineer, to avoid the issues of the 'problem of familiarity.'