CIS-TR-82-07

THE ANATOMY OF PROGRAMMING

Jozef Hvorecky
Department of Computer Science
Komensky University
842 15 Bratislava, Czechoslovakia

1. Introduction

The most important feature of school education is that it has been concentrated on general information which presents the main ideas and notions of its subjects. Such knowledge not cluttered by odd details can be easily used as a starting point on the way to some deeper particular knowledge.

In contradiction to that common fact, many current programming courses have usually been based on the recent state of hardware and software. Thus, they do not present the kind of knowledge apt for both future programmers and casual users, because they (willingly or unwillingly) show the given state as the best and/or the only possible one. On the other hand, some sources (for example, the set of books written by Ledgard et al.[8, 10, 11, 13]) indicate that there existe the knowledge which does not depend on any particular language or machine and which represents the core of programming.

An analogy with creative writing course can be offered: The creative writing is not the typing just as the programming is not the sitting at a terminal (though people tend to think so). In both cases the ideas of a performer are much more important than their actual writing. Therefore, the matter taught in the creative writing course is how to arrange those ideas and how to present them in a reasonable and legible manner. The completion of the course made nobody a writer But a good course can provide future writers with substantial knowledge about writing and can give general information about the profession to the others. Students write their own stories

in order to apply those abstract principles and to better understand them.

In this paper we present an outline of the introductory programming course. The ideas presented in the course are based on the belief that the core of a programmer's activity is the development and the implementation of algorithms.

The basic ideas of algorithm development had been known long before the first computer appeared. They have been successfully used by scientists and they do not change with the advancement of technology. The common use of computers has only made them more important. On the other hand, the fast development of both hardware and software makes the properties of future machines and programming languages difficult to predict. Thus, programming courses should afford knowledge about properties of algorithms, the algorithm development, and their connection to computers. Practical experience with work on computers (which is an unavoidable part of a programmer's skills) should be introduced in the form of a case study. In the actual course given by the author students start running their programs only when they have acquired the preliminary information as presented in Sections 2 - 5.

The course concentrates on the following issues:

a) Every problem may have several solutions.
b) Every proposed solution must be verified to see whether or not it really solves the problem.
c) The main differences among solutions are rooted in their computational complexity.

d) New solutions should be sought until one finds a solution with the "reasonable" complexity or the one that can not be improved.

The presentation starts with the general ideas of problem solving and deals with problems which need not be (and usually are not) solved by computers. Conversely, we will assume that the computation is to be performed by humans because in that case students are much more interested in improving their solutions. This means that the analogy between common human thinking and programming is utilized (see Fig. 1).

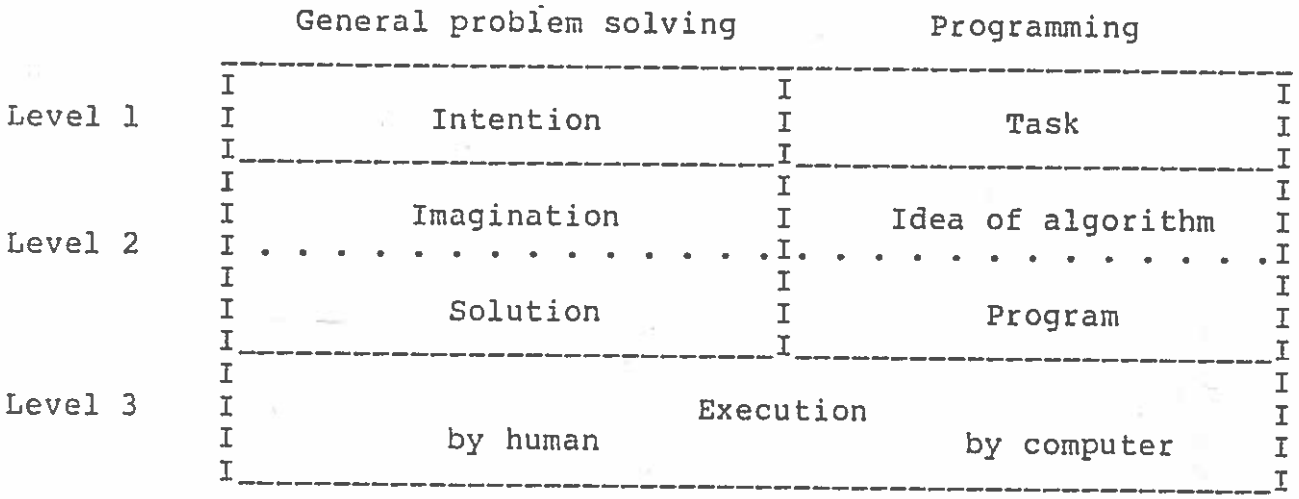

Fig. 1

The first two issues represent the relation between a task and a program satisfying it (i.e., between levels 1 and 2 on Fig.1). The relation is briefly explained in Section 2 of the paper. This section also shows a method by which a programming language can be introduced indirectly - as a tool for solving easy problems.

In Section 3 some ideas on the development of programs are presented, namely, the connection between the idea of an algorithm and its expression by a program. This section is an informal introduction to structured programming techniques.

The last two issues above address the connection between a program and its execution . They will be illustrated by the stepwise construction of the program for integer multiplication in Section 4. As it is shown there, during the construction many important notions of the theory of programming, of the computational complexity and of data structures can be introduced as results of solving advanced problems.

A brief evaluation of the proposed method of teaching programming is given in Section 5.

2. Programming language as a result of problem solving

In the presented course students become familiar with partial pieces of knowledge during problem solving in dialogue with their teacher. The teacher introduces problems, asks for their solutions, conducts dialogue and helps students, if necessary.

The execution of algorithms on computers is absolutely precise. To simulate this property, mathematical problems are solved, because they also require precise solutions.

The notion of an algorithm is introduced informally, as a prescription describing the process of the solution of a problem. To be understood the algorithm has to be written in some fixed form. This agreed upon form is called a programming language. The constructs of such a language must have the imperative form in order to change one given state of information to another. That is why they are called commands or statements.

The commands manipulate elementary objects - constants and variables. For the purpose of this paper it is sufficient to start with integer constants and integer variables, and two operations over them, addition and subtraction. Other operations and data types will be defined later in the moment when they are needed for solving some particular problem.

The existing programming languages are the results of long-term searches for an appropriate tool for problem solving. We simulate this development by a construction of a programming language from simple problems. Such an approach has another

advantage: It shows indirectly that commands of a programming language are correct tools for solving some problems, but solutions of other problems have to be expressed either by their combinations or by choosing another language more appropriate for that particular problem.

A problem is defined as a triple

$$\{ P \}\ X\ \{ Q \}$$

where P is used for a precondition (i.e., the property of data which is valid at the moment the problem is formulated),

Q means a postcondition (i.e., a desired relation which should be held after solving the problem),

X is an (unknown) algorithm with the following property: Every execution of X which starts with data satisfying the precondition P finishes with results satisfying the postcondition Q.

Let us solve the following simple problem:

$$\{ A < B \}\ S\ \{ A <= B \} \qquad (1)$$

which can be formulated in words as:

'Given two integers A and B for which relation A < B holds. Find the algorithm S such that after the execution of S, the relation "A is less or equal to B" will be valid.'

Because

$$(A < B) => (A <= B) \qquad (2)$$

it is easy to see that the postcondition holds for input values A and B, i.e., the problem can be solved by the command "do nothing".

This command which does not change the value of any variable is called the dummy command.

We are interested in finding as many solutions of (1) as possible. Therefore, students are asked if other solutions of the problem exist. Since

$$(A = B) \Rightarrow (A \leq B) \tag{3}$$

any algorithm guaranteeing the equality A = B after its execution solves the problem.

According to the precondition, A is not equal to B before S starts. Thus, the algorithm must change the value of at least one of those variables. The change must be brought about by an execution of a command. It implies that we must introduce the command which allows the changing of the value of a variable. That command is called the assignment statement.

Changing just the value of one variable, we obtain two different algorithms

$$A := B \tag{4}$$

and

$$B := A \tag{5}$$

We can also change the values of both variables to another value, say, five:

$$A := 5; B := 5$$

These two commands must not be separated, because none of them solves the problem by itself. To express this requirement we enclose them by special brackets "begin" and "end". The command which is a result of enclosing several command by these brackets is called a compound statement. Consequently,

```
begin A := 5; B := 5 end                                    (6)
```

is a solution of (1), too.

Now we can ask: Can the value of A be changed independently of B. (i.e., neither to B nor simultaneously with the change of B)?

The answer is yes. The value of A can be increased by 1, because the relation

```
(A < B) => (A + 1 <= B)
```

holds for integers. Consequently, after the assignment

```
A := A + 1                                                  (7)
```

the relation A <= B will be true. Thus, (7) is another solution of (1). This example illustrates the assignment statement with the same variable on both sides of the assignment sign.

If the relation A < B still holds after the execution of (7), the command can be repeated. The number of repetitions depends on the validity of the relation. It allows us to introduce the notion of a loop

```
while A < B do A := A + 1                                   (8)
```

Since the validity of A < B is implied directly from the precondition, the loop body is executed at least once. This means that it is reasonable to execute the loop body A := A + 1 first and then ask if the equality A = B was achieved.

The next loop

```
repeat A := A + 1 until A = B                               (9)
```

will be executed that way.

Even when there are other interesting solutions of (1), let us solve the reverse problem

$$\{ A <= B \} \quad T \quad \{ A < B \} \tag{10}$$

One can easily find its next solution

$$A := A - 1 \tag{11}$$

But there is no reason for subtraction, if the relation A < B holds for the initial values of A and B. Thus, the precondition can be partitioned into two cases:

$$(A < B) \text{ or } (A = B).$$

The first case can be solved using a dummy command and the second one by (11). The switch is made in the conditional statement

```
if  A = B  then  A := A-1  else  dummy        (12)
```

which can be also written in the shortened form

```
if  A = B  then A := A-1                      (13)
```

In this way we have introduced the main control structures of the programming language Pascal.

3. Programming techniques

The ideas of structured programming are aimed at the construction of large programs. However, their advantages can also be presented at the construction of small programs. In this section, we use those ideas to illustrate the connection between the idea of an algorithm and its expression by commands.

3.1 Direct programming

If the problem is simple or its solution is known, it can be programmed directly. To illustrate this notion, let us consider the following simple problem:

$$\begin{array}{c} \{\ A > B\ \} \\ \quad\cup \\ \{\ A < B\ \} \end{array} \qquad (14)$$

One can propose a direct solution: "Decrease the value of A by as much as it is less than B". From the many possible realizations we have chosen

$$A := B - 1 \qquad (15)$$

as an example.

3.2 Reformulation of a problem

If you are not able to solve a problem directly, you can try another technique - the reformulation. The reformulation does not solve the problem; it only produces another version of it (which ought to be more suitable for programming).

A reformulation of the problem (14) is

$$\begin{matrix}\{\ A > B\ \} \\ \quad U' \\ \{\ B > A\ \}\end{matrix} \tag{16}$$

which offers the new idea: "Exchange the original values of A and B". Thus, the problem

$$\begin{matrix}\{\ A = a,\ B = b\} \\ \quad U'' \\ \{\ A = b,\ B = a\}\end{matrix} \tag{17}$$

describes a partial subproblem of (14). This well-known problem has many possible solutions. The most common of them can also be used for the introduction of the notion of the auxiliary variable:

```
begin AUX := A;
        A := B;                    (18)
        B := AUX
end
```

Note that any solution of (17) also solves problem (14), but the opposite assertion is not necessarily true. The algorithm (15) does not solve (17).

3.3 The partitioning of a problem

The following problem

```
{ A > B }
    V                    (19)
{ A <= B }
```

will be used in this section as an illustration of the basic idea of structured programming - the partitioning of a problem.

This approach to solving a problem is based on splitting the problem into several subproblems which are "easier to solve" in some sense.For example, in the above problem (19) we notice that the precondition of (19) is the same as the precondition of (14). Because that problem has been solved, it is reasonable to propose its solution as the first part of a solution of problem (19). Thus, the problem can be reformulated to the following form

```
{ A > B }
begin U;
      { A < B }          (20)
      V'
end
{ A <= B }
```

where U is an algorithm solving (14) and
V' is an algorithm solving the rest of problem,

i.e., the subproblem

```
{ A < B }
    V'                   (21)
{ A <= B }
```

But the latter problem is identical with problem (1), i.e., the algorithm

$$V = \text{begin } U;\ S \text{ end} \tag{22}$$

where U is a solution of (14) and
S is a solution of (1)

solves the problem (19).

4. Systematic improvement of programs

In this section we present systematic stepwise improvement of programs as the main idea to be used in the construction of real programs. Most textbooks prefer to explain the idea by means of problems unknown to students (sorting, numerical analysis etc.) In such cases students are not able to distinguish between new pieces of information connected with this particular problem and information typical of programming itself.

For that reason, the problem of multiplication of non-negative integers has been chosen. The purpose of multiplication, its properties and the need for finding a fast algorithm are known to students. Thus, what is new is that it is an algorithmic view of the problem and its solution.

4.1 Fundamental algorithm

The problem of multiplication is formulated as follows:

```
{ X >= 0, Y >= 0}
      MULT                          (23)
{ PRODUCT = X * Y }
```

where X * Y is an abbreviation for

```
 Y + Y + Y + ... + Y
\___________________/
        X times
```

Our first task will be finding an arbitrary algorithm which solves (23). This means that the proof of the existence of the multiplication algorithm is required. In addition, the algorithm can be used as a basis for the subsequent construction of other algorithms and for the comparison of their effectiveness.

Assuming that we do not know of any multiplication algorithm, we can follow the natural idea:

It is easy to multiply Y by some small integer M, say, by

0, 1, 2, ...,

but not by an arbitrary value of X. Accordingly, let us split the process of multiplication into two parts: the first will realize the multiplication by M and the second one will expand the multiplication for X. It implies the requirement M <= X and the following reformulation of the problem:

```
{ X >= 0, Y >= 0 }
begin MULT';
      { PRODUCT = M * Y, M <= X }
      MULT''
end
{ PRODUCT = X * Y }
```
(24)

The first subproblem MULT' has a simple solution

```
begin M := 0; PRODUCT := 0 end
```
(25)

Then, we will reformulate the subproblem MULT'' as

```
{ PRODUCT = M * Y, M <= X }
      MULT''
{ PRODUCT = M * Y, M = X }
```
(26)

to obtain the similar structure of its pre- and post- conditions. In this form it is easier to see what should be done: To increase

the value of M in such a manner that the equality PRODUCT = M*Y holds. Thus, the solution has the form

```
while M <> X do MULT'''                                    (27)
```

where MULT''' is an algorithm increasing M and preserving the equality. The problem MULT''' can be formulated as follows

```
{ PRODUCT = M * Y, M < X }
          MULT'''                                          (28)
{ PRODUCT = M * Y, M <= X }
```

with the solution

```
begin M := M + 1;
      PRODUCT := PRODUCT + Y                               (29)
end
```

Completing the partial solutions we obtain the algorithm for multiplication by sequential addition

```
begin M := 0; PRODUCT := 0;
      while M <> X do begin M := M + 1;
                            PRODUCT := PRODUCT + Y         (30)
                      end
end
```

Note that the condition

```
{ PRODUCT = M * Y, M <= X }
```

holds after the execution of MULT' as well as MULT''', i.e., before the entrance to the loop and after every execution of its body. A condition with this property is called the loop invariant.

4.2 Modifications of the fundamental algorithm and the theory of programming

Algorithm (30) needs 2*X additions for one multiplication, i.e., it is slow even for small numbers. Our ambition will be to improve it.

To decrease the number of additions one may want to increase the increment by which the partial product within the loop is changed. The first modification of (30) will use the increment 2*Y:

```
begin M := 0; PRODUCT := 0; INCREMENT := Y + Y;                (31)
      while M <> X do begin M := M + 2;
                            PRODUCT := PRODUCT + INCREMENT
                      end
end
```

This algorithm computes the product for even X's faster than (30) does, but for odd X's its computation never terminates.

An algorithm that computes the right results for some initial values, but does not terminates for the others, is called the partially correct one.

Changing of the loop condition into

```
while M < X do ...
```

guarantees the termination, but not the right results for odd X's.

An algorithm that terminates its execution for any initial values satisfying the precondition is called the terminating algorithm.

A correct algorithm must be terminating and partially correct (in that case it terminates and gives correct results for any allowed initial values).

The following algorithm which contains "the correction" of the product obtained in the first loop is correct and approximately twice as fast as algorithm (30):

```
begin M := 0; PRODUCT := 0; INCREMENT := Y + Y;
      while M < X do begin M := M + 2;
                           PRODUCT := PRODUCT + INCREMENT
                     end;                                          (32)
      if M > X then begin M := M - 1;
                          PRODUCT := PRODUCT - Y
                    end
end
```

The point of the previous example is that a small change in the algorithm (say, a misprint) can considerably change algorithm's behavior, i.e., the behavior of algorithms is discontinuous.

4.3 Further improvement of the algorithm and its computational complexity

By similar means we are able to construct the algorithms with increments 3*Y, 4*Y, etc., respectively. The larger the constant N in N*Y the better performance of the algorithm for large numbers, but the worse performance for small numbers.

Consequently, we need the algorithm with smaller increments for small numbers and greater increments for large numbers, i.e., the algorithm with non-constant increments of product.

In the simplest case the increment will change with the arithmetic progression

$$Y, 2*Y, 3*Y, \ldots, N*Y, \ldots$$

The next algorithm computes the product in this way

```
begin M := 0; PRODUCT := 0;
      N := 1; INCREMENT := Y;

      while M < X do begin M := M + N;
                           PRODUCT := PRODUCT + INCREMENT;
                           N := N + 1;
                           INCREMENT := INCREMENT + Y
                      end;
                                                              (33)
      while M > X do begin M := M - 1;
                           PRODUCT := PRODUCT - Y
                      end
end
```

The evaluation of the first loop ends for the least value of N for which

$$1 + 2 + 3 + \ldots + N >= X.$$

The corresponding quadratic inequality gives an approximate solution

$$N = \mathrm{sqrt}(2*X).$$

The execution of the second loop is repeated N-1 times in the worst case and approximately N/2 times in the average case. Accordingly, the algorithm (33) needs

$$4*N + 2*(N-1) <= 6 * \mathrm{sqrt}(2*X) <= 9 * \mathrm{sqrt}(X)$$

additions and subtractions in the worst case and

$$4*N + 2*(N/2) <= 5 * \mathrm{sqrt}(2*X) <= 7.5 * \mathrm{sqrt}(X)$$

additions and subtractions in the average case.

Both of the above functions express the relationship between the size of input data and the time necessary for computation. The function with this property is called the time computational complexity.

During the correction phase (i.e., inside the second loop) just the value

$$Y \text{ or } 2*Y \text{ or } 3*Y \text{ or } \ldots \text{ or } (N-1)*Y$$

will be subtracted from PRODUCT, depending on the difference M-X after termination of the first loop. Each of them was computed during the first loop as a value of an increment. Consequently, if we store the values of increments in the array, the second loop can be replaced by the conditional statement

```
begin M := 0; PRODUCT := 0;
      N := 1; INCREMENT[1] := Y;
                                                          (34)
      while M < X do begin M := M + N;
                           PRODUCT := PRODUCT + INCREMENT[N];
                           N := N + 1;
                           INCREMENT[N] := INCREMENT[N-1] + Y
                     end;

     if M > X then PRODUCT := PRODUCT - INCREMENT[M-X]

end
```

The length of the array INCREMENT is to be sqrt(2*X).

The function which expresses the relationship between the size of input data and the necessary capacity of memory space is called the computation space complexity.

The algorithm (34) is interesting also for another reason. The corrections are supposed to be of the same duration. In fact, this is possible only if the time for finding any member of the array is the same. This assumption is true only for random access memory.

4.4 The advanced algorithms

Improvement of the present algorithms was achieved by using the arithmetic progression for computing increments. The use of the geometric progression in the next algorithm paves the way for obtaining further improvement.

```
begin M := 0; PRODUCT := 0;
      N := 1; INCREMENT := Y;

      while M < X do begin M := M + N;
                           PRODUCT := PRODUCT + INCREMENT;
                           N := N + N;
                           INCREMENT := INCREMENT + INCREMENT
                     end;

      while M > X do begin M := M - 1;
                           PRODUCT := PRODUCT - Y        (35)
                     end

end
```

Since the first loop ends when

$$1 + 2 + 4 + 8 + \ldots + 2^t >= X$$

(where t is approximately lg(X), the logarithm based 2 of X) the first loop in (35) is faster than the first loop in (33) and (34). Unfortunately, the last member of the product is

$$2^t * Y = 2^(\lg X) * X = X * Y$$

i.e., we can "overshoot" by as much as X-1. Therefore in the worst case the second loop simulates the fundamental algorithm (30) (from the opposite direction).

Thus, the radical change of the correction phase is needed. In fact we can also use geometric progression for the correction

of M. Obviously, there is a new possibility of shooting M over X (in the opposite direction, i.e., before X). That is why in the next algorithm M swings around X like a pendulum until it stops stops on the right value. The similarity of actions can also be observed from the form of the algorithm.

```
begin M := 0; PRODUCT := 0;
    while M <> X do
        begin N := 1; INCREMENT := Y;
              while M < X do
                        begin M := M + N;
                              PRODUCT := PRODUCT + INCREMENT;
                              N := N + N;
                              INCREMENT := INCREMENT + INCREMENT
                        end;                                        (36)
              N := -1; DECREMENT := -Y;
              while M > X do
                        begin M := M + N;
                              PRODUCT := PRODUCT + DECREMENT;
                              N := N + N;
                              DECREMENT := DECREMENT + DECREMENT
                        end
        end
end
```

The number of additions used for multiplication is a linear function of lg(X)^2. It implies that this algorithm is faster than any previous algorithm (except for a few small numbers).

But we need not allow M to overshoot X. Should M overshoot X, we will start the generation of the progression from 1 again. This is the main idea of the next algorithm.

```
begin M := 0; PRODUCT := 0;
      N := 1; INCREMENT := Y;

      while M <> X do
            if M + N <= X then
                  begin M := M + N;
                        PRODUCT := PRODUCT + INCREMENT;
                        N := N + N;
                        INCREMENT := INCREMENT + INCREMENT
                  end
                                                          (37)
               else

                  begin N := 1;
                        INCREMENT := Y
                  end
end
```

Not only is the notation of (37) shorter than the notation of (36), but the same relation is valid for their computation time. In algorithm (37) we do not waste time computing redundant arithmetic operations caused by overshooting. The improvement represents a small factor and the computational complexities of both previous algorithms are within the same range lg(X)^2.

We notice a drawback of the last algorithm. The computation uses a large number of small increments, but only a few large ones. Better usage of large increments could be a means of further improvement.

The values of large increments are computed through the smaller ones. Consequently, if we prefer the larger increments in the computation of the product, all the increments have to be evaluated before the computation of the product starts. During the computation we will always use the largest increment as many times as possible. Only if it could cause an overshooting, will we start adding the next smaller one.

This idea is presented by algorithm (38).

```
begin M := 0; PRODUCT := 0;
      i := 1;
      N[i] := 1; INCREMENT[i] := Y;

      while N[i] < X do
            begin i := i + 1;
                  N[i] := N[i-1] + N[i-1];
                  INCREMENT[i] := INCREMENT[i-1] + INCREMENT[i-1]
            end;

      while M <> X do                                          (38)
               begin
                   while (M + N[i]) <= X do
                          begin M := M + N[i];
                                PRODUCT := PRODUCT + INCREMENT[i]
                          end;
                    i := i - 1
               end

end
```

The first loop computes the increments and stores them (together with the information about the number of Y's in the i-th increment). The second loop computes the product in the manner discussed above.

It can be shown that in this particular case the inner loop (i.e., the one inside the second loop) is repeated at most once. Accordingly, it can be replaced by the conditional statement

```
if (M + N[i]) <= X then
         begin M := M + N[i];                                  (39)
               PRODUCT := PRODUCT + INCREMENT[i]
         end
```

This version of algorithm (38) is used in computers for the hardware integer multiplication. Surprisingly, this algorithm was not developed for computers. Its oldest form is written on Rhind's papyrus dated 1800 B.C.

5. Conclusion

The last example shows that the ideas presented here are very old and that a process similar to the one discussed had to be made at least four thousands years ago. But it is also the fact that many of these ideas had been forgotten and (for example) multiplication has been taught through drill.

The author likes the presented ideas because he prefers thinking to drill. Unfortunately, there are many teachers who prefer drill to thinking and start teaching programming with the syntax of a programming language or (even worse) explaining principles of computer construction. They should note one argument: Programming as a problem solving activity existed many years before computers (though it had no name) and it has been only reinforced by the technological advancement. On the other hand there are also many examples that show that technology has adopted ideas of programming.

A reader can also object that our method is too formal and, consequently, oriented to students with highly developed mathematical reasoning ability. In our opinion programming and mathematics are both formal disciplines. A reader who does not believe this is asked to read Section 4.2 once more. It shows that small changes in programs have big consequences (often financial ones, too).

Thus, all future programmers must be prepared to use formal reasoning. Of course, the level of formalization can be a

matter of discussion and it will probably differ depending on the student's age and specialization. From this point of view the last objection is not the issue. It is rather a call for the development of similar methods oriented to the other groups of students.

Fortunately, publications espousing this or similar approach have begun to appear. Some of them are mentioned in the lost of references.

6. References

1. H. Abelson, A. di Sessa: Turtle geometry, MIT Press, Cambridge, Ma., 1981

2. E. W. Dijkstra: A discipline of programming, Prentice-Hall, Englewood Cliffs, N. J., 1976

3. A. P. Ershow: Programming - the second literacy, 3rd World Conference Computers in Education, North Holland, Amsterdam, 1981

4. A. P. Ershow, G. A. Zvenigorodsky, Yu. A. Pervin: School informatics, Report 152, Computing Center of Siberian Division Academy of Sciences of the U.S.S.R, Novosibirsk, 1980 (in Russian)

5. D. Gries: What should we teach in an introductory programming course, Proceedings of 5th SIGCSE Symposium, Detroit, 1974

6. P. Grogono, S. H. Nelson: Problem solving and computer programming, Addison-Wesley, Reading, Ma., 1982

7. J. Hvorecky, J. Kelemen: Algoritmizacia - elementarny uvod, Alfa Publishers, Bratislava, 1983 (in Slovak)

8. L. J. Chmura, H. F. Ledgard: Cobol with style: Programming proverbs, Hayden book company, Rochelle Park, N.J., 1976

9. H. W. Lawson: Understanding computer systems, Computer Science Press, Rockville, Md., 1982

10. H. F. Ledgard, L. J. Chmura: Fortran with style: Programming proverbs, Hayden book company, Rochelle Park, N.J., 1978

11. H. F. Ledgard, P. Nagin, J. F. Hueras: Pascal with style: Programming proverbs, Hayden book company, Rochelle Park, N.J., 1979

12. H. F. Ledgard, A. Singer: Elementary Pascal, Random House, New York, 1982

13. P. Nagin, H. F. Ledgard: Basic with style: Programming proverbs, Hayden book company, Rochelle Park, N.J., 1978

14. S. Papert: Mindstorms, Basic Books, New York, 1980

15. R. Pattis: Karel the Robot, John Wiley, New York, 1981

16. K. W. Smillie: A service course in computing science presented from a historical point of view, SIGCSE Bulletin, Vol. 13, No. 2, June 1981

17. N. Wirth: Systematic programming. An introduction, Prentice-Hall, Englewood Cliffs, N. J., 1973

18. N. Wirth: Algorithms + Data structures = Programs, Prentice-Hall, Englewood Cliffs, N. J., 1976