Copyright Notice

My dear Friend or Relation, Master, Colleague or Pupil,

Paraphrasing the ominous sentence: "This ..... has been placed here for your convenience.", which is usually used to explain the presence of all sorts of American hotel room contraptions, I should like to say "The enclosed manuscript has been sent to you for your enjoyment.".

I would not dare to send it to you if you regarded it as the next item for the evergrowing pile of tasks still to be done. I know that the manuscript is long but I have let it grow that way in the hope that the intellectual effort needed for its digestion is inversionally proportional to its length. And your enjoyment may be proportional to it. So I don't apologize for its length. *)

There are no shattering discoveries in it: it is the kind of peaceful prose that I write (mainly for my own distraction?) when a somewhat poor condition forces me for some period of time to some sort of inactivity. It will certainly be less gloomy than this evening's front page news!

When you have read it and feel like dropping me a line, please don't hesitate to do so; I will receive it gladly.

Yours sincerely

Edsger W.Dijkstra
Department of Mathematics
Technological University Eindhoven
P.O. Box 513
EINDHOVEN
The Netherlands

*) There is no point in denying it: I do like Franz Schubert's music.

Stepwise Program Construction.

Over the past years I have been (heavily) engaged in a number of (at that time) advanced programming projects that could be considered as large in comparison to the available manpower. I am still in the active process of learning from the experience gained, one of the immediate goals of this learning process being the discovery of better ways to construct even "small programs" in a reliable fashion. Although large, advanced and sophisticated programming efforts are more spectacular, we must not forget that quite a lot of machine time and programmer's energy is really spent on small, down-to-earth projects and the present efforts to make computing facilities more directly accessible for the individual user will only reinforce this tendency.

For the interested reader I am going to make two programs and, besides that, I am going to show the individual steps in which they have been constructed. The examples serve to illustrate parts of my present understanding of the demands, that the task of programming makes upon the human mind.

In my approach there are some central themes that I shall just mention for the proper understanding of the following. The one theme is that, although the program made by the programmer is his final product, the computations evoked by it are the true subject matter of his trade: he has to guarantee that the computations -the "making" of which he leaves to the machine- evoked by his program will have the desired effect. As a result he has the duty to structure his program in a useful way, where usefulness (among other things) implies that the form of the program admits trustworthy statements about the coresponding computations. The second theme is that the mental aids available to the human programmer are, in fact, very few. They are enumeration, mathematical induction and abstraction, where the appeal to enumeration has to satisfy the severe boundary condition that the number of cases to be considered separately, should be very, very small. The introduction of suitable abstractions is our only mental aid to reduce the appeal to enumeration, to organize and master complexity. Mathematical induction has been mentioned explicitly because it is the appropriate (and only!) established pattern of reasoning by which we can understand programs with either repetitive clauses or recursive procedures. As a corollary I mention the fact that for some time I knew that as a programmer I

could live quite happily without any form of go to statements but that in the mean time my considered opinion is that I cannot live happily with the go to statement.

To avoid misunderstanding I should like to state explicitly that I do not claim that the two programs produced are the best possible, measured (probably!) in terms of your private yard-stick. I do claim that they are fairly good and reasonable in terms of the average yard-stick, i.e. that they present utterly realistic solutions. I do claim to have achieved a degree of clarity and transparency of an order of magnitude better than the average programmer's solution, that my solutions have been reached with an intellectual effort considerably below average and that they admit exhaustive verification. And that is more than can be said about many a program.

The reason to treat two examples is because they have been drawn from vastly different fields: the one dealing with prime numbers is a so-called scientific application, the other, dealing with the idiosynchrasies of Flexowriters, is a so-called clerical application. These two fields are often regarded as completely foreign to each other: the succesful application of the same discipline as illustrated below gives a strong support to the assumption that the difference between scientific and clerical machine usage is by no means an inherent difference, but more probably the result of a difference in intellectual level and professional training of the people engaged.

(Note. I do not feel myself called to justify the choice of my examples, which are a kind of random draws from what is happening around me: emotionally speaking, prime numbers leave me as unaffected as Flexowriters.)

The construction of a table of the first 1000 prime numbers.

"Given an integer array p[1:1000], make a program making its elements in order of increasing subscript value equal to the successive prime numbers, where 2 is considered as the first prime number."

Well-defined as this task may seem to the benevolent reader, as we go along we shall discover an undefined boundary between the amount of mathematical knowledge the programmer is willing to embody in his program and the amount of

computation he leaves to the machine.

To start with: for the task to make sense it must be known that at least 1000 primes actually exist. We grant the programmer this knowledge and at a certain stage of program construction we allow him to appeal to this fact when he has to prove that his program does indeed halt.

We shall now give the coarsest version of the program, viz.
version 0:

```
begin"assign to the array p the prime table as described" end
```

When this action occurs among the well-understood and well-defined repertoire of actions from which the computation has to be composed, version 0 solves our problem. For the sake of argument we now assume that this action does not occur among the repertoire; particularly we restrict ourselves to repertoires in which we can operate on arrays only element wise. This implies that in our next version the order in which the elements of the array p will get their desired value has to be expressed and in it we shall try to express just that and preferably nothing more.

An obvious version of the program then starts with

```
begin p[1]:= 2; p[2]:= 3; p[3]:= 5; p[4]:= 7; p[5]:= 11;.....
```

implying that the programmer's knowledge includes a table of the first 1000 primes. We shall not pursue this version, as it would imply that the programmer hardly needed the machine at all.

The first prime number being given (=2), the thousandst being assumed unknown to the programmer, the most natural order to fill the elements of the array p is in order of increasing subscript value and if we express just that (with a simple repetitive while do clause) we come to
version 1a:

```
begin  integer k,j; k:= 1; j:= 1;
       while k ≤ 1000 do
       begin "increase j until the next prime number";
             p[k]:= j; k:= k + 1
       end
end
```

Identifying k as the subscript value of the element whose turn it is to be filled the correctness of version 1a is easily proved by mathematical induction (under the assumption of the existence of a sufficient number of primes).

Version 1a is a perfect program when the operation described by "increase j until the next prime number" occurs among the repertoire, but let us suppose that it does not. In that case we have to express how j is increased and in our next elaboration we shall try to express just that and preferably nothing more. With a simple repetitive repeat until clause (which may act upon a sequence of statements) we come for "increase j until the next prime number" to version 2a:

```
begin  boolean jprime;
       repeat j:= j + 1; "give to jprime the meaning: j is a prime number"
       until jprime
end
```

If we substitute version 2a for the appropriate operation in version 1a our resulting program is undoubtedly correct. But if we assume that the programmer knows that, apart from 2, all prime numbers are odd, then we may expect that he will be dissatisfied with the obvious inefficiency of version 2a. The price to be paid for this, call it "lack of clairvoyance" is a revision of version 1a, in which the prime number 2 is dealt with separately, after which the cycle can deal with the odd primes. So we come to
version 1b:

```
begin  integer k,j; p[1]:= 2; k := 2; j:= 1;
       while k ≤ 1000 do
       begin "increase odd j until the next odd prime number";
             p[k]:= j; k:= k + 1
       end
end
```

where the analogous elaboration of the operation between quotes leads to version 2b:

```
begin  boolean jprime;
       repeat j:= j + 2;
              "give to jprime for odd j the meaning: j is a prime number"
       until jprime
end
```

The above oscillation between versions 1 and versions 2 is in fact nothing else but moving the interface between the overall structure and the primitive that has to fit in this structure. This is definitely not attractive, but with a sufficient lack of clairvoyance and being forced to take our decisions in sequence, I see no other way: we can regard our efforts as experiments to explore where the interface can be most conveniently chosen.

Encouraged by the success of treating 2 apart, we investigate what can be gained by treating 3 apart as well. For this purpose we introduce the property "throdd", i.e. neither divisible by 2, nor by 3. The throdd numbers are of the form 6N+1 or 6N+5. By definition, 2 and 3 are the only prime numbers not contained in the set of throdd numbers and so we come to version 1c:

```
begin  integer k,j; p[1]:= 2; p[2]:= 3; k:= 3; j:= 1;
       while k ≤ 1000 do
       begin "increase throdd j until the next throdd prime number";
             p[k]:= j; k:= k + 1
       end
end
```

where the analogous elaboration of the operation between quotes leads to version 2c:

```
begin  boolean jprime;
       repeat "increase throdd j until the next throdd value";
              "give to jprime for throdd j the meaning: j is a prime number"
       until jprime
end
```

This is only an improvement, when the operation "increase throdd j until the next throdd value" is easily implemented. The proper increase of j is a function of j: call it "INC(j)". Its value is =4 when j=6N+1, its value is =2 when j=6N+5. Instead of freshly evaluating the function INC(j) whenever we need it, we introduce a separate variable, inc say, to record the current value of INC(j), corresponding to the current value of j. The variable inc has to be set initially when j is set, it has to be adjusted whenever the value of j

is changed. (The introduction of inc is an instance of a standard programmer's device to trade variable space for computation speed.) Using list-assignments to stress that inc is just a companion of j, the introduction of inc and the elaboration of "increase throdd j until the next throdd value" leads to version 1d:

```
begin integer k,j,inc; p[1]:= 2; p[2]:= 3; k:= 3;
      (j,inc):= (1,4);
       while k ≤ 1000 do
       begin "increase throdd j, adjustment of inc included, until the next
              throdd prime number";
              p[k]:= j; k:= k + 1
       end
end
```

where the elaboration of the operation between quotes leads to version 2d:

```
begin  boolean jprime;
       repeat (j,inc):= (j + inc, 6 - inc);
              "give to jprime for throdd j the meaning: j is a prime number"
       until jprime
end
```

There is no indication that any gain will result from taking the next prime (i.e. 5) out of the cycle as well and we shall not try it.

Again, when "give to jprime for throdd j the meaning: j is a prime number" is an operation from the presupposed repertoire, then our program is finished. We now assume that it is not, in other words we have to evoke a computation deciding whether a given throdd j has a factor. It is only at this stage that the algebra really enters the picture. Here we make use of the knowledge that we only need to try prime numbers as factors; furthermore we shall use that the prime numbers to be tried can already be found in the filled portion of the array p.

We use the facts that

a) j being a throdd value, the smallest potential factor to be tried is

p[3], i.e. the first prime above 3;

b) the largest prime factor we have to try is p[ord-1], when p[ord] is the smallest prime number whose square exceeds j.

If this set is not empty, we have a chance of finding a factor and as soon as a factor has been found, the investigation of this particular j value can be stopped. We have to decide in which order the prime numbers from the set will be tried and we shall do so in order of increasing magnitude because the smaller a prime number the larger the probability of its being a factor of j.

In our first elaboration of "give to jprime for throdd j the meaning: j is a prime number" we come to

version 3d:

```
begin integer n,ord; boolean nofactorfound;
      ord:= 1; while p[ord]↑2 ≤ j do ord:= ord + 1;
      n:= 3; nofactorfound:= true;
      while n < ord and nofactorfound do
      begin "give to nofactorfound the meaning: p[n] is no factor of j";
            n:= n + 1
      end;
      jprime := nofactorfound
end
```

Here we make two observations. The boolean variable called "nofactorfound" is superfluous, we could have used jprime instead, so that the last assignment statement can be removed. Furthermore, ord is a function of j that we need not recompute freshly every time, but that we can and should treat along the same line as inc. The latter remark causes the final revision of version 1, leading to version 1e:

```
begin  integer k,j,inc,ord; p[1]:= 2; p[2]:= 3; k:= 3;
       (j,inc,ord):= (1,4,1);
       while k ≤ 1000 do
       begin "increase throdd j, adjustment of inc and ord included, until
              the next throdd prime number";
              p[k]:= j; k:= k + 1
       end
end
```

where the elaboration of the operation between quotes leads to version 2e:

```
begin  boolean jprime;
       repeat (j,inc):= (j + inc, 6 - inc);
              while p[ord]↑2 ≤ j do ord:= ord + 1;
              "give for throdd j, using p and ord, to jprime the meaning:
               j is a prime number"
       until jprime
end
```

(Remark: here "while p[ord]↑2 ≤ j do" can be replaced by "if p[ord]↑2 ≤ j then", but to my taste the marginal gain in efficiency is not worth the intellectual effort to prove its validity. A programmer should learn to be lazy at the right moment and to let the principle "Safety First" prevail!)

Elaboration of the operation between quotes gives a variant of version 3d, viz. version 3e:

```
begin  integer n; n:= 3; jprime:= true;
       while n < ord and jprime do
       begin "give to jprime the meaning: p[n] is no factor of j";
             n:= n + 1
       end
end
```

For "give to jprime the meaning: p[n] is no factor of j" we may write under the assumption of decent real arithmetic

```
begin  real q; q:= j / p[n]; jprime := (entier(q) ≠ q) end     ;
```

we shall assume the availability of the integer division and write version 4e:

```
jprime := (j ≠ (j ÷ p[n]) * p[n])
```

Finally we perform all substitutions to construct a single statement.

```
begin  integer k,j,inc, ord; p[1]:= 2; p[2]:= 3; k:= 3;
       (j,inc,ord):= (1,4,1);
       while k ≤ 1000 do
       begin  begin  boolean jprime;
                     repeat  (j,inc):= (j + inc, 6 - inc);
                             while p[ord]↑2 ≤ j do ord:= ord + 1;
                             begin  integer n;
                                    n:= 3; jprime:= true;
                                    while n < ord and jprime do
                                    begin  jprime:=(j ≠(j ÷ p[n])* p[n]);
                                           n:= n + 1
                                    end
                             end
                     until jprime
              end;
              p[k]:= j; k:= k + 1
       end
end
```

We could have made the inner blocks into compound statements by moving the declarations for jprime and n to the outside. We have not done so: clarity does not gain by it and whether there is a point in doing it is rather dependent on the implementation.

And thus ends the treatment of the first example.

The unique reporting of the printed page as produced on a Flexowriter.

For our purpose we can regard a Flexowriter as a kind of electric typewriter which is operated only via the keys of its keyboard. Whenever a key is pressed, a configuration characteristic for this key is punched in a paper tape which is then moved on over one position. Typing a page thus implies the production of a paper tape specifying what has been typed. (Indeed: besides the punching station the Flexowriter has also a reading station, from which the printing mechanism can be controlled. By inserting the paper tape just produced into the reading station one can obtain another copy of the printed page.)

We want to program a routine which reads such a paper tape and gives, when called repeatedly, a unique description of the coresponding page image, according to conventions to be described below. As we go along we shall see that this is no trivial matter, because (mainly due to the construction of the Flexowriter) the very same page image may corespond to many paper tapes, greatly varying among each other. (In our example we shall simplify the real situation slightly: we shall exclude the unexpected occurrence of "end of tape" and exclude the situation that the paper tape reader of the computer discovers -due to some error in punching or reading- an illegal configuration. Even thus simplified, the problem is messy and intricate enough to serve our purpose!)

Two remarks about the form in which we shall present our solution:

1) the routine will be coded as an operator, operating in a local universe of permanently existing variables; we shall use small letters for their identifiers.

2) constants, referring to the integer values associated with characters will be denoted by identifiers composed from capital letters.

In its coarsest form the local universe contains one integer variable, called "charf" and the operator can be described by

version 0:

begin "assign to charf the next value" end .

Our Flexowriter has equal spacing, i.e. each line has a fixed number of print positions. There is a finite number of so-called "position characters" (this thanks to the absence of a backspace key on our Flexowriters, that would allow a practically unlimited number of superpositions) and each position character can occur at each print position of the page. A numerical code for the position characters has been chosen and the operator reports by assigning to charf the numerical value associated with the position character in the current print position, dealing with the print positions in each line in order from left to right and with the lines in order from top to bottom.

With respect to the left margin we assume that its position on the printed page is given; to indicate the right hand end of a line we have extended the range of charf values with an additional one, denoted by "RET" (i.e. New Line,

Carriage Return) and require for the sake of uniqueness that all "invisible" spaces at the right hand end of a line are suppressed. It is as if RET is counted among the visible position characters but that its (symbolic) printing position has to be aligned to the left as far as possible.

It is the purpose of version 1 to suppress any spaces at the right hand end of each line; for its benefit the local universe has been extended with two integer variables:

charf1: the range of this variable equals that of charf, but in the time sequence of its values invisible spaces at the right hand end of each line will still occur (if present, of course)

stock: this is a counter; its value equals the number of times that charf can be filled with a next value, before charf1 has to be refilled. It requires the initial setting "stock:= 0".

Version 1 implements the look ahead whenever via charf1 one or more spaces are reported; when followed by RET they have to be suppressed, otherwise they have to be transmitted.

verion1:

```
begin  if stock = 0 then
       begin  repeat "assign to charf1 the next value";
                     stock:= stock + 1
              until charf1 ≠ SPACE;
              if charf1 = RET then stock := 1
       end;
       charf:=(if stock > 1 then SPACE else charf1); stock:= stock - 1
end
```

Our next complication is that the "position character" as reported in charf1 (with the exception of RET) may be composed of three parts: by means of the mechanism of a so-called non-escaping key (i.e. one that leaves the carriage position as it is) one can superpose various "key characters" in the same print position. We have in fact two such key characters, viz. underlining and a vertical stroke. It is the purpose of version2 (an elaboration of "assign to charf1 the next value") to combine the key characters referring to the same print position.

We have to take into account

1) that non-escaping key characters have to be combined with the first following escaping key character

2) that repetition of the same non-escaping key character in the same print position must be considered as equivalent to its single occurrence.

For the benefit of version2 we extend the local universe with one integer variable,

charf2: the range of this variable covers those charf values coresponding to position characters, produced without non-escaping key characters plus the values denoted by UNDER and STROKE.

As a matter of fact, $0 \leq$ charf2 ≤ 127 will be satisfied; the presence of underlining will be coded in charf1 by an increase of 128, that of a stroke by an increase of 256.

Our tentative elaboration of "assign to charf1 the next value" gives rise to version2 (here CRAZY2 denotes a constant value well beyond the range of charf2)

version2:

```
begin  integer under, stroke;
       under:= 0; stroke:= 0; charf1:= CRAZY2;
       repeat "assign to charf2 the next value";
              if charf2 = UNDER then under:= 128
                                else
                if charf2 = STROKE then stroke:= 256
                                   else
                   charf1:= charf2
       until charf1 ≠ CRAZY2;
       charf1:= charf1 + under + stroke
end
```

We have said "tentative elaboration", because as it stands, this version will not prevent, say, the transmission of an underlined RET: "charf2 = UNDER" followed by "charf2 = RET" requires the insertion of an additional space to be

underlined. As pure spaces (i.e. without underlining or stroke) preceding RET will be suppressed by version 1 anyhow, we can (and shall) remedy this situation by imposing upon "assign to charf2 the next value" the requirement that it will never transmit RET unless immediately preceded by a transmission of SPACE.

The next complication is that our Flexowriters are equipped with a tabulator key TAB which, when pressed, gives rise to a punching in the paper tape, while the carriage moves on until the next tabulator stop that is more than one position to the right of the current position: the carriage moves over at least two positions. The positions of the tabulator stops are standardized (once every eight positions) but it implies that the algorithm deriving the number of spaces coresponding to TAB must be aware of the current position of the carriage (at least modulo 8). It is the purpose of version3 -the elaboration of "assign to charf2 the next value"- to translate tabulations into the equivalent number of spaces and to insert a SPACE before RET.

For its benefit we introduce into the local universe three integer variables.

charf3: the range of this variable is that of charf2, extended with TAB;

pos: keeps track of the current carriage position; when "charf3 = RET" occurs it will be set to zero, when "charf3 = TAB" occurs it will be increased until the proper multiple of 8 etc. It requires an initial setting, say "pos:= 0".

substock: this is a counter; its value equals the number of times that charf2 can be filled with its next value before charf3 has to be refilled. It requires the initial setting "substock:= 0".

We arrive at the following elaboration of "assign to charf2 the next value". (Note. As it stands I am not very much satisfied with the coding of version3. The way in which SPACE before RET is smuggled in, for instance, is too tricky. As it stands it is, however, the first version I wrote down for it.)

```
version3:
begin  if substock = 0 then
       begin  "assign to charf3 the next value";
              if charf3 ≠ UNDER and charf3 ≠ STROKE then pos:= pos + 1;
              if charf3 ≠ RET and charf3 ≠ TAB then charf2 := charf3
                                               else
              begin  charf2:= SPACE;
                     if charf3 = RET then
                     begin substock:= 1; pos:= 0 end
                                     else
                     begin substock:=(pos + 8 + 1)* 8 - pos;
                           pos:= pos + substock end
              end
     end                 else
     begin charf2:=(if charf3 = TAB then SPACE else RET);
           substock:= substock - 1
     end
end
```

The last complication presented by the structure of the Flexowriter is its built-in memory element, called "the case". It is in one of two states, called "upper case" and "lower case" respectively. When it is in the state upper case it remains in this state until the key "LOWER CASE" is pressed, what furthermore results in punching the value "LC" in the paper tape; when it is in the state lower case, it remains in this state until the key "UPPER CASE" is pressed, what results in punching the value "UC" in the paper tape. When pressing any of the other keys, the punching is only dependent on the key pressed, the printing is (except for the space bar, the tabulator and the carriage return) dependent on the current case as well.

In version4 -an elaboration of "assign to charf3 the next value"- we have to implement the influence of the case punchings. For the benefit of it we extend the local universe with two integer variables

octade: used to record the next punching on the paper tape

case: this variable may have the values LC or UC (or possibly a third one, meaning "undefined", because space, tabulation and carriage return can be processed case independently). It must get an initial value, say "case:= LC".

At this same level we implement that two legal punchings (BLANK and ERASE, coresponding to no holes and all holes respectively) are skipped without any possible effect on the page image. CRAZY3 denotes a constant outside the legal range for charf3.

```
version4:
begin  charf3:= CRAZY3;
       repeat "give octade its next value";
              if octade ≠ BLANK and octade ≠ ERASE then
              begin if octade = LC or octade = UC
                        then case:= octade
                        else charf3:= fun(case, octade)
              end
       until charf3 ≠ CRAZY3
end
```

With "give octade its next value" I indicate the paper tape read instruction and I shall not elaborate it any further. The function "fun(case,octade)" is also left undescribed: it is too much dependent on the special numerical codes; we only mention that upper and lower case space (tab or ret) must both be transmitted as SPACE (TAB or RET).

The successive insertions of version "i + 1" into version "i" is left to the industrious reader (or should I say "writer"?).

Concluding Remarks.

Before stressing the similarity of the ways in which our two problems have been solved I should draw attention to a difference. In the first example I have paid considerable attention to the decision where to put the interface between the successive levels, in the second one I did no longer do so. I do not believe that the origin of this difference is in any way related to the supposed contrast between "scientific" and "clerical" machine applications, for it has a perfect historical and psychological explanation. The historical

explanation is that I have used the prime number table generation problem in a number of oral examinations, the psychological explanation is that, treating the second example I am getting tired and perfectly willing to leave to my readers the intellectual satisfaction of improvement.

Personally I am much more impressed by the similarity of the ways in which the two rather different programs have been constructed. The successive versions appear as successive levels of elaboration. It is apparently essential for each level to make a clear separation between "what it does" and "how it works". The description of "what it does", the definition of its nett effect requires the introduction of the adequate concepts and both examples seem to show a way in which we can use our power of abstraction to reduce the appeal to be made upon enumeration.

As stated in the introduction we may expect that computers will become more directly accessible for the individual user and we may expect that the latter should like to use its capabilities for the text manipulations involved in program composition. At present I am rather unsure about the true nature of the text manipulations the user would then like to perform: it is certainly something more structured than just deletion and insertion of characters or lines! In the fervent hope of getting a better understanding of what these manipulations are I have reported two instances of program construction as detailed and as honestly as I possibly could.

Finally: if I did hit a worth-while nail on its head, then this manuscript should end with a proper acknowledgement, giving honour where honour is due. Under the present circumstances I can only express my gratitude to.... my Friends or Relations, my Masters, Colleagues and Pupils.

Eindhoven. February 1968