Programming Teaching Techniques, W.M. Turski (ed.).

AN ANALYSIS OF PROGRAM MAKING

EIITI WADA, KATSUHIKO KAKEHI, MASATO TAKEICHI
Faculty of Engineering, University of Tokyo, Japan

Abstract: A method of programming in a group which partly consists of postgraduate students is analyzed. It is not purely top-down nor purely bottom-up. Instead, it is somehow top-down and always anticipates the refinement and coding in later stages.

1. INTRODUCTION AND GENERAL CONSIDERATIONS

For several years we had been giving a lecture, the aim of which was to teach computer programming and elementary numerical analysis for the second-year students of the Faculty of Engineering. Those students had chances to write programs, run them on the computer and solve the simple problems in numerical analysis. The text of the lecture was edited from the carefully-prepared and arranged program examples, according to our principle.

The analysis of the results of the examinations and of the computer outputs indicated that the purpose of the lecture was almost fulfilled, although we could not help feeling that there would be another approach to the teaching of programming. Therefore, we read Professor Dijkstra's "A short Introduction to the Art of Programming" with much interest.

Triggered by this book, we began to think about what our programming style was, and what the important factors for accomplishing programs were. When analysis yields any commendable programming methods, we should tell them to the students, especially to those majoring in information engineering because they have more frequent opportunities to program at a higher level than general students.

With this aim in mind, we discussed our programming methods with researchers of other groups or institutes who were involved with programming, but excluded professional programmers from our discussions in spite of their experience.

Following is the list of our principles of programming. This list met the approval of those who joined the discussion because it was similar to theirs.

1. When we start a rough sketch of a program, we consider techniques that would produce a program that required a reasonable amount of work.

One example is the one-pass assembler for the mini-computer. This mini-computer came to our laboratory with manufacturer-made assemblers. The two-pass assembler was too tedious to operate, while the one-pass (i.e., assembly and go) assembler occupied more than half of 4K, 16-bit words memory. So we examined the possibility of a shorter, one-pass assembler. If the macro facilities were excluded, the only problem with the assembler is handling the undefined symbol table. Since we are trying to write assemblers in the 4K mini-computer, this undefined symbol table should be kept as short as possible to provide a wider area for the problem program. As soon as an undefined symbol becomes defined, all entries referring to this symbol become garbage. At this point, the table should be squashed, of course retaining the logical connections as before.

Fortunately this algorithm (program 1) was an established technique among us since we had used this algorithm already in other compilers. We concluded that the shorter one-pass assembler could be made without any difficulty and would bring forth considerable gain.

2. While designing the program, we frequently encounter the question of which functions to include and which to exclude. This is again closely connected with our knowledge of programming techniques, as well as our understanding of human nature. In designing the source program context editor for the mini-computer mentioned above, we abandoned the upward movement of the pointer since at that time we did not know the magic list, and we did not like having two links for a record in the linear list. This evidently decreases the number of records to be stored in the memory. A motto in programming for the mini-computers, we believe, is to make everything in the smallest limit.

3. Trying to keep every part of the program or data area as short as possible is recommended since this makes it easier for us to handle the whole program. Moreover this tendency has a double effect: it decreases program writing time and, later, increases program running speed. To this end, we are constantly trying to write a program that produces the same effect in fewer steps and then staharing the results among us. Examples are: (i) to construct a word in the accumulator from some sections of word A and complementary sections of word B, assuming that the words C and D contain the masking patterns for the requested sections of A and B, respectively, and word T serves as a temporary storage; instead of writing a program like:

```
Load  A
And   C
Store T
Load  B
And   D
Xor   T ,
```

write it as:

```
Load  A
Xor   B
And   C
Xor   B ;
```

and (ii) to test whether a word in the accumulator contains a code of letters from A to Z in ASCII, granting that the code of A is 41 in hexadecimal or 65 in decimal and that of Z is 5A or 90, write in the 16-bit accumulator as

```
Add   = (32765-90)
Add   = 26
Skip if overflow
Jump to NONLETTER.
```

4. However, if we have no excellent technique at hand, we put off improving the offending part until a later stage, and hurry to complete the whole program whilwe are still interested in it. This is because, on the one hand, to complete the whole program is very much harder than to make the local improvement and requires great concentration, and we cannot endure such deep concentration for long. On the other hand, trial usage of the first version program might reveal that the basic design of the program is undesirable. This should be corrected before too many users become accustomed to the undesirable interface with the program. The improvement of the local efficiency is never too late even after discovering the disagreeable points. According to our experience in the mini-computer, almost all first version programs were coded in less than a

week except for the Fortran compiler-interpreter. After every function of the program is certified (though it is often very difficult to do this perfectly), we start to work on the extensive improvements. One example of this improvement is in the length of the program as in the case of the one-pass assembler, introduced in the preceding section, in which the first version program occupied 860 wordswhile the shortened one requires only 780 words of memory.

5. In programming, we decide the data structure at the earliest chance, and spend considerable time in designing it and the basic routines of data handling before the detailed planning starts. We admit hato having the tendency Professor Dijkstra described in hitook, of putting the cart before horse. However, in the case of the mini-computer, memory storage is so precious that we contrive many possibilities in data structure focusing attention on compactness of the data area and on speed of data handling. One of the strategies in table searching is to construct the inner loop to contain only one test of pattern matching. Table end is tested in a tricky way as a special case of the match by a slight provision at the beginning of the search.

6. As might be observed from the above description, our programming method is not systematic at all. We have based our programming on several intuitive attitudes, acquired unconsciously. The fact that our method is less systematic might be related to the properties of system programs, which are less systematic than typical problem programs. Because of this unsystematic approach, we think, we are often stuck in the mud during the programming. The ohlnly escape is to change our point of view, although nobody knows explicitly the most effective change. Actually we saved many programs this way. Even if the program is not apparently in the mud, we strongly recommend throwing out the first version of a program and writing it again from a new angle. In reality, however, this is not often done.

2. PROGRAMMING THE TYPE-OUT ROUTINE

In this section, we will trace our real experience in designing and coding the type-out routine, again on the mini-computer. By type-out routine, we mean one such as the RUNOFF routine in the MAC system of MIT, which, according to the MAC programmer's guide, types out memorandum files of English text in manuscript format. Many such programs are known to exist. The function and program of the routine are now of no interest. The current presentation of it is just to serve for observing, from a higher level, how the programming is actually performed. The first trigger to this program was pulled when one of the authors saw a computer-controlled manuscript many years ago. The first step to the programming of this routine began, probably, while we were preparing the first version of the Algol N report in 1968, and at that time and later on, the possibility of implementation was discussed occasionally. The first step in those days would be a discussion something like the following.

Step 1

Q. What sort of functions should be performed by the type out routine?

A. Something like the report of the Algol N or Algol 68 should be edited or typed out.

Q. To do so, what would be the most serious problem?

A. Since these reports contain many type fonts, the typing element could be replaced and the manuscript of the multi-font should be printed.

Q. How would the problem be solved?

A. If the typing element could be replaced automatically, it is solved. But that seems almost impossible. It would also be good if, at every change in the font, the machine could stop and request that the operator replace the typing element, but that would be far from practical. The most applicable solution would be as follows. Prepare one page image in memory; then, let the operator set a typing element of the font which is used in that page and let the computer scan the page image and print all the characters of the font. When the computer has finished this process, let the operator set the second element, put the paper in the initial position and start the computer. This time, the computer would fill in the blanks with the characters of the second font. A by-product of this is the possibility of automatically printing the subscripts or superscripts if we treat them as different fonts from the main text. The subscript positions are left blank while the main text is printed. To print the subscript, return the paper, adjusting it slightly higher than before, and start the computer with the same typing element. This seems very useful for mathematical manuscripts.

Q. How is the input tape prepared?

A. We have a very convenient source program editor, so editing and correction would be very easy.

Q. How about the output?

A. The on-line connected type-element typewriter would be indispensable.

Q. Is it practical?

A. If casette tape memory is supplied, it surely is. But even without it, the system would prove useful.

Q. What other functions are expected?

A. A footnote editor and conversion of the symbolic reference from one part of the text into page-number reference, just like the elimination of the symbolic address reference by the assembler. This would be a burden for the mini-computer, however, and if we would like to realize the system as quickly as possible, these functions should be excluded. The material mentioned above is probably still an idea for development at some other time. Because of these ideas for other possibilities, once the project starts, we are at once ready to undertake it.

Meanwhile the typewriter was connected to the mini-computer, and we started the implementation. The following was the second step in planning.

Our usual practice soon after the installment of a new input or output device is to treat the related commands on it. But as the typewriter commands were very simple we omitted this practice.

Step 2

First, we designed the input rules. The input device is the ordinary teletype with 64 graphic characters including the space. As the general principle or the fundamental requirement, there should be plain correspondence between the letters and digits of input and output. On the standard typing element, the character set is very similar to that of teletype. But there are such typing elements, for instance, "symbol", with characters on them which are quite different from the teletype character set. This produces difficulty in deciding the correspondence between the special characters and the

teletype codes. Accordingly, the input rules for the characters of the non-standard typing elements were not settled. Instead, correspondence tables were designed between the set of the standard element characters and that of non-standard elements. For example, since symbol " " corresponds to the capital F according to this table, input F with specification of "symbol" font will appear as on the output.

Even within the limits of the standard element, there are still some characters which exist either on the teletype or on the typewriter but not on both. After various considerations, the correspondence between the sets was decided upon as follows:

teletype	typewriter
/	¢
	o
	-

Two angle brackets " " and " " of teletype would be used, first, as case-shift codes from lower to upper and upper to lower, respectively, and, second, as control delimiters in the input record. These decisions were really very exciting. Since this decision fixes the quality of the routine, it should be made carefully yet promptly. It is very important to prepare an excellent system program with good interface in the early stage of the computer's life. If the first programs installed are not good ones, they inspire installation of similar programs and thus cause confusion. Ian excellent program is supplied at a later time, the users are unfortunately accustomed to the former, clumsy interface, and nobody wants to use the new one, thus making thorough debugging of the new system impossible.

If the design is analyzed at this level, there might be another kind of interest. One criterion among us in this type of decision is ease of remembrance. Does " " remind one of the crescendo from small to large and " ", of decrescendo?

Next, types of control were selected. Contrary to the input character set, the control set is open ended, allowing new controls to be added later somewhat freely. So, as the initial set, we selected from those of the RUNOFF, "indent", "undent", "begin page", "adjust", "nojust", "fill", "nofill" and "break", and to this set we added "end" and "back space". As a first experiment, we thought, this control set would be sufficient. The same abbreviations of control as the RUNOFF were adopted.

"End" is the indication of the end of file and the effect of "back space" is understood literally. Since the teletype keyboard has no back-space key, we had to include "back space" control to type symbols like ≠ or real.

Step 3

The next consideration is the setting up of the page image in memory. First, we estimated whether the 4K memory was enough for holding one page. Suppose one page consists of 60 lines and one line, of 60 characters. Then, one word-per-character configuration occupies 3.6K words, which compels extremely compact programming and almost prohibits the enlargement of the page size. Though the image buffer handling becomes rather complicated, the half word or 8 bits-per-character configuration had to be used.

Next step is the design of the 8-bit patterns as the elements of the page image. Because the input to the routine is placed into the image, character-by-character, without changing the order, no special provision would seem necessary except that for the treatment of spaces. To justify the right side, space

in the line should be extended according to an algorithm. If this treatment causes the movement of the already packed patterns in the image, though it would not be impossible, it would take longer than, say, by adding a counter to the space pattern and increasing the counter which corresponds to widening the space. The slowness in input would be less favorable since input tape is fed from the photo-electric tape reader. As for the two shift controls, font shift and case shift, usually one of the two is used, one to store the shift code separately from the character code, thus indicating the shift as an internal state, and the other, combining shift information with the code, thus expanding the total length of code. In the limit of 8 bits, our solution was like this. Font-shift was to be stored separately and case shift combined with code. We cannot recall why we reached this solution. Probably, past experience and some unconscious anticipation of the later coding stage made us decide so. If hindsight is permitted, the reasons for our decision would have been: (i) combined font shift lengthens code length, for example, by 4 bits if ten sets of font are used, while combined case shift lengthens only 1 bit for 26 characters, (ii) in the output stage, a change in font has an important meaning, while a character, along with code shift, is mapped into the output code by the code conversion table. Whether it was the best solution or not is still unknown. It would be very interesting to observe how the novice programmer in this sort of programming reaches a solution when he encounters the present problem for the first time. The solved 8-bit patterns are summarized below.

```
1xxxxxxx space, last 7 bits are space count
0111xxxx font shift, last 4 bits are font number
01100010 end of page
01100001 new line
01100000 back space, without space counter
00000000 characters, (=mod(ASCIIcode,64)+
  ...                  if uppercase then 64 else 0)
01011010
```

One of the planned local improvements is the modification of the back space pattern. As will be described later, the main scheme of the output subroutine changed during the course of programming. At the beginning we considered completing and testing the program with a single typing element. In that version of the program, at each occurrence of the back space pattern, the computer would send the back space control out to the typewriter. However, in the multi-font output subroutine, the virtual position of the typing element is only accumulated in the counter until the actual printing becomes necessary. For this modified version, the back space with counter would be suitable. The end-page pattern was employed to detect that the page had ended with the same mechanism as for the pattern distribution, i.e., by way of the multi-jumping table. The algorithm attached to the character patterns was adopted as the simplest to prepare the short domain of argument for the code conversion table.

Then we stepped into the coding of the input analyzer. The analyzer works like the lexical analyzer or basic symbol reader or, according to our terminology, the syllable reader, generally used in compiler construction. Consequently, the fundamental techniques are very familiar and as soon as the following decisions are made, this could be coded on the spot. Into what groups is the input analyzed? For each group, where is the additional information stored? Since, in assembly language, the multi-exit method is usually employed to return from the subroutine, the order of the exits must be chosen. The current version classifies input into, (i) word, (ii) space, (iii) carriage return and (iv) controls. The line feed and other non-graphic code are neglected in the input analyzer. On designing the first version, no attention was paid to detecting the hyphen. But some trial use made us feel that the synthesized words should be separated at the hyphenated position, when they bridged the lines,

although this improvement is not so urgent because it is irrelevant to the user-machine interface.

The page image construction was not difficult. This is the main routine of the type out program. The design of this part started from the main loop, postponing the adjustment of the boundary conditions or the detailed preparation of information for the subroutines.

Call the input analyzer. If the unit is a word, and if the remaining space in the line is enough to hold it, then place the word. If the word is too long to place, call the right-adjust subroutine and prepare the next line and place the word in the new line. But if it is impossible to prepare the next line because of the page limit, then call the output subroutine. Then try to prepare the next line. If the next unit is a space, and if it is possible to place it in the current line, place it there. If placing is impossible. prepare the next line and discard the space. If a carriage return is acquired, and it is the nonfill mode, prepare the next line, but otherwise, treat it like a space. The font shift is analyzed by the input analyzer, and it is attached to the word though it is not included in the word length. Other controls are treated independently. "Fill", "nonfill", "adjust" and "nojust" set or reset the related flip-flops. "Indent" and "undent" set the related counters. "Back space" places the frequency of the back space pattern parameter. "Begin page" places an end of-page pattern and calls the output subroutine, then prepares the new page. "Break" places a new line pattern and prepares the next line. "End" also puts the end of page pattern and calls the output subroutinebut this time it exits from the main control loop and completes the program. The precision of the initial coding was something like this, though inconsistencies were noticed here and there; for example, the above description refers to a placing a new line pattern when a "break" control appeared while it neglected a new line after calling the adjust subroutine or after the overflowed space. Of course, they are taken into account on the second version of the control loop.

A possible starting point of the whole control loop would be the new page initialization after the output calling of "begin page" control. However, the final form consisted of another page initialization and main control loop with an input analyzer call at the entrance, in that order. This configuration seems more natural than initiating the program from a special point in the loop. The final configuration would not lengthen the entire program very much. After this, detailed planning for the various parts of the loop were made. Counting up the number of spaces for the right-adjust subroutine was one of the examples. There was a place to be programmed carefully. When the next line was prepared anew, a space pattern with space count zero was inserted even though there was no indentation at all. This is simply for the adjusting routine's sake, because in adjusting the right end, the leftmost space should not be adjusted if it is an indentation. It is desirable to treat various conditions as uniformly as possible. So even with zero indentation, an extra space was inserted. However, care must be taken in setting the first space counter. We could not set this when the new line was prepared, since immediately after this and before the next word was placed, another "indentation" control might come. This sort of need for care might be easily forgotten. We ourselves were also on the verge of neglecting the condition because all the programming was advancing at full speed.

Reaching this point, we began coding the main control loop in the assembler language and attached the input analyzer to it. Without output and adjusting routines, the first test was initiated. Of course, the main program calls adjusting and output subroutines. The adjusting routine is a dummy. It returns control immediately without any processing. The output routine is a hexadecimal dump of the page image. Time between the coding initiation and running the test was two or three hours; the output made us feel that we were at the height of

programming or even that we had already crossed over the saddle point.

Then the adjusting subroutine and single-font output subroutine were coded. These routines made it possible for us to write letters in these early days. The whole process from the beginning of the second step of the completion of the single-font system was a matter of two or three days.

There was a comparatively quiet period when trial use of the program was made. Those who examined the output suggested their own favorite adjusting algorithms; for example, the wider spaces should be distributed from the neighbourhood of the longest word in the line, etc.

Soon we resumed programming. This time, the multi-font output routine was planned.

The first design was something like this.

There were two control loops, each of which was essentially the main control loop of the single-font output routine. In the single-font control loop, the font shift pattern took no action. But now they play the role of switching between loops. One of the loops controls the "current" font and the other loop undertakes the "other" font. The "other" font loop treats a character as a space, while the "current" font loop prints a character on its position. But it is undesirable to move the typing element in vain in the "other" loop by a space or a space equivalent character. Therefore, in the "other" font loop, the spacing was only counted in the computer, and on transferring to the "current" loop, the counted spaces should be sent out. The new line of the "other" font sends out a return control to the typewriter and resets the space count. On transferring from the "current" font to the "other" one, a proper space count should be calculated. Somewhere in the planning, it looked more complicated than necessary. The whole planning was reconstructed from the foundation. The new plan employed the concept of four counters, namely, the actual vertical (horizontal) counter which reflects the vertical (horizontal) position of the typing element and the virtual vertical (horizontal) counter which reflects the scanned vertical (horizontal) position in the page image. On the initiation of the page image scan, the four counters are reset to zero. During the scan, the virtual counters go up and down following the scanning position. If a character comes and the character is of the other font, the virtual horizontal counter increases by one. If a character of the current font comes, then a subroutine is called to locate the typing element to the page image position, assigning the actual counters to the values of the corresponding virtual ones. The attached program 2 shows this algorithm, though the declarations of some predicates or procedures are omitted wherever their meaning is obvious. The tabulations are set at every 8 positions because the mini-computer has no division instruction, and the calculation of quotient and remainder, when the division is 8, is exceptionally simple on such a machine.

Acknowledgement. We wish to express our most sincere thanks to Professor S. Moriguti for his constant guidance and to the colleagues of our laboratory for their help. Our gratitude is also expressed to Professor T. Iwamura and the members of the Algol N group, as well as to those who happened to participate in this work for their enthusiastic and valuable discussions.

```
program 1

procedure garbagecollection(integer i0, integer j0);
begin integer i,j;
      i:=i0; j:=j0;
      while i   j do
      begin while i   j and not garbage(i) do i:=i+1;
            while i   j and garbage(j-1) do j:=j-1;
            if i   j do
            begin info[i]:=info[j-1];
                  link[i]:=link[j-1]; link[j-1]:=i
            end
      end;
      i:=i0;
      while i   j do
      begin if link[i]   j and link[i]   j0 do
            link[i]:=link[link[i]];
            i:=i+1
      end
end


program 2

procedure multifontoutput();
begin integer av,vv,ah,vh,i,code;
      boolean fontflag,loopcontrol,lowercase;
      procedure locate();
      begin if vv   av and vh   ah do
            begin writereturn();
                  ah:=0; av:=av+1;
            end;
            while av   vv do begin writevfeed(); av:=av+1 end;
            while ah div 8   vh div 8 do
            begin writetab();
                  ah:=(ah div8 + 1)x8
            end;
            while ah   vh do begin writespace(); ah:=ah+1 end;
            while ah   vh do begin writebackspace(); ah:=ah-1 end
      end
      print("load new paper"); wait();
      i:=0;
      while i   10 do
      begin if fonttable[1]   0 do
            begin print("set type element", i); wait();
                  av:=vv:=ah:=vh:=0;
                  initializeimagepointer();
                  loopcontrol:=true;
                  while loopcontrol do
                  begin getpattern();
                        if space() do vh:=vh+spacecount;
                        if font() do fontflag:=fontnumber=i;
                        if backspace() do vh:=vh-1;
                        if return() do begin vv:=vv+1; vh:=0 end;
                        if endpage() do loopcontrol:=false;
                        if letter() do
```

```
                    begin if fontflag do
                          begin locate();
                                code:=codetable[letternumber];
                                if code  64 and lowercase do
                                begin writeuppercase();
                                      lowercase:=false
                                end;
                                if code   64 and lowercase do
                                begin writelowercase();
                                      lowercase:=true
                                end;
                                writeletter(code);
                                ah:=ah+1
                          end
                          vh:=vh+1
                    end
              end;
              i:=i+1
        end
  end
end
```

DISCUSSION

Ershov:

You stated that your method is not so systematic as it could be; nevertheless it is highly systematic, and it seriously induces some discipline in developing a program from the analysis of specification up to final debugging and delivery. I would like to know how you administer the programmers to act in this way.

Wada:

I think our method is not so systematic as lectures given in the classroom, but I think programming can be taught in the so-called trade guild system: one master and many pupils. From the beginning of my programming life I have been using that system; and I think it is the best system for students, especially for postgraduate students with a lot of ability. We just live wit the postgraduate students in our group everyday and provide a lot of computing facilities for them; and we talk about the comiler that should be coded, the language that must be implemented, or what sort of utility programs might speed up the debugging and so on. In these discussions many graduate students grasp the atmosphere of our programming. Of course, for undergraduate students we have to use quite a different system, but for postgraduate students I think this is thbes wt way.

Woodger:

You mentioned that you had used some programming techniques because the core store in your mini-computer was so precious. Do you think that in consequence of those methods you suffer afterwards the cost of programming time or of re-programming, which if you put it in terms of money, would equal the cost of more core storage?

Wada:

I think if you are writing system programs, for instance an assembler, the programming cost of making the whole program as short as possible is less expensive in comparison with the core storage left for the users. For instance, suppose we left an amount a of core storage for users. We can also leave them an amount b (b a) of core by improving our assembler. The users receiving amount a of core memory, might try to write, using the assembler, a program of length c. If a c b this task is quite impossible for them. But if we give them the shorter version of the assembler, the one leaving amount b of core, we can make this task possible. There is of course, some physical limit, but we should try to provide users with as much core memory as possible.