On Teaching the Art of Compromising in the Development of External Specifications*

IZUMI KIMURA**

Programming in the broad sense includes the development of an external specification. No systematic method, however, has been made available for training programmers in this side of programming. This paper presents a sample problem which, it is hoped, is useful for improving this state of the art. The problem concerns the formatting of a letter, and is so designed as to require compromises between the implementor's and the user's convenience. A possible solution is outlined. In a coding example, a state diagram is used as a vehicle for pseudo-coding. Use of a table-driven technique is also described. Experiences with the use of the problem in classrooms and other environments are discussed.

Keywords and Phrases. External specification, compromise, man-machine aspects, programmer's training, computer science education, text formatting, state diagrams, table-driven techniques.

0. Introduction

Programming in the broad sense includes the development of an external specification of the product. This first step of programming often plays a decisive role in the success or failure of the project as a whole. A programmer cannot claim his professional maturity until he acquires a good command of this aspect of programming.

This, however, is an extremely difficult subject. An external specification is usually a result of subtle compromises between the implementor's and the user's convenience. It is all too well-known that a little thoughtless twist in the external specification could make the implementation hopelessly difficult. It is also well-known that a seemingly innocent omission made for a bit of ease of implementation often results in a product which is very hard to use. Some may well insist that this side of the story can be learned only through bitter experiences and by face-to-face guidance of senior colleagues.

This classical approach, however, has the disadvantage of wasting the time and effort of both the instructor and the student. To improve programmer's training, it is therefore desirable to reduce at least some of the art of developing external specifications from unteachable to teachable.

A standard method for teaching what is otherwise impossible to teach is to use examples. A good example often improves the efficiency of an educational process by orders of magnitude. In fact, the only method for teaching Basic, Fortran, Algol 60, PL/I, APL, Cobol, Snobol 4 and Lisp in one semester(!) is, as was done by Peterson [1], to use examples. A good example visualizing good ways of developing external specifications will greatly contribute in improving the way in which programmers are trained.

This paper is concerned primarily with gaining an insight into how these educational examples may be developed. We try to do this according to our own philosophy, i.e., by developing an instance of such an example.

Our example relates to faircopy preparation of letters, and simulates in a diminished scale a situation which a professional programmer would encounter in a real environment. In particular, it is so designed as to require compromises between the user's and the implementor's convenience. Our example consists of a sample problem along with an outline of its solution. The solution may not be unique, just as the solutions of any realistic programming problems are not. The example has actually been used for several times in classrooms, and, as far as the author sees, has achieved a considerable success.

The problem is presented in Section 1. Section 2 outlines a possible solution. The algorithms sketched there may be of some interest for their own sake. Section 3 describes how our problem was useful in classroom and other situations. Section 4 gives concluding remarks.

1. The Problem

The sample problem of this paper reads as follows:

Write a program that reads a letter in English of the form as illustrated in Fig. 1, and edits and prints it in a better-looking format as shown in Fig. 2. Use a procedure-oriented language of your choice, say Fortran. Before reading the original letter, a line giving the left and the right margin positions of the output, e.g. of the form "␣10␣50", is read, and

*A preliminary version in Japanese of this paper appears in the Proceedings of the 18th Programming Symposium at Hakone (Jan., 1977) 161-168.

**Department of Information Science, TokyoInstitute of Technology,

used for controlling the subsequent editing operations; no further controlling information is made available. The blank lines and the heading lines (the addresses, date, closing and the like are so termed here) may be left as they are. Only the text (body) of the letter is edited with the right margin justified. Although the sample letter given in Fig. 1 has one paragraph only, two or more paragraphs may be present, of course. The method of distributing blanks may be chosen freely, provided that the quality of the result is comparable to that of Fig. 2. The end of data is indicated by placing a line starting with "-␣" immediately after the last line of the letter. This additional line is just read, but is not transfered to the output.

Your program must be accompanied with a document sufficient for enabling other people to use it, understand the algorithm, and modify it if the occasion demands. The document, however, should be as brief as possible.

```
                                LOCK WILLOW,
                                  SEPTEMBER 19TH.

DEAR DADDY,
     SOMETHING HAS HAPPENED AND I NEED ADVICE.
I NEED IT FROM YOU, AND FROM NOBODY ELSE IN THE
WORLD.  WOULDN'T IT BE POSSIBLE FOR ME TO SEE YOU?
IT'S SO MUCH EASIER TO TALK THAN TO WRITE; AND I'M AFRAID
YOUR SECRETARY MIGHT OPEN THE LETTER.

                                                JUDY.

P.S. I'M VERY UNHAPPY.
```

Fig. 1 Sample letter, original form—by Jean Webster.

```
                                LOCK WILLOW,
                                  SEPTEMBER 19TH.

DEAR DADDY,
     SOMETHING HAS HAPPENED AND I NEED ADVICE.
I NEED IT FROM  YOU,  AND  FROM NOBODY ELSE IN
THE WORLD.  WOULDN'T IT BE  POSSIBLE FOR ME TO
SEE YOU?  IT'S SO MUCH  EASIER TO TALK THAN TO
WRITE; AND  I'M  AFRAID  YOUR  SECRETARY MIGHT
OPEN THE LETTER.

                                                JUDY.

P.S. I'M VERY UNHAPPY.
```

Fig. 2 Sample letter, edited output.

Notes

(1) Here, and throughout the rest of the paper, "␣" stands for a blank.

(2) The reader will note that in some places the wording of the problem has been left intentionally vague. For one thing, what do we mean when we say that the heading lines are left "as they are"? The problem does not even bother to define a heading line. It is the most important point that from such a vague description the student must formulate his external specification.

(3) The programming language to be used in solving this problem is left open. Any programming language may be used provided that it is "procedure-oriented". It is intended by this restriction to prohibit the use of a language which is of a too high-level allowing the programmer not to think himself about the implementation details. Apart from this point, the choice of programming language does not matter very much. Fortran is mentioned just because it is all too popular (in Japan). Note that the problem does not talk about ANSI Fortran. It says just Fortran. In fact, if we require an absolute conformity to the ANSI Fortran standard effective at the time of this writing, it will become impossible to solve our problem.

(4) The use of "-␣" as a terminator is not beyond criticisms. A better way to indicate the end of data might be to use the end-of-file condition. Also, the minus sign might better be "?". Nevertheless this choice was made for making the problem solvable in a wide variety of Fortran processors.

(5) As stated, this problem is primarily concerned with the art of developing an external specification. It still requires that a completed product consisting of a program with accompanying documentation be submitted. By this requirement we intend to force the student to make real compromises.

(6) It is customary with this type of text formatters to use control codes for telling the computer about the position of a heading, the beginning of a paragraph, and so on. It is by design that this problem does not mention a control code. For avoiding control codes man-machine aspects must be taken into account, and it is these very aspects that are most important in our problem.

(7) A similar text-editing example has been published recently by Abrahams [2]. The author shares with him the opinion that educational programming examples should not be restricted to the eight-queen's problem and topological sorting.

2. A Solution

A final solution cannot be given to this problem. Our solution, by definition, is not final since the problem itself has been made intentionally vague. Here, our main objective will be to indicate possible directions where the student could profitably search for his solution.

The following subsections describing the steps of solving the problem in fact correspond to the sequence of sessions actually used by the author in training his students. See Section 3.3.

2.1. Decomposing the Problem

It is first of all necessary to identify the difficult points of the problem. As noted previously, our problem has been intentionally made vague, and therefore has many possible interpretations. Interpretations, however, can be good or bad. The central issue lies in achieving a good one. Clearly, the most basic question about this side of the problem is:

Question a:

How do we discriminate between the heading and the text lines in a letter containing no control codes?

In addition, we have a question concerning the general structure of the program written:

Question b:

How do we control the process of extracting words from the input lines and repacking them to form output lines?

A third basic question is:

Question c:

How do we determine the way the additional spaces are inserted in order to right-justify the output lines?

Also, there are some additional questions of interest:

Question d:

Why don't we hyphenate? By hyphenating big words, we would easily obtain a beautiful result.

However, hyphenation is one of the major subjects in text formatting [3]. The student should not be overambitious by reading more out of the problem than it actually implies.

Question e:

How about justifying, or centering as appropriate, the heading lines?

Attempting to do this, however, can adversely interact with solution of other parts of the problem, notably Question a. It turns out later that a straightforward solution will be invalidated if heading lines are to be modified.

In fact, the decision to leave the heading lines as they are has been made in connection with another decision, inclusion of the facility for setting the right margin. This alleviates the user's difficulty by making it possible for him to type the heading lines first, and to adjust the placement of the body of his letter afterwards.

Question f:

How about obtaining better results by considering two or more lines in determining which of the words are to be packed in a particular output line?

To illustrate this point, see Fig. 3. In Fig. 3(a), everything that can be packed into a line is actually packed. In Fig. 3(b), sometimes we refrain from packing what we can. Clearly, the latter approach can result in a better-looking output, as it actually does in Fig. 3(b). However, in a straightforward design, this would heavily complicate the problem.

```
BUT  BEFORE THIS CAN
BE       ADEQUATELY
ANSWERED,  A HOST OF
METHODOLOGICAL
QUESTIONS MUST FIRST
BE RESOLVED.
```

(a)

```
BUT    BEFORE    THIS
CAN  BE    ADEQUATELY
ANSWERED,    A    HOST
OF    METHODOLOGICAL
QUESTIONS MUST FIRST
BE RESOLVED.
```

(b)

Fig. 3 Refraining from packing what can be packed.

We shun the tempting Questions d, e, f. It is an important part of the programmer's job to control his own ambition. Our problem has been designed to heavily penalize overdesigns.

Moreover, extracting and repacking of words (Question b) can be handled adequately in terms of coroutines or pseudo-parallel processing. Even in Fortran, we can arrange the algorithm as a collection of multiple passes using work files, and then systematically rewrite the program into a more conventional form if efficiency is found important. As all these techniques are standard in the literature, we shall hereafter restrict our attention to Questions a and c.

Exercises

Although the Questions d, e, f should be avoided in a solution of our problem *per se*, the following exercises derived from them are still interesting and suitable as assignments for the students, each requiring a considerable amount of labor:

Exercise 1:

Read Gimpel's book [3] on Algorithms in Snobol 4, and try to fit his method of hyphenation into our letter formatting program.

Exercise 2:

Introduce suitable control codes so as to make right-justification and centering of the heading lines possible. Also try pagination. Keep your use of control codes to a minimum.

Exercise 3:

Devise schemes that avoid the difficulty as illustrated in Fig. 3 by considering no more than two lines at a time. Test your scheme for a variety of texts.

2.2 Question a—Discriminating between Heading and Text Lines

This part of the problem, if programmed in Fortran in a straightforward manner, would be intertwined with Question b. In order to cleanly separate this portion out, we shall consider a subproblem as follows:

Subproblem a:

Write a program that reads a letter of the form as shown in Fig. 1, discriminates between the heading and the text lines, and writes the lines of the letter to an output file, with an "*" attached to each heading line. To each text line is attached a "␣". As an additional service, we require that the first line of each paragraph be marked with "(" instead of "␣", and that a line of the form ")␣␣ . . . ␣" be written immediately after each paragraph of the body of the letter.

As before, "␣" stands for a blank. A blank line is considered as a heading line throughout the rest of this paper. Note that the output file of the above is suitable

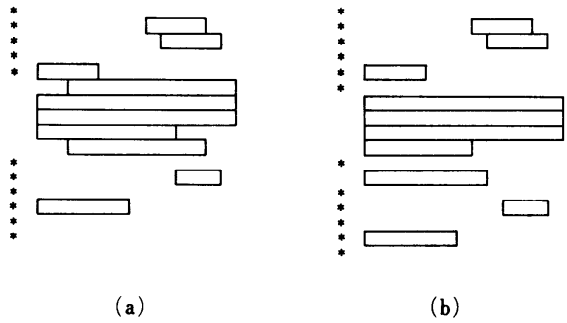

Fig. 4 Schematic representation (a "cartoon") of a letter, (a) like that of Fig. 1, (b) another style.

for use as an input for later stages of processing the letter.

A good way for finding a solution of a problem of this kind is to draw "cartoons", i.e., schematic pictures. Fig. 4(a) is a schematically redrawn version of Fig. 1 (in "indented" form), with a paragraph added. Fig. 4(b) shows another possible style (in "blocked" form). The stars indicate heading and blank lines, while those not starred are text lines. What we are to produce is something like these stars, with additional indications of the placement of the paragraphs.

Note that the paragraphs are drawn as though they are already right-justified. In drawing a "cartoon", it is very important to cut unnecessary details. The subject here is discrimination rather than justification. Our picture should not catch the eye by being overrealistic with the unjustified ends of lines.

First look at Fig. 4(a) "with your eyelids narrowed". Then you will find that all those lines having long runs of no less than, say, twelve leading spaces are heading lines. In addition, there are some other heading lines which can be distinguished by long runs of spaces to the right of them. However, a line of the latter type must be regarded as a part of the text if it immediately follows a text line. A consistent interpretation is obtained if we regard a long run of spaces to the right of a line as a sign of a heading line whenever the line is at the beginning of the letter, or follows another heading line.

The break points of paragraphs can be found by looking at a text line whether it is immediately followed either by a heading line, or by a text line starting with a space (not forming a long run). The state transition diagram of Fig. 5 summarizes these considerations. Here, the hatches indicate the runs of spaces, i.e., the absence of text strings, and the dotting indicates don't-care: both the presence and absence of text strings are allowed.

It is interesting to note that Fig. 5 remains valid even if we consider the format of Fig. 4(b). Fig. 4(b) includes a paragraph consisting of a single, short line. Fig. 5 regards this as a heading. Since the heading lines are printed as they are, however, this will not cause any trouble provided that the original line is typed gracefully.

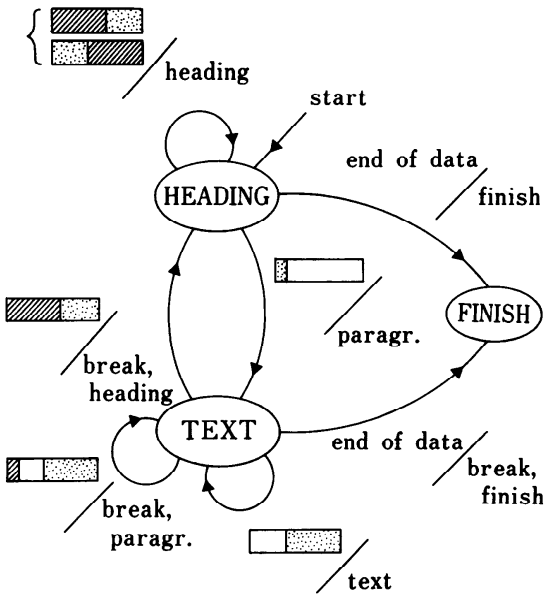

Fig. 5 Discriminating between heading and text lines. The hatches and the dotting indicate spaces and don't-care, respectively.

(Recall what we have said about Question e.) Alternatively, the user could type the last word of the short line near the end of it, with the space preceding the word boosted as necessary. This requires that the line has two or more words, of course. However, for a single-word paragraph, there is absolutely nothing to worry about.

Now, what do we mean by saying that there is a long run of spaces to the right? For leading runs of spaces to the left of a line, we can establish a rule of thumb: the number of spaces used for starting a paragraph doesn't usually exceed ten; longer runs are sometimes used, but are rare; to ten spaces we may add one slack to obtain our bound, i.e., eleven spaces; twelve or more spaces may be safely regarded as starting a heading line.

No similar standard is available for runs of spaces to the right. Thus, even the width of platen of a terminal can affect the threshold. Here, we shall be content with an arbitrary bound: a long run of spaces to the right is defined to exist if and only if column 40 and the following columns of a line are all blanks.

Exercise 4:

A better solution for determining which line has a long run of spaces to the right might be to use another pass, in which the input data is scanned and an appropriate threshold is computed. Try it. Note that this exercise involves a significant amount of man-machine considerations.

Our method will naturally fail if the user desires a heading line, with a long run of spaces to the right, to follow a text line immediately. This problem can be

```
C
C**** MAIN DRIVER ****
C
      INTEGER COUNT
C
   10 CALL DISCR(COUNT)
        IF(COUNT.GT.1) GOTO 10
      STOP
      END
C
C**** DISCRIMINATE BETWEEN HEADING AND TEXT LINES ****
C
      SUBROUTINE DISCR(COUNT)
      INTEGER COUNT
      LOGICAL LEFT,RIGHT,TOP,LAST
      INTEGER CARD(80),DUMMY(80),
     $          TITLE,TEXT,PARAGR,BREAK,FINISH
      DATA    DUMMY /80*' '/,
     $          TITLE /'*'/,  TEXT  /' '/,
     $            PARAGR/'('/,  BREAK /')'/,
     $              FINISH/'-'/
C
      COUNT = 0
C
C                                  ***  TITLE/HEADING STATE  ***
   10 CALL GET(CARD,COUNT)
        IF(.NOT.LAST(CARD)) GOTO 20
          CALL PUT(FINISH,DUMMY)
            RETURN
   20   IF(.NOT.(LEFT(CARD).OR.RIGHT(CARD))) GOTO 30
          CALL PUT(TITLE,CARD)
            GOTO 10
   30   CALL PUT(PARAGR,CARD)
C
C                                  *** * *  TEXT  STATE  * * ***
   40 CALL GET(CARD,COUNT)
        IF(.NOT.LAST(CARD)) GOTO 50
          CALL PUT(BREAK,DUMMY)
            CALL PUT(FINISH,DUMMY)
              RETURN
   50   IF(.NOT.LEFT(CARD)) GOTO 60
          CALL PUT(BREAK,DUMMY)
            CALL PUT(TITLE,CARD)
              GOTO 10
   60   IF(TOP(CARD)) GOTO 70
          CALL PUT(TEXT,CARD)
            GOTO 40
   70   CALL PUT(BREAK,DUMMY)
          CALL PUT(PARAGR,CARD)
            GOTO 40
C
      END
```

```
C
C**** INPUT AND OUTPUT ****
C
      SUBROUTINE GET(CARD,COUNT)
      INTEGER COUNT
      INTEGER CARD(80)
      READ(5,500) CARD
  500  FORMAT(80A1)
      COUNT = COUNT + 1
      RETURN
      END
C
      SUBROUTINE PUT(FLAG,LINE)
      INTEGER FLAG,LINE(80)
      WRITE(6,600) FLAG,LINE
  600  FORMAT(' ',A1,'  ',80A1)
      RETURN
      END
C
C**** PROPERTIES OF INPUT LINES ****
C
      LOGICAL FUNCTION LEFT(CARD)
      INTEGER CARD(80),SPACE
      INTEGER I
      DATA SPACE/' '/
      LEFT =.FALSE.
      DO 10 I = 1,12
        IF(CARD(I).NE.SPACE) RETURN
   10 CONTINUE
      LEFT =.TRUE.
      RETURN
      END
C
      LOGICAL FUNCTION RIGHT(CARD)
      INTEGER CARD(80),SPACE
      INTEGER I
      DATA SPACE/' '/
      RIGHT =.FALSE.
      DO 10 I = 40,80
        IF(CARD(I).NE.SPACE) RETURN
   10 CONTINUE
      RIGHT =.TRUE.
      RETURN
      END
C
      LOGICAL FUNCTION TOP(CARD)
      INTEGER CARD(80),SPACE
      DATA SPACE/' '/
      TOP = CARD(1).EQ.SPACE
      RETURN
      END
C
      LOGICAL FUNCTION LAST(CARD)
      INTEGER CARD(80),MINUS,SPACE
      DATA MINUS/'-'/,SPACE/' '/
      LAST = CARD(1).EQ.MINUS
     $         .AND. CARD(2).EQ.SPACE
      RETURN
      END
```

Fig. 6 A Fortran program for Fig. 5

solved simply by requiring a little discipline on the part of the user. For, what will be gained by allowing such a sloppy practice?

Fig. 6 shows a Fortran program coded on the basis of Fig. 5. Here, the subproblem has been slightly modified to allow two or more letters, each being terminated by a line starting with "–␣". This has been done for easier experimentation. The program can be stopped by an empty letter, i.e., an immediate succession of terminating lines. Fig. 7 shows a sample output.

Central to this program is the subroutine DISCR. The rest are a driver and support routines. The DISCR routine follows the pattern of Fig. 5 very closely. There are two parts corresponding to the Heading and the Text states. Each part begins with a call to the GET subroutine for reading a card. After a test for an end of data, it successively tests for the applicability of the state transition arcs that go out from that state, and yields an output using PUT.

The last call to PUT in DISCR could be dispensed with if the succeeding "GOTO 40" is changed into "GOTO 30". We don't do this, however. This will make the correspondence to Fig. 5 less clear. This type of a saving is best left to an optimizing compiler.

For coping with the six-letter restriction of Fortran, some of our identifiers are strained. Thus, TITLE and TEXT are the names of marks for a heading and a text lines. PARAGR and BREAK marks the start and end of a paragraph. FINISH marks the end of output.

LEFT, RIGHT, TOP, and LAST are conditions holding for a run of spaces to the left, a run of spaces to the right, a leading space, and an end of data, respectively.

The program as a whole is driven by a main program looping for successive letters. COUNT counts the number of input cards for each letter. The card starting with "–␣" for terminating the input is included in the count.

Some may object that the this program is not "structured" in the "narrow" sense. To this we answer that it is, we hope, still structured in the spirit of Dijkstra. The seemingly intertwined goto's are merely the result of compiling the "program" of Fig. 5 into a machine language, Fortran. For a related idea, see [4].

The program could be made more "structured" by rearranging the statements to form nested loops with

```
*                              LOCK WILLOW,
*                                SEPTEMBER 19TH.
*
*  DEAR DADDY,
(       SOMETHING HAS HAPPENED AND I NEED ADVICE.
   I NEED IT FROM YOU, AND FROM NOBODY ELSE IN THE
   WORLD.  WOULDN'T IT BE POSSIBLE FOR ME TO SEE YOU?
   IT'S SO MUCH EASIER TO TALK THAN TO WRITE; AND I'M AFRAID
   YOUR SECRETARY MIGHT OPEN THE LETTER.
)
*                                        JUDY.
*
*  P.S. I'M VERY UNHAPPY.
*
-
-
```

Fig. 7 A sample output.

occasional exits. It can be made even more "structured" if we note that a long run of spaces to the left in a line automatically implies at least one leading space. However, this is a really small point. The programs thus obtained are by far less understandable than the state transition diagram of Fig. 5. Besides, the additional knowledge about the relationships between the conditions should best be left unused: its use might well invalidate unforeseen areas of future application.

Certainly, we have the following

Exercise 5:

Use the ideas of this paper in a general purpose formatter suitable for typing a scientific paper.

Here, the additional knowledge mentioned above could still be used safely. However, this exercise indicates that there are areas that require similar but subtly differing considerations.

A similar exercise could be formulated for pretty printing of programs.

2.3 Question c—Right-Justifying the Text Lines

This part of the problem is old. Various solutions are known and actually used in production text formatters. No final solution, however, seems to exist for the situation considered here: no half spaces are available, and hyphenation is not attempted.

A commonly-used solution is to distribute the excess spaces evenly to the breaks of words, and to allocate those that still remain to as many word breaks that stand to the right (or left) of the line. Alternatively, the remainder could be allocated to the middle, or to both ends of the line.

A known drawback of this method is that it is apt to give rise to "islands" and "rivers" of blanks. It is also well-known that this difficulty can be alleviated by switching the direction of adding the remaining blanks alternately [3, 5].

A solution of our problem could use any of these known techniques, say, the alternating method. Gries [6] gives a detailed analysis about how this latter method might be implemented in a well-organized manner.

Before closing our discussion, however, we shall try another application of our principle: by requiring a little discipline on the part of the user, we wish to make him happy. In a sense, we are ourselves committing an overdesign here. The rest of this section might best be regarded as an exercise for ourselves.

```
**** INTELLIGENT ****

     ALICE  WAS BEGINNING  TO  GET  VERY TIRED  OF
SITTING BY HER SISTER  ON THE BANK,  AND OF HAVING
NOTHING TO DO :  ONCE OR TWICE SHE HAD PEEPED INTO
THE BOOK  HER SISTER WAS READING,  BUT IT  HAD  NO
PICTURES OR CONVERSATIONS IN IT,  "AND WHAT IS THE
USE OF A BOOK,"  THOUGHT ALICE,  "WITHOUT PICTURES
OR CONVERSATIONS?"

**** RIGHT FIRST ****

     ALICE WAS BEGINNING  TO  GET  VERY  TIRED  OF
SITTING BY HER SISTER ON THE BANK,  AND  OF  HAVING
NOTHING TO DO: ONCE OR TWICE SHE HAD  PEEPED  INTO
THE BOOK HER SISTER WAS READING,  BUT  IT  HAD  NO
PICTURES OR CONVERSATIONS IN IT,  "AND WHAT IS  THE
USE OF A BOOK,"  THOUGHT ALICE,  "WITHOUT  PICTURES
OR CONVERSATIONS?"

**** ALTERNATE ****

     ALICE WAS BEGINNING  TO  GET  VERY  TIRED  OF
SITTING  BY  HER SISTER ON THE BANK,  AND OF HAVING
NOTHING TO DO: ONCE OR TWICE SHE HAD  PEEPED  INTO
THE  BOOK  HER  SISTER  WAS READING,  BUT IT HAD NO
PICTURES OR CONVERSATIONS IN IT,  "AND WHAT IS  THE
USE  OF  A BOOK,"  THOUGHT ALICE,  "WITHOUT PICTURES
OR CONVERSATIONS?"

**** DELIMITER ORIENTED ****

     ALICE WAS BEGINNING  TO  GET  VERY  TIRED  OF
SITTING  BY HER SISTER ON THE BANK,  AND OF HAVING
NOTHING TO DO:  ONCE OR TWICE SHE HAD PEEPED  INTO
THE  BOOK  HER  SISTER  WAS READING,  BUT IT HAD NO
PICTURES OR CONVERSATIONS IN IT,  "AND WHAT IS THE
USE OF A BOOK,"  THOUGHT ALICE,  "WITHOUT PICTURES
OR CONVERSATIONS?"

                                   -- LEWIS CARROLL --
```

Fig. 8 Right-Justification Methods and Possibilities.

What we are going to do concerning Question c is formulated as follows:

Subproblem c:

Obtain a better-looking right-justified output by allocating more spaces to those semantical breaks of the text that are apparent from punctuation.

For a motivation, look at Fig. 8. The paragraph marked "INTELLIGENT" has been composed manually using the formatter's human intelligence. In general, it leaves more spaces in those breaks at which a pause is expected if the text is read aloud, but it does not carry this principle to the extreme, because it is still desirable that adjacent breaks are evenly spaced.

It would be nice if we could mechanize this "intelligent" justification, but of course we can't. Natural language understanding is one of the greatest unsolved problems of the computer industry; we should not open our Pandora's box.

Compare this "INTELLIGENT" paragraph with those marked "RIGHT FIRST" and "ALTERNATE" illustrating previously outlined methods. They are tolerable, but certainly less comfortable to human eyes when compared with the "INTELLIGENT" paragraph. We desire something in between.

The worst aspect of the known methods is that they make no distinction between ordinary breaks and the breaks that delimit larger units of the text. Hence, the Subproblem c.

In Fig. 8, the "DELIMITER ORIENTED" paragraph illustrates what we have in mind. We don't require our formatting program to know that the phrase "ON THE BANK" forms a unit, but do wish that, if a single excess space is to be allocated to a line, and the line has a single comma and no period, then the space is given to that comma. (See, however, a discussion in Section 2.3.2 about being conservative.)

In this connection, the following exercise for a very ambitious student deserves mention:

Exercise 6:

Devise a simplified grammar of the English language, and use it in a justification scheme for obtaining something better than our "DELIMITER ORIENTED" paragraph.

The student, however, should be forewarned that his solution must be such that the cost of computation is not excessive considering the relative importance of a beautiful right-justification of the text. He must be reminded that the "RIGHT FIRST" paragraph is after all tolerable.

We now briefly sketch a possible implementation of our delimiter oriented right-justification.

2.3.1 Decomposition of Subproblem c

Let us first identify the difficult points of our subproblem. The questions are:

Question c1:

How do we determine the kinds of the breaks of a given line?

Question c2:

How do we compute the number of spaces actually allocated to the breaks?

We wish to separate our implementation efforts cleanly into two portions corresponding to these questions. To do so requires some means for transmitting information about the kinds of the breaks from one module to another. For this we shall use a vector consisting of a small number of integers. They might eventually mean: the minimum number of blanks that must exist at the given break, and a first and a second relative weights for use when the number of excess spaces is small and large. However, at this stage of the design, we shall leave the significance of the components of the vector open. For, various man-machine considerations are coming in, and hence experimentation is a must. It is definitely unwise to fix the interface at this early stage.

A possible alternative is to transmit the information in a single code number indicating the kinds of breaks. If a higher sophistication is desired, however, there could be a great number of possible combinations. For example, we may wish to handle a period followed by two blanks (".␣␣") differently from a period followed by a right parenthesis, which in turn is followed by two blanks (".)␣␣"). The reader will see that it would become very hard to stabilize the design of the code numbers if this latter approach is taken.

2.3.2 To Determine the Kinds of Breaks—Question c1

Here we must make the computer behave as though it understands the material it has been told to type.

How do we find a fullstop in a line? It is not so simple as it might at first seem. There are at least four kinds of periods: (1) periods within a word; (2) periods indicating abbreviation; (3) dots; and (4) fullstops or sentence separators. Examples are: (1) I.E. (2) C. L. DODGSON (3) BUT . . . (4) THAT'S ALL.

Challenged in this way, a typical reaction from the student is to suggest a "rule" which regards those periods immediately following a single (capital) letter as abbreviations. To this we answer by another example: "WE ARE FRIENDS– –YOU AND I." Excluding "I."? Never! The author of this paper happens to be I. Kimura, not to advertise himself.

Our solution consists in asking the user to type with a bit of care. If he wishes that "C." of "C. L. DODGSON" be regarded as an abbreviation, he must be sure to put no more than one blank immediately after it. Our program will simply distinguish three classes: periods followed by no space, one space, and two or more spaces. This should be enough for our purposes. A user will get an unsatisfactory result if he is sloppy enough to type "C." at the end of a line, but it is his fault!

A similar problem exists between apostrophes and single quotes. For example, in

THE TELEVISION SHOUTED 'READ READERS' DIGEST.'

the first and the third occurrences of " ' " are single quotes. (We apologize to the Editor of Reader's Digest for an intentional misspelling.) We may wish to boost the breaks before and after them. The second occurrence is an apostrophe. The break after it should not be boosted.

Our solution is simply to prohibit the use of single quotes as quotation marks. The user is requested to use double quotes instead. If he does use " ' " for a quotation, he would get a tolerable but less pleasing result, which is, again, his fault.

Still another problem area is the distinction between hyphens and dashes. Thus, "A- AND B-REGISTERS" and "WE HAVE TWO REGISTERS– –A AND B" must be distinguished if hyphens are to indicate connection, and dashes, separation. Here, a case-by-case analysis based on the number of contiguous minus signs, and their relations to blanks gives a usable result.

Commas may be problematic: they may delimit from a very long clause down to a single word. Thus, in "WE HAVE SIX QUESTIONS A, B, C, . . . , F.", it would be undesirable if the blanks after the commas are markedly more in number than those after the ordinary words. This indicates that we must be conservative about commas. Otherwise, a control code would become necessary. In fact, we must be conserva-

tive not only with commas but also with virtually every side of our design.

As noted earlier, we may wish to put more spaces after ".)␣␣" than after ".␣␣". This may be handled by first calculating a vector for " . ", and then adding an increment vector for ")". However, a detailed design of our vector is best postponed until enough experimentation is over.

2.3.3 To Compute the Number of Spaces—Question c2

Here, we are facing a human reader. We are to give him comfort by allocating spaces appropriately.

If we were using a phototypesetter, in which the unit length is, say, one-sixth of the width of a thinnest character, our job would be very simple. We could use any of the methods shown in Fig. 8. In all cases the spacings would be very nearly of equal lengths. Each of the methods would appear as beautiful.

Actually, even half spaces are not available here. We can manipulate only hopelessly large units, the full spaces. Integer programming? Okay, try it. But certainly, it's too much.

Our solution here is as follows: First, a weight for each break of the given line is obtained as a function of (1) the weight parameters given in the vector mentioned above, and (2) the number of extra spaces per break. Second, the extra spaces are distributed to the breaks in proportion to the weights, but here the fractions of spaces are cut off. Finally, those extra spaces that still remain are successively distributed to the breaks, where those, of which the ratio of the weight to the number of actually distributed spaces is the highest, are given the highest priority. Those breaks having equal weights are treated in sequence from right or from left according to a prescribed order. The order is switched alternately each time a new line comes in.

3. Experience

Our sample problem has a rather long history. To describe some of it will be of help for understanding implications and possible uses of our problem. It is noted that the academic year begins in April in Japan.

3.1 Early Attempts

In the fall of 1971, the author for the first time used an earlier version of this problem for second-year undergraduates of the Department of Information Science (D.I.S.) of Tokyo Institute of Technology (T.I.T.). The version, to be solved in Fortran on a minicomputer, required that a commercial correspondence answering an inquiry be composed on the basis of data given in conversational mode for date, customer name, merchandise, quantity, price, and delivery. Right-justification was optional. Less ambitious students were allowed to use a fixed format into which the variable items were written appropriately. The students had no previous experience in programming except for another simple exercise requiring to print a table of square and cubic roots of 1, 2, . . . , 10. One of the students discovered the "alternate" method for himself.

Similar problems were used by the author and by one of his colleagues for a number of times in their classes subsequently. It is interesting to note that the author always used no control codes, while his colleague used a large set of them (leaving the design of the codes to the students).

3.2 A Readers' Contest in a Tutorial Journal

A problem almost identical to that of this paper was used by the author in August, 1973 in a contest for the readers of a tutorial journal [7]. A different letter was used as a sample, and the terminator was "–" rather than "–␣", but the main difference was that the problem explicitly stated the criterion for the choice of the winner: the solutions were to be evaluated mainly on the basis of the ease of understanding the program as combined with the accompanying documentation. It was also stated that the contest results were to be determined by a jury consisting of two or more people in an attempt to be fair.

The contest, as did all other contests given by the journal each month, allowed only 40 days or so to the contestants. The result was at the same time gratifying and disappointing. Three persons submitted solutions, and two of them made some efforts to cope with the man-machine aspects of the problem. However, their programs were basically brute-force ones, and contained several bugs. The bugs were of such a character that the author could not, and in fact had no intention to, complete debugging within about 50 days allowed to him in the presence of other duties. In [7] the contest results were published with a record of discussions of the jury, and complete listings of these two programs as modified by the author. A warning was included that there might be remaining bugs, and this was actually the case.

A third contestant intentionally misinterpreted the problem. He submitted a program which considered only that particular letter given as a sample in the problem.

3.3 Later Classroom Experiences

In the spring of 1976, the author used exactly the same problem as was used in the contest of [7] in a laboratory course for third-year undergraduates at D.I.S. of T.I.T. The following assignments were given to the students: (1) Deciphering the contestants' programs of the above; (2) Drawing Nassi-Shneiderman charts [8] for the contestants' programs; (3) Preparing test data that reveal the remaining bugs in the programs; (4) Evaluating the previous reports of other students, by exchanging them (borrowing the idea of Weinberg [9]); (5) Overall design of the student's own solution, by drawing a picture of data structures and writing a general pseudo-code; (6) Carefully redrawing on a large sheet what was

written in (5), which was used in oral presentations by a few students selected by a lottery; (7) Overall design and a pseudo-code preparation for Subproblem a; (8) Coding in Fortran for Subproblem a, with the method of Fig. 5 disclosed at this point; (9) Pseudo-coding a multipass algorithm for a combination of Questions a and b; (10) Recoding in Fortran for (9); (11) Incorporating right justification, with the algorithm left to the students; (12) Rewriting the Fortran program into a form using no work files; (13) Additional assignment on a separate topic: Snobol.

The pseudo-codes were written in Japanese in the standard, unromanized notation of the language. In the "nationalistic" view of the author, this form of a pseudo-code is one of the world's best means for designing programs.

In parallel with these assignments the following topics were covered: Guy de Balbine's structuring engine [10]; Coroutines and multipass algorithms; A brief summary of "The Elements of Programming Style" by Kernighan and Plauger [11]; Top-down design explained using Mills' example on supermarket checkout control [12]; JCL on an OS, files; The Snobol language. The above corresponded to a fifteen-week semester with each week containing 100 minutes of lecture and 400 minutes of laboratory work.

Although it is difficult to give a long-term assessment now, it is at least true that the course evoked the students' unusual enthusiasm, especially when the algorithm of Fig. 5 was disclosed to them. All in all, the author feels that he was successful. (However, it is noted that the use of the contestants' programs in the initial stages had some negative effects. Some of the less diligent students imitated bad points of these programs in writing theirs.)

In the fall of 1976, the same was tried at the Department of Information and Computer Sciences, Osaka University for third-year undergraduates. This time twelve 100-minute classes in three chunks each of which lasted two days were available to the author. Here, our problem was covered less extensively, with more emphasis on the decomposition of the problem, and the right-justification algorithms. Exercise 4 was used as one of the assignments. Apart from our problem, something like what Kernighan and Plauger did in their book [11] was done for a sample Fortran program obtained from a Japanese textbook. Again, it is difficult to assess now, but the author feels that he was more successful than previously expected.

4. Conclusion

This paper discussed a text formatting problem. However, it was merely an example. A single text formatting problem is clearly not enough. For programmer's training we need a great many other problems. It is very important to accumulate similar problems, and they can be found in virtually every field of computer application. In fact, any nontrivial programming project involves some compromises between the convenience of the user and the implementor.

To accumulate these problems, however, requires much more than an author's effort. By this paper the author wishes to advertise, perhaps with a fanfare, the importance of thus accumulating similar problems.

Acknowledgement

The author is indebted to the following persons for discussions and comments: Eiiti Wada, Masako Takahashi, Hirohiko Nishimura, Ken Hirose, Takashi Tsuji and Kojiro Kobayashi. He is also indebted to Gerald M. Weinberg for encouragements, and to Joseph C. Berston for suggestions regarding presentation.

References

1. PETERSON, W. W. *An Introduction to Programming Languages*, Prentice-Hall, New Jersey, 1974.
2. ABRAHAMS, P. W. On Realism in Programming Examples, *SIGPLAN Notices* **11**, 2 (Feb., 1976), 17–19.
3. GIMPEL, J. F. *Algorithms in SNOBOL* **4**, John-Wiley, New York, 1976, Chap. 10.
4. TAKAHASI, H. Programs and the GOTO, (in Japanese), In "*Suri to Gensho*" (Mathematical Reasoning and Physical Phenomena), Iwanami Shoten, Tokyo, 1975, 214–218; originally in *Sugaku Seminar* **11**, 8 (Aug. 1972), 38–40.
5. GRISWOLD, R. E. *String and List Processing in SNOBOL* **4**, Prentice-Hall, Englewood Cliffs, New Jersey, 1975, Section 6.2.
6. GRIES, D. An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs, *IEEE Transactions on Software Engineering*, SE-2, 4 (Dec., 1976), 238–244.
7. KIMURA, I. Nano-Pico Kyoshitsu (a readers' contest, in Japanese), *bit* **5**, 9 (Aug., 1973), 63 and **5**, 12 (Oct. 1973), 63–71.
8. NASSI, I. AND SHNEIDERMAN, B. Flowchart Techniques for Structured Programming, *SIGPLAN Notices* **8**, 8 (Aug., 1973), 12–26.
9. WEINBERG, G. M. Personal communication.
10. DE BALBINE, GUY. Better Manpower Utilization Using Automatic Restructuring, *Proc. 1975 National Computer Conference, AFIPS* **44**, (May, 1975), 319–327.
11. KERNIGHAN, B. W. AND PLAUGER, P. J. *The Elements of Programming Style*, McGraw-Hill, New York, 1974.
12. MILLS, H. D. (with appendix by LINGER, R. C.) On the Development of Systems of Men and Machines, In Hackel ed., *Programming Methodology, Springer Lecture Notes on Computer Science* **23**, (1975), 1–10.