literate programming

Moderated by
Christopher J. Van Wyk

PROCESSING TRANSACTIONS

Moderator's Introduction to Column 2

Judging from the amount of mail I received about Column 1, the readership of "Literate Programming" is gratifyingly large. Part of the reason for the large mail volume, however, just might be that the program in Column 1 contains a serious error.

The first letter I received about the column came from Michael Shook: "The sort in the routine `printwords` requires a bucket for each possible frequency of occurrence, not for each different word. The error would be exercised by input of exactly two identical words." I asked David Hanson to run his program on the input "`hello hello`"; he quickly reported back that his program had dumped core.

In his review of Hanson's program, John Gilbert had said, "'`total`' violates the rule that a global variable should have the most descriptive name possible." This is a strong hint to why this error was hard to catch: Although Section 6 of the program points out that "`total` ... counts the number of distinct words in the table," the name `total` is misleading enough that one could forget this in Section 7 and imagine that `total` is the total number of words in the input.

Many others wrote to point out that Hanson's program uses an inefficient hash function. Hanson measured the performance of his program on the sample input, and found that it left 3000 hash slots empty and three hash slots contained more than 100 items. Hanson experimented with Hans Boehm and Joe Warren to devise a better hash function; on the same sample input, it leaves only 1500 hash slots empty, and the longest hash chain it constructs has only seven items. Readers interested in more details can write directly to Hanson at the Department of Computer Science, Princeton University, Princeton, NJ 08544.

There was a very tiny overlap between correspondents who noticed the bug and those who noticed the poor performance of the hash function. How much easier it is to read a program attentively to assess its efficiency *or* its correctness than to read a program with an eye to *both*, not to mention its literary qualities.

In reply to last year's "Programming Pearls" about literate programming, Richard Botting wrote to suggest that very different solutions would be proffered by a programmer trained in Michael Jackson's method of program design. I was intrigued enough by his letter to want to find out more about this method and finally settled down to read Jackson's 1975 book, *Principles of Program Design* [4]. The book presents the method through a sequence of problems. The second problem, "Printing Invoices," is described as follows:

> A serial master file contains customer name and address records, arranged in ascending sequence by customer number. Another serial file contains billable item records, arranged in ascending sequence by date within invoice number within customer number.
>
> These two files are to be used to print invoices. There may be more than one invoice for a customer, but some customers will have no invoices. Due to punching errors, there may be billable item records for which no customer record exists; these are to be listed on a diagnostic file of messages. [4, p. 7]

I set the book aside and sketched this pseudocode solution:

```
customer ← first record from the master
           file
item ← first billable item
repeat
  while customer has a smaller number
        than item,
    customer ← next record from the
               master file
  if customer has a larger number than
     item,
     add item to the diagnostic file;
  otherwise,
    print the name and address data
        for customer
    repeat
      print the item
      item ← next billable item
    until item and customer have
        different numbers
until all billable items have been
  processed
```

I returned to the book to find a very similar pseudocode solution on pages 7 and 8. This left me feeling happily self-satisfied until I read further and found that "the structure is utterly wrong" and "absurd."

Unfortunately the book does not seem to contain a solution that Jackson would deem correct or reasonable. Thus, I posed this problem to Jackson himself. The problem has connections to the word-counting application of the first column: After extracting the words from a document and alphabetizing them, processing similar to this could be used to tabulate word frequencies and find spelling errors.

Jackson's Solution

1. The problem is Problem 2 mentioned earlier.

```
{COMMENTARY:
  The solution described below uses the JSP design
  method; for a general account of JSP, see [1] and
  [4]. It is only an illustrative example: There are
  many report writers that can solve this problem in
  a few lines. The notations used are chosen to min-
  imize the amount of prior explanation needed for
  most readers.
    The description of the program is divided into
  sections. Each section ends with a commentary
  (e.g., this is the commentary of Section 1) contain-
  ing notes and remarks that are intended chiefly to
  explain the JSP method and how it is being used
  to readers for whom it is unfamiliar.
END COMMENTARY}
```

2. We are given the following TYPE declarations for the files:

```
⟨Type declarations⟩ =
  t_cusno = string[6];
    {customer number}
  t_invno = string[8];
    {invoice number}
  t_date = RECORD
             year: 0..99;
             month: 1..12;
             day: 1..31;
           END;
  t_cus_rec =
    RECORD
        {customer name and address record}
      cus_cusno: t_cusno;
        {customer number of this record}
      cus_name: string[15];
      cus_street: string[15];
      cus_state: string[15];
    END;
  t_itm_rec =
    RECORD
        {billable item record}
      itm_cusno: t_cusno;
        {number of customer to bill}
      itm_invno: t_invno;
        {number of invoice containing item}
      itm_date: t_date;
        {date item was delivered}
      itm_desc: string[20];
        {item description}
      itm_value: 0..999999;
        {item value in cents}
  END;
```

We are also given the following VAR declarations for the files:

```
⟨Variable declarations⟩ =
  cus_file: FILE OF t_cus_rec;
    {Customer name and address file}
  itm_file: FILE OF t_itm_rec;
    {Billable item file}
  diag_file: text;
    {Diagnostic file for erroneous
     billable items}
  inv_file: text;
    {Invoice file}
```

Both cus_file and itm_file contain a last record at the end of the file whose value of cus_cusno and itm_cusno, respectively, is the string of 9's. This record is viewed as an eof marker and contains no significant data. We are required to write a similar record at the end of the diagnostic file.

```
⟨Constant declarations⟩ =
  eof_cusno = '999999';
```

```
{COMMENTARY:
  In a realistic data-processing application, file han-
  dling would use techniques adopted as standard
  for the organization. See [2] for some good ideas
  on techniques for use with Pascal implementa-
  tions. In this instance, as in many others, we are
  greatly interested in brevity of exposition. That is
  why we have allowed ourselves text files for the
  outputs, thus avoiding further record and other
  declarations; the eof marker records, thus avoid-
  ing the horrors of Pascal's failure to provide short-
  circuit Boolean evaluation; the unrealistically
  simplified invoice format (which lacks even pagi-
  nation); and other expedients. A realistic program
  that could be regularly used for this application in
  a data-processing environment would be at least
  2500 lines of Pascal text.
END COMMENTARY}
```

3. The file structures (grammars) are apparently as follows:

```
⟨itm_file⟩ ::= ⟨i_f_cus_group⟩*;
  {ascending sequence by itm_cusno}
⟨i_f_cus_group⟩ ::= ⟨i_f_inv⟩+;
  {ascending sequence by itm_invno}
⟨i_f_inv⟩ ::= ⟨itm_rec⟩+;
  {nondescending sequence by itm_date}
```

An i_f_cus_group is an *item file customer group*, containing all the item records for one customer. An i_f_inv is an *item file invoice* group, containing all the item records for one invoice. In the Backus-normal form (BNF) used, "*" means "0 or more occurrences," and "+" means "1 or more occurrences."

```
⟨cus_file⟩ ::= ⟨cus_rec⟩*;
  {ascending sequence by cus_cusno}
```

For the files itm_file and cus_file, the terminals of the grammars represent data records that can be read by Pascal get or read operations.

```
⟨diag_file⟩ ::= ⟨d_f_rec⟩*;
  {nondescending sequence by itm_date
   within ascending sequence by
   itm_invno within cusno}
⟨inv_file⟩ ::= ⟨v_f_cus_group⟩*;
  {ascending sequence by cusno}
⟨v_f_cus_group⟩ ::= ⟨v_f_inv⟩+;
  {ascending sequence by itm_invno}
⟨v_f_inv⟩
  ::= ⟨inv_hdr⟩⟨inv_body⟩⟨inv_total⟩;
⟨inv_body⟩ ::= ⟨inv_itm⟩+;
  {nondescending sequence by date}
```

A v_f_cus_group is an invoice *file customer group*, containing all invoices for one customer. A v_f_inv is an invoice *file invoice*, which consists of an *invoice header*, an *invoice body*, and an *invoice total* line. The inv_body contains inv_itms, which are *invoice item* lines.

For the files diag_file and inv_file, the terminals of the grammars represent groups of data that can be written by a consecutive set of write or writeln operations.

{COMMENTARY:

We have added to the specification in the grammars and in the comments on the grammars for diag_file and inv_file: We have not only specified some structure within invoices in inv_file, but also specified that both the invoices (with their contents) and the diagnostic messages are to appear in the order that is most convenient for our program (i.e., the order of the input files). We certainly need to check this and the rest of the file structure specification with the user.

The first JSP design step is always to define the structures (grammars) of the input and output data, and to check them with the specifier or customer. The usual JSP notation is diagrammatic: Here we are using a form of BNF merely because it is likely to be more familiar to most readers. The JSP notation can be seen in Figure 1, in which the leftmost tree shows the structure of inv_file; in the diagrammatic notation, no distinction is made between iterations containing zero or more occurrences and those containing one or more occurrences.

The diagrammatic notation has a number of advantages, including the ease with which it can be taught and understood. One very important advantage is its ability to clarify relationships among

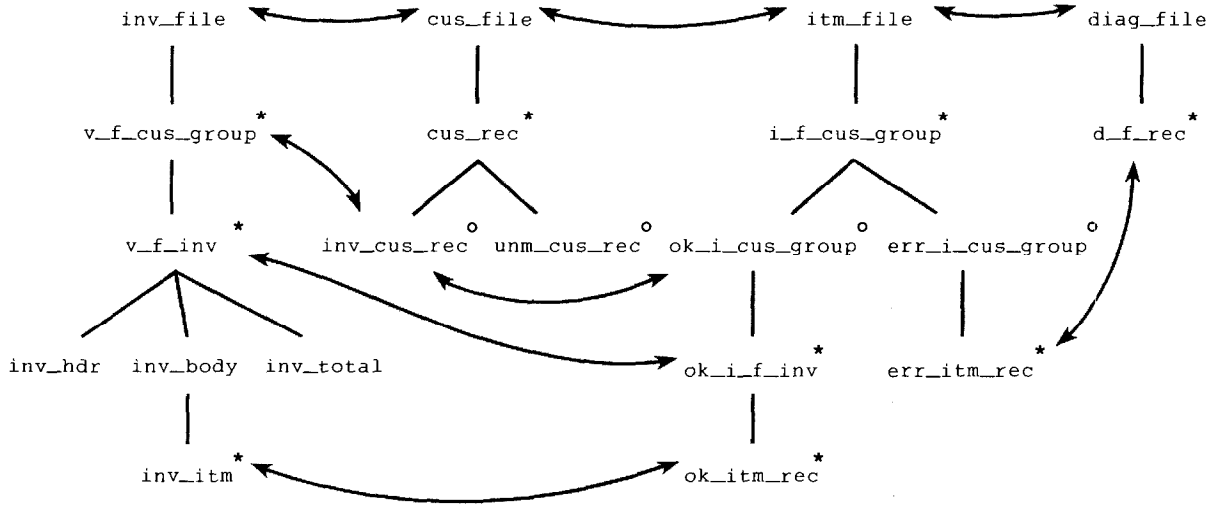

FIGURE 1. Data Structures and Correspondences

structures, and we use it for this reason in Section 6.
END COMMENTARY}

4. Now we must form a program structure that correctly reflects all of the data structures and the correspondences among their components. At the top level, we have the following correspondence:

```
(cus_file, itm_file,
 diag_file, inv_file);
```

Therefore, the top level of the program structure is as follows:

```
begin
  {cus_file, itm_file,
   diag_file, inv_file}
:
end.
  {cus_file, itm_file,
   diag_file, inv_file}
```

However, we can progress no further: `i_f_cus_group` does not correspond to `cus_rec`, nor does `v_f_cus_group`, because there may be `cus_recs` for which there are no billable items and no invoice, and there may be billable items for which there is no `cus_rec`. We are forced to reconsider the data structures.

{COMMENTARY:
The idea of *correspondence* is central to JSP. `inv_file` and `itm_file` correspond because each execution of the program consumes one `itm_file` instance and produces one `inv_file` instance; generally, two data components correspond if their occurrences can be put into 1-to-1 correspondence without reordering.

We have, in fact, made a false start: The data structures we defined in Section 3 are inadequate for our purpose. It is an important aim of the JSP method to reveal errors as early as possible by preventing further progress on an erroneous basis. In practice, no experienced JSP designer would have made the error shown here: It is intended only to illustrate this point about revealing design errors.

The idea of correspondence may be compared to the notion of parallel composition in CSP [3]. We may regard a data structure as a trace set if we consider each data component to have a **begin** and an **end** event. If two data components correspond, their **begin** events are one common event, and their **end** events are another. In forming a JSP program structure, we are choosing one of the allowable interleavings of the traces of the data structures and fixing this choice in the program design. Sometimes we will find it necessary to avoid making such a choice, where data structures are incompatible: This point is briefly discussed at the end of Section 5. There are, of course, other situations in which we would want to keep the interleaving of traces indeterminate; these are the subject matter of JSD (e.g., see [1]) rather than JSP.
END COMMENTARY}

5. After reconsideration of this "matching" problem, we produce the following grammars:

```
⟨cus_file⟩ ::= ⟨cus_rec⟩*;
  {ascending sequence by cus_cusno}
⟨cus_rec⟩
  ::= ⟨unm_cus_rec⟩|⟨inv_cus_rec⟩;
```

An `unm_cus_rec` is an *unm*atched `cus_rec` (no billable items); an `inv_cus_rec` has at least one *inv*oice for the `cus_rec` (i.e., it has at least one `itm_rec`).

```
⟨itm_file⟩ ::= ⟨i_f_cus_group⟩*;
  {ascending sequence by itm_cusno}
⟨i_f_cus_group⟩
  ::= ⟨ok_i_cus_group⟩
      |⟨err_i_cus_group⟩;
⟨ok_i_cus_group⟩ ::= ⟨ok_i_f_inv⟩+;
  {ascending sequence by itm_invno}
⟨ok_i_f_inv⟩ ::= ⟨ok_itm_rec⟩+;
  {nondescending sequence by itm_date}
⟨err_i_cus_group⟩ ::= ⟨err_itm_rec⟩+;
  {nondescending sequence by itm_date
   within ascending sequence by
   itm_invno}
```

An `ok_i_cus_group` is an *ok group* of *i*tems for *cus*tomer whose record is present in `cus_file`; an `err_i_cus_group` is an *err*oneous *cus*tomer *group* of *i*tems for which no customer record is present in `cus_file`. We do not need the structuring by invoice within `err_i_cus_group`, because the diagnostic file is essentially an iteration of item records in the order they happen to occur in the input.

{COMMENTARY:
We might have found that our failure to form a program structure was due to the presence of a "structure clash," not merely to our inadequate description of the structures. In that case, we would have identified the type of clash, and respecified the program as two or more processes communicating by message streams, the resulting network being determined by the type of clash found. This technique of decomposition into two or more processes ensures that the processes to be designed in a JSP program are always structurally simple: Any structural complexity is resolved by the decomposition. JSP includes an implementation technique (program inversion) by which process communication can be easily programmed in the most unpromising languages such as Cobol or Assembler.
END COMMENTARY}

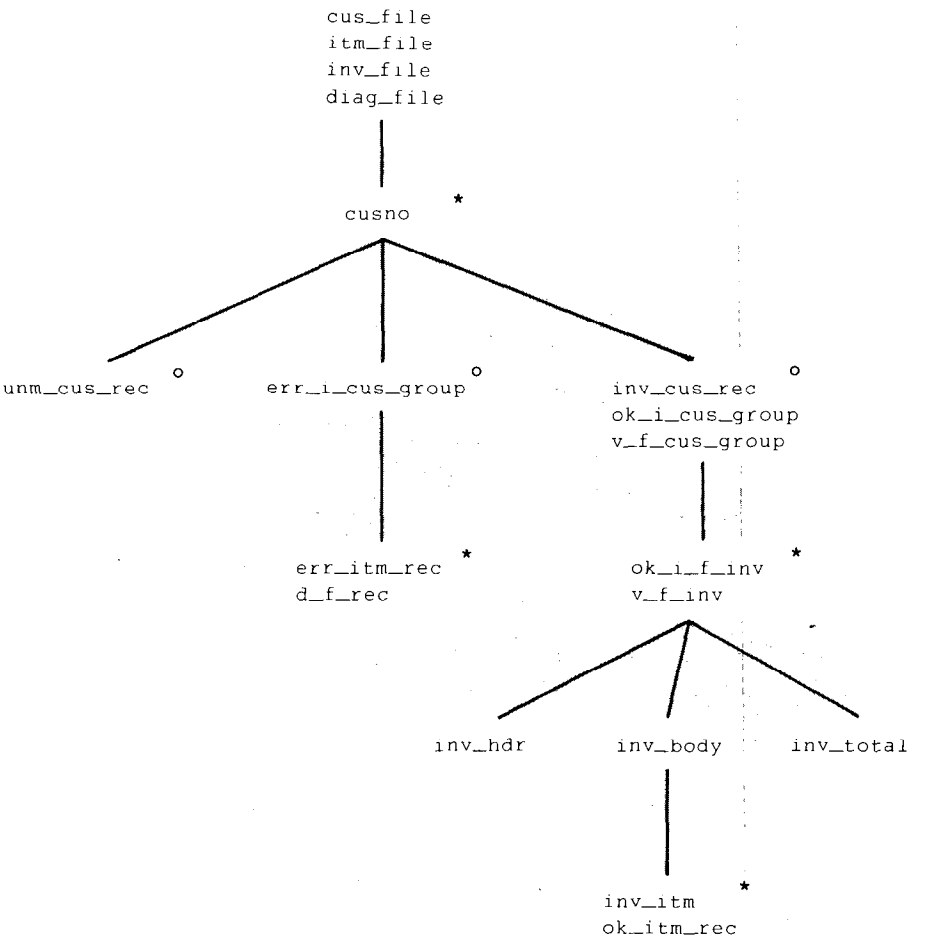

FIGURE 2. Program Structure Derived from Data Structures

6. We now have these correspondences:

```
(cus_file, itm_file,
 diag_file, inv_file);
(inv_cus_rec, ok_i_cus_group,
 v_f_cus_group);
(unm_cus_rec);
(ok_i_f_inv, v_f_inv);
(ok_itm_rec, inv_itm);
(err_itm_rec, d_f_rec);
```

These correspondences are shown in Figure 1, and the resulting program structure is shown in Figure 2 in the Pascal outline shown in Figure 3.

The correctness of this program structure with respect to the data structures is easily verified.

```
{COMMENTARY:
  Verification of the program structure requires
  (1) examination and confirmation of each corre-
  spondence it implies, and (2) a check that each
  data structure can be recovered from the program
  structure by pruning the program tree and apply-
  ing certain simple equivalences between regular
  expressions (e.g., a selection of one part is identi-
  cal to that part).
    The style of Pascal may seem cumbersome; its
  virtue is that it allows us to express each part of
  the program structure in a consistent way that
  documents its association with the corresponding
  data parts. In JSP a suitable pseudocode known as
  Structure Text is used for this purpose.
END COMMENTARY}
```

7. The list of operations to be executed by the program, with declarations of required program variables, is as follows:

```
{11} rewrite(inv_file);
{12} (write inv_hdr from cusno,
       cus_file^.cus_name, invno)
```

```
{13} ⟨write inv_total from itotal⟩
{14} ⟨write inv_itm from itm_file^⟩

⟨Variable declarations⟩ +=
  cusno:t_cusno;
    {current customer number, not
     eof_cusno}
  invno:t_invno;
    {current invoice number}
  itotal:integer;
    {total value for one invoice}

{21} rewrite (diag_file);
{22} ⟨write d_f_rec from itm_file^⟩
{23} ⟨write eof marker to diag_file⟩

{31} if cus_file^.cus_cusno
        < itm_file^.itm_cusno then
      cusno := cus_file^.cus_cusno
    else
      cusno := itm_file^.itm_cusno;
        {determine next value of
         cusno for processing by the
         program}

{41} itotal := 0;
{42} itotal
       := itotal + itm_file^.itm_value;

{51} invno := itm_file^.itm_invno;

{61} reset(cus_file);
{62} get(cus_file);

{71} reset(itm_file);
{72} get(itm_file);

{COMMENTARY:
```

JSP aims to separate model from function. We may think of the program structure as a model of the problem universe in the form of a composition of all data structures; we must now consider the program's function: What it should do as it traverses this structure.

We therefore begin by listing the most obvious operations: those that are needed to produce the output file records (operations {11}-{14} and {21}-{23}). These give rise to a need for local variables (cusno, invno, and itotal), which require operations to assign their values (operations

```
begin {cus_file, itm_file, diag_file, inv_file}
  while ⟨Another cusno value is present⟩ do
    begin {one cusno value}
      if ⟨Unmatched cus_rec⟩ then
        begin {unm_cus_rec}
        end   {unm_cus_rec}
      else
      if ⟨Unmatched billable item group⟩ then
        begin {err_i_cus_group}
          while ⟨Another err_itm_rec for this cusno⟩ do
            begin {err_itm_rec, d_f_rec}
            end;  {err_itm_rec, d_f_rec}
        end {err_i_cus_group}
      else
        begin {inv_cus_rec, ok_i_cus_group, v_f_cus_group}
          while ⟨Another ok_i_f_inv for this cusno⟩ do
            begin {ok_i_f_inv, v_f_inv}
              begin {inv_hdr}
              end;  {inv_hdr}
              begin {inv_body}
                while ⟨Another ok_itm_rec for this invoice⟩ do
                  begin {ok_itm_rec, inv_itm}
                  end;  {ok_itm_rec, inv_itm}
              end;  {inv_body}
              begin {inv_total}
              end;  {inv_total}
            end; {ok_i_f_inv, v_f_inv}
        end; {inv_cus_rec, ok_i_cus_group, v_f_cus_group}
    end; {one cusno value}
end. {cus_file, itm_file, diag_file, inv_file}
```

FIGURE 3. The Pascal Outline for Figure 2

```
⟨Program Text⟩ =
begin {cus_file, itm_file, diag_file, inv_file}
  {11} rewrite(inv_file);
  {21} rewrite(diag_file);
  {61} reset(cus_file);
  {71} reset(itm_file);
  while ⟨Another cusno value is present⟩ do
    begin {one cusno value}
      {31} if cus_file^.cus_cusno < itm_file^.itm_cusno then
             cusno := cus_file^.cus_cusno
           else
             cusno := itm_file^.itm_cusno;
               {determine next value of cusno
                for processing by the program}
    if ⟨Unmatched cus_rec⟩ then
      begin {unm_cus_rec}
        {62} get(cus_file);
      end {unm_cus_rec}
    else
    if ⟨Unmatched billable item group⟩ then
      begin {err_i_cus_group}
        while ⟨Another err_itm_rec for this cusno⟩ do
          begin {err_itm_rec, d_f_rec}
            {22} ⟨write d_f_rec from itm_file^⟩
            {72} get(itm_file);
          end; {err_itm_rec, d_f_rec}
      end {err_i_cus_group}
    else
      begin {inv_cus_rec, ok_i_cus_group, v_f_cus_group}
        while ⟨Another ok_i_f_inv for this cusno⟩ do
          begin {ok_i_f_inv, v_f_inv}
            {51} invno := itm_file^.itm_invno;
            {41} itotal := 0;
            begin {inv_hdr}
              {12} ⟨write inv_hdr from cusno,
                     cus_file^.cus_name, invno⟩
            end; {inv_hdr}
            begin {inv_body}
              while ⟨Another ok_itm_rec for this invoice⟩ do
                begin {ok_itm_rec, inv_itm}
                  {14} ⟨write inv_itm from itm_file^⟩
                  {42} itotal := itotal + itm_file^.itm_value;
                  {72} get(itm_file);
                end; {ok_itm_rec, inv_itm}
            end; {inv_body}
            begin {inv_total}
              {13} ⟨write inv_total from itotal⟩
            end; {inv_total}
          end; {ok_i_f_inv, v_f_inv}
        {62} get(cus_file);
      end; {inv_cus_rec, ok_i_cus_group, v_f_cus_group}
    end; {one cusno value}
  {23} ⟨write eof marker to diag_file⟩
end. {cus_file, itm_file, diag_file, inv_file}
```

FIGURE 4. The Program Structure with Allocated Operations

{31}, {41}, {42}, and {51}); these in turn give rise to a need for operations to obtain data from the input files.

Of course, it is only because the program structure is simple that we can determine an operation list in this simple way, but we aim to avoid complexity of program structure by the recognition and resolution of structure clashes mentioned in Section 5.

Inevitably, the separation between model and function is far from perfect: We make the model with some function in mind and describe the function (as a list of operations) with the model in mind. But the separation is still salutary, and gives an added check on our design by forcing us to consider the problem from several different points of view.

END COMMENTARY}

8. We now allocate these operations to the program structure. The program structure with allocated operations is shown in Figure 4.

{COMMENTARY:

Allocating the operations is an important step: If we find difficulty in any allocation, we should suspect that our program structure, the problem model, is inadequate to support the required function. For example, if we are required to list the customers for whom no invoice item is present, we could allocate the appropriate `write` operation to the component `unm_cus_rec`: The model supports this function. But we could not allocate an operation to diagnose an empty item file, because there is no program component corresponding to an empty item file: The model does not support this function.

Operations are allocated by considering each operation in the list in turn and deciding where in the program structure it must be executed; they are not allocated by mentally executing the program and deciding what it should do next.

Readers who find this idea difficult or bizarre may think of the analogy of designing a one-pass compiler by forming the structure required for syntax analysis and embedding the "semantic routines" in this structure.

The regime for the reading of input files is a simple "read ahead one record" scheme. This works for our present problem because of the nature of the file grammars. The initial `read` for each input file is implicit in the `reset` operation; later `read` operations are allocated at the end of each program component that consumes a record of the file.

We have deferred definition of the conditions in the program text until a later step (see Section 10).

END COMMENTARY}

```
⟨write inv_hdr ...⟩ =
  page(inv_file);
  writeln(inv_file,  'CUSTOMER NO', cusno,' :', cus_file^.cus_name);
  writeln(inv_file,  'INVOICE NO', invno,' :', cus_file^.cus_street);
  writeln(inv_file,  ' ':21, cus_file^.cus_state);
  writeln(inv_file);
  writeln(inv_file;  'DATE',' ':4, 'DESCRIPTION', ' ':4, 'VALUE');
  writeln(inv_file);

⟨write inv_total ...⟩ =
  writeln(inv_file);
  writeln(inv_file, 'INVOICE TOTAL = $', (itotal DIV 100):1, '.',
                                          (itotal MOD 100):2);

⟨write inv_itm ...⟩ =
  WITH itm_file^ do
    writeln(inv_file), itm_date.day:2, '/', itm_date.month:2, '/',
      itm_date.year:2, ' ':4, itm_desc:20, ' ':4, '$',
      (itm_value DIV 100):4, '.', (itm_value MOD 100):2);

⟨write d_f_rec ...⟩ =
  WITH itm_file^ do
    writeln(diag_file, itm_cusno:6, ' ':2, itm_invno:8, ' ':2,
      itm_date.year:2, itm_date:month:2, itm_date.day:2, ' ':2,
      itm_desc:20, ' ':2, itm_value:6);

⟨write eof ...⟩ =
  writeln(diag_file, eof_cusno:6);
```

FIGURE 5. Expansions of Operations

9. The expansions of the operations not yet fully defined are shown in Figure 5, page 1007.

```
{COMMENTARY:
  We have made many simple implementation decisions here, and earlier, purely with a view toward brevity of exposition. The main theme of JSP design is achievement of the correct program structure; if the program structure is right, the rest follows with relative ease.
    We might, of course, have expanded some operations into calls of procedures provided by some bottom-up design.
END COMMENTARY}
```

10. The expansions of the conditions not yet defined are as follows:

```
⟨Another cusno ...⟩ =
  ((cus_file^.cus_cusno ⟨⟩ eof_cusno)
   or
   (itm_file^.itm_cusno ⟨⟩ eof_cusno))

⟨Unmatched cus_rec⟩ =
  ((itm_file^.itm_cusno ⟨⟩ cusno) and
   (cus_file^.cus_cusno = cusno))

⟨Unmatched billable ...⟩ =
  ((cus_file^.cus_cusno ⟨⟩ cusno) and
   (itm_file^.itm_cusno = cusno))

⟨Another err_itm_rec ...⟩ =
  (itm_file^.itm_cusno = cusno)

⟨Another ok_i_f_inv ...⟩ =
  (itm_file^.itm_cusno = cusno)

⟨Another ok_itm_rec ...⟩ =
  ((itm_file^.itm_cusno = cusno) and
   (itm_file^.itm_invno = invno))

{COMMENTARY:
  The conditions are defined by considering static properties of the data structures. For example, Unmatched cus_rec is a group defined by a cusno value for which there is a cus_rec but no itm_rec. Note that we have ignored some optimization possibilities here: We could delete the second conjuncts for Unmatched cus_rec and Unmatched billable if we wished, but such "improvements" usually cost more over the program life in unnecessary errors than they save in execution time.
    Had our file structures not permitted the simple "read one record ahead" regime, we would have discovered it at this step from an inability to express the condition on some selection or iteration without looking more than one record ahead. We would then have considered how to deal with the difficulty without damaging the program structure, either by using a "multiple read ahead" regime for input, or by a JSP three-stage technique of introducing backtracking into the program behavior.
END COMMENTARY}
```

11. We need now to put the complete program text together:

```
PROGRAM problem2
    (cus_file, itm_file,
     diag_file, inv_file);
  CONST
    ⟨Constant decl ...⟩
  TYPE
    ⟨Type decl ...⟩
  VAR
    ⟨Variable decl ...⟩
⟨Program Text⟩
```

Michael Jackson
Michael Jackson Systems Limited
22 Little Portland Street
London W1N 5AF, England

Review of Jackson's Solution

[David Wall is a member of the technical staff at Digital Equipment Corporation's Western Research Laboratory.]

If you travel a bit in Europe, you will notice a frustrating fact: Half of the great cathedrals and castles of Europe are covered with scaffolding. Each has an elegant, beautiful structure that is discernible through the planks and ropes, but terribly hard to see; you do better to buy the picture postcard from a souvenir stand.

I was reminded of this by Jackson's program. Jackson advocates a particular approach to programming, and therefore finds it reasonable to document his program by including all the scaffolding that he used to construct it. In theory, this should make it easier to understand or modify the program, because it ties the program text to the first principles that led to it. In practice, I am sorry to say that it obscures the program considerably. I read Jackson's book [4] carefully before I read his essay, and was still unable to get any clear idea of the program's structure until I had stripped out all of the "{inv_cus_rec, ok_i_cus_group, v_f_cus_group}" comments that relate his program text to his file structure diagrams. Call me reactionary, but I still think comments should be in English and should not appear on nearly every line.

Jackson's approach reminded me strongly of formal verification. Like literate programming, verification is

best done while writing the program, with the verification driving and being driven by the programming itself. A program and verification produced simultaneously may have a higher probability of correctness than a program alone, because the programmer has been forced to state the program in two different formal mechanisms, namely, logic and code; a mistake in one form may be caught in the other. And providing a formal verification along with a program may even help document the program. But it does not make the program more readable, and the increased burden of notation may even make it less readable.

Regardless of what technique one uses to write a program, exposing that technique is unlikely to be the best way to present the solution. Mathematicians have long known this; to discover a proof, one goes down many blind alleys and applies a lot of questionable analogies, none of which show up in the actual proof. What shows up in the proof is exactly as much structure as needed, with a minimum of notation; the same thing should be true of a literate program.

None of this is to say that Jackson's technique is flawed; on the contrary, I was impressed that it led apparently automatically to the best structure for the program. Jackson's program and Van Wyk's pseudocode have different structures because of their disagreement over the correct unit of iteration. The item correctly processed by one iteration is not one or more records from one or the other of the input files, but rather one "customer," who is represented by records from both files.

In his book Jackson proposes four modifications to this problem, each of which is easy to make to his program, but harder to make to Van Wyk's. All four modifications are stated as things to do for each customer, such as "Print on the diagnostic listing the customer numbers of those customers for whom at least one invoice has been printed." My first thought was that they had been cooked to justify the program structure, but I was wrong. I thought of several other modifications that would require large changes to Jackson's program structure, but all of them seemed to me to be a large change to the problem statement as well; the programmer would be justified in saying, "Wait, that changes the whole problem." I concluded that the decision to iterate over customers was correct.

This conclusion was consistent with another approach to programming that has intrigued me ever since I heard of it. Michael Clancy has been teaching programming for many years and has formulated a heuristic called "simple decomposition." The idea is that the technique of top-down decomposition needs a companion technique for deciding how extreme a decomposition to make, and that the right answer is to decompose as little as possible each time. Clancy puts it this way:

> If you have a choice among ways to decompose a problem, make the choice as follows:
>
> 1. Choose a sequence rather than a selection or a loop.
> 2. Choose a selection rather than a loop.
> 3. Choose the loop that iterates over the largest possible chunk of data.

The third rule is the one that caught my attention. It is what kept the word-counting programs of the previous columns from having structures like

```
for each letter in the input file, do
  if this letter is eof then
    print the most common words in data
      structure
  elseif this is not the last letter
         in a word then
    accumulate the letter in the current
      word
  else
    record the accumulated word in data
      structure
  end
end.
```

In the word-counting programs, the largest chunk of input with which we are concerned is a word, not a letter. In the invoice-printing program, the largest chunk of input is the customer record and billing items associated with a single customer.

I would have guessed that Jackson would agree with this heuristic and argue that JSP gives you a mechanical way of determining what the largest possible chunk is. Botting, an advocate of JSP, partially disagreed with me, saying, "This heuristic only works when there is no 'boundary clash' because it assumes that there is a linear ordering 'larger than' which can be applied to chunks of data." The term *boundary clash* refers to data-processing problems in which the input and output files have structures that cannot be unified. JSP deals with this by connecting two programs together. The first program processes the input into an intermediate file that is then used by a second program to produce the output. This approach could also be thought of as imposing a pipe or coroutine structure on the program, which is a possibility not explicitly considered by Clancy's heuristic, or for that matter by top-down decomposition.

In any event, focusing the programmer's attention on the question of the proper chunk size is valuable. As Jackson points out in his book, one sign that we have selected the right iteration unit is that we can state simply what that unit is; we can do that for Jackson's solution, but not for Van Wyk's.

This problem is rather easier than the word-counting problem of the previous columns, and there is less that we can learn from a study of this or any solution. Nevertheless, Jackson has shown us a technique worthy of study; it lets many data-processing programs be structured deterministically, and bypasses the explicit use of heuristics like Clancy's, without contradicting them. Jackson's technique is founded on an important insight, which is that the structure of the program depends on the structure of the data. This fact must be applied more creatively to problems that are not driven

so directly by their input and output, but we can still be grateful for this small chance to demonstrate it in practice.

David W. Wall
DEC Western Research Laboratory
100 Hamilton Street
Palo Alto, CA 94301

REFERENCES

1. Cameron, J.R. JSP and JSD: The Jackson Approach to software development. IEEE Tutorial Text, IEEE Press, New York, 1984.
2. Elder, J. *Construction of Data Processing Software.* Prentice-Hall, Englewood Cliffs, N.J., 1984.
3. Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall International, U.K., 1985.
4. Jackson, M.A. *Principles of Program Design.* Academic Press, New York, 1975.

For Correspondence: Christopher J. Van Wyk, AT&T Bell Laboratories, Room 2C-457, 600 Mountain Avenue, Murray Hill, NJ 07974.

Programming Pearls *(continued from p. 999)*

tions described in Solution 2 are more efficient when N is large compared to M.

2. Bob Floyd described several possible data structures to implement the set S in Algorithm F2: "A bit array is appropriate if N is no more than perhaps $100M$; if b is the number of bits per word, then the run time is virtually constant at $O(M) + O(N/b)$.

   "For larger N, use an array of size near M indexed by the high order bits of the data, of pointers to sorted linked lists containing (the low order bits of) the data. Mean execution time is $O(M)$, variance is $O(M)$, and maximum is $O(M^2)$; storage is $O(M)$.

   "A cautious implementation uses a balanced order tree. Mean and worst-case times are $O(M \log M)$, with small variance; storage is $O(M)$."

3. A data structure that is efficient for Algorithm S might be slow when used in Algorithm F2. When $M = N$, for instance, a binary search tree gives logarithmic expected search time in Algorithm S. In Algorithm F2, though, the elements are inserted in increasing order, so the binary search tree degrades into a linked list with linear search time.

5. Any algorithm for generating a random M-element permutation from $1..N$ must use at least

$$\lg(N \times N-1 \times \cdots \times N-M+1) = \sum_{I=N-M+1}^{N} \lg I$$

   random bits, where lg denotes the base-two logarithm. Algorithm P consumes

$$\sum_{I=N-M+1}^{N} \lceil \lg I \rceil$$

   random bits, so it is within M bits of optimal.

   Doug McIlroy developed this algorithm to store the Gth combination of M of N items in array A:

```
procedure Comb(N, M, G, A)
    D := 1
    while M > 0 do
        T = C(N - D, M - 1)
        if G - T < 0 then
            M := M - 1
            A[M] := D
        else
            G := G - T
        D := D + 1
```

   The function C(N, M) returns $\binom{N}{M}$. Both the array A and the integer G use a "zero-origin": the array is indexed over $0..M-1$ and the first permutation corresponds to $G = 0$. One can therefore generate a random M-element subset of $1..N$ with the call

```
Comb(N, M, RandInt(0, C(N,M)-1), A)
```

   This method uses precisely the optimal number of random bits.

6. Floyd writes, "An appropriate data structure for the sequence S in Algorithm P is a hash table with a linked list connecting the entries. If the hash table size is about $2M$, the expected running time is $O(M)$. A cautious version of this representation is a balanced ordered tree with a linked list running through it, for expected and worst-case times $O(M \log M)$."

8. Burstall and Darlington described a system for transforming recursive programs in *Acta Informatica* (6, 1, 1976, 41–60) and in *JACM* (24, 1, Jan. 1977, 44–67).

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974.