programming pearls

by Jon Bentley

SELF-DESCRIBING DATA

You just spent three CPU hours running a simulation to forecast your company's financial future, and your boss asks you to interpret the output:

```
Scenario 1:   3.2% -12.0%   1.1%
Scenario 2:  12.7%   0.8%   8.6%
Scenario 3:   1.6%  -8.3%   9.2%
```

Hmmm.

You dig through the program to find the meaning of each output variable. Good news—Scenario 2 paints a rosy picture for the next fiscal year. Now all you have to do is uncover the assumptions of each. Oops—the disaster in Scenario 1 is your company's current strategy, doomed to failure. What did Scenario 2 do that was so effective? Back to the program, trying to discover which input files each one reads. . . .

Every programmer knows the frustration of trying to decipher mysterious data. The first two sections of this column discuss two techniques for embedding descriptions in data files. The third section then applies both methods to a concrete problem.

Name-Value Pairs

Many document production systems support bibliographic references in a form something like:

```
%title   The Art of Computer Programming,
         Volume 3: Sorting and Searching
%author  D. E. Knuth
%pub     Addison-Wesley
%city    Reading, Mass.
%year    1973

%author  A. V. Aho
%author  M. J. Corasick
%title   Efficient string matching:
         an aid to bibliographic search
%journal Communications of the ACM
%volume  18
%number  6
%month   June
%year    1975
%pages   333-340
```

Blank lines separate entries in the file. A line that begins with a percent sign contains an identifying term followed by arbitrary text. Text may be continued on subsequent lines that do not start with a percent sign.

The lines in the bibliography file are *name-value pairs*: each line contains the name of an attribute followed by its value. The names and the values are sufficiently self-describing that I don't need to elaborate further on them. This format is particularly well suited to bibliographies and other complex data models. It supports missing attributes (books have no volume number and journals have no city), multiple attributes (such as authors), and an arbitrary order of fields (one need not remember whether the volume number comes before or after the month).

Name-value pairs are useful in many databases. One might, for instance, describe the aircraft carrier USS *Nimitz* in a database of naval vessels with these pairs:

```
name         Nimitz
class        CVN
number       68
displacement 81600
length       1040
beam         134
draft        36.5
flightdeck   252
speed        30
officers     447
enlisted     5176
```

Such a record could be used for input, storage, and output. A user could prepare a record for entry into the database using a standard text editor. The database system could store records in exactly this form (we'll soon see a representation that is more space efficient). The same record could be included in the answer to the query "What ships have a displacement of more than 75,000 tons?"

Name-value pairs offer several advantages for this hypothetical application. A single format can be used for reading, storing, and writing records; this simplifies life for user and implementer alike. The application is inherently variable-format because different ships have different attributes: submarines have no flight decks and aircraft carriers have no submerged depth. Unfortunately, the example does not document the units in which the various quantities are expressed; we'll return to that shortly.

Some database systems store records on mass memory in exactly the form shown above. This format makes it particularly easy to add new fields to records in an existing database. The name-value format can be quite space efficient, especially compared to fixed-format records that have many fields, most of which are usually empty. If storage is critical, however, then the database could be squeezed to a compressed format:

```
naNimitz¦clCVN¦nu68¦di81600¦le1040¦
be134¦dr36.5¦fl252¦sp30¦of447¦en5176
```

Each field begins with a two-character name and ends with a vertical bar. The input and the stored formats are connected by a data dictionary, which might start:

```
ABBR NAME         UNITS
na   name         text
cl   class        text
nu   number       text
di   displacement tons
le   length       feet
be   beam         feet
dr   draft        feet
fl   flightdeck   feet
sp   speed        knots
of   officers     personnel
en   enlisted     personnel
```

In this dictionary the abbreviations are always the first two characters of the name; that may not hold in general. Readers offended by hypocrisy may complain that the above data is not in a name-value format. The regular structure supports the tabular format, but observe that the header line is another kind of self-description embedded in the data.

Name-value pairs are a handy way to give input to any program. (They are perhaps the tiniest of the "little languages" described in the September 1986 column.) They can help meet the criteria that Kernighan and Plauger propose in Chapter 5 of their *Elements of Programming Style* (2nd ed., McGraw-Hill, 1978):

> Use mnemonic input and output. Make input easy to prepare (and to prepare correctly). Echo the input and any defaults onto the output; make the output self-explanatory.

Name-value pairs are useful in code far removed from input/output. Suppose we wanted to provide a subroutine that adds a ship to a database. Most languages denote the (formal) name of a parameter by its position in the parameter list. This mechanism leads to remarkably clumsy calls:

```
addship("Nimitz", "CVN", "68",
        81600, 1040, 134, 36.5,
        447, 5176,,,30,,,252,,,,)
```

The missing parameters denote fields not present in this record. Is 30 the speed in knots or the draft in feet? A little discipline in commenting helps unravel the mess:

```
addship("Nimitz",   # name
        "CVN",      # class
        "68",       # number
        81600,      # disp
        1040,       # length
          ...)
```

Many programming languages support named parameters, which make the job easier:

```
addship(name = "Nimitz",
        class = "CVN",
        number = "68",
        disp = 81600,
        length = 1040,
          ...)
```

Even if your language doesn't have named parameters, you can usually simulate them with a few routines:

```
shipstart()
shipstr(name, "Nimitz")
shipstr(class, "CVN")
shipstr(number, "68")
shipnum(disp, 81600)
shipnum(length, 1040)
  ...
shipend()
```

The name variables `name`, `class`, `number`, etc., are assigned unique integers.

Provenances in Programming

The provenance of a museum piece lists the origin or source of the object. Antiques are worth more when they have a provenance (this chair was built in such-and-such, then purchased by so-and-so, etc.). You might think of a provenance as a pedigree for a nonliving object.

The idea of a provenance is old hat to many programmers. Some software shops insist that the provenance of a program be kept in the source code itself: in addition to other documentation in a module, the provenance gives the history of the code (who changed what when, and why). The provenance of a data file is often kept in an associated file (a transaction log, for instance). Frank Starmer, a Professor in the Departments of Computer Science and Medicine at Duke University, tells how his programs produce data files that contain their own provenances:

"We constantly face the problem of keeping track of our manipulations of data. We typically explore data sets by setting up a UNIX® pipeline like

```
sim.events -k 1.5 -l 3 |
sample -t .01 |
bins
```

The first program is a simulation with the two parameters `k` and `l` (set in this example to 1.5 and 3).[1] The vertical bar at the end of the first line pipes the output into the second program. That program samples the data at the designated frequency, and in turn pipes its output to the third program, which chops the input into bins (suitable for graphical display as a histogram).

"When looking at the result of a computation like this, it is helpful to have an 'audit trail' of the various command lines and data files encountered. We therefore built a mechanism for 'commenting' the files with various annotations so that when we review the output, everything is there on one display or piece of paper.

"We use several types of comments. An 'audit trail' line identifies a data file or a command-line transformation. A 'dictionary' line names the attributes in each column of the output. A 'frame separator' sets apart a group of sequential records associated with a common event. A 'note' allows us to place our remarks in the file. All comments begin with an exclamation mark and the type of the comment; other lines are passed through untouched and processed as data. Thus the output of the above pipeline might look like:

```
!trail sim.events -k 1.5 -l 3
!trail sample -t .01
!trail bins -t .01
!dict bin_bottom_value item_count
0.00    72
0.01    138
0.02    121
  ...
!note there is a cluster around 0.75
!frame
```

All programs in our library automatically copy existing comments from their input onto their output, and additionally add a new `trail` comment to document their own action. Programs that reformat data (such as `bins`) add a `dict` comment to describe the new format.

"We've done this in order to survive. This discipline aids in making both input and output data files self-documenting. Many other people have built similar mechanisms; wherever possible, I have copied their enhancements rather than having to figure out new ones myself."

Tom Duff of Bell Labs uses a similar strategy in a system for processing pictures. He has developed a large suite of UNIX programs that perform transformations on pictures. A picture file consists of text lines listing the commands that made the picture (terminated by a blank line) followed by the picture itself (represented in binary). The prelude provides a provenance of the picture. Before Duff started this practice he would sometimes find himself with a wonderful picture and no idea of what transforma-

UNIX is a registered trademark of AT&T Bell Laboratories.

[1] Note that the two parameters are set by a simple name-value mechanism. —*J. B.*

tions produced it; now he can reconstruct any picture from its provenance.

Duff implements the provenances in a single library routine that all programs call as they begin execution. The routine copies the old command lines to the output and then writes the command line of the current program.

A Sorting Lab

To make the above ideas more concrete, we'll apply them to a problem described in the July 1985 column: building "scaffolding" to experiment with sort routines. That column contains code for several sort routines and a few small programs to exercise them. This section will sketch a better interface to the routines. The input and output are both expressed in name-value pairs, and the output contains a complete description of the input (its provenance).

Experiments on sorting algorithms involve adjusting various parameters, executing the specified routine, then reporting key attributes of the computation. The precise operations to be performed can be specified by a sequence of name-value pairs. Thus the input file to the sorting lab might look like this:

```
n          100000
input      identical
alg        quick
cutoff     10
partition  random
seed       379
```

In this example the problem size, `n`, is set to 100,000. The input array is initialized with `identical` elements (other options might include `random`, `sorted`, or `reversed` elements). The sorting algorithm in this experiment is `quick` for quicksort; `insert` (for insertionsort) and `heap` (for heapsort) might also be available. The `cutoff` and `partition` names specify further parameters in `quicksort`.

The input to the simulation program is a sequence of experiments in the above format, separated by blank lines. Its output is in exactly the same format of name-value pairs, separated by blank lines. The first part of an output record contains the original input description, which gives the provenance of each experiment. The input is followed by three additional attributes: `comps` records the number of comparisons made, `swaps` counts swaps, and `cpu` records the run time of the procedure. Thus an output record might end with the fields:

```
comps      4772
swaps      4676
cpu        0.1083
```

Given the sort routines and other procedures to do the real work, the control program is easy to build. Its main loop is sketched in Program 1: the code reads each input line, copies it to the output, and processes the name-value pair. The `simulate( )` routine performs the experiment and writes the output variables in name-value format; it is called at each blank line and also at the end of the file.

```
loop
    read input line into string S
    if end of file then break
    if S = "" then simulate()
    write S on output
    F1 = first field in S
    F2 = second field in S
    if F1= "n" then
        N = F2
    else if F1 = "alg" then
        if F2 = "insert" then
            alg = 1
        else if F2 = "heap" then
            alg = 2
        else if F2 = "quick" then
            alg = 3
        else error("bad alg")
    else if F1 = "input" then
        ...
simulate()
```

PROGRAM 1. A Loop to Process Name-Value Pairs

This structure is useful for many simulation programs. The program is easy to build and easy to use. Its output can be read by a human and can also be fed to later programs for statistical analysis. Because all the input variables (which together provide a provenance of the experiment) appear with the output variables, any particular experiment can be repeated and studied in detail. The variable format allows additional input and output parameters to be added to future simulations without having to restructure previous data. Problem 8 shows how the basic structure can be gracefully extended to performing sets of experiments.

Principles

This column has only scratched the surface of self-describing data. Many systems, for instance, allow a programmer to multiply two numeric objects of unspecified type (ranging from integers to arrays of complex numbers); at run time the system determines the types by inspecting descriptions stored with the operands, and then performs the appropriate action. Some tagged-architecture machines provide hardware support of self-describing objects, and some communications protocols store data along with a description of its format and types. It is easy to give even more exotic examples of self-describing data.

This column has concentrated on two simple but useful kinds of self-descriptions. Each reflects an important principle of program documentation.

- The most important aid to documentation is a clean programming language. Name-value pairs are a simple, elegant, and widely useful linguistic mechanism.
- The best place for program documentation is in the source file itself. A data file is a fine place to store its own provenance: it is easy to manipulate and hard to lose.

Problems

1. Self-documenting programs contain useful comments and suggestive indentation. Experiment with formatting a data file to make it easier to read. If necessary, modify the programs that process the file to ignore white space and comments. Start your task using a text editor. If the resulting formatted records are indeed easier to read, try writing a "pretty printing" program to present an arbitrary record in the format.

2. Give an example of a data file that contains a program to process itself.

3. The comments in good programs make them self-describing. The ultimate in a self-describing program, though, is one that prints exactly its source code when executed. Is it possible to write such a program in your favorite language?

4. Many files are implicitly self-describing: although the operating system has no idea what they contain, a human reader can tell at a glance whether a file contains program source text, English text, numeric data, or binary data. How would you write a program to make an enlightened guess as to the type of such a file?

5. Give examples of name-value pairs in your computing environment.

6. Find a program with fixed-format input that you find hard to use, and modify it to read name-value pairs. Is it easier to modify the program directly or to write a new program that sits in front of the existing program?

7. A general principle states that the output of a program should be suitable for input to the program. For instance, if a program wants a date to be entered in the format "06/31/87" then it should not write:

```
Enter date (default 31 June 1987):
```

   Give other contexts in which this rule holds.

8. The text sketched how to specify a single experiment on a sorting algorithm. Usually, though, experiments come in sets, with several parameters systematically varied. Construct a generator program that will convert a description like

```
n          [100 300 1000 3000 10000]
input      [random identical sorted]
alg        quicksort
cutoff     [5 10 20 40]
partition  med-of-3
```

   into $5 \times 3 \times 4 = 60$ different specifications, with each item in a bracket list represented by a single element in a cross product. How would you add more complex iterators to the language, such as

```
[from 10 to 130 by 20]
```

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