LITTLE LANGUAGES FOR PICTURES IN AWK

Jon L. Bentley

Jon L. Bentley *is a member of technical staff in the Computing Principles Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He does research in algorithm design and programming methodology. He is also an adjunct associate professor of computer science at Carnegie Mellon University. He has a B.S. in mathematical sciences from Stanford University and an M.S. and Ph.D., both in computer science, from The University of North Carolina. He joined AT&T in 1982.*

Specialized pictures for some problem domains can be easily described by little languages. This paper shows how a "compiler" for a small but useful language can be built in a few dozen lines of `awk` code. Example problem domains include graphical display of data, chemistry, algorithm animation, and computer documentation.

The `awk` language was originally designed in 1977 to perform tasks in data processing.[1] A number of features were added in the 1985 version of `awk` to make the language more powerful.[2] In this paper we use `awk` to process little languages, in the tradition of `troff` preprocessors.[3] (The `awk` language has proven useful for implementing various document production programs, including tools for preparing indexes[4] and cross references.[5] Simple versions of these are presented in Section 5.3 of Reference 2.) The `awk` programs we describe transform textual descriptions into `pic` pictures.[6]

The following section describes one little language in detail. The next four sections survey languages for display of data, algorithm animation, chemistry, and computer hardware documentation. Principles of language design are sketched in the subsequent section.

A Language for Data Formats

Data formats ranging from computer words to packets on a data network are often described by pictures composed of rectangles. The `dformat` program allows such diagrams to be included in `troff` documents. The first part of this section describes the `dformat` input language, which is a typical little language: its simple syntactic structure allows common pictures to be drawn easily, while advanced bells and whistles are provided for complex pictures. The second part describes the `awk` implementation of `dformat`; the prototype program was written in an hour in a few dozen lines, while the final version took a day and 100 lines.

The `dformat` Language. Figure 1 shows the format that many computers use for storing short integers, standard integers, and floating-point numbers. Each line of boxes shows a *record* that contains several *fields*. The figure was described in `dformat` by this text:

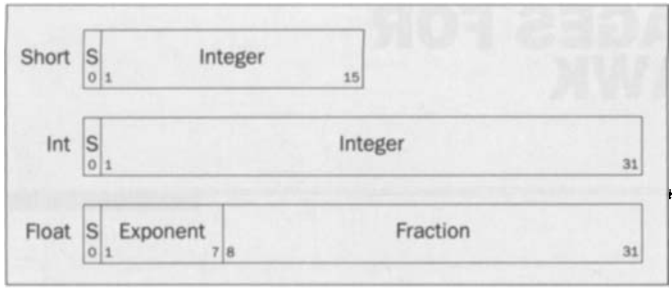
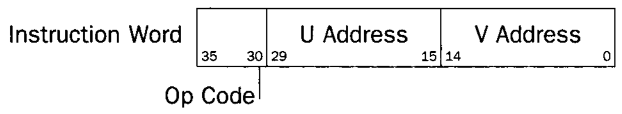


Figure 1. Formats for storing short integers, integers, and floating-point numbers.

```
.begin dformat
style bitwid 0.09
Short
   0 S
   1-15 Integer
Int
   0 S
   1-31 Integer
Float
   0 S
   1-7 Exponent
   8-31 Fraction
.end
```

The `.begin dformat` and `.end` lines delimit the `dformat` input. Lines that start in column 1 name the records; subsequent lines that begin with white space (blanks or tabs) give the field ranges and names, in order. The `style` command can be viewed as an assignment of the value 0.09 inches to the parameter `bitwid`. The 32-bit record is therefore 2.88 inches across.

Bits can also be numbered in descending order; the diagram below, for example, shows the instruction format of the UNIVAC 1103A computer:



It was produced by this input:

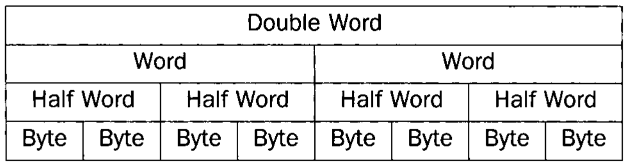
```
.begin dformat
style bitwid 0.06
Instruction Word
   35-30 Op Code
   29-15 U Address
   14-0 V Address
.end
```

Because "Op Code" is too long to fit in its box, `dformat` places the text below the box and connects it with a line.

Because `dformat` is a `pic` preprocessor, a document that contains `dformat` pictures is usually compiled with a pipeline like

```
dformat paper.in | pic |
   troff -ms > paper.out
```

These examples illustrate the typical use of `dformat`. Simple pictures are simply described; this level of detail is sufficient for most users. We will now examine a few more complex figures. Many computer systems have memory organized as in this diagram:



This picture was drawn by this `dformat` description:

```
.begin dformat
style bitwid 0.05
style recspread 0
style addr off
style recht 0.2
noname
   0-63 Double Word
noname
   0-31 Word
   32-63 Word
noname
   0-15 Half Word
   16-31 Half Word
   32-47 Half Word
   48-63 Half Word
 ...
.end
```

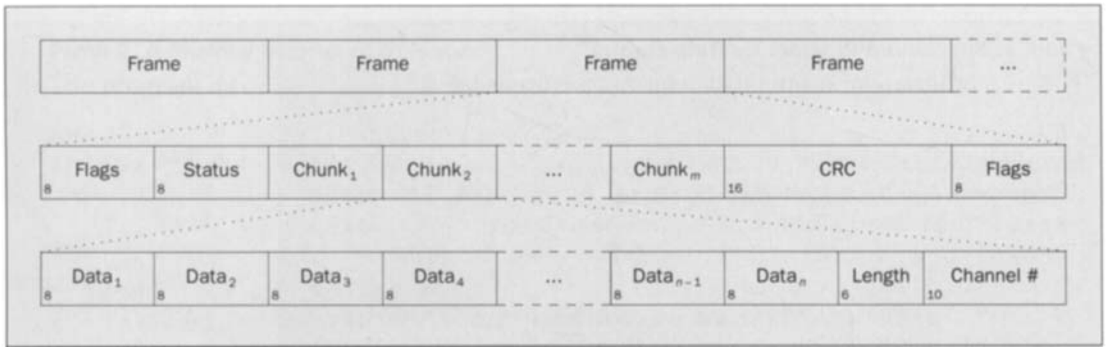


Figure 2. Packets in a data network.

The `bitwid` parameter of 0.05 makes the 64-bit record 3.2 inches wide. The `recspread` parameter is the vertical spread between records; the value of 0 causes the records to be stacked with no intervening space. Assigning `off` to `addr` turns the addresses off (the default assignment is `both`; `left` and `right` also work). The `recht` parameter ensures that each record is depicted by a rectangle 0.2 inches high. All records have the special name `noname`, so no text appears to the left of the records.

The last example uses four `style` parameters; `dformat` has five other variables that control the resulting picture. The values of variables are retained from one `dformat` display within a document to another. This allows one to define a style for a document by setting all values in the first display.

For drawing more complicated figures, `dformat` has additional capabilities. For example, arbitrary text may be placed at the left and right addresses of a field, and boxes may be dashed or dotted. In addition, a field within a record may be named, and `pic` commands can be used to draw lines connecting fields. Figure 2 uses these features to depict the packets used in a data communications network. This figure represents the most complex kind of diagram that can (and should) be drawn with `dformat`.

The `dformat` language is typical of little languages suitable for processing by `awk`. At its core is a small subset that is sufficient for many users:

- The syntax models the two-level hierarchy inherent in the problem: records begin in column 1, and fields begin in later columns.
- The textual description is included in a `troff` document, set off by the `.begin` and `.end` delimiters.
- The width of the picture is adjusted by setting the `bitwid` variable.

Advanced features are available for drawing more esoteric pictures. Variables that control pictures (about a dozen) retain their values between pictures. Fields may be named, and `pic` commands that refer to those names may be passed through to `pic` to produce objects in the resulting picture.

Implementing `dformat`

We will turn now to the construction of the "compiler" that translates `dformat` descriptions into `pic`. We will begin by studying a tiny version in detail, and then sketch the complete program.

The minimal version of `dformat` draws simple data formats. It does not support parameters or style assignments, nor does it place long strings below their boxes. For instance, this sample input

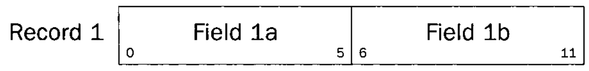
```
.begin dformat
Record 1
   0-5 Field 1a
   6-11 Field 1b
.end
```

Panel 1. Specialized Program for Data Format

The `pic` program for a simple data format picture is:

```
.PS
BoxA: box invis ht 0.3 wid 0
 "Record 1  " rjust at BoxA.w
 BoxB: box invis ht 0.3 wid 0 at BoxA
  BoxB: box ht .3 wid 1.2 with .w at BoxB.e
   "Field 1a" at BoxB.c
     " \s-40\s+4" ljust at BoxB.sw + (0,.06)
     "\s-45\s+4 " rjust at BoxB.se + (0,.06)
  BoxB: box ht .3 wid 1.2 with .w at BoxB.e
   "Field 1b" at BoxB.c
     " \s-46\s+4" ljust at BoxB.sw + (0,.06)
     "\s-411\s+4 " rjust at BoxB.se + (0,.06)
.PE
```

produces this picture



by writing the (highly specialized) `pic` program in Panel 1.

The program in Panel 2 translates this subset of dformat into `pic`. (Readers unfamiliar with `awk` should skim this section to appreciate that the implementation is concise; for more information on the language, see Reference 2.) The program consists of four `awk` pattern-action pairs:

- The first pattern recognizes that the variable `inlang` is zero (its initial value), which means that we are not currently in a `dformat .begin/.end` block. If the current line is not `.begin dformat` then it is printed; otherwise, `inlang` is set to one and `dformat` initializes its state.
- The second pattern recognizes the `.end` of a `dformat` block; its action sets `inlang` to zero and terminates this picture.
- The third pattern recognizes text in column 1, which marks a new record. The action prints two invisible `pic` boxes (subsequent boxes will be placed adjacent to them), and writes the record name.
- The fourth pattern recognizes white space in column 1, which marks a new field. The action prints a box, the field name, and the left and right indices.

The complete `dformat` program is about 100 lines of `awk` code. The most significant change from the minimal version is a symbol table for variables, implemented by an `awk` associative array named `parm`. The nine style parameters are initialized using this `awk` idiom:

```
s = "recht 0.3          addrht 0.055 "\
    "recspread 0.15     charwid 0.07 "\
    "textht 0.167       addrdelta 4 " \
    "bitwid 0.125       linedisp 0.04 "\
    "addr both"
n = split(s, x)
for (i = 1; i < n; i += 2)
    parm[x[i]] = x[i+1]
```

Panel 2. A Minimal Version of `dformat`

The program `dformat.min` is as follows:

```
awk '
inlang == 0     { if ($0 !~ /^\.begin / || $2 != "dformat") print
                  else { inlang = 1; print ".PS"; boxacnt = 0 }
                  next
                }
/^\.end/        { inlang = 0; print ".PE"; next }
/^[^ \t]/       { printf "BoxA: box invis ht 0.3 wid 0"
                  if (boxacnt++) printf " with .n at BoxA.s - (0,0.15)"
                  printf "\n"
                  printf " \"%s  \" rjust at BoxA.w\n", $0
                  printf " BoxB: box invis ht 0.3 wid 0 at BoxA\n"
                  next
                }
 /^[ \t]/       { split($1, r, "-")
                  $1 = ""; sub(/^[ \t]+/, "")
                  printf "  BoxB: box ht .3 wid %g with .w at BoxB.e\n",
                        0.2*(r[2]-r[1]+1)
                  printf "     \"%s\" at BoxB.c\n", $0
                  printf "\t\" \\s-4%s\\s+4\" ljust at BoxB.sw + (0,.06)\n", r[1]
                  printf "\t\"\\s-4%s\\s+4 \" rjust at BoxB.se + (0,.06)\n", r[2]
                }
 ' $*
```

A variable is set by a `style` command, which is processed by this pattern-action pair:

```
$1 == "style" { if ($2 in parm)
                    parm[$2] = $3
                else
                    error("bad name: " $2)
                next
              }
```

Some of the constants that are hard-wired into the program in Panel 2 are replaced by accesses to the `parm` array in the complete program; other options are interpreted by conditional code.

The complete version of the program includes several additional features. It checks for errors of several forms; this pattern-action pair detects when a `.begin dformat` display does not have a matching `.end`:

```
END { if (inlang)
            error("eof inside begin/end")
    }
```

Statements are passed through to the underlying `pic` processor by this line:

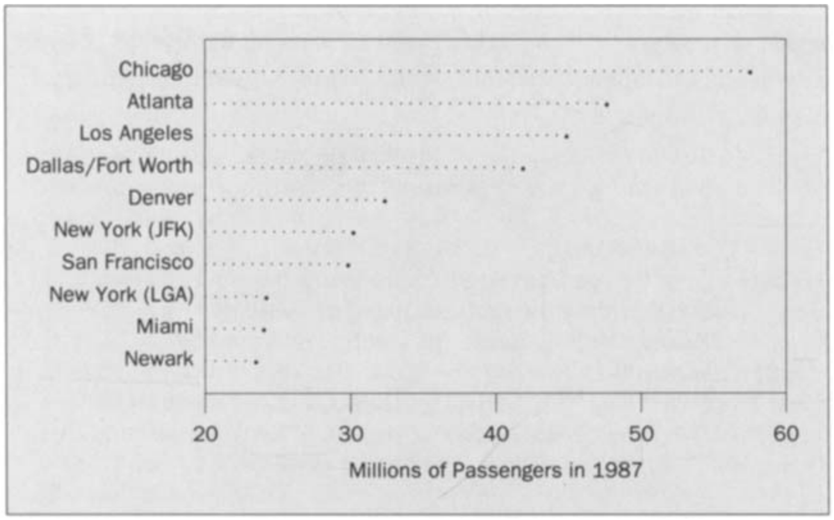


Figure 3. Chart of airport passenger arrivals and departures.

```
$1 == "pic" { $1 = ""; print $0; next }
```

When a field name is too large to fit in its box, a routing algorithm puts the overflow names in channels beneath the box.

The `dformat` program was originally built for a colleague who wanted to include data formats in a document. He described the problem on a Friday afternoon, and the first version was written in a couple of hours on Saturday; it was a slightly larger and "dirtier" than the program in Panel 2. After the colleague agreed that the output was approximately correct, I spent six hours on Sunday adding parameters, checking errors, and making other refinements. Although the program had been built for one particular user, its description spread by word of mouth over the next couple of months, and several other people requested it. With that motivation, a few days of work sufficed to polish the program and to write documentation.[7] Thus the final product, code plus documentation, represents roughly one week of programmer time. Over the next few months, the program acquired a dozen more users.

My colleague had originally planned to sketch the diagrams with a mouse-based drawing program and then to cut and paste them into his document; the `dformat` language offers many advantages over that approach. It takes just a fraction of the time to draw the formats in the first place, and stylistic changes (such as changing the height of all boxes) may be accomplished by changing a single `style` command at the beginning of the document, instead of by redrawing every format.

Graphical Displays of Data

In this section we will study study pictures that represent data. The `grap` language[8] is a general tool for displaying data. We will first study a preprocessor for `grap` and then consider `grap`'s implementation.

The "dotchart" in Figure 3 shows the 10 busiest airports in the United States in 1987 and the number of passengers that arrived and departed in that year. A dotchart is an effective way of displaying string/number

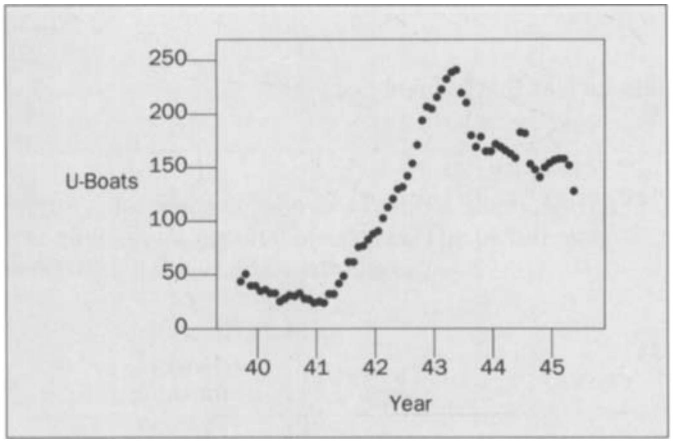


Figure 4. Plot of operational German submarines.

pairs. The picture was described by this `dot` text:

```
.begin dot
ticks bot out at 20, 30, 40, 50, 60
label bot "Millions of Passengers in 1987"
quoted
57.54 "Chicago"
47.65 "Atlanta"
44.87 "Los Angeles"
41.88 "Dallas/Fort Worth"
32.36 "Denver"
30.19 "New York (JFK)"
29.81 "San Francisco"
24.22 "New York (LGA)"
24.03 "Miami"
23.48 "Newark"
.end
```

The `.begin` and `.end` lines delimit the input. The `ticks` and `label` statements are passed through to `grap`, and the `quoted` statement states that the quotation marks are present in the input. The remainder of the input is the string/number pairs.

A minimal implementation of `dot` is presented in Reference 8; it is about 30 lines of `awk`. The complete language takes about 70 lines of code; it offers a number of style parameters and a choice of accessing data placed in the `dot` text (as above) or data in other files. That paper also describes a `grap` preprocessor named `scatmat` for producing "scatterplot matrices." Other little languages for specialized data displays include those for box plots, pie charts, and least-squares fits.

The complete `grap` language is at the large end of little languages: its user manual is 40 pages long[9] and its compiler is about 3000 lines of `lex`, `yacc`, and C code. We will now study the `scatter` language, which can be viewed as a small `grap` file (perhaps useful as a prototype). It displays a set of *x*, *y* pairs contained in a file.

For instance, the file `uboat.d` tells how many German U-boats were operational each month from September 1939 to May 1945; the fact that 34 submarines were active in February 1940 is represented by the pair 40.1667, 34; the *x* value is 40 + 2/12. The data are displayed by this `scatter` file:

```
.begin scatter
label x Year
label y U-Boats
range x 39.3 45.9
range y 0 270
ticks x 40 41 42 43 44 45
ticks y 0 50 100 150 200 250
file uboat.d
.end
```

The `label` statements give names for the *x* and *y* axes, `range` statements tell the minimum and maximum data values, `ticks` place tick marks, and the `file` statement names the input file. Figure 4 is the resulting scatterplot. Panel 3 presents the `scatter` program; it requires about 30 lines of `awk`. While `dformat` produces output as it reads each input line, `scatter` gathers all the data

Panel 3. The scatter Processor

The following 30-line program is used to generate scatter plots such as that in Figure 4.

```
awk '
BEGIN          { x = "x"; y = "y"; size[x] = 2; size[y] = 1.5 }
inlang == 0    { if (/^\.begin/ && $2 == "scatter") inlang = 1; else print
                 next
               }
$1 == "size"   { size[$2] = $3 }
$1 == "label"  { lab[$2] = $3 }
$1 == "range"  { min[$2] = $3; max[$2] = $4 }
$1 == "ticks"  { for (i = 3; i <= NF; i++) ticks[$2,tickct[$2]++] = $i }
$1 == "file"   { fname = $2 }
/^\.end/       { inlang = 0
                 print ".PS"
                 print "fw = " size[x] "; fh = " size[y]
                 print "B: box ht fh wid fw with .sw at 0,0"
                 print "\"" lab[y] "\" at B.w - (.4, 0) rjust"
                 print "\"" lab[x] "\" at B.s - (0, .4) below"
                 print "define tx { (($1)-" min[x] ")*fw/" max[x]-min[x] " }"
                 print "define ty { (($1)-" min[y] ")*fh/" max[y]-min[y] " }"
                 for (i = 0; i < tickct[x]; i++) {
                         print " line from tx(" ticks[x, i] "), 0 down .15"
                         print " \"" ticks[x, i] "\" at last line .s below"
                 }
                 for (i = 0; i < tickct[y]; i++) {
                         print " line from 0, ty(" ticks[y, i] ") left .15"
                         print " \"" ticks[y, i] "\" at last line .w rjust"
                 }
                 print "copy \"" fname "\" thru { \"\\(bu\" at tx($1), ty($2) }"
                 print ".PE"
               }
' $*
```

into variables and then writes its output at the `.end` statement. As alternative approaches to producing simple graphs, Section 6.2 of Reference 2 produces the venerable character array output, and Reference 10 compiles into the `ideal` language.

Algorithm Animation

We will turn now to a system that animates algorithms.[11] Unfortunately, this paper cannot present the primary output of the system, which is a primitive movie. We can, however, present "stills" from the movie.

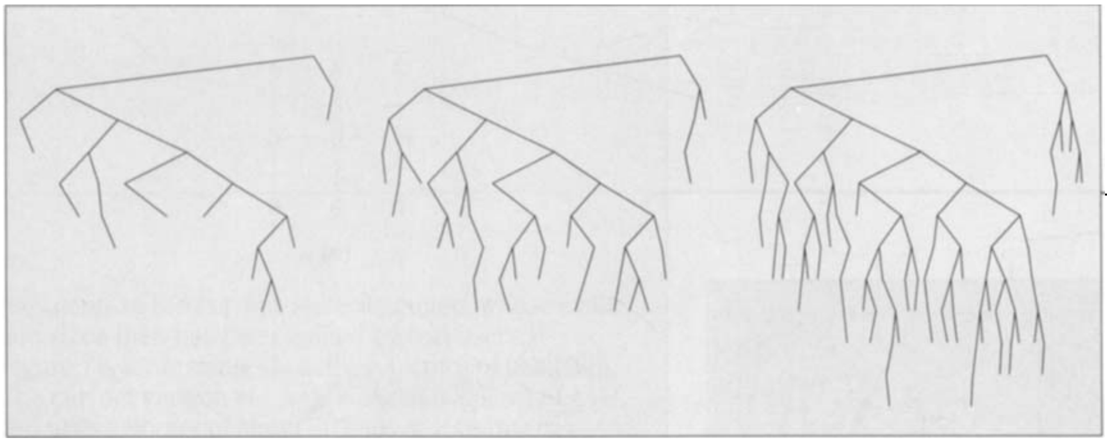


Figure 5. Frames for a growing search tree after 25, 50, and 100 insertions (left to right).

Figure 5, for instance, shows a random binary search tree after 25, 50, and 100 insertions. The picture was described in the `stills` language:

```
.begin stills
frameht  2.0
framewid 1.9
frames   invis
times    invis
across   3
view     def.view ""
print    insert 25 50 100
file     bst.s
.end
```

The size of each snapshot is given by the `frameht` and `framewid` statements. The next two statements turn off (that is, make invisible) surrounding frames and times. The `across` statement tells the direction in which to print the pictures, and the `print` statement tells which snapshots to print.

The file `bst.s` is a "script" file that contains a geometric history of the computation. Its first few lines are:

```
click insert
line 93 -1 14 -2
click insert
line 93 -1 99 -2
click insert
line 14 -2 33 -3
click insert
```

The `line` statements draw a line from the first *x*, *y* pair to the second *x*, *y* pair, and the `click` statements mark each insertion event (the marks are used by the `print` statement). The script language provides for other geometric primitives (text, boxes, and circles) and other housekeeping operations (erasing objects and switching among several windows on a computation).

The complete animation system consists of three programs: `movie` allows a user to explore a computation interactively, `stills` includes snapshots in a `troff` document, and `develop` is a preprocessor used by both. Brian Kernighan and I built the system in five versions; Table I shows the size of the programs in each version. All programs were originally written in `awk`; `develop` and `movie` were eventually converted to C for performance and function. Working with small `awk` prototypes made it easy for us to experiment with many different language designs.

It is easy to animate many algorithms with movies that manipulate lines, boxes, circles, and text. Sorting algorithms, for instance, might permute a sequence of sticks to go from shortest to tallest. We used the animation system extensively in experiments on "traveling salesman" algorithms for graphs defined by Euclidean point sets. For planar point sets, we simply drew lines

Table I. Sizes of Programs in Animation System

Version number	Lines of code		
	`develop`	`stills`	`movie`
1	110	70	60
2	110	150	100
3	300	320	500C
4	240	250	1000C
5	1000C	300	1500C

Note: C represents code in C language.

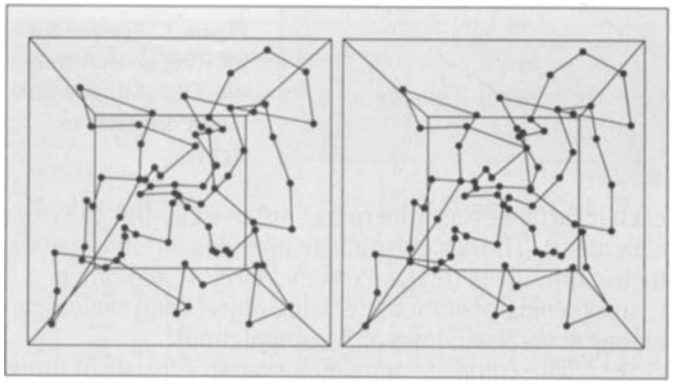


Figure 6. Stereoscopic views of a traveling salesman tour.

between points in the set. To study point sets in three-dimensional space, we built a preprocessor that allows us to draw three-dimensional lines and text. Figure 6, for instance, is an (almost optimal) traveling salesman tour among 70 points in three-dimensional space, which is easy to perceive with a stereo viewer. The first version of the three-dimensional preprocessor was built in an hour in about 50 lines of `awk`; the final version is 150 lines.

Chemistry

Brian Kernighan and I became interested in typesetting chemical diagrams through collaboration with our Bell Laboratories colleague Lynn Jelinski. During an afternoon discussion, she described the hassles of including chemical structures in her documents, in particular, the high cost and long delays of manual drafting. We suspected that her task might be appropriate for a `pic` preprocessor, so she lent us a monograph rich in chemical diagrams. That evening, we designed a tiny language that could describe many of the structures, and implemented it with an `awk` processor, about 50 lines long. Because the monograph dealt with polymers, our prototype languages aimed at the specialized figures typical of polymer chemistry, such as that in Figure 7a. The input for this picture was just

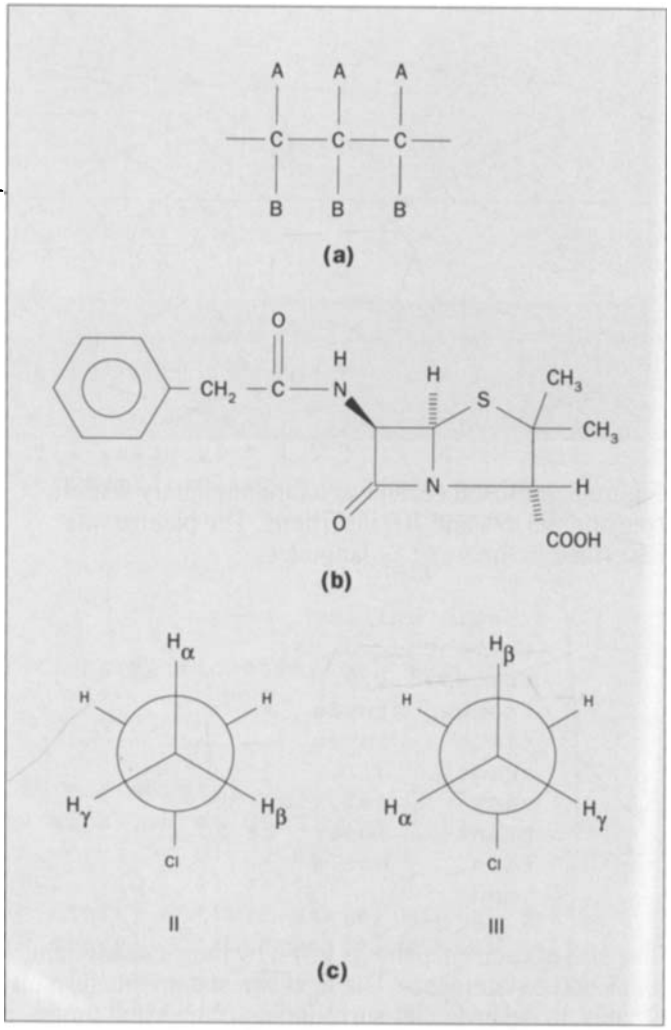


Figure 7. Pictures drawn with `chem`. (a) Basic polymer picture. (b) Penicillin molecule. (c) Sample Newman diagrams.

```
A C B
A C B
A C B
```

Jelinski was interested in drawing the broader class of figures typical of organic chemistry, so our languages were of little use to her. The phototypesetter output, however, was sufficiently attractive that it trapped Jelinski into working on a more ambitious language.

Over the next few days the three of us built and threw away several little languages. A week after starting the project, we had designed and implemented the rudiments of the current `chem` language, whose evolution since then has been guided by real users.[12] Figure 7b, for instance, is a `chem` picture of penicillin. The current version of `chem` is about 500 lines of `awk` and uses a library of about 70 lines of `pic` macros.

Along the way to `chem`, we studied several other kinds of pictures that arise in chemistry documents. Newman diagrams are a device for describing the six atoms adjacent to a given carbon bond; Figure 7c shows a sample pair of diagrams. That figure is described by this `newman` text:

```
.begin newman
# From Morrison and Boyd, p. 446
    label front Halpha Hbeta Hgamma
    label back H Cl H
    title II
next
    label front Hbeta Hgamma Halpha
    label back H Cl H
    title III
.end
```

The original `newman` program was a few dozen lines of `awk`; the final version required about 150 lines and a day of programmer time.

Swizzling Bits

We turn now to a little language that we built as we were writing this paper. A Bell Laboratories colleague, Bart Locanthi, was documenting a hardware device to permute the bits in a 32-bit computer word; such a permutation is referred to as "swizzling" the bits in the word. The swizzle in Figure 8, for instance, interleaves bit pairs by pushing even pairs to the right and odd pairs to the left. The picture is a concise way of saying that the two leftmost bits (F and E) remain stationary, bits D and C are placed in positions f and e, etc. Locanthi prepared several of the diagrams by hand, using a mouse to manipulate the image on a computer screen; he estimates that each figure took about 20 minutes to draw.

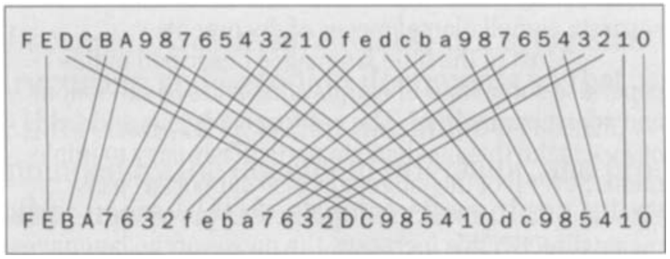

F E D C B A 9 8 7 6 5 4 3 2 1 0 f e d c b a 9 8 7 6 5 4 3 2 1 0
F E B A 7 6 3 2 f e b a 7 6 3 2 D C 9 8 5 4 1 0 d c 9 8 5 4 1 0

Figure 8. Swizzle diagram.

When Locanthi was sufficiently frustrated by these laborious drills, he asked us to design a little language for the task. We spent about 10 minutes proposing a language design, and finally settled on this input for Figure 8:

```
.begin
swizzle
style ht 0.8
style wid 3.2
FEBA7632feba.7.6.3.2DC985410dc.9.8.5.4.1.0
.end
```

After the two `style` lines, the next line contains the final permutation of input bits. (The notation `.9` refers to the right 9; the left 9 is unadorned.) The `swizzle` program was coded in under an hour in two dozen lines of `awk`. It is a cheap one-of-a-kind program; we have no intention to pursue it further.

Hints for Language Design

After experiences with many irregular languages, we have found it convenient to design languages to meet certain conventions. Descriptions in the `langname` language are delimited by the statements `.begin langname` and `.end`. An external file is usually copied by the statement `file filename`. These two conventions alone allowed us to build (in `awk`, of course) a facility for separate compilation of pieces of documents.

Most of the little languages described in this paper use some kind of name/value pairs to set options and adjust parameters. The associative arrays and fields of `awk` make this mechanism particularly easy to implement. Several of the languages also support a "pass-through" mechanism to allow the user to write `pic` statements directly; this increases the power of the languages at the cost of a single line of `awk` code in the processor.

For languages with a restricted syntactic structure, `awk` is an effective translator. The `scatter` language contains only a few kinds of statements; the order of appearance of statements is immaterial. The `dotchart`, `swizzle`, and `polymer` languages have a linear structure: the objects are described in sequential order. The `dformat` language supports the slightly more complex structure of a two-level hierarchy (fields within records), noted by indentation. The syntactic structure of a language is usually closely related to the implementation structure: `dformat` processes one line at a time, while `dotchart` reads all of its input before generating a single output line.

Acknowledgment

I am grateful for the helpful comments of Brian Kernighan, with whom I collaborated on much of the work in this paper.

References

1. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "AWK—A Pattern Scanning and Processing Language," *Software Practice and Experience*, Vol. 9, April 1979, pp. 267-280.
2. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Massachusetts, 1988.
3. B. W. Kernighan, "The UNIX® System Document Preparation Tools: A Retrospective," *AT&T Technical Journal*, Vol. 68, No. 4, July/August 1989, pp. 5-20.
4. J. L. Bentley and B. W. Kernighan, "Tools for Printing Indexes," *Electronic Publishing*, Vol. I, No. 1, April 1988, pp. 3-17.
5. A. V. Aho and R. Sethi, "Maintaining Cross References in Manuscripts," *Software Practice and Experience*, Vol. 18, No. 1, January 1988, pp. 1-13.
6. B. W. Kernighan, "PIC—A Language for Typesetting Graphics," *Software Practice and Experience*, Vol. 12, No. 1, January 1982, pp. 1-20.
7. J. L. Bentley, "dformat—A Language for Typesetting Data Formats," Computing Science Technical Report 142, AT&T Bell Laboratories, Murray Hill, New Jersey, April 1988.
8. J. L. Bentley and B. W. Kernighan, "GRAP—A Language for Typesetting Graphs," *Communications of the ACM*, Vol. 29, No. 8, August 1986, pp. 782-792.
9. J. L. Bentley and B. W. Kernighan, "GRAP—A Language for Typesetting Graphs; Tutorial and User Manual," Computing Science Technical Report 114, AT&T Bell Laboratories, Murray Hill, New Jersey, December 1984.
10. C. J. Van Wyk, "AWK as Glue for Programs," *Software Practice and Experience*, Vol. 16, No. 4, April 1986, pp. 369-388.
11. J. L. Bentley and B. W. Kernighan, "A System for Algorithm Animation," Computing Science Technical Report 132, AT&T Bell Laboratories, Murray Hill, New Jersey, January 1987.
12. J. L. Bentley, L. W. Jelinski, and B. W. Kernighan, "CHEM—A Program for Phototypesetting Chemical Structure Diagrams," *Computers and Chemistry*, Vol. 11, No. 4, 1987, pp. 281-297.

(Manuscript received March 1, 1989)