An Introduction to Literate Programming

by Jeffreys Copeland and Haemer

This month, we start a short series on literate programming. This is a technique that includes some interesting features, some odd mis-features and has generated fanatic adherents and detractors. We will pay some attention to literate programming's advantages and disadvantages with respect to portability issues.

Literate History

Literate programming is, in a sense, an accident. It's the side result of an author who liked the way his books looked, and a paradigm shift in technology.

In the mid-1960s, a bright combinatorialist who was interested in computers was finishing his Ph.D. and post-doctoral work at Caltech. He sat down to write several survey volumes about the 12 most important topics in computer science, beginning with bits and ending with compilers. Those 12 topics were supposed to cover seven volumes. The mathematician, of course, was Donald Knuth, and the series was titled *The Art of Computer Programming*. The first volume appeared in 1967.

Well, computer science marches on, and quickly. By the time Knuth had finished the third volume (on sorting and searching) it was time to revise the first volume. Then it was time to prepare a revision on the second volume. Unfortunately, before that could be done, Addison-Wesley, which had done all its mathematical typesetting in Ireland, moved from hot-lead Monotype equipment to cold-type phototypesetting. Knuth's books were mathematically intensive, and the equations set by the new technology were not as pretty.

So, what's a good computer scientist to do? Build a typesetting sys-

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting and cats and children.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

tem, obviously. Knuth's solution was to take a year off from writing *The Art of Computer Programming* and build a system to set type and a complementary system to build fonts because he wanted to duplicate the Monotype Modern typeface used in his original books.

The first versions of those programs, which were called TEX and Metafont, were written in SAIL, a language that hadn't found much favor outside the artificial intelligence communities at Stanford University and the Massachusetts Institute of Technology. (Let's note here that TEX is a trademark of the American Mathematical Society and Metafont is a trademark of Addison-Wesley, just to keep the lawyers happy.)

The first versions of TEX and Metafont were used to produce the second edition of *Seminumerical Algorithms*, but their quality still wasn't up to the old hot-lead Monotype equipment. Because portability was an issue, versions of both programs were eventually written in Pascal. But even Pascal wasn't sufficient; to be portable, the programs had to have good internal documentation.

This was where literate programming came in. Knuth and his colleagues invented a language called WEB, which combined a text description, in TEX, with Pascal code. The WEB program text could be compiled with one tool to extract the program source code, and with another to make a printable document that described WEB's internal workings. The most famous example is the TEX program itself, which is written in WEB. It's described in Knuth's *TEX: The Program* (Addison-Wesley, 1986).

This enterprise was an example of Pournelle's Law ("Everything takes longer and costs more"). Ten years after the one year "to write some typesetting software," the final versions of TEX and Metafont were completed, and Knuth is finally working on Volume 4 of *ACP*.

Literate vs. Structured Programming

The Art of Computer Programming begins with the following sentence: "The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music." Likewise, writing a program in the literate programming discipline can be a very good way to share that aesthetic experience.

More important, computer time is now cheap. Maintenance costs and human time to interact with programs outweigh the cost of time to run them and frequently outweigh the cost of the computer itself. Today's goal

Circle No. 17 on Inquiry Card

should not be to describe the task to the computer, but to describe it sufficiently to the next human who will need to understand the program.

In his paper "Literate Programming" (*The Computer Journal*, May 1984, Vol. 27, pp. 97-111) Knuth admits to a more subtle reason for choosing the term: "During the 1970s, I was coerced like everyone else into adopting the ideas of structured programming, because I couldn't bear to be found guilty of writing *unstructured* programs. Now I have a chance to get even. By coining the phrase 'literate programming,' I am imposing a moral commitment on everyone who hears the term; surely nobody wants to admit writing an *illiterate* program."

From Knuth's statement, you may conclude that structured programming and literate programming cannot coexist. Not true. By allowing for an orderly exposition of a program, we can attack the problem in any order that's appropriate for correct structure without having to worry about that structure all the time.

Literate programming allows us to work on the problem in both top-down and bottom-up, or sometimes even middle-out, order, as needed. We've found that often our programs are more logically thought out, and the control flow is better structured.

In Knuth's original WEB system, in which TEX and most of its friends are written, TEX is the formatting engine, and Pascal is the programming language. However, there's no reason not to make other pairings. Knuth reports on a literate programming environment, built by Harold Thimbleby at the University of York, which uses `troff`. Using TEX and C, C. Silvio Levy built a tool called CWEB, to which Knuth has made some additions and modifications. There are tools combining other pairs of formatter and programming languages. These are in the pointers of the "Further Reading" section, at the end of this article.

For the moment, we will use CWEB for our examples. There are several reasons for this. In general, C is more portable than Pascal, so we like it better. All our previous columns have assumed knowledge of C. Pascal is not a standard language on the RS/6000, and we don't have a compiler handy. Most important, our Pascal is rusty.

So how does CWEB work? A program is written in CWEB, and includes both C text and TEX text (see Listing 1).

The CWEB source is processed by a program called `ctangle` to extract the C code. This is not the equivalent of a simple sed operation. We can present the program in the CWEB source in a completely different order than the one we wish to present to the compiler, as we've discussed.

Once we've processed the CWEB source through `ctangle`, we can compile the resulting code normally. Similarly, we can process the CWEB source through the `cweave` program and end up with TEX source containing both the program text and the explanatory text we've written to make it a literate program. Processed through TEX, this file can be made a pretty document. An example of `cweave` output is shown in the next section. Note, however, that the `cweave` output, which would normally be typeset by TEX, was converted by hand into input appropriate for this magazine's typesetting software. Unfortunately, this means that much of the program text's nice typesetting is lost. (If you're interested, we'll be happy to email you a PostScript file of the TEX output from `cweave` and the C source file from `ctangle`. Send requests to one of our email addresses.)

Listing 1.

```
@ This means that we now have to strip
the search string from the command
line, and put it in |search|.

@<isolate search string@>=
if( ac <= 1 ) @<usage message@>@/
else if( strlen(*(++av)) < 1 ) @<usage message@>@/
else {@;
    search = *av;
    else {@;
      search = *av;

      ac--;
}@;
```

An Example of CWEB

1. An example of `fgrep` written in CWEB. Here, we provide a very simplistic version of the UNIX utility `fgrep`. We will take a command-line argument to find a character string and will search for that character string in the lines on standard input. If the string is found, the line is written to standard output. Normally, we'd read from an arbitrary list of files, but the point of this program is to show you some of the techniques, not write a complete filter.

2. A CWEB program consists of numbered *sections* and named *modules*. The numbered sections contain (optional) descriptive text and code. The sections with boldfaced names appear in the (optional) table of contents. The modules are pointers to code, which are expanded in the sections.

A basic filter consists of a pretty standard structure. CWEB allows us to outline it now, and in a process of step-wise refinement, fill it in later. We provide each module with a name, which has text later. This section contains an unnamed module. Unnamed modules are strung together sequentially to form the main body of the code.

```
<Header files 3>
<Global variables 5>
<Main program 4>
```

3. We begin with the obvious two header files. Note that we are providing text to an earlier module name, and that the modules are cross-referenced to section numbers.

```
<Header files 3> ≡
#include <stdlib.h>
#include <stdio.h>
(See also section 12.)
(This code is used in section 2.)
```

4. We also need to start laying out the main program in order to sketch it in the control flow. Again, we are using the step-wise refinement technique that Dijkstra first discussed.

Note that once we've given a module its full name, we need to use only the minimum recognizable prefix, followed by an ellipsis, to name it again. This section begins to define the module "<Main program>," but the text in our source below actually uses the name "<Main...>;" cweave does the appropriate expansion to print the full name, seen below.

Our modules don't need to neatly define groups of statements. See the "while" statement below for an example.

```
<Main program 4> ≡
main(ac, av)
int ac;
char **av;
{
  <isolate search string 6>
  while ( <we have read a line 8> )
    <search for the string 10>
  exit(0);
}
(This code is used in section 2.)
```

5. We have just skipped over vast amounts of detail. What kind of data structures are we using? How are we reading the next line?

Let's start with some variables.

```
<Global variables 5> ≡
char *search; /* the search string itself */
char buffer[BUFSIZ] /* the input buffer */
(This code is used in section 2.)
```

6. This means that we now have to strip the search string from the command line, and put it in *search*.

```
<isolate search string 6> ≡
  if(ac ≤ 1) <usage message 7>
  else if(strlen(*++av)) <usage message 7>
  else {
    search = *av;
    ac--;
  }
(This code is used in section 4.)
```

7. Notice that we've ignored the messy business of an error message in the last section and can take it up at our leisure. The encouraging news is that because the error processing may well be more complicated than the mainline code, we won't feel bad about writing an error routine. Too often, if the error condition requires more code than the correct branch of the *if*, we just pretend the error will never happen.

```
<usage message 7> ≡
{
  fprintf(stderr,"Usage: %s string\n", *av);
  exit(1);
}
(This code is used in section 6.)
```

8. The next module we need reads lines in the `while` loop above. We pass this off to a function we will define later. This function needs to return `true` if there's a line, and `false` if we've reached the last line of the input.

We could just use `fgets()` here, but we want to allow for later expansion to an arbitrary list of files. As a result, we use an intermediate routine instead.

```
<we have read a line 8> ≡
get_next_line(buffer)
(This code is used in section 4.)
```

9. We provide another unnamed module, which contains the text of `get_next_line()`.

```
get_next_line(buffer)
char *buffer;
{
  if(fgets(buffer,BUFSIZ,stdin) == NULL)
    return 0;
  else
    return 1;
}
```

10. How do we search for the string? Let's populate this module with a simple *if*.

```
<search for the string 10> ≡
  if(find_string(buffer,search))
    printf(" s",buffer);
(This code is used in section 4.)
```

11. What string search algorithm to use? We could choose any algorithm from Chapter 5 of *The Art of Computer Programming*. Instead, let's do something really simple-minded with the standard C string-handling routines.

```
find_string(b,s)
char *b, *s;
{
  char *initial;
  for(initial=b; initial; initial++)
  {
    initial = strchr(initial,*s);
    if(initial      NULL)
      return 0;
    if(strncmp(initial,s,strlen(s)) == 0)
      return 1;
  }
  return 0;
}
```

12. However, if we're going to use the string routines, we really need to declare them:

```
<Header files 3> + ≡
#include <string.h>
```

13. At the end of the TEX file generated by cweave, we get two useful indexes. The first index includes variables and the section numbers in which they appeared (underlined entries are sections where the variable is declared). The second index is an alphabetical list of the module names and the sections where they are defined and used.

An important thing to notice in our example above is that we presented the program as it came to us, sometimes top-down, sometimes inside-out, sometimes backtracking to add a variable or an include file, but the ctangle processor untangled it all and put the pieces into the correct order for the compiler.

In CWEB, because C is a little strict about newlines, the text to be compiled is pretty readable. However, in the original Pascal-based WEB, the code output from tangle became what's known as the "Pascal brick": The code is filled to 70 columns, with no line breaks for convenient reading. This is intentional. Humans are supposed to read the WEB code or the printed output of weave, not the extracted Pascal code. The compiler, particularly a Pascal compiler, doesn't care how the code is formatted.

We are reminded here of a colleague who was teaching a course in Pascal to some students at a German industrial concern that will remain nameless. He asked the students to write a pretty printing program to format their code. Several of them returned the next day with programs to fill-and-justify the source.

Further Reading, More to Come

The fountain of all knowledge is the Internet. Consult the newsgroup comp.programming.literate for ongoing discussions about it. Of particular interest are two tools called noweb and nuweb, which allow literate programming without imposing large overhead.

In addition to Knuth's original paper in *The Computer Journal*, two of Jon Bentley's "Programming Pearls" columns were devoted to literate programming, with examples provided by Knuth (*Communications of the ACM*, Vol. 29, May 1986, pp. 264-369, and June 1986, pp. 471-483).

The original WEB tools are described in detail, with programs, in Stanford Computer Science Technical Report 980 (September 1983). They are available on the Net from the Comprehensive TEX Archive Network (CTAN) sites: ftp.dante.de, ftp.tex.ac.uk, pip.shsu.edu. The Levy-Knuth CWEB is available on the Net in the same places.

Next month, we'll begin by discussing portability issues and then move on to a full example of a useful tool in CWEB. Stay tuned. ▲

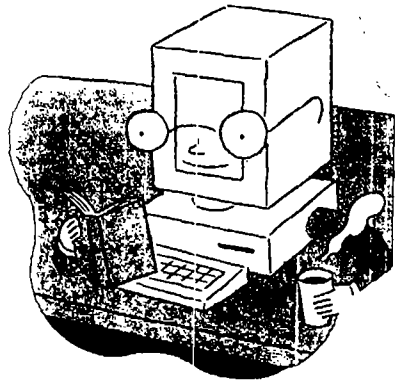
A Real Example, Part 1

by Jeffreys Copeland and Haemer

If you were with us last time, you must have read our introduction to literate programming, a discipline originally developed by Donald Knuth. We described the technique and provided a toy example using the CWEB tool. As promised, we're back this month with a larger example.

One of our references from last month was from Jon Bentley's "Programming Pearls" column in *Communications of the ACM*. In his June 1986 column, Bentley posed a problem to Knuth, for which Knuth wrote a solution in literate Pascal. Bentley then pointed out that because Knuth was proposing a new literary form, it should be reviewed as such, and asked Doug McIlroy to do so. Suffice it to say that the literary effort did not wow the critics.

Our intention in this and next month's columns is to replay that bit of history. We've chosen a useful problem, and one of us (J. Copeland) will write a literate program to solve it. Afterwards, the other of us (J. Haemer) will review the solution.



Like last month, the program was retypeset from its original version. If you'd like to see the original, drop us an email note, and we'll be glad to send you a PostScript file to produce it.

A Bit of History

"Ted, are you more fluent in English or French?"

"troff."

–conversation between Chris Kostanick and Ted Dolotta, circa 1984.

As we've discussed in our previous series, the history of `troff` is long and interesting. This ubiquitous UNIX utility began as a tool for driving the C/A/T phototypesetter, a huge mechanical monster that Wang Labs stopped manufacturing more than a decade ago.

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

Also, the tools built for and around troff have grown over the years. First were the preprocessors, like eqn and tbl. Later, there were tools to "transmogrify" the bits destined for the C/A/T into bits that could be read by Versatec plotters or other devices. One of our favorites allowed you to preview a *very* rough approximation of your final typesetter output on a Tektronix 4014 graphics terminal. The transmogrification business developed into quite a cottage industry, until folks realized that the typesetting engine should output a device-independent form of typesetting instructions rather than bits tied to a particular device, and then post-process that intermediate form into device codes for a C/A/T, or Autologic typesetter or Imagen laser printer.

Beginning in 1981, Interactive Systems Corp. developed INroff, from a clean base, then spent a great deal of time and effort trying to make it bug-for-bug compatible with troff.

Meanwhile, in the late 1970s, Brian Kernighan had developed a typesetter-independent troff, known as ditroff, based on the original C/A/T sources. When ditroff was released in 1982 (accompanied by the much-cited *Bell Labs Technical Report 97*) Kernighan's sources became the reference version that nearly everyone else adopted.

Interactive Systems Corp. developed *INroff*, from a clean base, then spent a great deal of time and effort trying to make it bug-for-bug compatible with *troff*.

An internationalized version of ditroff is distributed on the RS/6000 as part of AIX. The Free Software Foundation provides James Clark's groff, a version free of AT&T license restrictions, which is distributed with systems shipped by BSDI.

One of troff's failings is in page makeup. It is a wonderful galley system–that is, it is very good at setting characters into slugs (printer talk for lines of type) and getting those lines justified. But it is very bad at ensuring that groups of lines make pleasant paragraphs and that those paragraphs are well-composed into pages. (Until recently, page makeup was done from galleys, by trained pasteup people.)

Here are some of the most important page-layout tricks:

- Avoid widows (a header alone, or single lines from the beginning of a paragraph, at the bottom of a page).
- Avoid orphans (the last line of a paragraph alone at the beginning of a page).
- In general, the pages in a document should be the same length.
- In particular, facing pages should be made the same overall length, by adding extra leading between paragraphs if necessary.
- Figures and display equations will need to "float," or move from their original position in the galleys, but should not appear very far away from their original references.

One of TEX's notable features is that it makes a great deal of effort to do good page makeup. Both Michael Plass' work developing TEX's paragraph-building algorithm and Frank M. Liang's work on TEX's hyphenation algorithm resulted in doctorates.

During the original era of C/A/T-only output, in the late 1970s, our polyglot friend Kostanick and colleague Dolotta developed a post-processor at AT&T Bell Labs that took C/A/T output bits and did some page makeup. The processes vertically justified facing pages and rearranged slugs as necessary to prevent widows and orphans. It needed to have some knowledge of the page's design, in order to prevent destruction of headers and footers. As a result, you needed to tune and rebuild it for each new page layout. It was not a general-purpose tool.

Later, Kernighan and Chris Van Wyk developed the pj program, in order to justify ditroff output. Unfortunately, pj was created on the erroneous assumption that troff finds good page breaks and just needs a little help with justification. They followed this with a general-purpose page makeup program pm, which they describe in "Page Makeup by Postprocessing Text Formatter Output," (*Computing Systems*, [2] [Spring 1989], pp. 103-132).

We realized over the course of last summer that we needed just this program. We were producing too many one- and two-page troff documents that needed jiggering and adjusting and reformatting to get their pages to look nice. As is often the way with life, it's not until the snow started falling in the Front Range of the Rockies that we had time to sit down and write our version of pm.

The Problem: A Short Set of Rules

Our pages are made up of several kinds of slugs:

- A slug containing text (which Kernighan and Van Wyk call a *vbox*, in homage to TEX);
- *sp* slugs, which contain space;
- *ne* slugs of parameter *h*, which force a page break if there is not *h* vertical space left on the page.

Adjacent *sp* slugs are combined to contain the maximum of their heights; when an *sp* slug is output, its size may be increased to allow vertical justification. Text slugs are grouped, with tags between them, and come in two flavors: breakable and unbreakable. An unbreakable group–a page

header block, for instance–cannot be split across a page boundary. Breakable groups–for example, a paragraph–have a parameter *k*, which tells us how many lines must stay together if we break the group. Page title groups, *pt*, are a special case: We gather them as they appear and output one per composed page. If we have some groups left over when the document is complete, they populate the bit bucket. If we have too few, we use the last one again.

Note that we have postulated a problem that is slightly smaller than the one Kernighan and Van Wyk solved. We don't allow floating groups, and we ignore the problems of footnote blocks, so we are only dealing with streams of slugs and not with floats. Furthermore, we ignore the multicolumn problem and only work on making up pages of single columns.

How do we process the groups? How do we get tags into them? Read on.

s*n*	set the point size to *n*
f*n*	set the font to *n*
c*x*	print character *x*
C*xyz*	print special character named *xyz*
t*text*	print *text* with each character at its natural width (used only by `groff`)
H*n*	go to absolute horizontal position *n*
V*n*	go to absolute vertical position *n*
h*n*	move *n* units horizontally
v*n*	move *n* units vertically
nnc	move *nn* right, print character *c*; *nn* must be two digits
D*t* ... \n	draw a graphic of type *t*
n*b a*	end of line: *b* space before, *a* space after
w	paddable word space
p*n*	begin new page *n*; set V to 0
x... \n	device control

Device-independent `troff` produces strictly ASCII output. (Not true for the internationalized version on AIX–it produces output strictly in the codeset of the target printer.) There are a number of directives, as seen in the following table. Output is parsable by a post-processor.

Note in particular the device control directive x.... We can provide arbitary device controls with the `ditroff` directive `\X'...'`. Each tag and group marker takes the form of a device control and is generated by a `\X'...'` in a macro.

The Page Makeup Algorithm: An Overview

At the highest level, we are adopting Kernighan and Van Wyk's "algorithm 1" in its entirety. To wit:

```
<algorithm one > ≡
  <while slugs remain to be output > {
      <fill currpage with enough eligible slugs >
      <compose currpage into a page >
      if ( <this is not the last page > )
          <justify currpage to height pageht >
      <output the next pt group and currpage >
  }
```

Notice how we adopted the algorithm bodily from Kernighan and Van Wyk's article, and that CWEB allows us to presently ignore issues of data structure and keywords like *main*. This also means that we have completely ignored the difficult details of data structures for now.

Kernighan and Van Wyk use a series of queues to populate the current trial page with slugs. Slugs are read into a queue called *Input* and tagged with a serial number. They are identified as to type and routed into either *Bqueue*, containing breakable streams, or *Uqueue*, containing unbreakable streams of slugs. Slugs flow from *Input* to *Bqueue* or *Uqueue* and are processed onto the trial *currpage* immediately; this means that only one of *Bqueue* or *Uqueue* is populated at a time. Also, when *Bqueue* is populated, it only contains the minimum number of slugs to honor the parameter *k*, which tells us the minimum number of slugs from the group that can appear on any page, and which slugs are attached to each breakable group.

You may observe that since only one queue is occupied at a time, this is exactly the same as reading one slug at a time from the main input stream. This is true. However, we set up the queue management now, because it will be easier–here's one of our famous exercises for the reader–to add handling for floating blocks of slugs later.

Let's expand the first step of algorithm one, given the preceding discussion of queues.

We always want to add as many unbreakable groups as we can to the page before we begin to add text from breakable streams. However, when we reach a second page title group, we stop adding slugs to the page from *Uqueue*.

Similarly, since *Bqueue* is populated with the minimum number of slugs we can add without causing a widow or orphan, we either add all its contents or recycle them for the next try.

```
<fill currpage with enough eligible slugs > ≡
  <unblock all queues >
  <while there is a queue neither empty nor blocked>
  {
    <while Uqueue is available >
    {
      <try to add head of Uqueue to currpage >
    }
    <try to add all of Bqueue to currpage >
    if ( <Bqueue did not fit> ) {
      <empty Bqueue back to Input>
      <block Input>
    }
  }
```

This begs an important question: how to "try to add..." some or all of a queue to *currpage*, since we must check the height of the trial page at each addition.

```
<do a trial add > ==
    <add the trial item to the end of currpage >
    if( <height of trial page is greater than pageht > ) {
        <remove the trial item from currpage >
        <recycle trial item to the input queue >
        <block the input queue >
        <return failure status >
    }
    else {
        <return success status >
    }
```

Notice that checking the height of the trial page is complicated. First, we need to preserve the "natural" height of paddable spaces. We will do this by having two heights for each paddable space slug with each slug's given and expanded height; after each trial, we will reset the expanded height to the given height. Also, we cannot blindly increase all the paddable space on the page. Every trial page would succeed after the first paddable space was added, but they would have too much white space.

Kernighan and Van Wyk discuss three shortcuts to compute the height of the trial page. The goal is to avoid performing a complete calculation of the trial page's height, which includes justifying the page, for each slug we add. Each of these shortcuts has some failings for their larger problem, which involves floating items, but any of them would work in our postulated universe, which only includes running text.

For simplicity, we adopt the same shortcut Kernighan and Van Wyk reported, which adds slugs while the sum of the natural heights of slugs on the page is less than *pageht*, and then adds slugs only if a trial justification ensures that the page doesn't overflow. So, we can add another step:

```
<height of trial page is greater than pageht > ==
    (sum_of_heights() > pageht) | (trial_justification() > pageht)
```

Here We Take a 30-Day Break

That's about all we have time for this month. We've succeeded in outlining the problem of page justification. However, we've neatly ignored nearly all the issues of implementation and data structure. We'll return next month with some remaining details of the algorithm and some conclusions about it, and our on-the-spot exercise in literary criticism. ▲

Tomorrow and Beyond....
IBM® RS/6000™ and idp
Complete System Integration
• Hardware
• Software
• Connectivity
• RAID
Call for unparalleled service
International Data Products
Minnesota
Florida • California
800.846.7254
IBM Business Partner
IBM is a registered trademark of International Business Machines Corporation

Circle No. 16 on Inquiry Card

SALES • RENTALS
CONVERSIONS • INTEGRATIONS
RS/6000
RT/6150
SERIES/1
SYSTEM/36
AS/400
IBM Authorized Distributor Product Integrator
CALL:
(800) 888-2000
Dempsey BUSINESS SYSTEMS
Where IBM Quality Is Second Nature.
18377 Beach Blvd., Suite 323
Huntington Beach, California 92648
(714) 847-8486 • FAX: (714) 847-3149
IBM is a registered trademark of International Business Machines Corporation.

Circle No. 11 on Inquiry Card

Literate Programming: An Example, Part 2

by Jeffreys Copeland and Haemer

We Return from Our Break

11. We've had a month to mull over the algorithm we began presenting in February's column. If you were with us then, you'll remember that we were working on a `troff` filter to do page makeup as a literate programming problem. We've based this on Kernighan and Van Wyk's program `PM`, which they describe in "Page Makeup by Postprocessing Text Formatter Output" (*Computing Systems*, [2] (Spring 1989), pp. 103-132).

As before, this program has been re-typeset from its original form; to obtain the original, feel free to send us email.

To complicate matters, we goofed. We deleted the section numbers when we submitted last month's column. That's why this month's segment starts with section 11. That's also why there are references to section numbers that don't appear in this month's column.

Without further delay, let's jump into the problem where we left off.

Data Structures

12. We have a number of data structures we've ignored so far. We're about to need them, so we should decide how they will be arranged.

13. We begin by defining the data structure for the slug itself. We need the type of slug (we define the possi-

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

bilities as manifest constants), the slug's height (both its natural height and trial expanded height), its parameters and the actual `troff` output composing it. We add a `block_count`, which tells us how many of these slugs immediately preceeded this one.

```
#define     SLUG_BS 1
            /* breakable stream */
#define     SLUG_US 2
            /* unbreakable stream */
#define     SLUG_PT 3       /* page title */
#define     SLUG_SP 4       /* space slug */
#define     SLUG_NE 5       /* need slug */

struct slug {
    int type;
    int natural_ht;
    int trial_ht;
    int k;
    int block_count;
    char *text;
};
```

14. We also need to define the queues of working slugs. We'll implement these as arrays up to some size, and operate them as circular buffers. We'll also need a flag to indicate if the queue is blocked. We define queues for breakable and unbreakable streams and page title slugs, and the main input queue. Let's define the actions of the queue pointers now: *qhead* and *qtail* are equal if the queue is empty, *qtail* points at the last slug entered into the queue while *qhead* points at the last slug read from the queue. This means that both pointers are pre-increment.

One more (possibly subtle) point: Notice that we define a lookahead queue. We will populate this queue with slugs we need to preview for some reason. These slugs can't be read directly into *Input* because *Input* is expected to contain the minimum number of slugs to be processed at each stage. If *Input* did not perform this function, when we recycled rejected slugs to it, the order of text on the page would become garbled.

```
#define QSIZE 1024
struct queue {
    struct slug *Q[QSIZE];
    int qhead, qtail;
    int blocked;
} Bqueue, Uqueue, PTqueue, Input, Lookahead;
```

15. Lastly we need to define a structure for the current trial page. We could just define it as another queue, but it is a bit cleaner to implement a separate data structure.

```
struct currpage {
        struct slug *sp[QSIZE];
        int slugcount;
} currpage;
```

Utility Routines

16. Now that we've defined data structures, we need some routines to manage them. We can write these as freestanding functions.

Let's declare all the routines that don't return *int* here, to make our lives easier later on.

```
struct slug *get_next_lookahead();
struct slug *get_slug();
struct slug *read_slug();
```

17. We begin by writing routines to get a slug from the queue *Input* and to push rejected slugs back. In *get_slug*, we grab all available slugs from the *Input* queue; otherwise, we need to get one from *Lookahead*. We transfer some data from the previous slug if it is of the same type. This prevents us from needing global variables to keep track of needs and record counts. We also need a general-purpose *enqueue* routine to put slugs on a queue. In *unget_slug*, on the other hand, we back up the *Input* queue by one entry.

```
struct slug * get_slug()
{
 int wtype, n, k;

 if ( empty(Input)) {
   next(Input.qtail);
   Qtail(Input) = get_next_lookahead( );
 }
 /* get last type */
 wtype = Input.Q[Input.qhead]->type;
 n = Qhead(Input)->block_count;
 k = Qhead(Input)->k;                    /* point to next slug */
 next(Input.qhead);   /* transfer data from previous slug */
 Qhead(Input)->block_count =
 (wtype§Input.Q[Input.qhead]->type ) ? ++n : 0;
 Qhead(Input)->k =
 (wtype§Input.Q[Input.qhead]->type) ? k : 0;
 return( Qhead(Input));
}

enqueue(Q, s)
    struct queue Q;
    struct slug *s;
{
 next(Q.qtail);
 Qtail(Q) = s;
}

unget_slug()
```

```
{
 prev(Input.qhead);
}
```

18. To do the preceding operations, it helps to have macros to do modulo arithmetic on the queue pointers.

```
#define     next(x)
            {
              (x)++;
              (x)%=QSIZE;
            }
#define     prev(x)       (((x)== 0) ? (x) = QSIZE :(x)--);
#define     empty(q)      (q.qhead == q.qtail)
#define     Qhead(q)      (q.Q[q.qhead])
#define     Qtail(q)      (q.Q[q.qtail])
```

19. Next we introduce the routine to get a slug from *Input* into the appropriate queue for the slug type.

```
process_next_slug_from_Input_queue()
{
 int i;
 struct slug *s;

 s = get_slug();
 switch( s->type ) {
 case SLUG_US:
  do
  {
     enqueue(Uqueue, s);
  }
     while( (s=get_slug())->type == SLUG_US );
  unget_slug(s);
  break;
 case SLUG_PT:
  enqueue(PTqueue, s);
  break;
 case SLUG_BS:
  <special input processing for breakable stream slugs 20>;
  break;
 }
 return 1;
}
```

20. Why do breakable streams need to be handled specially? Notice that we read all of the US-type slugs we can into the queue. This is because they need to be treated as a block. (Kernighan and Van Wyk treat blocks of US slugs as a single compound slug.) With BS slugs, it is not as easy: We must read the minimum number consistent with the parameter *k*, but no more.

As a result, we need to do some special handling for *Bqueue* to make sure that there are at least *k* slugs in the queue at each end of the paragraph.

```
<special input processing for breakable stream slugs 20>=
 enqueue(Bqueue,s);
<prevent widows 21>
<prevent orphans 22>
```

This code is used in section 19.

21. Preventing widows and orphans is remarkably simple, in principle. At each end of a breakable stream, we just ensure that there are *k* slugs in *Bqueue*.

```
<prevent widows 21>=
  if( s->block_count == 0 ) {
    for( i = 1; i < s->k; i++ ) {
      if( (s = get_slug())->type≠SLUG_BS ) {
            unget_slug(s);
            break;
    }
          enqueue(Bqueue, s);
  }
}
```

This code is used in section 20.

22. Preventing orphans is marginally harder. We need to look ahead into the input stream to see how many BS slugs remain ahead.

```
<prevent orphans 22>=
  i = BS_slugs_to_come();
if( i > 0 && i < s->k ) {
  while( (s=get_slug())-> type == SLUG_BS)
          enqueue(Bqueue,s);
  unget_slug(s);
}
```

This code is used in section 20.

23. We also need a routine to populate the *Lookahead* queue. Note that this routine is essentially a poor man's version of *process_next_slug_from_Input_queue()*. The heart of this routine is the one that actually reads slugs from the troff input.

```
struct slug *get_next_lookahead()
{
  if( empty(Lookahead)) {
    next(Lookahead.qhead);
    Qtail(Lookahead) = read_slug();
  }
  next(Lookahead.qhead);
  return (Qhead(Lookahead));
}
```

24. *BS_slugs_to_come()* is essentially a dirty trick on

the data structures. We need to count up the BS slugs in the *Input* and *Lookahead* queues, and to read ahead until we run out of them.

```
BS_slugs_to_come()
{
  int count = 0;
  int i, type;

  <check Input 25>
  <check Lookahead 26>
  <read more, if needed 27>
}
```

25. We look in the *Input* queue first:

```
<check Input 25> ≡
for( i = Input.qhead;  i < Input.qtail;  i++ ) {
  if( (Input.Q[i])->type≠SLUG_BS ) return count;
  count++;
}
```

This code is used in section 24.

26. Similarly, we check *Lookahead*.

```
<check Lookahead 26> ≡
for( i = Lookahead.qhead; i < Lookahead.qtail; i++ ) {
  if( (Lookahead.Q[i])->type ≠ SLUG_BS ) return count;
  count++;
}
```

This code is used in section 24.

27. Lastly, we read as many more slugs as we need to get to the end of the current string of BS slugs.

```
<read more, if needed 27> ≡
do
{
  next(Lookahead.qtail);
  Qtail(Lookahead) = read_slug();
  if((type=(Qtail(Lookahead)->type)) ≡ SLUG_BS )
      count++;
}
while( type ≡ SLUG_BS );
return count;
```

This code is used in section 24.

Composing the Page

28. We have already discussed the page makeup algorithm in overview. The heart of that algorithm was outlined in last month's column, and now that we have utility routines to work with, it is time to expand it.

29. We begin with the procedure for unblocking all queues. Since we have a flag associated with each queue, this is quite simple.

```
<unblock all queues 29 > ≡
  Bqueue.blocked = Uqueue.blocked = Input.blocked = 0;
```

This code is used in section 7.

30. We need to check if queues are available. If a queue is not empty and not blocked, we can continue to add slugs to the page. If the ready queue is *Input*, we must process its head before proceeding to the main loop. We check using a convenient macro, which we define first.

```
#define  ready(q)  ((¬q.blocked) & (¬empty(q)))
<while there is a queue that is neither empty
   nor blocked 30> ≡
while( ready(Bqueue) || ready(Uqueue) ||
   (ready(Input) && <get from Input 31>))
```

31. Next, we can process the *Input* queue.

```
<get from Input 31 > ≡
process_next_slug_from_Input_queue()
```

This code is used in section 30.

32. Now we get to a slightly more difficult part. We will expand the processing for *Uqueue*.

```
<while Uqueue is available> ≡
while( ready(Uqueue) )
```

This code is used in section 7.

33. For the trial add, we follow the outline of the code in last month's issue. (This code–which was the module `do a trial add`–is intended as an outline of the code for processing both *Uqueue* and *Bqueue*, which we neglected to mention when we wrote it down. We won't expand that code further.) Remember that the "head" of *Uqueue* is conceptually a compound slug, so we add all the slugs that make up this unbreakable block. We save a pointer to the place we started in *Uqueue*, so we can back out the group of slugs if they don't fit.

```
<try to add head of Uqueue to currpage 33>≡
n=0;
while( ¬empty(Uqueue))
{
    next(Uqueue.qhead);
    currpage.sp[currpage.slugcount++] = Qhead(Uqueue);
    n++;
}
if( <height of trial page is greater than pageht 9> ) {
    <recycle last n slugs back to Input 34>
    Uqueue.blocked++;
}
```

This code is used in section 7.

34. Recycling the compound slug to *Input* is roughly the same loop as we use to put the text on *currpage*. We also have to remove the slug from *currpage*.

```
<recycle last n slugs back to Input 34> ≡
 currpage.slugcount -= n;
 for( i = 0; i < n; i++ )
       enqueue (Input,
          currpage.sp
            [currpage.slugcount + i]);
```

This code is used in sections 33 and 37.

35. We perform a similar set of operations for putting the contents of *Bqueue* onto the page, again using the code in section 8 as a model.

```
<try to add all of Bqueue to currpage 35> ≡
  n = 0;
  while(¬empty(Bqueue) ) {
    next(Bqueue.qhead);
    currpage.sp[currpage.slugcount++] = Qhead(Bqueue);
    n++;
}
```

This code is used in section 7.

36. We check if we've got too much for the trial page.

```
<Bqueue did not fit 36> ≡
      <height of trial page is greater than pageht 9>
```

This code is used in section 7.

37. We may need to recycle those slugs back to *Input*. For this we can use the same code we used for *Uqueue*.

```
<empty Bqueue back to Input 37> ≡
  <recycle last n slugs back to Input 34>
```

This code is used in section 7.

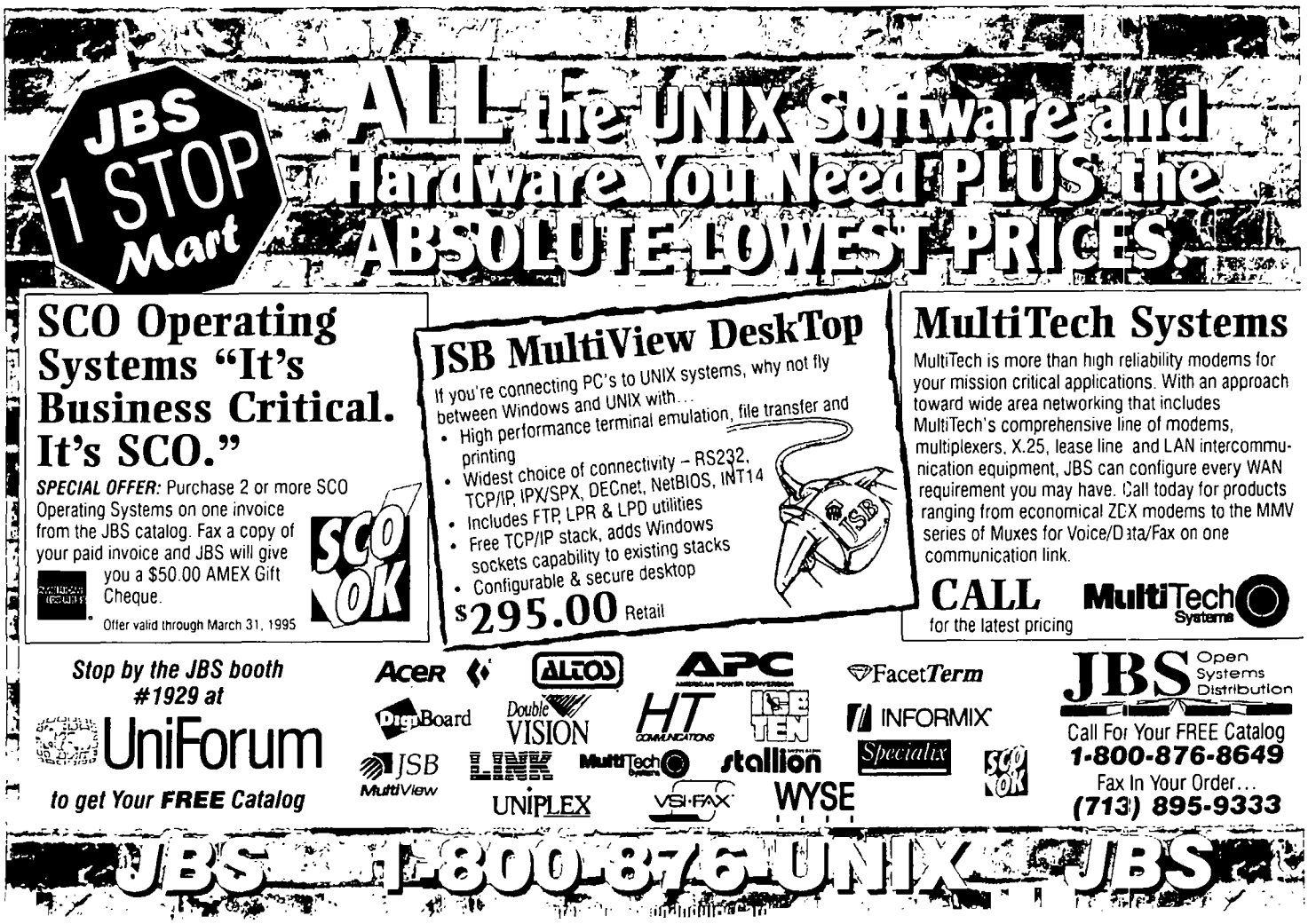
38. We can also simply dispose of blocking the *Input* queue.

```
<block Input 38> ≡
  Input.blocked++;
```

This code is used in section 7.

39. Justifying and Outputting the Page

Now that we've got slugs on the page, we need to have some utility routines to determine whether the page is full, and to justify it if it is.



We begin with the routine to check the "natural" height of the page. We could skip this routine entirely and gain some efficiency if we kept running track of the natural height of the page.

```
sum_of_heights()
{
 int i, sum = 0;
 for( i = 0; i < currpage.slugcount; i++ )
 sum += (currpage.sp[i])->natural_ht;
 return( sum );
}
```

40. The more complicated utility routine is that which does a trial justification of the page. We need to do a number of things with spaces as part of a trial justification. We first coalesce adjacent space slugs, zeroing space above the first and below the last text on the page, and leaving only the maximum of adjacent space slugs. If the page is still short, we can add more slugs.

```
trial_justification()
{
 int sum = 0, seen_first_text = 0, last_text = 0;
 int max_space = 0;
 int i;
 for ( i = 0; i < currpage.slugcount; i++ ) {
  if ( currpage.sp[i]->type == SLUG_SP ) {
   max_space = currpage.sp[i]->natural_ht;
   if ( seen_first_text   0 ) currpage.sp[i]->trial_ht = 0;
   else if ( currpage.sp[i-1]->type== SLUG_SP ) {
     max_space = max( max_space,
     currpage.sp[i]->natural_ht );
     currpage.sp[i-1]->trial_ht = 0;
   }
  }
  else {     /* not a space slug */
   if ( max_space > 0 || seen_first_text > 0 ) {
     currpage.sp[i-1]->trial_ht = max_space;
     max_space = 0;
   }
   if ( seen_first_text =0 )seen_first_text = i;
   last_text = i;
  }
 }  /* at end of page, ensure that trailing space is zero'd */
 for( i = last_text; i < currpage.slugcount; i++ )
  if( currpage.sp[i]->type =SLUG_SP )
  currpage.sp[i]->trial_ht = 0;
     /* now we must return the trial height */
 for( i = 0; i < currpage.slugcount; i++ )
  sum += currpage.sp[i]->trial_ht;
 return( sum );
}
```

41. Now we can actually compose the page. Actually, since we've got the page put together and done a good deal of work in our trial justification, this operation is actually null. The only remaining thing we need to do to the page before output is to justify it if we have a nonterminal page.

```
<compose currpage into a page 41>=
  {}
```

This code is used in section 4.

42. To justify the page, all we really need is the proper height. We judge this by calculating the excess space on the page and allocating it proportionately to the existing space slugs. We end the process by having each slug's printing height in *trial_ht*.

```
<justify currpage to height pageht 42>=
  justify_currpage(pageht);
```

This code is used in section 4.

```
43.  justify_currpage(pageht)
   int pageht;
{
 int total_space = 0, total_text = 0;
 int excess_space;
 int i;
 for( i = 0; i < currpage.slugcount; i++ )
  if( currpage.sp[i]->type = SLUG_SP  ) total_space +=
  currpage.sp[i]->trial_ht;
  else total_text += currpage.sp[i]->natural_ht;
  excess_space = pageht - (total_text + total_space);
 for( i = 0; i < currpage.slugcount; i++ )
  if( currpage.sp[i]->type == SLUG_SP )
   currpage.sp[i]->trial_ht +=
   currpage.sp[i]->trial_ht * excess_space / total_space;
  else currpage.sp[i]->trial_ht = currpage.sp[i]->natural_ht;
}
```

The Main Program

44. The main program is fairly straightforward from here. We get the page height and file names from the command line, and invoke Algorithm 1.

```
main(ac,av)
     int ac;
     char *av[];
{
  int i, n, pageht;

  process_args();
  <algorithm one 4>
  exit( 0 );
}
```

Out of Time and Space

45. We're out of space for the program exposition, so we won't show you the routines for reading slugs and outputting the page. (If you want to look at these, they're included in the original listing we discuss in the opening paragraphs.)

46. We couldn't leave you without some extra problems to think about. We've built a stripped-down version of the program Kernighan and Van Wyk describe, without accounting for floating blocks. What additional simplifying assumptions could we have made in the code, in particular in our queue structures, for a program that only uses static blocks?

We also leave you with the inverse problem: What additional code will allow us to handle floating blocks?

47. As promised, we now provide literary criticism by JSH of JLC's literate programming.

"Laziness is Next to Godliness"

48. A few years back, just for the heck of it, I spent an entire Usenix C++ conference wandering around asking people what object-oriented programming *wasn't* good for. Surprisingly many attendees told me that *all* code should be object-oriented.

(That other) Jeff (Copeland) has done a great job in the past few columns illustrating how to use CWEB by showing his literate solution to generating galleys from `troff` source. *This* Jeff (Haemer) wonders whether this wasn't overkill. Let's go back to last month's article and reread at the original problem: "We were producing too many one- and two-page `troff` documents that we were jiggering and adjusting and reformatting to get their pages to look nice." Personally, for a two-page memo, I don't need anything more than to get rid of widows and orphans. Will UNIX let me cobble together something to do that in less time than it took me to learn that when a typesetters say "slug," they aren't talking about me?

First off, the `troff` directive `.ne` *k* says that unless there are *k* lines left to the bottom of the page, `troff` should begin a new page.

This seems like a fine way to prevent widows. Here, for example, is a Perl program that will insert a `.ne 2` directive in front of every paragraph break.

```
#!/usr/bin/perl
            # how paragraphs begin
$breakslug = "^[.,]%%s*P%%s*[0-2]%%s*$";
            # loop through every input line
while (<>)
            # coating with Widow-Off
{  print ".ne 2%n" if /$breakslug/;
           # and printing it
   print;
}
```

If I preprocess my memos with this before printing, each paragraph or section will have at least two lines on the page. As an exercise, I'll leave the reader to add the few more lines needed to handle headers. You can't run this program through CWEB to produce fancy documentation, but it's only four lines of code.

I need to do more work to eliminate orphans, but usually that just means backing up in the source before each `.P` directive, and sticking in another `.ne 2` like this:

```
#!/usr/bin/perl
$troff_directive = "^[.,]";
    # how paragraphs begin
$breakslug = "^[.,]%%s*P%%s*[0-2]%%s*$";
    # suck all input into array
@a = <>;
    # one line at a time
LINE: for ($i=0 ; $i<=$#a ; $i++)
        # coat with Widow-Off
  if ($a[$i] =~ /$breakslug/) {
        # look for potential orphans
   for ($c=0, $n = $i; $c < 3; ) {
   $a[$i] .= ".ne 2%n";
   $_ = $a[--$n];
   next LINE if (/$breakslug/ $n < 0);
   $c++ unless /$troff_directive/;
   }
   $a[$n] .= ".ne 2%n"; # and fix
  }
}
 # avoid a one-line last page
$a[$#a-2] .= ".ne 2%n";

print @a;
```

Now let me critique my own critique. I've proposed a very UNIX-y alternative: a quick hack that provides a 99% solution to a small problem. Knuth's literate programming is a methodology for solving a large problem in which the added complexity of learning and using a new programming style is small. Nearly all software disciplines impose artificial constraints and structure on programmers in order to control a large, complex body of developing code. The UNIX tools-and-filters approach, in contrast, turns medium-size problems into small ones.

T-t-t-that's All, Folks!

49. Next month we begin an open-ended series on tools for the office. We've built a number of tools to handle our consulting ventures over the years, and find that they have some interesting educational features. We'll also be building tools specifically for the column, so feel free to write in with suggestions.

Until then, happy trails. ▲