Literate Programming in PAL and ObjectPAL

Lee Wittenberg*
Tipton Cole & Company
3006 Bee Cave Road, Suite B-200
Austin, TX 78746
(512) 329–0060
leew@pilot.njin.net

January 16, 1995

1 Introduction

A quiet revolution is taking place in programming circles—a revolution called "literate programming." In 1972, Edsger Dijkstra [1] wished for a "program written down as I can understand it, I want it written down as I would like to explain it to someone." Ten years later, Donald Knuth developed the original `WEB` system, coining the phrase "literate programming" in the process. Literate programming is precisely what Dijkstra wanted: the ability to write a program as you would explain it to another human being, rather than as your compiler would have it.

2 Literate Programming

The literate programming philosophy is that a program needs to be readable by both humans and computers, but these two audiences have quite different needs. Computers need the program to have a fairly rigid syntax, usually with names declared before their use. Human beings, on the other hand, prefer a somewhat looser style, and often like to see how a name is used before worrying about how it is defined. Section headings, cross-references, charts, graphs, footnotes, and the like are similarly helpful to humans, but irrelevant to a computer. The best possible documentation, therefore, will recognize the needs of both audiences.

A literate program, commonly called a *web*, consists of alternating *chunks* of descriptive text and code. The chunks are organized for the human reader. Usually, each text chunk describes the following code chunk, and makes full use of appropriate word processing techniques. Each code chunk has a name, which can be used in other code chunks to refer to that particular piece of code.

The code chunks can be extracted from the web and placed in order suitable for a compiler (or interpreter). This process is called *tangling* the web. The process of typesetting the web to produce a human-readable document is called *weaving*.

*Current address: Computer Science Department, Kean College of New Jersey, Union, NJ 07083.

3 Programming with `noweb`

I do all of my PAL and ObjectPAL programming using `noweb` [4], a literate programming tool by Norman Ramsey. A `noweb` program is a text file, usually with a '`.nw`' extension. Text chunks are introduced by the symbol '`@`' on a line by itself, code chunks by a line containing the chunk's name enclosed in a set of double angle brackets, '`<<`' and '`>>`', followed by an equals sign. Figure 1 gives an example of an extremely simple PAL web. Note the use of

```
@
This is a trivial example of a PAL
web.  It displays a ''hello'' message
and then sleeps for a bit
before it's done.
<<*>>=
MESSAGE "Hello, world!"
SLEEP <<An appropriate interval>>
@
How long is appropriate?
Let's say 2000, for now.
<<An appropriate interval>>=
2000
```

Figure 1: A simple PAL web.

the code chunk `<<An appropriate interval>>` in the definition of `<<*>>`.

Tangling in `noweb` is accomplished with the `notangle` command. Assuming that the web in Figure 1 is in the file `example.nw`, the DOS command

```
notangle example.nw > example.sc
```

tangles it. The output is redirected to create the file `example.sc`. Figure 2 shows the resulting script. By default, tangling begins with the `<<*>>` chunk (it is possible to specify a different chunk on the command line). As `notangle` copies this "root" chunk to its output, it replaces each chunk name it encounters with the appropriate definition, so `SLEEP <<An appropriate interval>>`

```
MESSAGE "Hello, world!"
SLEEP 2000
```

Figure 2: The result of tangling Figure 1

becomes `SLEEP 2000` in the output, and the resulting program is pretty much what you would expect.

But the real payoff comes with `notangle`'s companion program, `noweave`, which produces a beautifully formatted version of the web, designed to be read by humans. Figure 3 is the result. The text chunks are formatted in a

> This is a trivial example of a PAL web. It displays a "hello" message and then sleeps for a bit before it's done.
>
> ⟨ * 1⟩≡
> ```
> MESSAGE "Hello, world!"
> SLEEP ⟨An appropriate interval 2⟩
> ```
> Root chunk (not used in this document).
>
> How long is appropriate? Let's say 2000, for now.
>
> ⟨*An appropriate interval* 2⟩≡
> ```
> 2000
> ```
> This code is used in chunk 1.

Figure 3: Formatted output from Figure 1

roman font, with paragraphs properly indented. The code chunks are set in a fixed-width typewriter font, with chunk names in italics (and the << >> pairs replaced by ⟨ ⟩). The chunks are also *cross-referenced*: each chunk is numbered, and has a footnote that tells where it is used, if at all. Since this example is so short, there isn't much need for cross-referencing, but this feature is invaluable in a web of any size.

4 noweb in Action

The best way to understand what literate programming is all about is to build a web. This article is actually a web containing two programs: one in PAL 4.x, the other in ObjectPAL. From a single source file, `litprog.nw`, `noweb` generated a typeset copy of this article, a PAL script, and an ObjectPAL method.

Since the purpose of this article is to introduce the concept of literate programming, and not to discuss interesting Paradox programming tricks, I have chosen to implement a fairly simple algorithm and to make the PAL script and the ObjectPAL method do pretty much the same thing. Both calculate dates of Easter for a given range of years, and create a table—keyed to the year—containing the results. The algorithm is from Knuth [2]. Since expressions and identifiers are pretty much the same in PAL and ObjectPAL, the same variable names can be used for both implementations. The algorithm-related code chunks accompany the algorithm.

Let y be the year for which the date of Easter is desired.

Step 1. Set $g \leftarrow (y \bmod 19) + 1$. Easter calculations depend on a 19-year cycle. A year's place in this cycle is called its "golden number."

⟨*Calculate the golden number* 1⟩≡
```
golden_number = MOD(easter_year, 19)
                    + 1
```
This code is used in chunk 9.

Note that, in place of the single letters y and g, I use `easter_year` and `golden_number` as identifiers, preferring good programming style to mathematical consistency. I use `easter_year` instead of `year`, because the latter is a reserved word in PAL 4.x.

Step 2. Set $c \leftarrow \lfloor y/100 \rfloor + 1$. This step determines the century in which the year occurs (more or less—this description isn't quite accurate for years that are divisible by 100, but it will do). Note that '$\lfloor\ \rfloor$' is mathematical notation for the "floor" function.

⟨*Calculate the century number* 2⟩≡
```
century = FLOOR(easter_year/100) + 1
```
This code is used in chunk 9.

Step 3. Set $x \leftarrow \lfloor 3c/4 \rfloor - 12$, and $z \leftarrow \lfloor (8c+5)/25 \rfloor - 5$. A couple of corrections are necessary due to idiosyncracies in the Gregorian calendar. The former, x, calculates the number of years divisible by 4 in which there is no leap year, such as 1900. The latter, z, is a special correction designed to keep Easter in step with the moon's orbit.

⟨*Calculate necessary corrections* 3⟩≡
```
leap_year_correction
    = FLOOR(3*century/4) - 12
lunar_correction
    = FLOOR((8*century+5)/25) - 5
```
This code is used in chunk 9.

Step 4. Set $d \leftarrow \lfloor 5y/4 \rfloor - x - 10$. This tricky little formula computes a date that falls on a Sunday.

⟨*Find Sunday* 4⟩≡
```
sunday = FLOOR(5*easter_year/4)
            - leap_year_correction
            - 10
```
This code is used in chunk 9.

Step 5. Set $e \leftarrow (11g + 20 + z - x) \bmod 30$. If $e = 25$ and the golden number g is greater than 11, or if $e = 24$, increase e by 1. The "epact," e, is used to calculate when a full moon occurs.

⟨*Calculate the epact* 5⟩≡
```
epact = MOD(11*golden_number + 20
            + lunar_correction
            - leap_year_correction,
            30)
IF (epact = 25 AND golden_number > 11)
    OR (epact = 24)
THEN
    epact = epact + 1
ENDIF
```
This code is used in chunk 9.

Step 6. Set $m \leftarrow 44 - e$. If $m < 21$ then set $m \leftarrow m + 30$. Easter is defined as "the first Sunday following the first full moon which occurs on or after March 21." This formula finds the first full moon after March 21.

⟨*Find the full moon* 6⟩≡
```
full_moon = 44 - epact
IF full_moon < 21
THEN
    full_moon = full_moon + 30
ENDIF
```
This code is used in chunk 9.

Step 7. Set $n \leftarrow m + 7 - ((d + m) \bmod 7)$. This formula finds the Sunday immediately after the aforementioned full moon.

⟨*Find Easter Sunday* 7⟩≡
```
easter_sunday = full_moon + 7
    - MOD(sunday + full_moon, 7)
```
This code is used in chunk 9.

Step 8. If $n > 31$, the date is $(n - 31)$ April; otherwise it is n March.

⟨*Get the month* 8⟩≡
```
IF easter_sunday > 31
THEN
    easter_sunday
        = easter_sunday - 31
    easter_month = "April"
ELSE
    easter_month = "March"
ENDIF
```
This code is used in chunk 9.

Putting it all together:

⟨*Calculate the date of Easter* 9⟩≡
⟨*Calculate the golden number* 1⟩
⟨*Calculate the century number* 2⟩
⟨*Calculate necessary corrections* 3⟩
⟨*Find Sunday* 4⟩
⟨*Calculate the epact* 5⟩
⟨*Find the full moon* 6⟩
⟨*Find Easter Sunday* 7⟩
⟨*Get the month* 8⟩
This code is used in chunks 10 and 18.

The desired date is contained in the variables `easter_sunday`, `easter_month`, and `easter_year`.

Aside: Notice that I've been working bottom-up rather than top-down, first coding the algorithm steps, then putting them together when all the steps have been coded. This goes against all received wisdom about good programming practice, but literate programming seems to change some of the rules. The primary focus in a literate program is explaining it to the reader. In a case like this, where the algorithm already exists, it's much more natural to develop a program as I have done, and easier for the reader to understand, as well. However, when I don't have an algorithm already prepared, I tend to write top-down, using the chunks for old-fashioned stepwise refinement, as you shall see in the following.

4.1 A PAL 4.x Script

Rather than bother with any input, the PAL script builds a table, `easter1.db`, that contains dates of Easter calculated for the years 1950 through 2000. After creating the table, the program iterates over the years, adding an entry to the table for each year.

⟨*EASTER.SC* 10⟩≡
```
⟨Create the easter1.db table 12⟩
FOR easter_year FROM 1950 TO 2000
    ⟨Calculate the date of Easter 9⟩
    ⟨Add the date to easter1.db 13⟩
ENDFOR
⟨Clean up the workspace 11⟩
```
This code is written to file `EASTER.SC`.

The `-R` option tells `notangle` which code chunk to begin with, so the command

```
notangle -REASTER.SC litprog.nw > easter.sc
```

extracts the PAL script, creating the `EASTER.SC` file.

The easiest way to clean up is to issue the `RESET` command. It has the disadvantage of removing anything that was on the workspace before the script started, but will do for this simple example.

⟨*Clean up the workspace* 11⟩≡
```
RESET
```
This code is used in chunk 10.

4.1.1 Creating the table

The `easter1.db` table will have three fields: *year*, *month*, and *day*. *Month* and *day* represent the date of Easter for the specified *year*, which naturally enough, is the key field. *Year* and *day* are both numeric (short numbers will do), while *month* is alphanumeric (5 characters will suffice as Easter occurs only in March or April).

The program has to put the table's image on the workspace and get into Coedit mode before adding entries. Since many table operations require subsidiary variables, which may need initialization, the ⟨*Any other initializations needed for* `easter1.db`⟩ chunk is useful for writing initialization code wherever in the web is most appropriate, yet insuring that the initialization is performed immediately after the table is created.

⟨*Create the* `easter1.db` *table* 12⟩≡
```
CREATE "easter1"
    "Year"  :   "S*",
    "Month" :   "A5",
    "Day"   :   "S"
⟨Any other initializations needed for easter1.db 14⟩
COEDIT "easter1"
```
This code is used in chunk 10.

4.1.2 Adding an Easter date to the table

`APPENDARRAY` appears to be the simplest technique PAL 4.x provides for adding records to a table.

⟨*Add the date to* `easter1.db` 13⟩≡
```
newdate[2] = easter_year
newdate[3] = easter_month
newdate[4] = easter_sunday
APPENDARRAY newdate
```
This code is used in chunk 10.

Unlike simple variables, the `newdate` array must be declared before it is used. Its first element refers to the `easter1.db` table.

⟨*Any other initializations needed for* `easter1.db` 14⟩≡
```
ARRAY newdate[4]
newdate[1] = "easter1"
```
This definition is continued in chunk 15.
This code is used in chunk 12.

4.1.3 A minor problem

Unfortunately, PAL 4.x does not provide a `FLOOR` function, although ObjectPAL does. Since the Easter algorithm never passes a negative number to `FLOOR`, the `INT` function provides a cheap substitute.

⟨*Any other initializations needed for* `easter1.db` 14⟩+≡
```
PROC FLOOR(n)
    RETURN INT(n)
ENDPROC
```

Aside: Notice the +≡ in this chunk definition. `noweave` uses this to indicate that the chunk has already been defined, and that the code in this definition will be concatenated with the code from previous definitions when the web is tangled.

4.1.4 PAL wrapup

The PAL script is now complete. Although simple, it demonstrates the literate programming style quite well. Each piece of the script is small and easy to understand. Each non-trivial part of the program is relegated to a chunk, whose name is usually a complete sentence. Since chunk names are typographically distinct from the actual code, they are much easier to read than procedure names would be. In addition, *there is no run-time overhead involved in using chunks*, as there would be if I had used procedures for refinements.

Each chunk is accompanied by a complete description of what the chunk does, including explanations of *why* things were (or weren't) done in a particular way. Observations that would be intrusive in standard program comments fit quite well in these descriptions. *The program is organized for the human reader, not the compiler.*

4.2 An ObjectPAL Method

`noweb` is language-independent; it doesn't care whether you are programming in PAL, ObjectPAL, or even Pascal or C. `notangle`'s `-R` option makes it possible to include programs written in several different languages in a single web.[1] As I mentioned earlier (Section 4), with a little bit of care, it is possible to share code between PAL and ObjectPAL applications. If it is necessary to maintain both a DOS and Windows version of a program, putting them in the same web will help keep them "in sync," making it much less likely that any changes will render the two versions incompatible.

A useful technique in ObjectPAL is to write a program as a `pushButton` method, linking its execution to a button on some form. I create such a method here, assuming that it will be attached to a button labelled "Generate Easter Table" on some form.

I use the `.txt` extension for ObjectPAL files, because that's what the Paradox for Windows "Edit|Paste From File" menu item seems to prefer.

[1] This is particularly useful when a program is to be invoked by one or more batch files. The batch files can be in the same web as the program they invoke, and can be easily updated whenever necessary due to program changes.

⟨*EASTER.TXT* 16⟩≡
```
METHOD pushButton(VAR eventInfo Event)
VAR
    ⟨Local pushButton variables 17⟩
ENDVAR
    ⟨Generate the easter2.db table 18⟩
ENDMETHOD
```
This code is written to file `EASTER.TXT`.

In ObjectPAL, all variables have to be declared. If the web didn't also contain a PAL 4.x script, I would have declared these variables as they made their appearence in the algorithm. Since I didn't do it then, I'd better do it now. The variables are declared to be of type `Number` (except for `easter_month`, which is a `String`), rather than `SmallInt` or `LongInt`, because for some inexplicable reason, ObjectPAL gets upset[2] when one tries to assign the result of `FLOOR` to an integer variable (even though the result *must* be integral).

⟨*Local* pushButton *variables* 17⟩≡
```
    easter_year             Number
    easter_month            String
    easter_sunday           Number
    golden_number           Number
    century                 Number
    leap_year_correction    Number
    lunar_correction        Number
    sunday                  Number
    epact                   Number
    full_moon               Number
```
This definition is continued in chunk 21.
This code is used in chunk 16.

Since the form can take care of all necessary input, `FirstYear` and `LastYear` will control the `FOR` loop, rather than the "hard-coded" years 1950 and 2000, and I assume that the form will contain Field objects with these names. Alternatively, it may provide them as global variables that are initialized somehow before `pushButton` is invoked, or as symbolic constants. In any event, this is the form's responsibility.

Not surprisingly, the control flow is similar to that of the PAL script.

⟨*Generate the* easter2.db *table* 18⟩≡
```
⟨Create the easter2.db table 20⟩
FOR easter_year FROM FirstYear TO LastYear
    ⟨Give other Windows programs a chance to run 19⟩
    ⟨Calculate the date of Easter 9⟩
    ⟨Give other Windows programs a chance to run 19⟩
    ⟨Add the date to easter2.db 23⟩
ENDFOR
⟨Give other Windows programs a chance to run 19⟩
⟨Clean up the desktop 22⟩
```
This code is used in chunk 16.

The only surprise here is the ⟨*Give other Windows programs a chance to run*⟩ chunk, which is scattered all over the place. Windows is a "cooperative multitasking" system, so each application must voluntarily give up control of the CPU from time to time, to let other applications run. Since ObjectPAL doesn't do this automatically, I like to give other programs a chance whenever possible. I use a smaller font for this chunk name so that it's fairly unobtrusive, and doesn't interfere with the program proper.

⟨*Give other Windows programs a chance to run* 19⟩≡
```
sleep()
```
This code is used in chunk 18.

There's no general desktop cleanup necessary yet; I'll add cleanup code as it becomes necessary.

4.2.1 Creating the table

It appears that the only way to create a table in ObjectPAL is to use the `create` pseudo-method with a `Table` variable. However, only a `TCursor` can actually add records to a table. There doesn't seem to be any getting around the fact that two variables are needed for what should be a one variable job.

The `easter2.db` table has the same structure as `easter1.db`, described in Section 4.1.1.

⟨*Create the* easter2.db *table* 20⟩≡
```
dummy = create "easter2.db"
            with
                "Year"  :   "S",
                "Month" :   "A5",
                "Day"   :   "S"
            key "Year"
        endcreate
easter_table.open("easter2.db")
easter_table.edit()
```
This code is used in chunk 18.

Since the `Table` variable is never used for anything other than the initial creation, "`dummy`" seems an appropriate name.

⟨*Local* pushButton *variables* 17⟩+≡
```
dummy           Table
easter_table    TCursor
```

The `easter_table` must be closed when the method ends to make sure the last record added gets posted to the table.

⟨*Clean up the desktop* 22⟩≡
```
easter_table.close()
```
This code is used in chunk 18.

[2] At least it used to; I don't know about 5.0.

4.2.2 Adding an Easter date to the table

Adding records to a table in ObjectPAL is straightforward: insert a new record, and assign the appropriate values to the appropriate fields.

⟨*Add the date to* `easter2.db` 23⟩≡
```
easter_table.insertRecord()
easter_table."Year"  = easter_year
easter_table."Month" = easter_month
easter_table."Day"   = easter_sunday
```
This code is used in chunk 18.

4.2.3 ObjectPAL wrapup

Now that the ObjectPAL method is complete, note that the only differences between it and the PAL script of Section 4.1 are syntactic.

4.3 Version Control Information

Since a `noweb` program is a text file, version control information can easily be included in the woven output. This information is updated automatically, whenever the program is "checked in." The following is the relevant information for the programs in this article.

File:	`litprog.nw`
Author:	LEEW
Revision:	2.2
Last Modified:	1995/01/16 11:19:51

4.4 List of Chunk Names

`noweb` can automatically generate an index of chunk names used in the web, and the pages on which they are defined and used. References to chunk definitions are underlined.

⟨*Add the date to* `easter2.db` 23⟩ 18, 23
⟨*Add the date to* `easter1.db` 13⟩ 10, 13
⟨*Any other initializations needed for* `easter1.db` 14⟩ 12, 14, 15
⟨*Calculate necessary corrections* 3⟩ 3, 9
⟨*Calculate the century number* 2⟩ 2, 9
⟨*Calculate the date of Easter* 9⟩ 9, 10, 18
⟨*Calculate the epact* 5⟩ 5, 9
⟨*Calculate the golden number* 1⟩ 1, 9
⟨*Clean up the desktop* 22⟩ 18, 22
⟨*Clean up the workspace* 11⟩ 10, 11
⟨*Create the* `easter2.db` *table* 20⟩ 18, 20
⟨*Create the* `easter1.db` *table* 12⟩ 10, 12
⟨*EASTER.SC* 10⟩ 10
⟨*EASTER.TXT* 16⟩ 16
⟨*Find Easter Sunday* 7⟩ 7, 9
⟨*Find Sunday* 4⟩ 4, 9
⟨*Find the full moon* 6⟩ 6, 9
⟨*Generate the* `easter2.db` *table* 18⟩ 16, 18
⟨*Get the month* 8⟩ 8, 9
⟨*Local* `pushButton` *variables* 17⟩ 16, 17, 21
⟨*Give other Windows programs a chance to run* 19⟩ 18, 19

4.5 List of Identifiers

`noweb` can also keep track of identifier definitions and usage.

`century`: 2, 3, 17
`dummy`: 20, 21
`easter_month`: 8, 13, 17, 23
`easter_sunday`: 7, 8, 13, 17, 23
`easter_table`: 20, 21, 22, 23
`easter_year`: 1, 2, 4, 10, 13, 17, 18, 23
`epact`: 5, 6, 17
`eventInfo`: 16
`FLOOR`: 2, 3, 4, 15
`full_moon`: 6, 7, 17
`golden_number`: 1, 5, 17
`leap_year_correction`: 3, 4, 5, 17
`lunar_correction`: 3, 5, 17
`newdate`: 13, 14
`pushButton`: 16
`sunday`: 4, 7, 17

5 Producing Formatted Output

The `noweave` program, discussed in Section 3, does not produce its formatted output directly. It produces a file that is then processed by the TEX[3] typesetting system. TEX is the typesetting system of choice for literate programming systems for a variety of reasons:

1. TEX is readily available. Implementations exist for pretty much every computer in existence, and are usually available free of charge. I use the excellent emTEX implementation for DOS, by Eberhard Mattes, which is freely distributable.
2. TEX is portable. It is designed to be as machine-independent as possible. Output generated by emTEX on my PC and that generated by, say, a UNIX implementation will be *identical*.
3. TEX is stable. Before a program can be certified "TEX," it must pass a rigorous test suite called the "trip test." Since TEX is not a commercial product, it is not subject to the whims of a manufacturer.
4. The quality of TEX output is significantly better than that of any commercial word processor or desktop publisher on the market today.

[3] "Insiders pronounce the χ of TEX as a Greek chi, not as an 'x', so that TEX rhymes with the word blecchhh. It's the 'ch' sound in Scottish words like *loch* or German words like *ach*; it's a Spanish 'j' and a Russian 'kh'. When you say it correctly to your computer, the terminal may become slightly moist." [3]

On the other hand, there is no reason a literate programming system *has* to use TEX. A number of non-TEX systems exist, and many are suitable for programming in PAL and ObjectPAL (see Section 6).

6 Availability

Although it is not in the public domain, `noweb` is freely distributable—this article (and the programs it generates) were created using the DOS implementation. Other literate programming systems suitable for PAL and ObjectPAL are also available. FunnelWeb and Nuweb are TEX-based; CLiP can be used with any word processor; and WinWordWEB is a collection of macros that provide a simple literate programming environment in Word for Windows. All of these systems are freely distributable.

If you would like to find out more about literate programming, I recommend that you subscribe to the "LitProg" discussion group on the Internet. All discussion is via electronic mail, so it should be possible to subscribe from CompuServe, or any other service that can send and receive Internet mail. To subscribe, send a message to `LitProg-Request@SHSU.edu`[4], and include

```
SUBSCRIBE LitProg "your name"
```

in the body of the message. As the official welcoming notice says, "Novices are welcome; it is intended that this group should be a place where newcomers can be welcomed into the fold as well as a place where seasoned literate programmers can discuss fine points of technique."

7 Discussion

In the introduction, I stated that literate programming is revolutionary. It is not "the only game in town," though. The "visual programming" revolution is also gaining ground. Although visual programming is receiving the lion's share of publicity, I believe that literate programming is much more significant for the professional programmer.

Visual programming, like BASIC before it, is designed for the neophyte. Writing a program is easier than ever before. However, just as BASIC's `GOTO` statement led to "spaghetti code" that was impossible to maintain, visual programming systems create their own maintenance problems. In Paradox for Windows, for example, the connections between objects are invisible, creating the potential for spaghetti of another kind. Objects can refer to other objects in many ways, and there is no way—short of examining every object—to determine whether a non-local identifier (e.g. `FirstYear` and `LastYear` in Section 4.2) is a variable, constant, or graphical object, and where in the "container hierarchy" it resides. Maintaining such a system can become a nightmare.

[4] CompuServe users should prepend "`>INTERNET:`" to this address.

Literate programming, on the other hand, is not for the beginner. The literate programmer must be proficient in at least two languages: a programming language, like PAL or ObjectPAL, and a text formatting language, like TEX or a commercial word processor. Since the bulk of a web is explanation, rather than code, the programmer has to take the time to think through the program in order to be able to explain it. As in any programming methodology, time spent in thought "up front" translates into reduced debugging and easier maintenance. What literate programming adds to the mix is that the programmer's thoughts no longer disappear into thin air once the program is written; they are preserved in the web. The programmer who maintains the web has these thoughts as a foundation for future work. For the professional programmer, *maintenance is everything.* A literate program is a maintainable program.

But, above all, literate programming is *fun.* The tedium of rearranging bits of code to cater to a compiler is gone. In its place is a process of discovery. As you write your explanations, the code unfolds naturally, as if it had always been there and you just now happened to find it. The phenomenon of "code that seems to write itself" is common among literate programmers, as is the practice of passing programs around for comments—and actually getting them!

I could go on and on. Literate programming techniques free the programmer to focus on the uniquely human aspects of programming, leaving the computer to perform the more mundane tasks. The benefits are there for the taking. Take them.

References

[1] Edsger W. Dijkstra. Notes on structured programming. In *Structured Programming*, pages 1–82. Academic Press, 1972.

[2] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, second edition, 1973.

[3] Donald E. Knuth. *The TEXbook*, volume A of *Computers & Typesetting.* Addison-Wesley, 1986.

[4] Norman Ramsey. Literate programming simplified. *IEEE Software*, pages 97–105, September 1994.