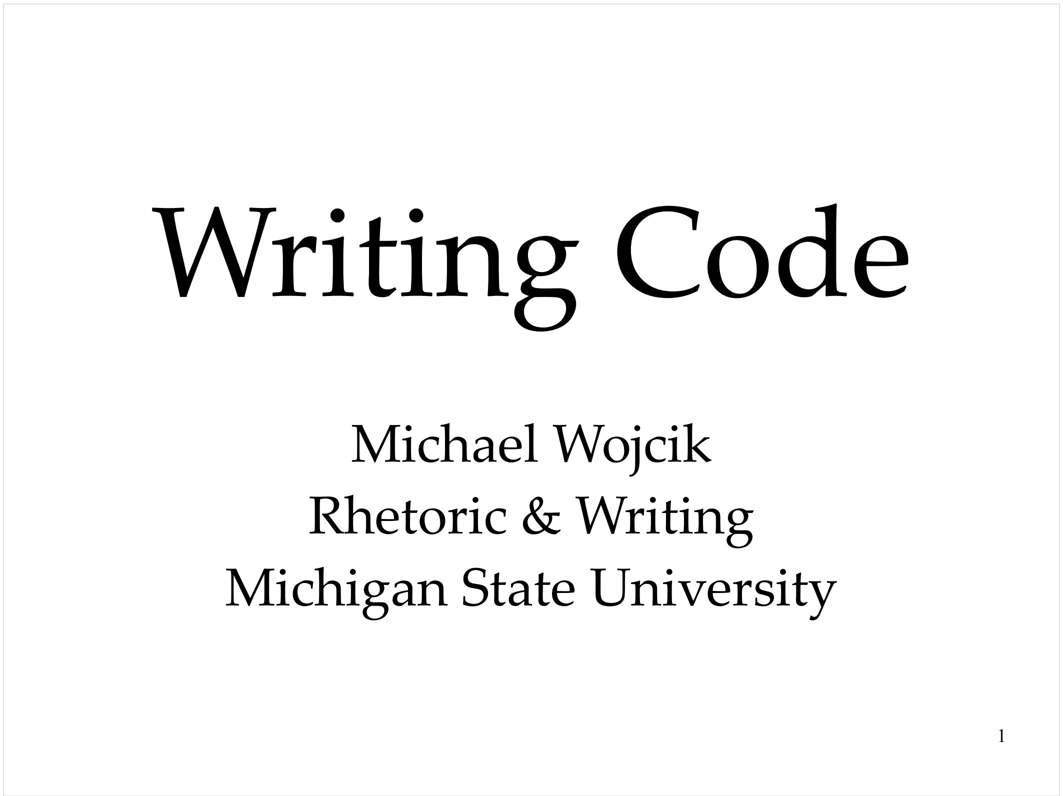


Writing Code

Michael Wojcik
Rhetoric & Writing
Michigan State University

Thanks, everybody, for coming here this afternoon. I'd also like to thank the Association of Teachers of Technical Writing for this opportunity to speak to you, and my fellow panelists for sharing it with me.

And I have to say a quick work of thanks to the faculty of the Rhetoric and Writing program at Michigan State University, particularly Bill Hart-Davidson, and my fellow graduate students, who provide an amazing intellectual community.

And finally, my deepest gratitude to Malea Powell for her inspirational example and unfailing support.

I am not a teacher of technical writing, or indeed a teacher of anything. I am on occasion a technical writer, and in fact I'm going to argue that I'm more often a technical writer than my timesheets suggest, but most of the time I'm a Principal Software Systems Engineer for Micro Focus International. I'm a programmer, a coder.

And what I'm going to talk about today...

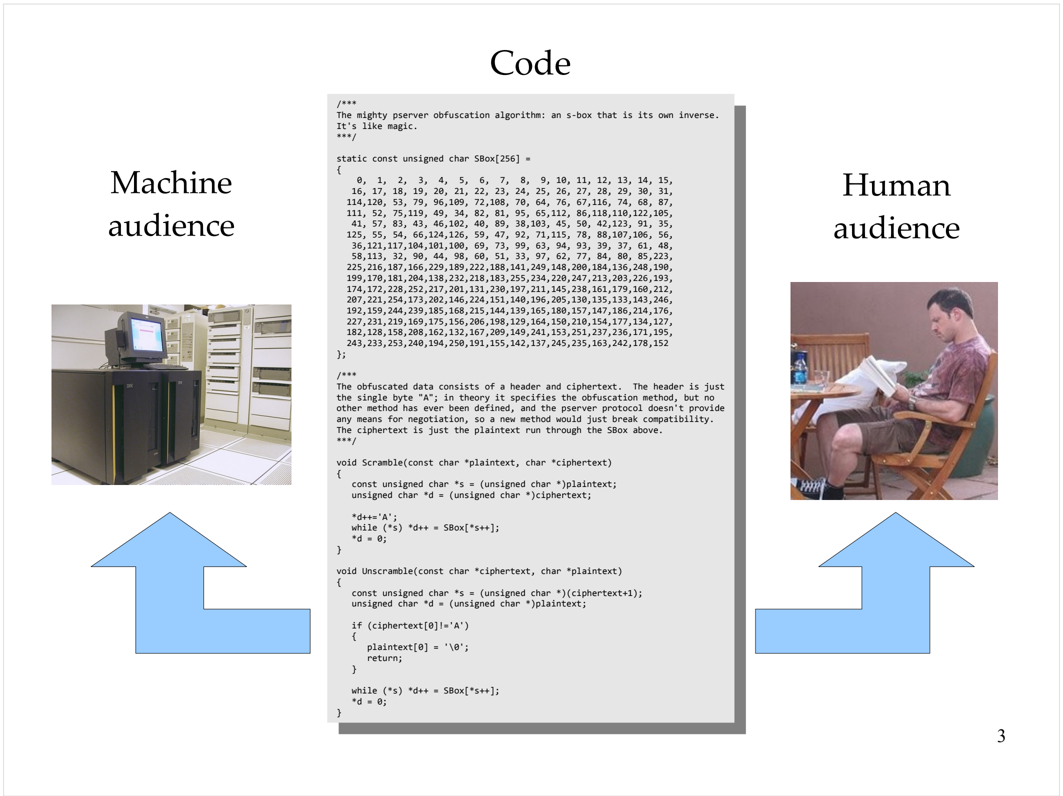
Writing Code

2

… is writing code – with an emphasis on the writing.

Programming, writing code, is technical writing. Programmers are technical writers. They generally don't know that they're technical writers, and most of them are poor technical writers.

And, I'd suggest, there are people who might be able to teach programmers to be better technical writers, and there's incentive to do that.



Code

Machine audience

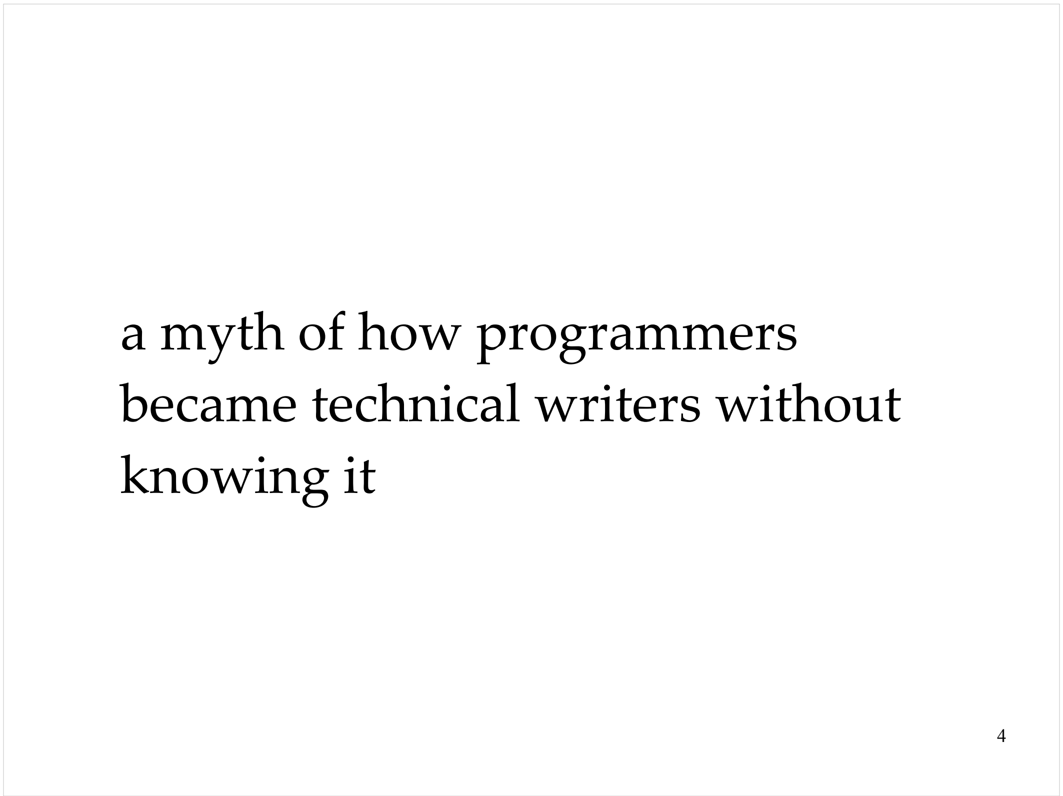
Human audience

```
/***
The mighty pserver obfuscation algorithm: an s-box that is its own inverse.
It's like magic.
***/

static const unsigned char SBox[256] =
{
    0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
   16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
  114,120, 53, 79, 96,109, 72,108, 70, 64, 76, 67,116, 74, 68, 87,
  111, 52, 75,119, 49, 34, 82, 81, 95, 65,112, 86,118,110,122,105,
   41, 57, 83, 43, 46,102, 40, 89, 38,103, 45, 50, 42,123, 91, 35,
  125, 55, 54, 66,124,126, 59, 47, 92, 71,115, 78, 88,107,106, 56,
   36,121,117,104,101,100, 69, 73, 99, 63, 94, 93, 39, 37, 61, 48,
   58,113, 32, 90, 44, 98, 60, 51, 33, 97, 62, 77, 84, 80, 85,223,
  225,216,187,166,229,189,222,188,141,249,148,200,184,136,248,190,
  199,170,181,204,138,232,218,183,255,234,220,247,213,203,226,193,
  174,172,228,252,217,201,131,230,197,211,145,238,161,179,160,212,
  207,221,254,173,202,146,224,151,140,196,205,130,135,133,143,246,
  192,159,244,239,185,168,215,144,139,165,180,157,147,186,214,176,
  227,231,219,169,175,156,206,198,129,164,150,210,154,177,134,127,
  182,128,158,208,162,132,167,209,149,241,153,251,237,236,171,195,
  243,233,253,240,194,250,191,155,142,137,245,235,163,242,178,152
};

/***
The obfuscated data consists of a header and ciphertext.  The header is just
the single byte "A"; in theory it specifies the obfuscation method, but no
other method has ever been defined, and the pserver protocol doesn't provide
any means for negotiation, so a new method would just break compatibility.
The ciphertext is just the plaintext run through the SBox above.
***/

void Scramble(const char *plaintext, char *ciphertext)
{
   const unsigned char *s = (unsigned char *)plaintext;
   unsigned char *d = (unsigned char *)ciphertext;

   *d++='A';
   while (*s) *d++ = SBox[*s++];
   *d = 0;
}

void Unscramble(const char *ciphertext, char *plaintext)
{
   const unsigned char *s = (unsigned char *)(ciphertext+1);
   unsigned char *d = (unsigned char *)plaintext;

   if (ciphertext[0]!='A')
   {
      plaintext[0] = '\0';
      return;
   }

   while (*s) *d++ = SBox[*s++];
   *d = 0;
}
```

Software, as source code – that is, as it's produced by programmers – has two audiences: a machine audience and a human audience.

We normally think of the machine audience: the compilers and interpreters that convert source code to machine instructions, and the hardware and supporting software that executes it.

But software is also read by people. It's read by its authors; programmers working on anything non-trivial have to refer back to the code they've written for the names they've used, interfaces they've defined, and so on. And it's almost inevitably read again later by maintenance programmers, who are fixing, updating, or enhancing the software. That maintenance programmer may be the original author – but even with well-written software, if you've been away from it for a year, you're going to have to spend some time reading it.

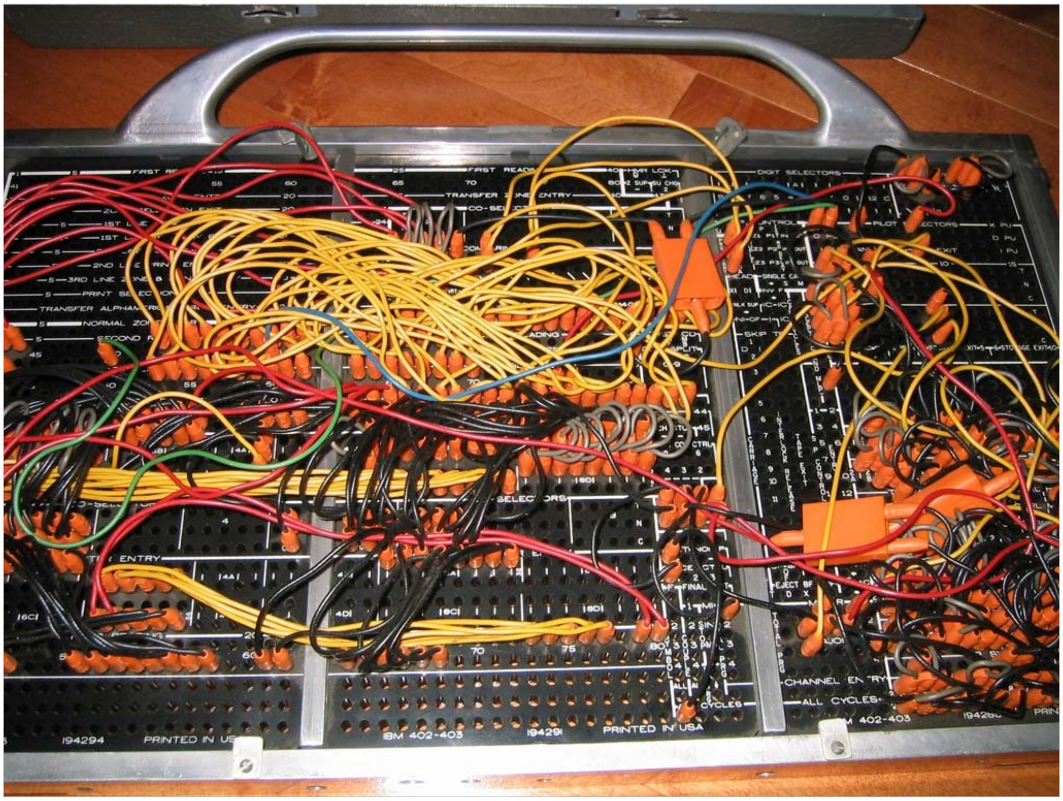


a myth of how programmers became technical writers without knowing it

I'm going to tell a story about some of the ways programmers have tried to address these two audiences. I want to emphasize that this is not a history, because my academic discipline, rhetoric, fetishizes history but often doesn't do it well. I haven't done the archival and human research to make this a proper history, and I'm not trained in historiography.

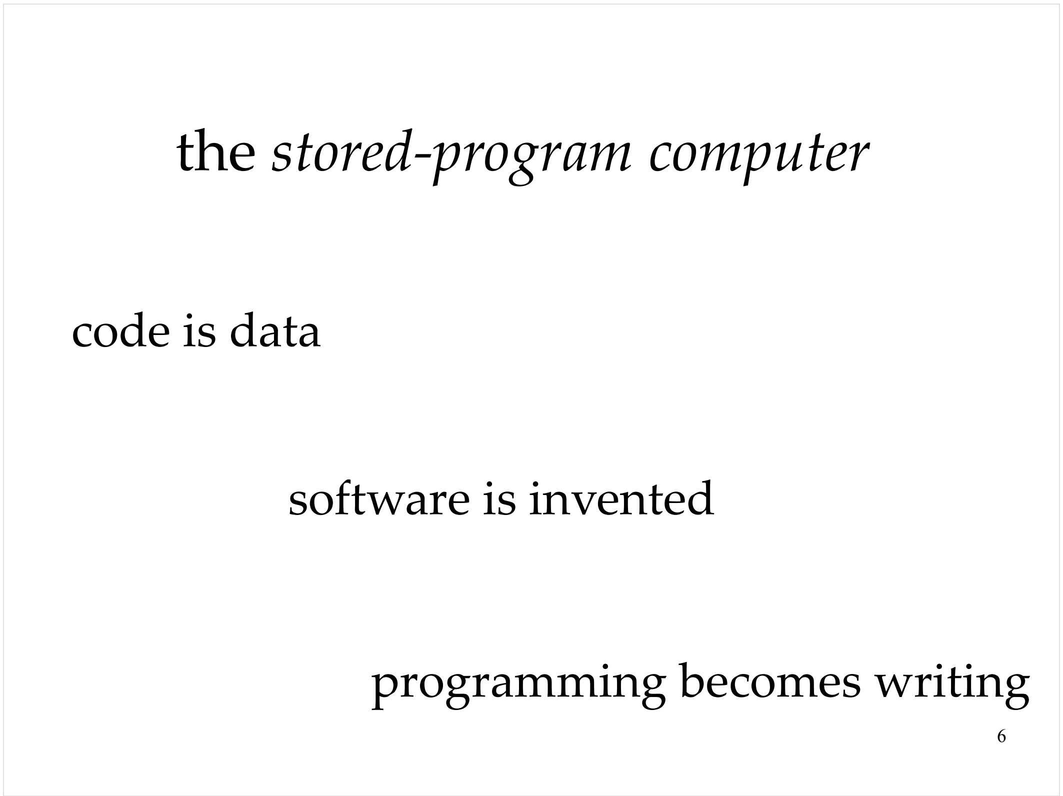
What I have are a collection of personal observations and anecdotes I've gathered informally, plus some fairly casual research. I think it tells a useful story about the general state of the software industry today and how we got here.

So think of this as a myth, about how programmers became technical writers without knowing it.



The earliest form of programming was mechanical, like this plugboard for an IBM 401, or card systems like the Jacquard loom.

This programming is not yet textual. It's a graph of functional points and machine states; it's engineering, not writing.



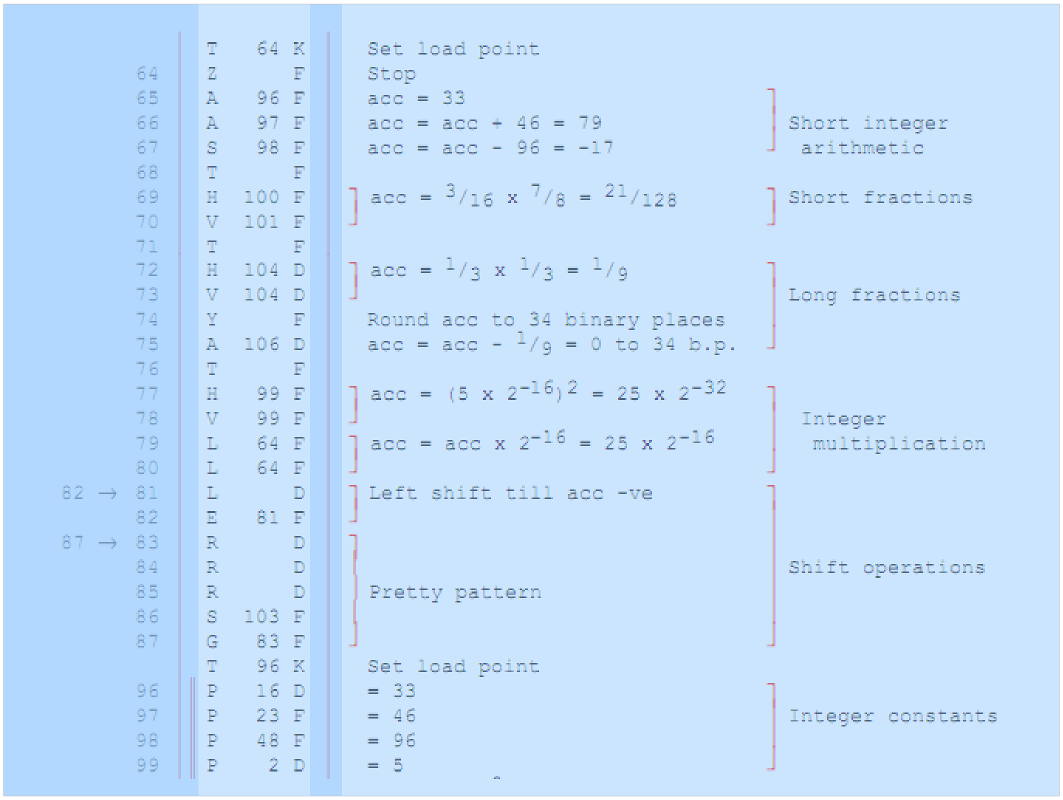
the *stored-program computer*

code is data

software is invented

programming becomes writing

But von Neumann, building on Turing's ideas, promotes the *stored program computer,* which takes its own programs as input. Code becomes data, we have software, and programming becomes writing.



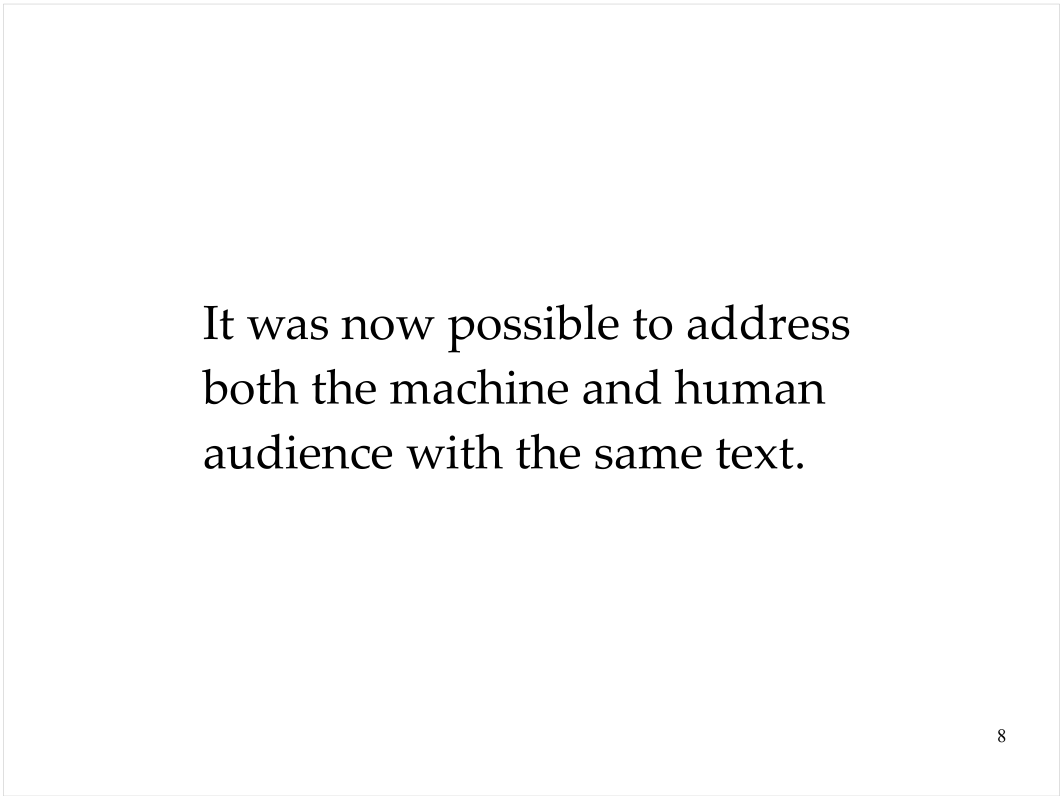
```
            T   64 K    Set load point
       64   Z      F    Stop
       65   A   96 F    acc = 33                                ] Short integer
       66   A   97 F    acc = acc + 46 = 79                     ]  arithmetic
       67   S   98 F    acc = acc - 96 = -17                    ]
       68   T      F
       69   H  100 F  ] acc = 3/16 x 7/8 = 21/128               ] Short fractions
       70   V  101 F  ]
       71   T      F
       72   H  104 D  ] acc = 1/3 x 1/3 = 1/9                   ]
       73   V  104 D  ]                                         ] Long fractions
       74   Y      F    Round acc to 34 binary places           ]
       75   A  106 D    acc = acc - 1/9 = 0 to 34 b.p.          ]
       76   T      F
       77   H   99 F  ] acc = (5 x 2^-16)^2 = 25 x 2^-32        ]
       78   V   99 F  ]                                         ]  Integer
       79   L   64 F  ] acc = acc x 2^-16 = 25 x 2^-16          ]   multiplication
       80   L   64 F  ]                                         ]
 82 -> 81   L      D  ] Left shift till acc -ve                 ]
       82   E   81 F  ]                                         ]
 87 -> 83   R      D  ]                                         ]
       84   R      D  ]                                         ] Shift operations
       85   R      D  ] Pretty pattern                          ]
       86   S  103 F  ]                                         ]
       87   G   83 F  ]                                         ]
            T   96 K    Set load point
       96   P   16 D    = 33                                    ]
       97   P   23 F    = 46                                    ] Integer constants
       98   P   48 F    = 96                                    ]
       99   P    2 D    = 5                                     ]
```

Programmers very quickly recognized that the equivalence of programs and data meant you could write programs down. They invented words – mnemonics – to represent machine instructions.

This is part of a "program manuscript" for Cambridge's EDSAC, one of the first stored-program computers. This document is for human readers; EDSAC's primary input devices were a card reader and a telephone dial. It didn't have the storage or processing power to translate a human-readable representation of a program into a machine-readable one.

I'd like to point out two key features of coding for human readers here:

- mnemonics for machine operations, and
- comments to abstract and explain portions of the code.

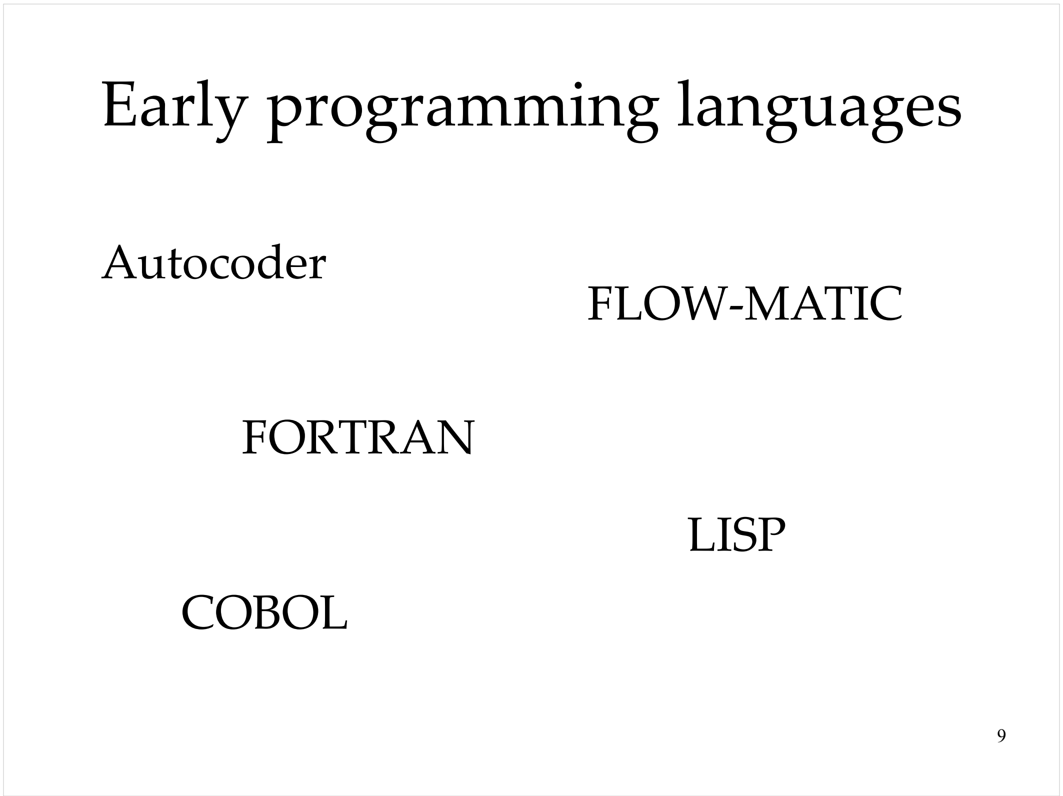


It was now possible to address both the machine and human audience with the same text.

As computers became more powerful in the early '50s, it became feasible to let them process these “program manuscripts”. The first assemblers – translation programs for these low-level programming languages – began to appear, such as IBM's Autocoder.

It was now possible to address both the machine and human audience with the same text.

The machine ignored everything but the opcode mnemonics, but maintenanc e programmers could read the comments in the source code that the program was assembled from.



Early programming languages

Autocoder

FLOW-MATIC

FORTRAN

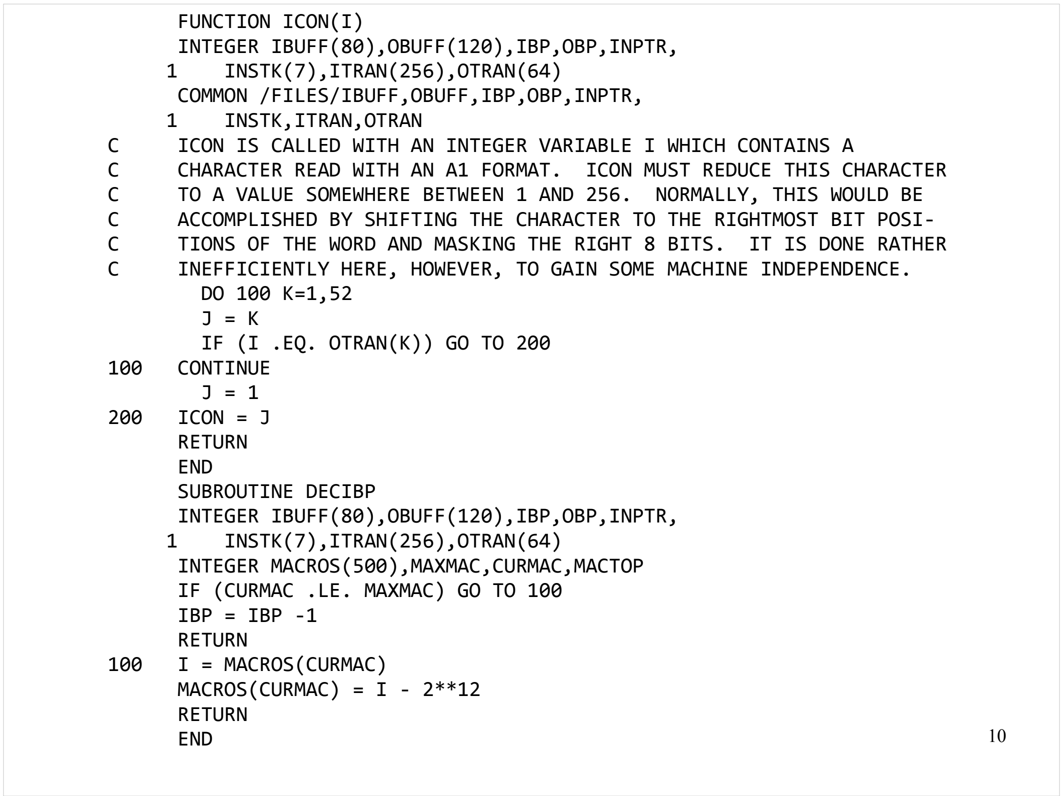
LISP

COBOL

During the '50s, Autocoder and the other assemblers were joined by the first higher-level programming languages. These gave up the close correspondence to machine operations in favor of more abstract representations. In other words, they shifted labor from the human audience (which had been interpreting a machine-friendly language) to the machine (which now had to interpret a more human-friendly language).

Around 1955, Grace Hopper's FLOW-MATIC was probably the first programming language designed to look something like human language. I'll have more to say about Admiral Hopper in a bit.

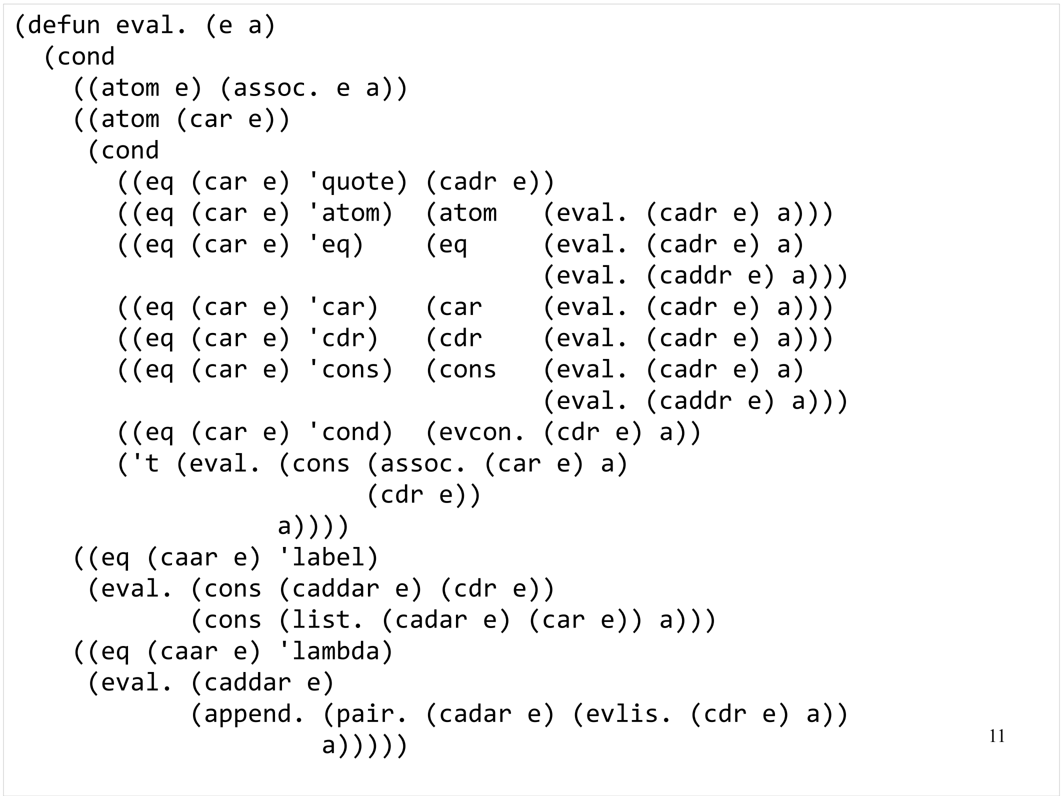
At about the same time, John Backus and his team produced FORTRAN, one of the oldest surviving programming languages.



```
      FUNCTION ICON(I)
      INTEGER IBUFF(80),OBUFF(120),IBP,OBP,INPTR,
     1    INSTK(7),ITRAN(256),OTRAN(64)
      COMMON /FILES/IBUFF,OBUFF,IBP,OBP,INPTR,
     1    INSTK,ITRAN,OTRAN
C     ICON IS CALLED WITH AN INTEGER VARIABLE I WHICH CONTAINS A
C     CHARACTER READ WITH AN A1 FORMAT.  ICON MUST REDUCE THIS CHARACTER
C     TO A VALUE SOMEWHERE BETWEEN 1 AND 256.  NORMALLY, THIS WOULD BE
C     ACCOMPLISHED BY SHIFTING THE CHARACTER TO THE RIGHTMOST BIT POSI-
C     TIONS OF THE WORD AND MASKING THE RIGHT 8 BITS.  IT IS DONE RATHER
C     INEFFICIENTLY HERE, HOWEVER, TO GAIN SOME MACHINE INDEPENDENCE.
        DO 100 K=1,52
        J = K
        IF (I .EQ. OTRAN(K)) GO TO 200
100   CONTINUE
        J = 1
200   ICON = J
      RETURN
      END
      SUBROUTINE DECIBP
      INTEGER IBUFF(80),OBUFF(120),IBP,OBP,INPTR,
     1    INSTK(7),ITRAN(256),OTRAN(64)
      INTEGER MACROS(500),MAXMAC,CURMAC,MACTOP
      IF (CURMAC .LE. MAXMAC) GO TO 100
      IBP = IBP -1
      RETURN
100   I = MACROS(CURMAC)
      MACROS(CURMAC) = I - 2**12
      RETURN
      END
```

Fortran – one of the three earliest languages, along with LISP and COBOL, still in use today – was designed for scientific computing. The name is a portmanteau of “formula translation”.

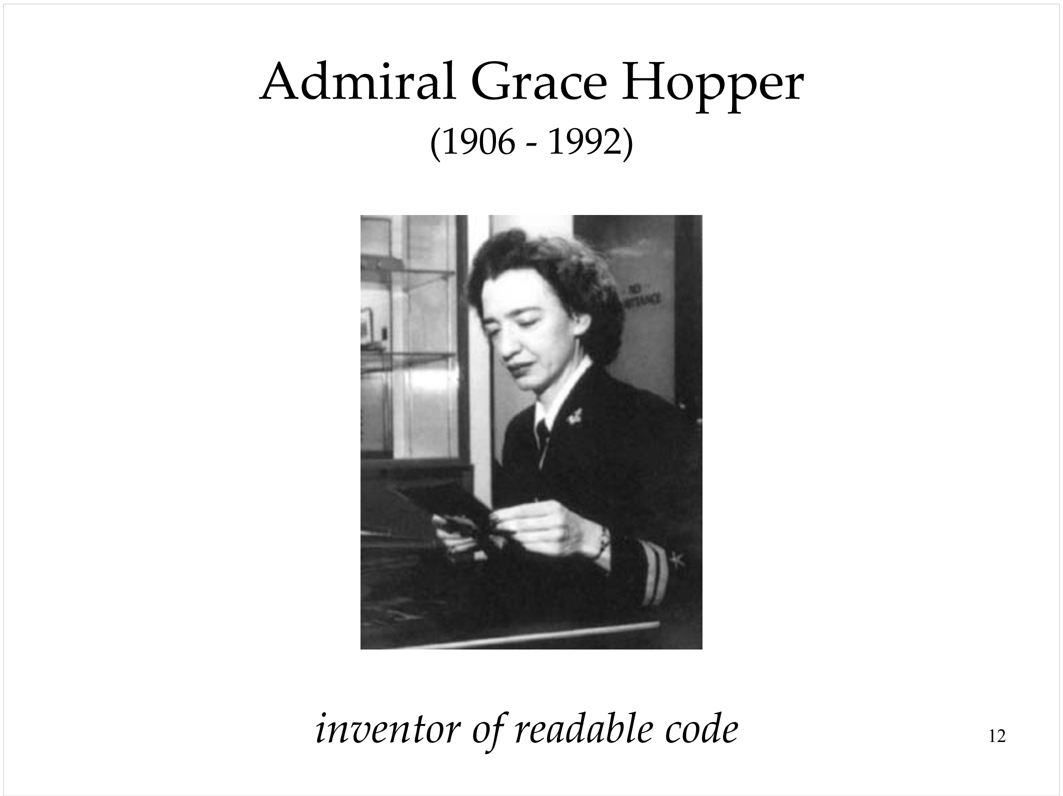
As you can see, it offers some abstraction, uses English words, allows comments. But though Fortran tries to make it easier for scientists to write code, it's not very reader-friendly. Whitespace is optional, the language is full of ideosyncratic constructs, and code written by undisciplined programmers quickly becomes an incomprehensible mess.



```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list. (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))
```

John McCarthy's LISP, which came a couple of years later, was designed to represent various concepts in computer science, such as Church's untyped lambda calculus. And it's about as readable as the untyped lambda calculus.

I don't mean to disparage these languages. FORTRAN and LISP were groundbreaking, and they're still in use because they solve problems for people. But they weren't designed with readability in mind.



Admiral Grace Hopper

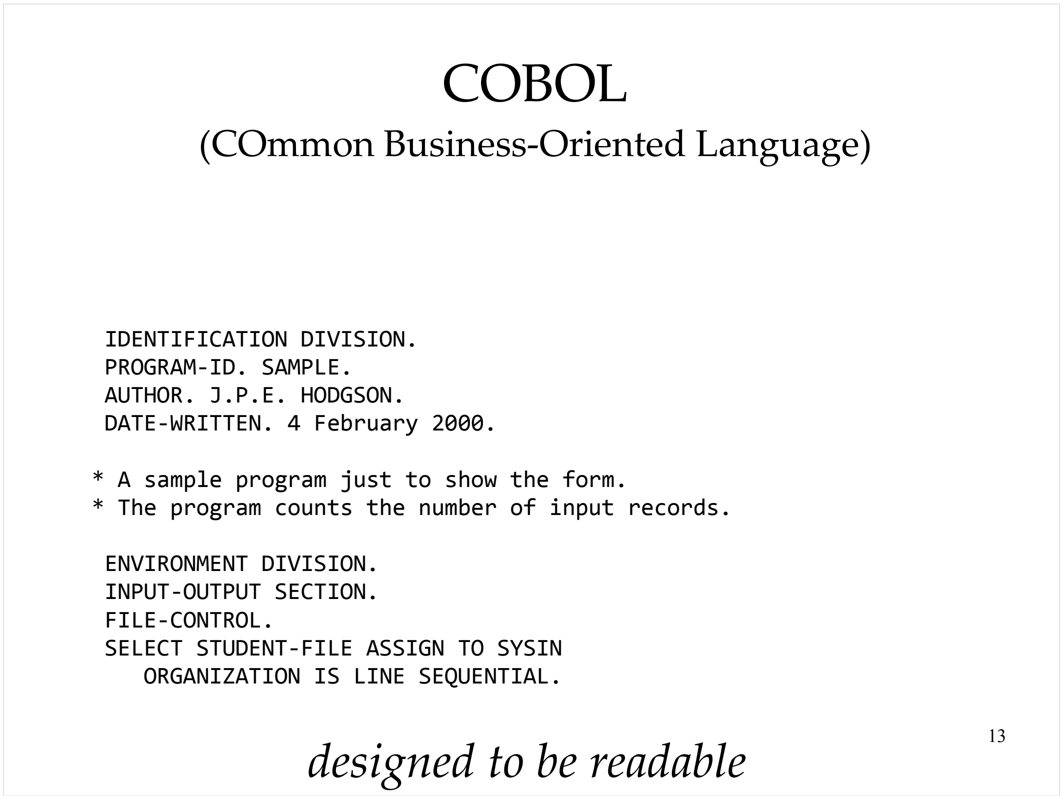
(1906 - 1992)

inventor of readable code

For code readability, we go back to Grace Hopper, of FLOW-MATIC fame. Rather than scientists or computing theorists, the programmers she was concerned with were writing business software. And she realized that business software has to be readable.

Business requirements change over time, so programs have to be maintainable. And businesses have legal and financial responsibilities, so business software has to be vetted by lawyers and accountants.

So in the late 1950s, Hopper formed a committee called CODASYL to create a business programming language with the goal that even a manager could read it.



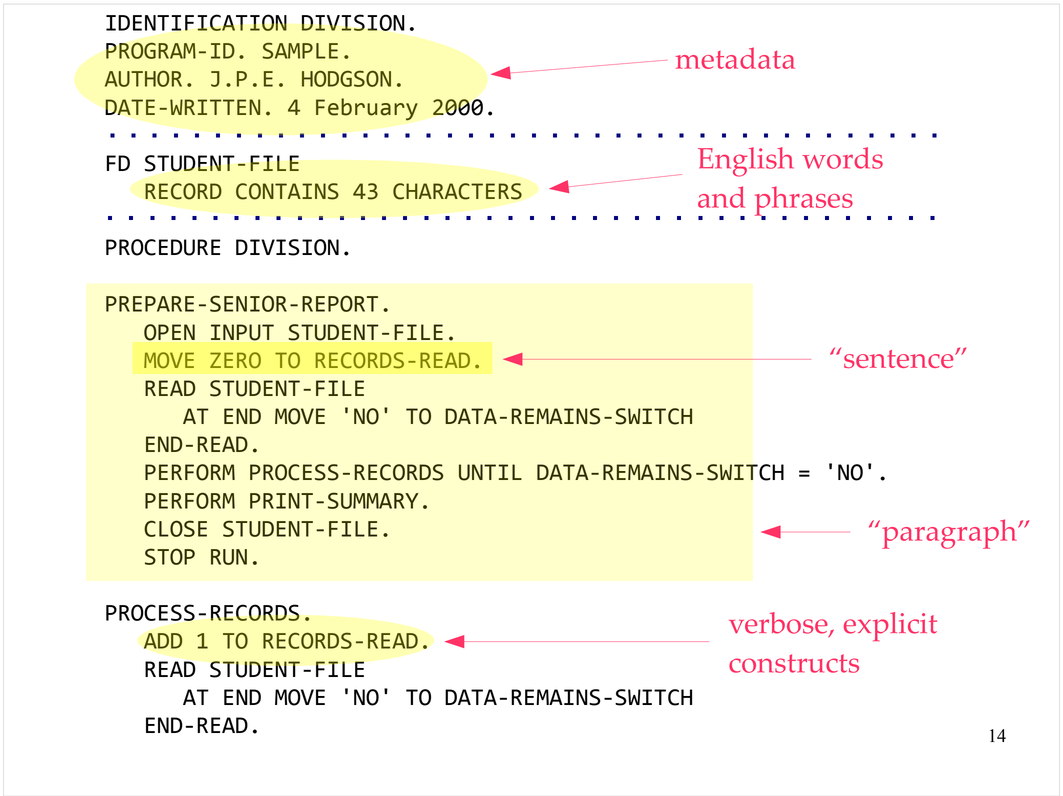
COBOL

(COmmon Business-Oriented Language)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SAMPLE.
 AUTHOR. J.P.E. HODGSON.
 DATE-WRITTEN. 4 February 2000.

* A sample program just to show the form.
* The program counts the number of input records.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 SELECT STUDENT-FILE ASSIGN TO SYSIN
    ORGANIZATION IS LINE SEQUENTIAL.
```

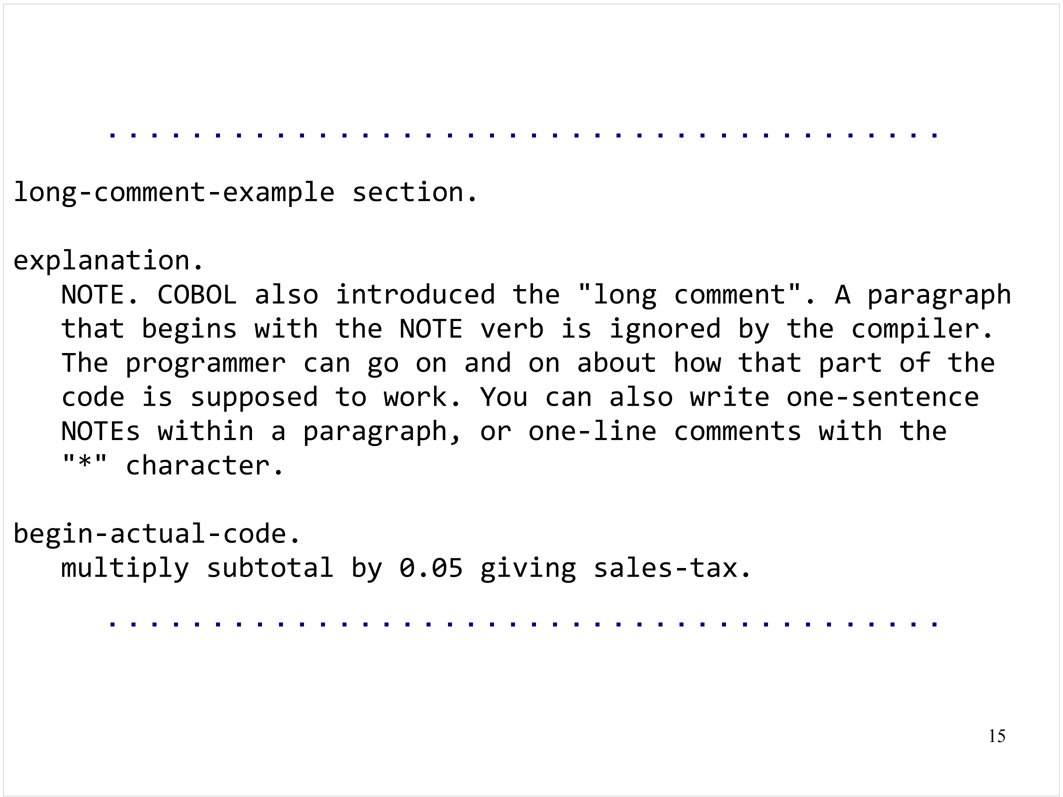
designed to be readable

And what they came up with was COBOL, the first programming language designed specifically to be readable by humans, even non-programmers.



```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
AUTHOR. J.P.E. HODGSON.
DATE-WRITTEN. 4 February 2000.

FD STUDENT-FILE
   RECORD CONTAINS 43 CHARACTERS

PROCEDURE DIVISION.

PREPARE-SENIOR-REPORT.
   OPEN INPUT STUDENT-FILE.
   MOVE ZERO TO RECORDS-READ.
   READ STUDENT-FILE
      AT END MOVE 'NO' TO DATA-REMAINS-SWITCH
   END-READ.
   PERFORM PROCESS-RECORDS UNTIL DATA-REMAINS-SWITCH = 'NO'.
   PERFORM PRINT-SUMMARY.
   CLOSE STUDENT-FILE.
   STOP RUN.

PROCESS-RECORDS.
   ADD 1 TO RECORDS-READ.
   READ STUDENT-FILE
      AT END MOVE 'NO' TO DATA-REMAINS-SWITCH
   END-READ.
```

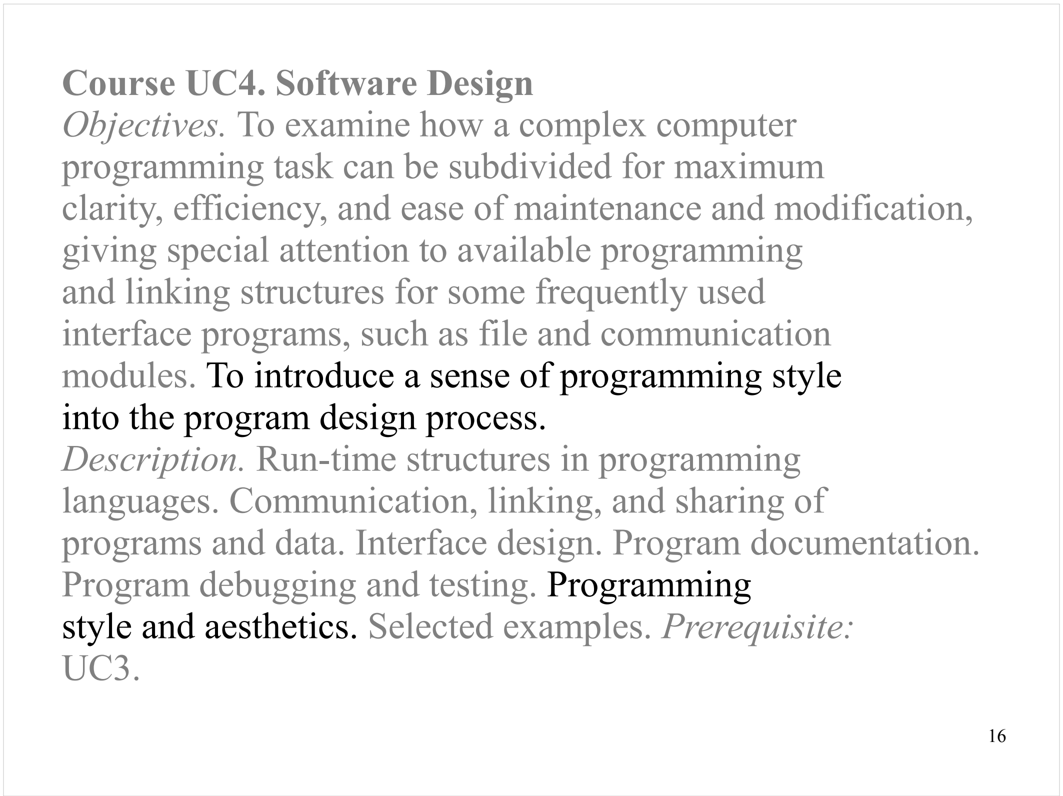
COBOL tried a bunch of things toaddress human readers. It adapted words, phrases, and structural elements from English. It used verbose constructions to increase redundancy and resist misinterpretation. It incorporated metadata into the program; that's an important feature that I'll have more to say about in a bit.



```
      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

long-comment-example section.

explanation.
   NOTE. COBOL also introduced the "long comment". A paragraph
   that begins with the NOTE verb is ignored by the compiler.
   The programmer can go on and on about how that part of the
   code is supposed to work. You can also write one-sentence
   NOTEs within a paragraph, or one-line comments with the
   "*" character.

begin-actual-code.
   multiply subtotal by 0.05 giving sales-tax.

      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

COBOL also encouraged writing detailed textual descriptions of parts of programs by introducing the “long comment” form. It was the first high-level language to make this kind of appeal to textuality.

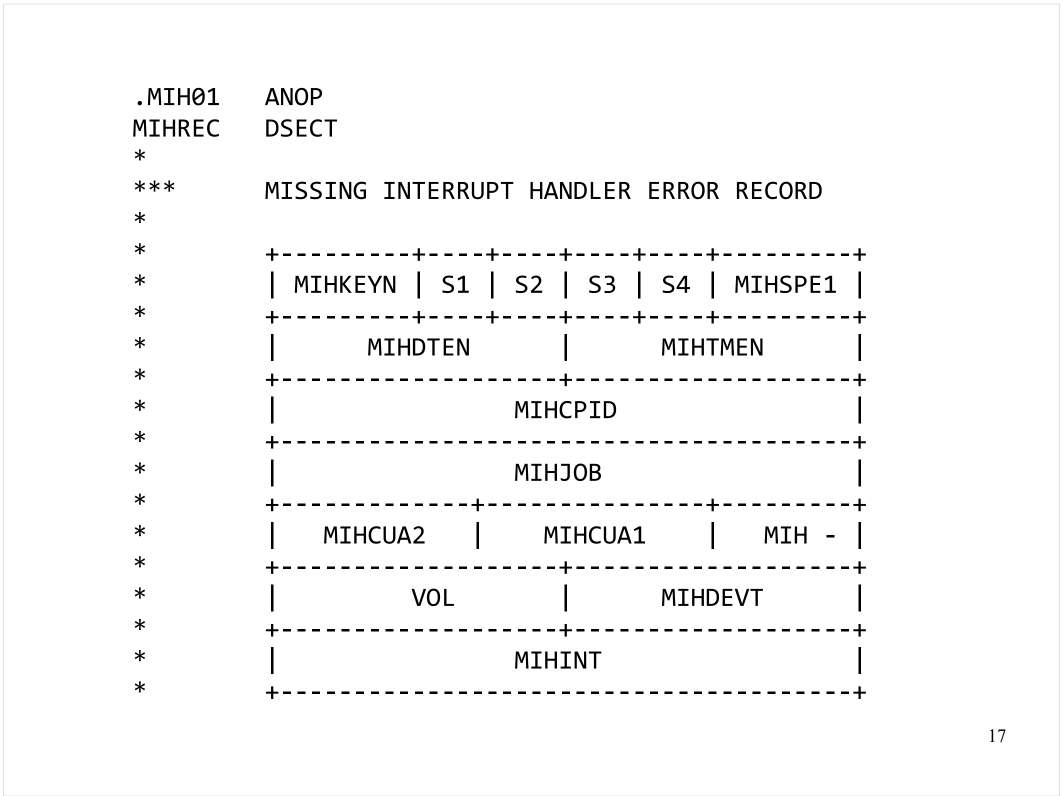
COBOL was a success – it may be the most-used programming language in history – but it's awkward for many purposes and it has a lot of competition. And most of those other languages are not inherently reader-friendly.



Course UC4. Software Design
Objectives. To examine how a complex computer programming task can be subdivided for maximum clarity, efficiency, and ease of maintenance and modification, giving special attention to available programming and linking structures for some frequently used interface programs, such as file and communication modules. To introduce a sense of programming style into the program design process.
Description. Run-time structures in programming languages. Communication, linking, and sharing of programs and data. Interface design. Program documentation. Program debugging and testing. Programming style and aesthetics. Selected examples. *Prerequisite:* UC3.

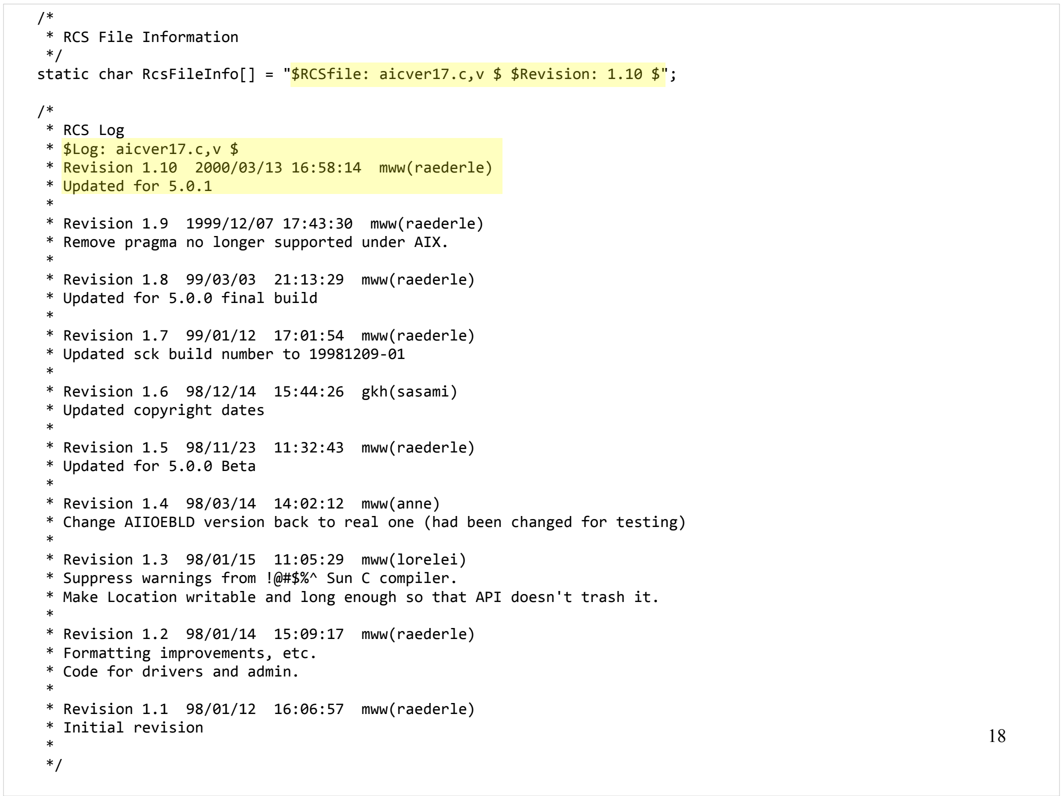
In the '60s and '70s some computer academics began to think that about training programmers to think about coding and readability, though they didn't really make the connection to technical writing.

This is from a set of recommendations for undergraduate IT curricula published by the Association for Computing Machinery in 1973. You can see that it at least acknowledges “programming style and aesthetics”.



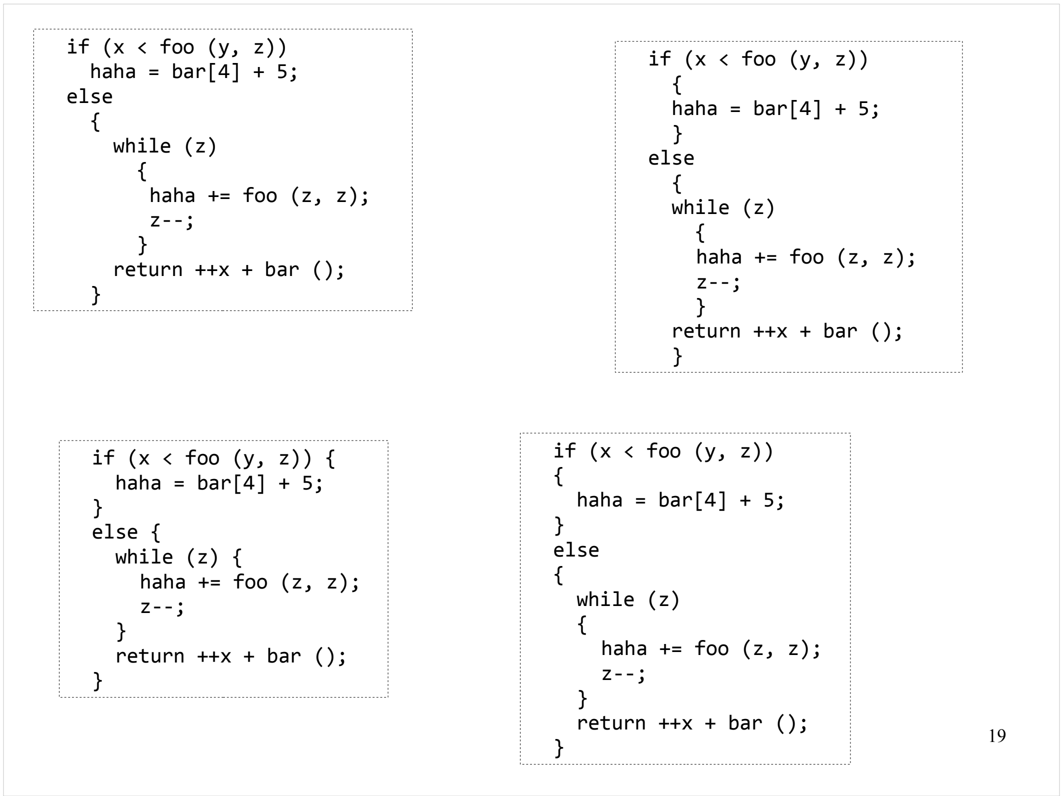
```
.MIH01   ANOP
MIHREC   DSECT
*
***      MISSING INTERRUPT HANDLER ERROR RECORD
*
*        +---------+----+----+----+----+---------+
*        | MIHKEYN | S1 | S2 | S3 | S4 | MIHSPE1 |
*        +---------+----+----+----+----+---------+
*        |      MIHDTEN      |      MIHTMEN      |
*        +-------------------+-------------------+
*        |                MIHCPID                |
*        +---------------------------------------+
*        |                MIHJOB                 |
*        +-------------+---------------+---------+
*        |   MIHCUA2   |    MIHCUA1    |   MIH - |
*        +-------------------+-------------------+
*        |         VOL       |      MIHDEVT      |
*        +-------------------+-------------------+
*        |                MIHINT                 |
*        +---------------------------------------+
```

Meanwhile practitioners were attacking the readability problem in various ways, and to implicitly acknowledge the writerly aspects of coding. We start seeing more ambitious use of comments in source code, for example “character art” diagrams, ASCII art, literary references, jokes.



```
/*
 * RCS File Information
 */
static char RcsFileInfo[] = "$RCSfile: aicver17.c,v $ $Revision: 1.10 $";

/*
 * RCS Log
 * $Log: aicver17.c,v $
 * Revision 1.10  2000/03/13 16:58:14  mww(raederle)
 * Updated for 5.0.1
 *
 * Revision 1.9  1999/12/07 17:43:30  mww(raederle)
 * Remove pragma no longer supported under AIX.
 *
 * Revision 1.8  99/03/03  21:13:29  mww(raederle)
 * Updated for 5.0.0 final build
 *
 * Revision 1.7  99/01/12  17:01:54  mww(raederle)
 * Updated sck build number to 19981209-01
 *
 * Revision 1.6  98/12/14  15:44:26  gkh(sasami)
 * Updated copyright dates
 *
 * Revision 1.5  98/11/23  11:32:43  mww(raederle)
 * Updated for 5.0.0 Beta
 *
 * Revision 1.4  98/03/14  14:02:12  mww(anne)
 * Change AIIOEBLD version back to real one (had been changed for testing)
 *
 * Revision 1.3  98/01/15  11:05:29  mww(lorelei)
 * Suppress warnings from !@#$%^ Sun C compiler.
 * Make Location writable and long enough so that API doesn't trash it.
 *
 * Revision 1.2  98/01/14  15:09:17  mww(raederle)
 * Formatting improvements, etc.
 * Code for drivers and admin.
 *
 * Revision 1.1  98/01/12  16:06:57  mww(raederle)
 * Initial revision
 *
 */
```

Programmers also developed tools that would automatically add comments with metadata to track revisions and so forth.



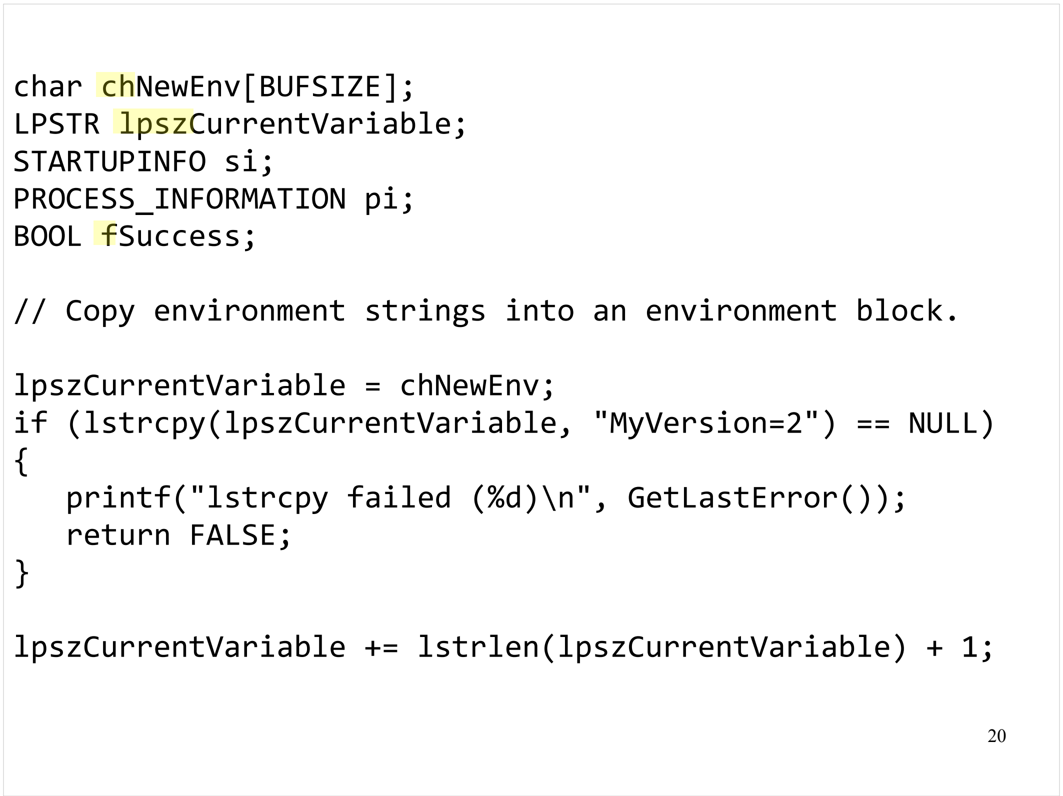
```
if (x < foo (y, z))
  haha = bar[4] + 5;
else
  {
    while (z)
      {
       haha += foo (z, z);
       z--;
      }
    return ++x + bar ();
  }
```

```
if (x < foo (y, z))
  {
  haha = bar[4] + 5;
  }
else
  {
  while (z)
    {
    haha += foo (z, z);
    z--;
    }
  return ++x + bar ();
  }
```

```
if (x < foo (y, z)) {
  haha = bar[4] + 5;
}
else {
  while (z) {
    haha += foo (z, z);
    z--;
  }
  return ++x + bar ();
}
```

```
if (x < foo (y, z))
{
  haha = bar[4] + 5;
}
else
{
  while (z)
  {
    haha += foo (z, z);
    z--;
  }
  return ++x + bar ();
}
```

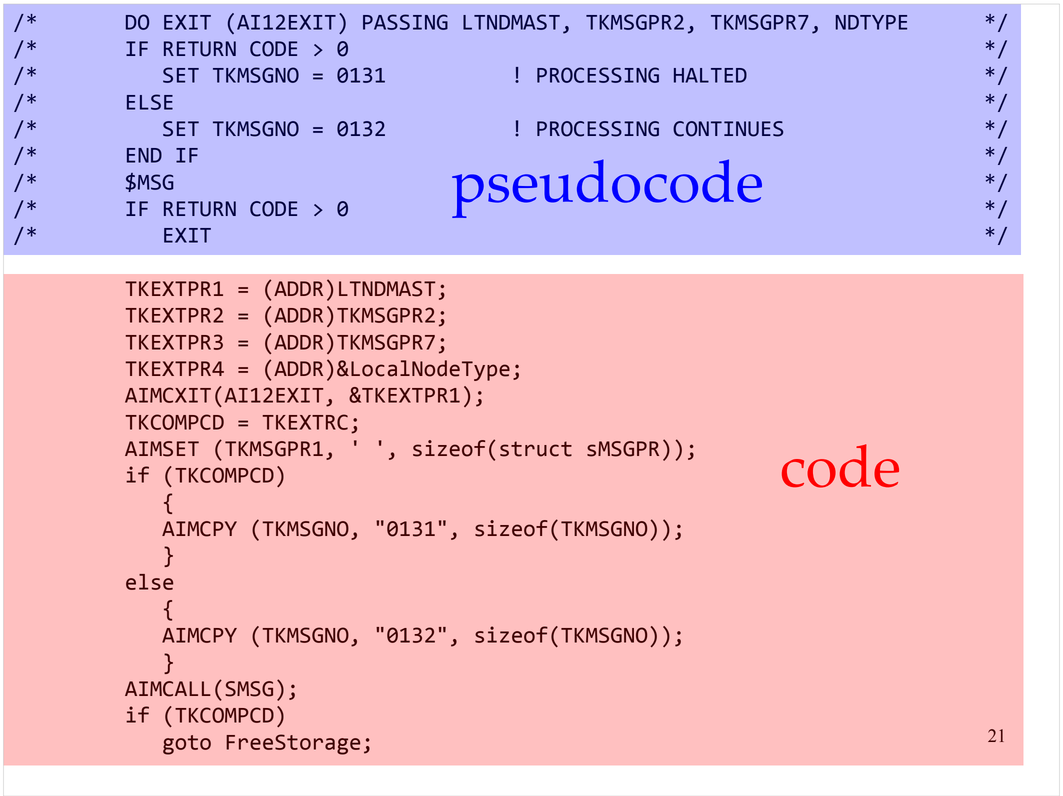
People developed preferences for style, layout, naming. Fierce battles are still waged over the placement of braces in C code – just one of many religious wars being fought over code readability.



```
char chNewEnv[BUFSIZE];
LPSTR lpszCurrentVariable;
STARTUPINFO si;
PROCESS_INFORMATION pi;
BOOL fSuccess;

// Copy environment strings into an environment block.

lpszCurrentVariable = chNewEnv;
if (lstrcpy(lpszCurrentVariable, "MyVersion=2") == NULL)
{
   printf("lstrcpy failed (%d)\n", GetLastError());
   return FALSE;
}

lpszCurrentVariable += lstrlen(lpszCurrentVariable) + 1;
```

During the '80s and '90s in particular, we see increasing efforts to create “coding standards” that dictate style conventions. Usually they're ad hoc, with no basis in theory or empirical research; just collections of unexamined personal preferences and and untested assumptions, made law to satisfy hobgoblin consistency. Microsoft's naive version of Hungarian notation – these obscure prefixes on names – is a famous example.

Maybe a more useful one was the “picture on a page” idea used by some teams at IBM and elsewhere, which says that a program must be broken down into conceptual units small enough to be read without scrolling. That's a real, if crude, tech-writing moment: forcing programmers to cater to human readers.



```
/*       DO EXIT (AI12EXIT) PASSING LTNDMAST, TKMSGPR2, TKMSGPR7, NDTYPE      */
/*       IF RETURN CODE > 0                                                   */
/*          SET TKMSGNO = 0131          ! PROCESSING HALTED                   */
/*       ELSE                                                                 */
/*          SET TKMSGNO = 0132          ! PROCESSING CONTINUES                */
/*       END IF                                                               */
/*       $MSG                                                                 */
/*       IF RETURN CODE > 0                                                   */
/*          EXIT                                                              */
```

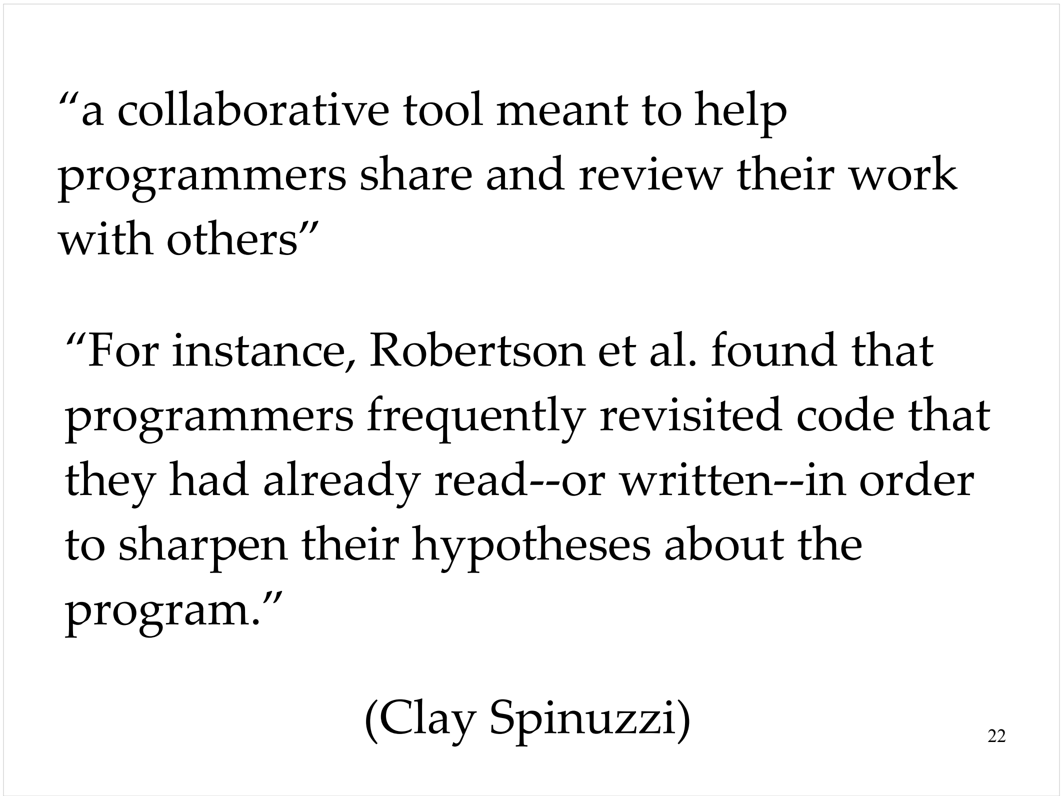
pseudocode

```
         TKEXTPR1 = (ADDR)LTNDMAST;
         TKEXTPR2 = (ADDR)TKMSGPR2;
         TKEXTPR3 = (ADDR)TKMSGPR7;
         TKEXTPR4 = (ADDR)&LocalNodeType;
         AIMCXIT(AI12EXIT, &TKEXTPR1);
         TKCOMPCD = TKEXTRC;
         AIMSET (TKMSGPR1, ' ', sizeof(struct sMSGPR));
         if (TKCOMPCD)
            {
            AIMCPY (TKMSGNO, "0131", sizeof(TKMSGNO));
            }
         else
            {
            AIMCPY (TKMSGNO, "0132", sizeof(TKMSGNO));
            }
         AIMCALL(SMSG);
         if (TKCOMPCD)
            goto FreeStorage;
```

code

This is an interesting convention from a product I worked on beginning in the late '80s. It was first written in pseudocode – stylized English that looks like code but omits some of the messy details. Then the pseudocode was turned into comments, and the actual code was inserted between the lines of psuedocode.

We'll see a more sophisticated version of this interweaving technique in a minute.



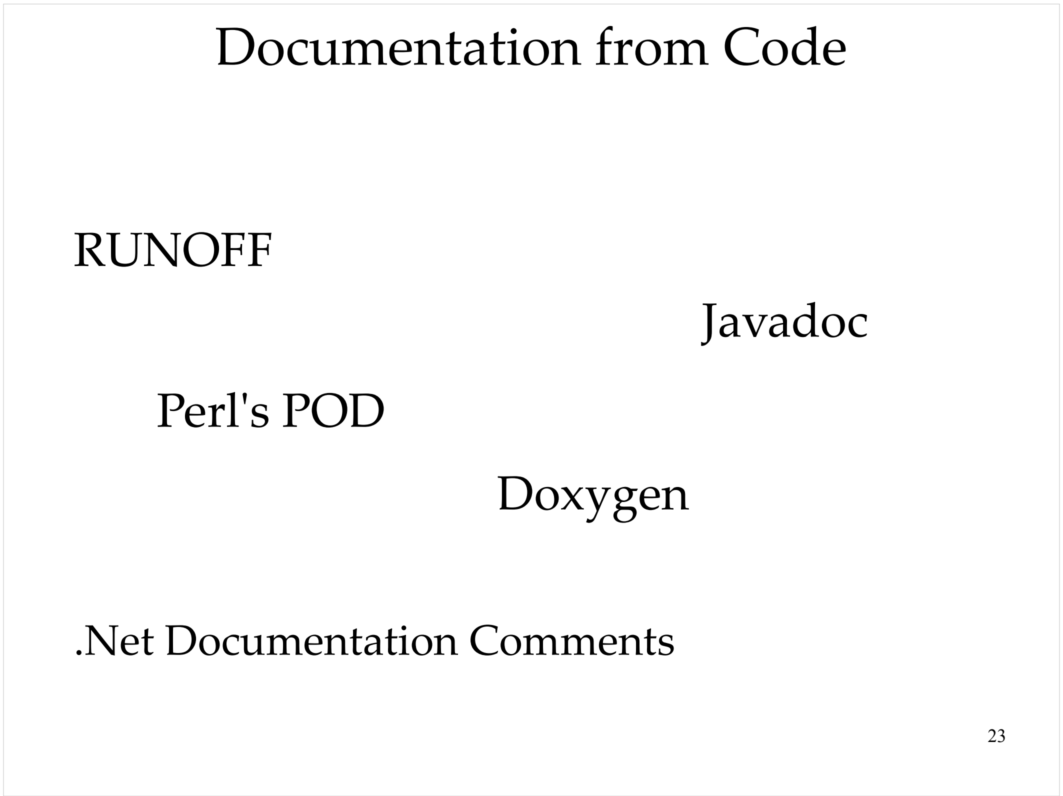
“a collaborative tool meant to help programmers share and review their work with others”

“For instance, Robertson et al. found that programmers frequently revisited code that they had already read--or written--in order to sharpen their hypotheses about the program.”

(Clay Spinuzzi)

By this point academics, mostly in computer science and information technology, had done some serious research into code readability. Clay Spinuzzi mentions some of this in his very nice short essay “Towards a Hermeneutic Understanding of Programming Languages”. He also makes much the same argument that I'm making here; as he puts it, code is “a collaborative tool meant to help programmers share and review their work with others”.

But there's surprisingly little discussion of teaching programmers to approach coding with an eye toward future readability.



Documentation from Code

RUNOFF

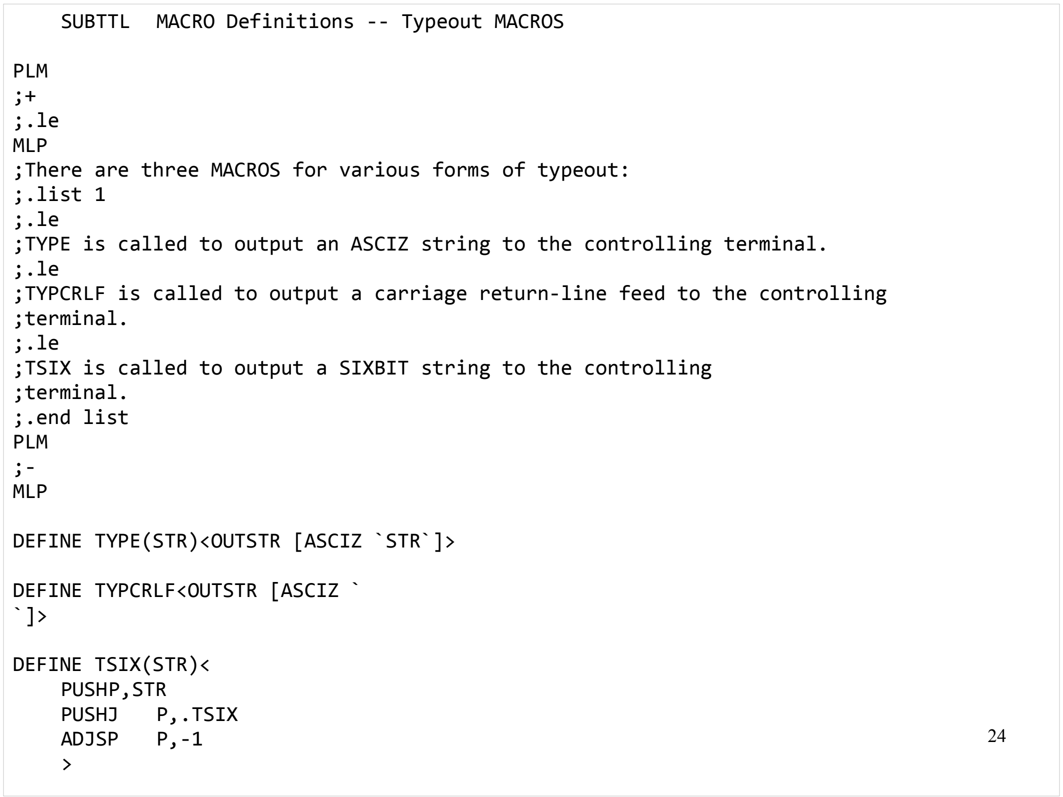
Javadoc

Perl's POD

Doxygen

.Net Documentation Comments

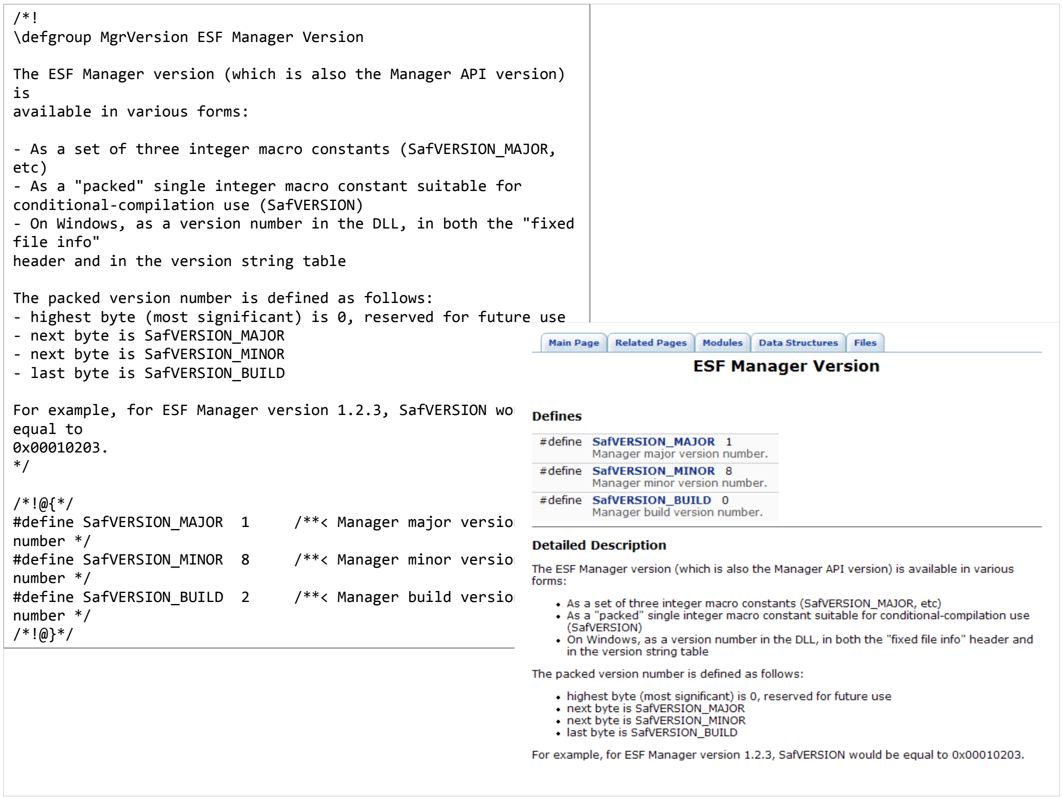
And something else interesting is happening: single-sourcing program documentation from code. Programmers who have to write documentation, particularly technical specs, want to do it all in one place, and avoid skew between code and documentation as the code is maintained.



```
    SUBTTL  MACRO Definitions -- Typeout MACROS

PLM
;+
;.le
MLP
;There are three MACROS for various forms of typeout:
;.list 1
;.le
;TYPE is called to output an ASCIZ string to the controlling terminal.
;.le
;TYPCRLF is called to output a carriage return-line feed to the controlling
;terminal.
;.le
;TSIX is called to output a SIXBIT string to the controlling
;terminal.
;.end list
PLM
;-
MLP

DEFINE TYPE(STR)<OUTSTR [ASCIZ `STR`]>

DEFINE TYPCRLF<OUTSTR [ASCIZ `
`]>

DEFINE TSIX(STR)<
    PUSHP,STR
    PUSHJ   P,.TSIX
    ADJSP   P,-1
    >
```

Peter Conklin at DEC may have been the first to do this. He put markup commands for the RUNOFF typesetting program into comments in some of the programs he wrote around 1970.

RUNOFF, by the way, was the inspiration for IBM's Script, which led to GML, which led to SGML, which led to HTML and XML.



```
/*!
\defgroup MgrVersion ESF Manager Version

The ESF Manager version (which is also the Manager API version)
is
available in various forms:

- As a set of three integer macro constants (SafVERSION_MAJOR,
etc)
- As a "packed" single integer macro constant suitable for
conditional-compilation use (SafVERSION)
- On Windows, as a version number in the DLL, in both the "fixed
file info"
header and in the version string table

The packed version number is defined as follows:
- highest byte (most significant) is 0, reserved for future use
- next byte is SafVERSION_MAJOR
- next byte is SafVERSION_MINOR
- last byte is SafVERSION_BUILD

For example, for ESF Manager version 1.2.3, SafVERSION wo
equal to
0x00010203.
*/

/*!@{*/
#define SafVERSION_MAJOR  1     /**< Manager major versio
number */
#define SafVERSION_MINOR  8     /**< Manager minor versio
number */
#define SafVERSION_BUILD  2     /**< Manager build versio
number */
/*!@}*/
```

ESF Manager Version

Defines

#define	**SafVERSION_MAJOR** 1 Manager major version number.
#define	**SafVERSION_MINOR** 8 Manager minor version number.
#define	**SafVERSION_BUILD** 0 Manager build version number.

Detailed Description

The ESF Manager version (which is also the Manager API version) is available in various forms:

- As a set of three integer macro constants (SafVERSION_MAJOR, etc)
- As a "packed" single integer macro constant suitable for conditional-compilation use (SafVERSION)
- On Windows, as a version number in the DLL, in both the "fixed file info" header and in the version string table

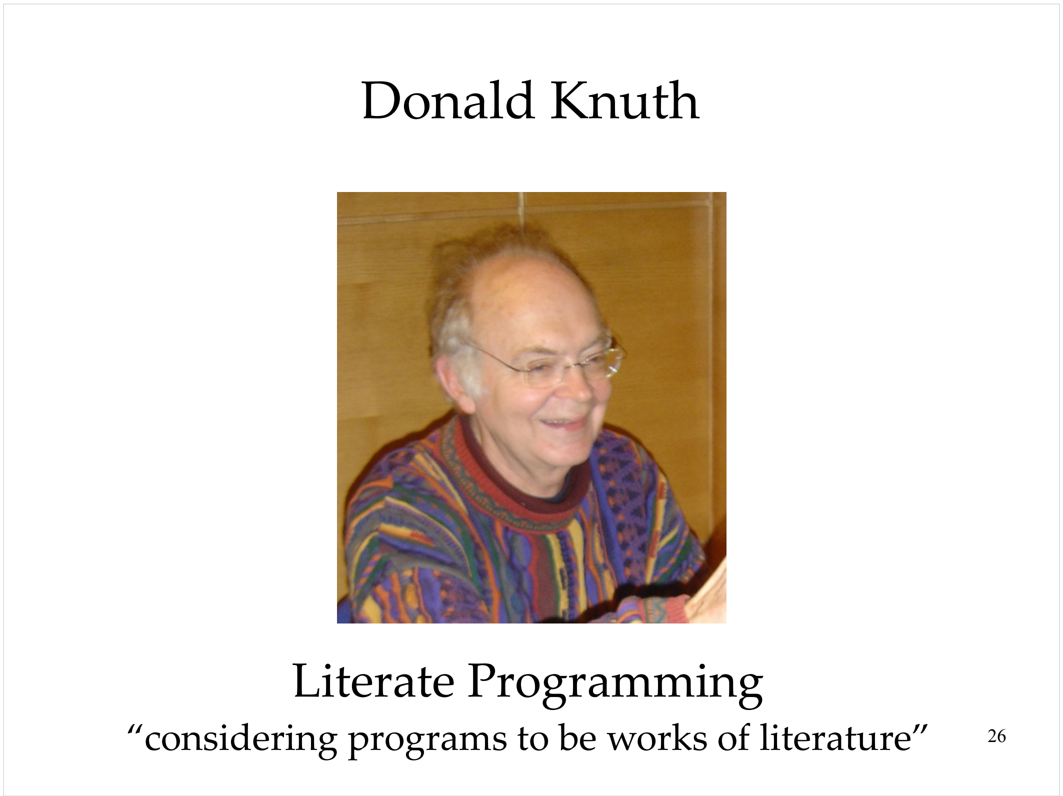
The packed version number is defined as follows:

- highest byte (most significant) is 0, reserved for future use
- next byte is SafVERSION_MAJOR
- next byte is SafVERSION_MINOR
- last byte is SafVERSION_BUILD

For example, for ESF Manager version 1.2.3, SafVERSION would be equal to 0x00010203.

The idea took a while to catch on, but became popular with Java's Javadoc documentation markup system, and a bit later with Perl's POD. Dimitri van Heesch's open-source Doxygen provides advanced documentation-markup features for a wide range of languages. Microsoft has an XML-based Javadoc-like system for .Net.

Doxygen and its competitors encourage programmers to think of technical writing as an integral part of their job. But they still put coding prior to writing, rather than seeing coding as writing.

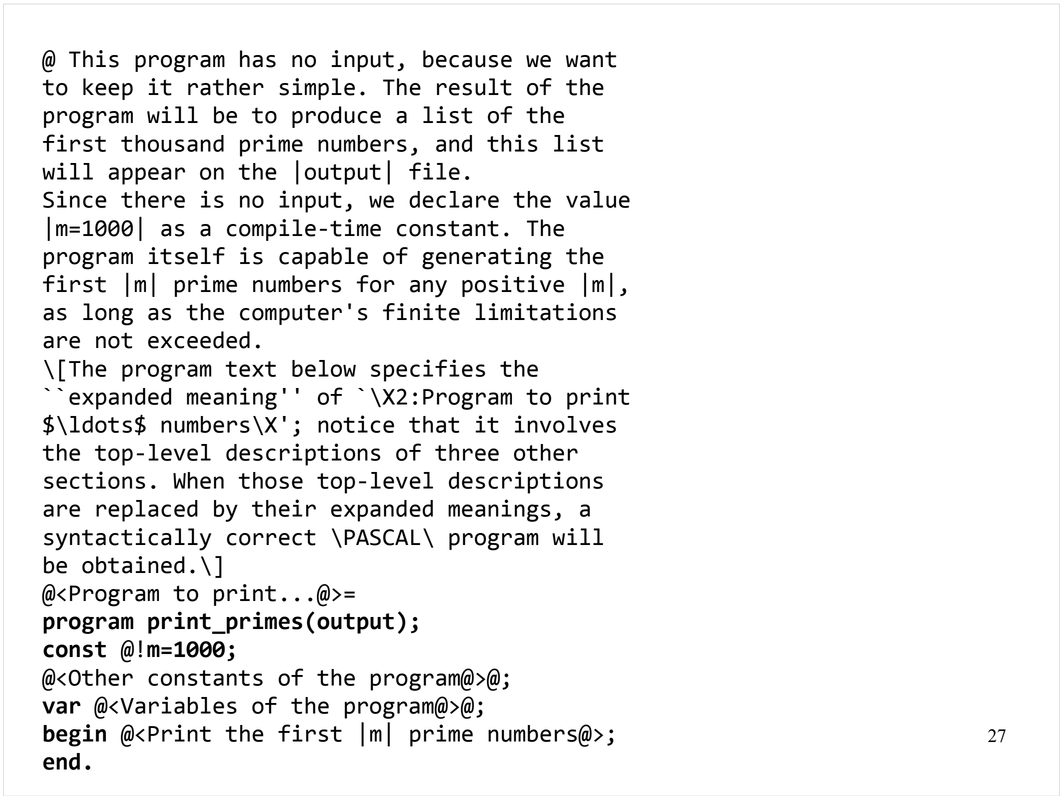


Donald Knuth

Literate Programming

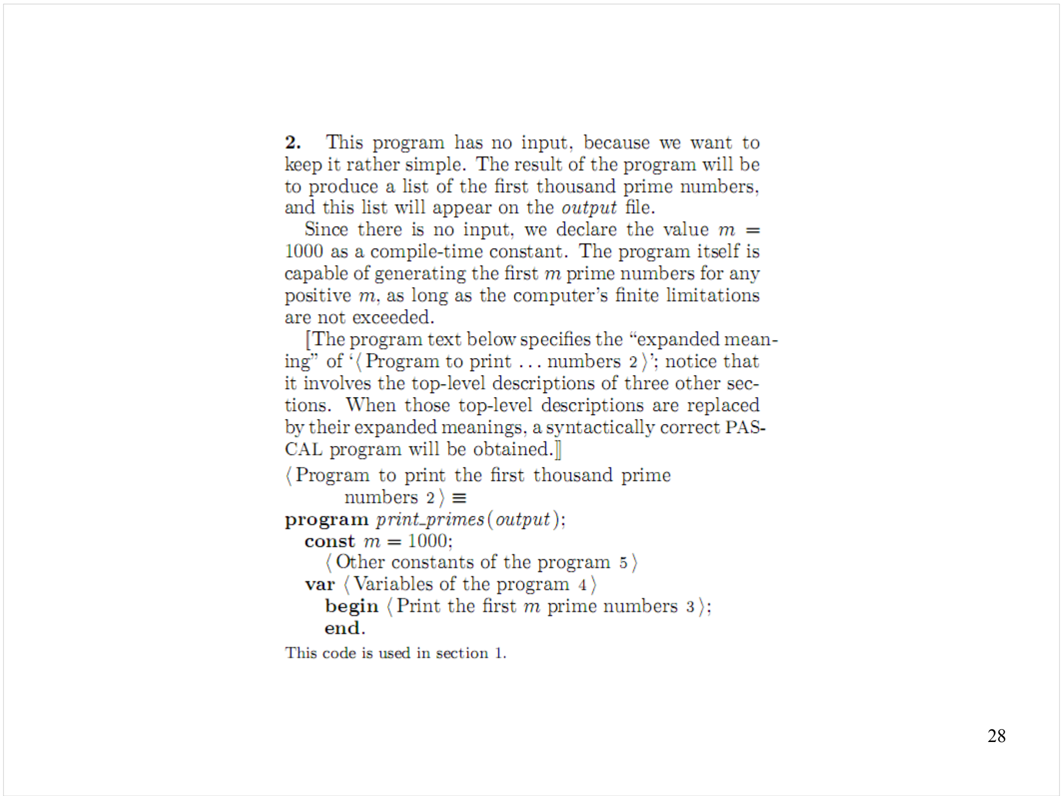
"considering programs to be works of literature"

But back in 1983 the legendary computer scientist Donald Knuth had presented a radical approach to coding called "literate programming". As Knuth's original paper on the subject put it, literate programming was about seeing software as literature – as something to be read.



```
@ This program has no input, because we want
to keep it rather simple. The result of the
program will be to produce a list of the
first thousand prime numbers, and this list
will appear on the |output| file.
Since there is no input, we declare the value
|m=1000| as a compile-time constant. The
program itself is capable of generating the
first |m| prime numbers for any positive |m|,
as long as the computer's finite limitations
are not exceeded.
\[The program text below specifies the
``expanded meaning'' of `\X2:Program to print
$\ldots$ numbers\X'; notice that it involves
the top-level descriptions of three other
sections. When those top-level descriptions
are replaced by their expanded meanings, a
syntactically correct \PASCAL\ program will
be obtained.\]
@<Program to print...@>=
program print_primes(output);
const @!m=1000;
@<Other constants of the program@>@;
var @<Variables of the program@>@;
begin @<Print the first |m| prime numbers@>;
end.
```

In literate programming, descriptive text predominates, and code is interwoven with it. It's a single source for a narrative of the program's design and construction, and the program itself.



2. This program has no input, because we want to keep it rather simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the *output* file.

Since there is no input, we declare the value $m = 1000$ as a compile-time constant. The program itself is capable of generating the first m prime numbers for any positive m, as long as the computer's finite limitations are not exceeded.

[The program text below specifies the "expanded meaning" of '⟨ Program to print . . . numbers 2 ⟩'; notice that it involves the top-level descriptions of three other sections. When those top-level descriptions are replaced by their expanded meanings, a syntactically correct PASCAL program will be obtained.]

```
⟨ Program to print the first thousand prime
        numbers 2 ⟩ ≡
program print_primes(output);
  const m = 1000;
    ⟨ Other constants of the program 5 ⟩
  var ⟨ Variables of the program 4 ⟩
    begin ⟨ Print the first m prime numbers 3 ⟩;
    end.
```

This code is used in section 1.

And the documentation includes all the code. It exists primarily for the programmer-as-reader.

But literate programming hasn't caught on, except in a few niches. (Statistical programming in the human and life sciences, for example.) And that's one of the problems that I think teachers of technical writing might attend to.

Quality

provisioning

security

Extreme Programming

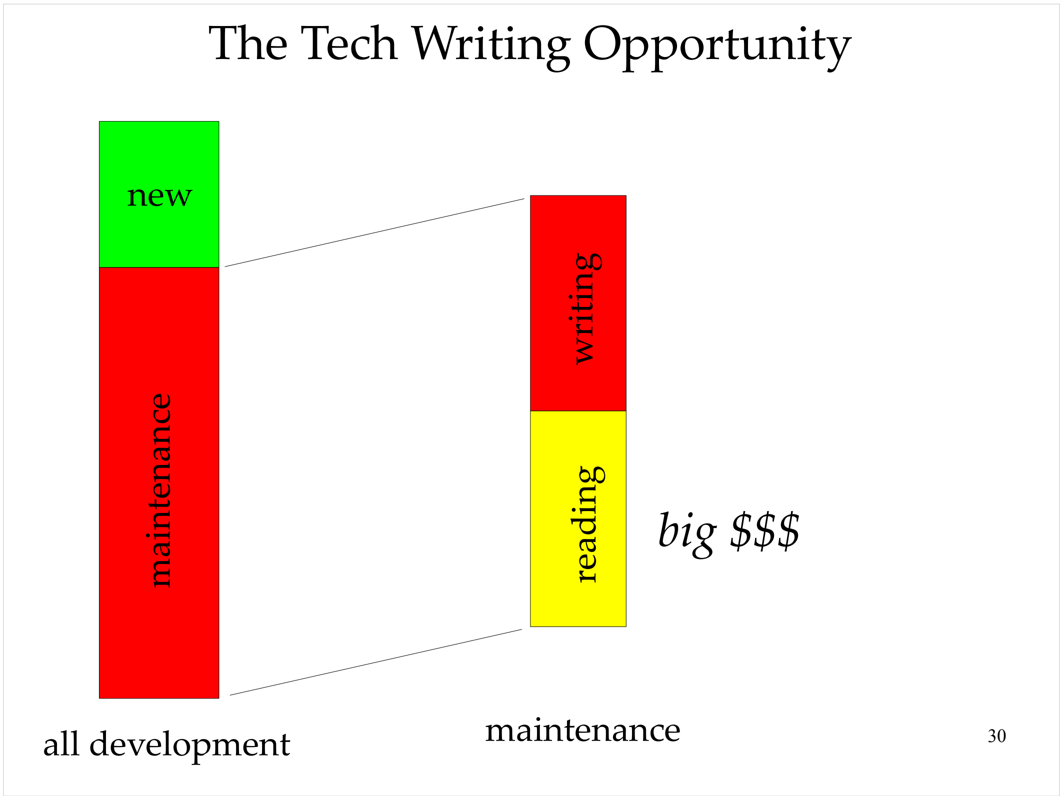
Scrum

tech writing?

Because there are some powerful arguments to be made for treating coding as technical writing. First, as those of you who have used computers know, software quality generally ranges from barely acceptable to abysmal.

Some powerful economic drivers – provisioning and security are two of them – are making customers very sensitive to quality issues, and developers are adopting new processes to try to address them. That's a bandwagon that has room for jumpers.

This applies doubly to open source, by the way, because the programming team is typically much more fragmented and uncoordinated. The code becomes a major communication channel.

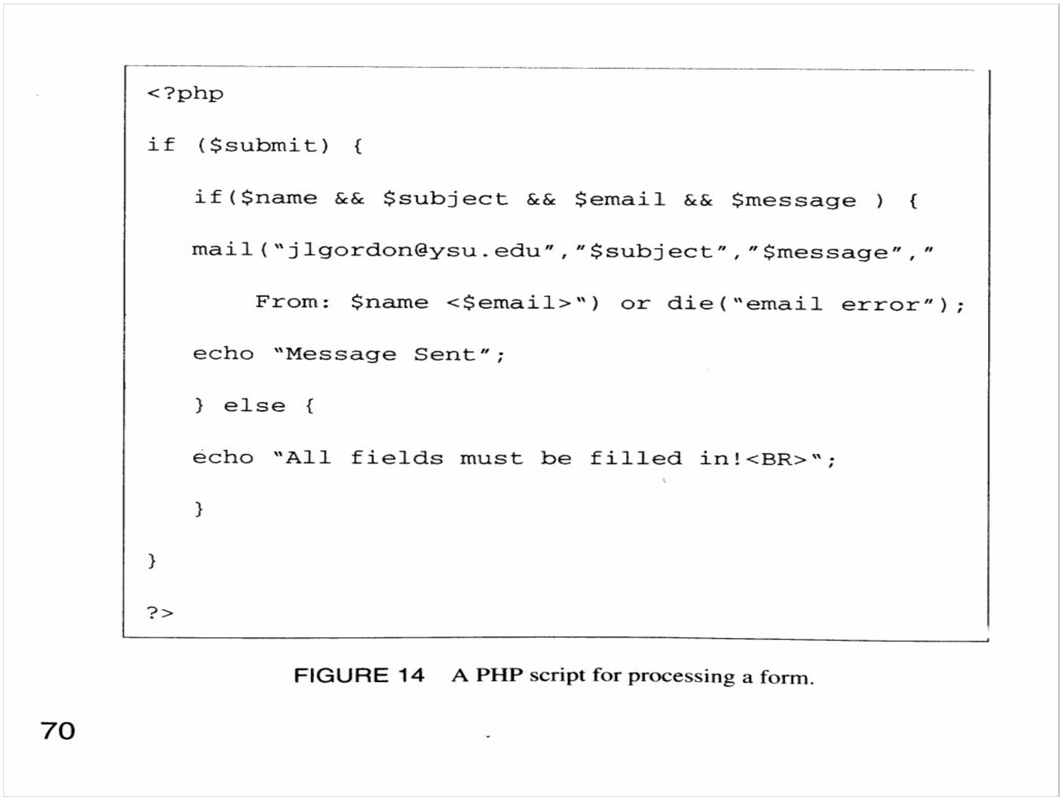


The Tech Writing Opportunity

Second, there's a big opportunity to capture efficiences by making code more readable. Estimates vary, but it's likely that around 75% of programming resources go to maintenance. And studies suggest that maintenance programmers spend more than half their time reading code to figure out what it's trying to do. That means potentially about a third of the resources spent on software development are available to be reclaimed by more readable software.

I've provided a management-style graph to illustrate that.

But tech writing teachers apparently haven't taken much note of this. Aside from the Spinuzzi piece, I didn't find any directly relevant articles on this in a couple of days of browsing the archives. It may be out there, but there's not a lot of it.



```
<?php

if ($submit) {

   if($name && $subject && $email && $message ) {

   mail(“jlgordon@ysu.edu”,”$subject”,”$message”,”

       From: $name <$email>“) or die(“email error”);

   echo “Message Sent”;

   } else {

   echo “All fields must be filled in!<BR>“;

   }

}

?>
```

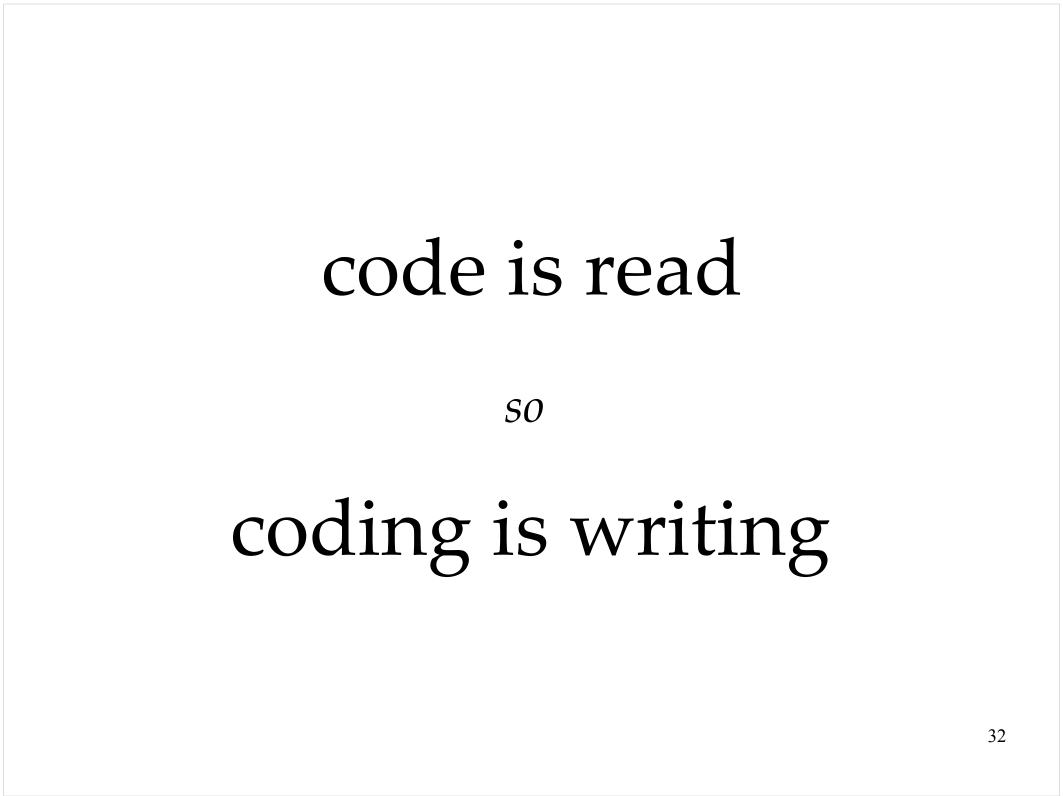
FIGURE 14 A PHP script for processing a form.

70

And tech writing teachers who do write about code often don't talk about reading it.

This is from Jay Gordon's 2005 TCQ article “Teaching Hypertext Composition” – a perfectly good article. Here Jay is showing a small PHP script to send email.He says “it may look a bit daunting” but “it is fairly simple and logical”. And it is at that. But I'd like to point out that a few comments and some unpacking here could make it quite a lot less daunting – less a magical incantation and more an assemblage of understandable basic functions.

Even technical writers generally don't think about coding as technical writing, even when they're writing technical documents about teaching technical writing.



code is read

so

coding is writing

Programmers think of technical writing as producing technical specs, product documentation, maybe user interfaces if they've had some HCI or interaction design training. But I think teachers of technical writing need to make the case for teaching technical writing as a task that's intrinsic to programming itself.