When Bad Programs Happen to Good People

Author: *Henry G. Baker, http://home.pipeline.com/~hbaker1/home.html; hbaker1@pipeline.com*

Why do good researchers/mathematicians/professors /students sometimes write bad programs? Why is it that mathematicians, for whom simplicity and elegance of presentation and proof is supreme, often consider representations of algorithms in the form of programs as a necessary evil, and don't consider their presentation to be worthy of elegance?

We might expect engineers to be less interested in the readability and elegance of their computer code than mathematicians, but we would be wrong. We might expect mathematicians will be more concerned with correctness of the overall algorithm and moderate efficiency, while engineers will be more concerned with 'bit-twiddling,' but instead we find outstanding mathematicians like Knuth teaching Ebo`MIX`.

I can only conjecture about the reasons for this state of affairs. The most innovative mathematicians seem happy when a new result is satisfactorily proved, and leave it to lesser lights to 'clean up' and simplify the proof for future generations. Such tasks merely 'make the obvious trivial,' and are not worthy of serious thought.

People of extremely high intelligence may have larger short-term memories, enabling them to easily juggle larger numbers of concepts simultaneously, and may therefore see no need to simplify these concepts for the merely mortal.

Researchers are usually in the position of *writing* code, rather than *reading* it, and therefore may not appreciate the problems of reading another's creation.

Professors often come from an earlier school before significant thought had gone into the problem of writing readable and understandable code. Too early an exposure to the `BASIC` language or to 'machine language' can cause irreversable brain damage. Sufferers are constantly fighting the 'ghosts' of dead machines and compilers; their programs show thick accents from the 'old country' of dead computer languages.

Theory of computation people have trouble 'step-counting' any language other than 'machine language,' even though Turing himself was fond of the lambda calculus.

Editors and reviewers must also share in the blame for bad programs that appear in published books and articles.

Perhaps those of us who care about quality programs have not spoken up often enough—'for bad programs to triumph requires only that good programmers remain silent.' I call this passivity the 'Silence of the Lambdas.'

We have found to our horror that computer programs live on for decades, long after the machines and compilers that caused their misshape have died. We thus live in the purgatory created by our hackerish enthusiasm.

But the tight feedback mechanism between hardware/compiler optimizations and the software 'literature' ensures that the poor programming styles of the past will persist (because they are 'efficient' on machines optimized for these poor programming styles) and will leave little room for optimizing better styles.

We must *force* ourselves to break out of this cycle by writing excellent programs, and then molding compilers and machines to make these programs efficient, rather than vice versa. Excellent programs do not happen by accident, but require very hard work. We must proactively seek elegance, as elegance will not find us on its own.

What makes a bad program? Typically, it is substantially longer than it needs to be, so that its sheer physical size alone becomes intimidating. Corollaries include an explosion of irrelevant and non-mnemonic names, and a rat's nest of boolean alternatives.

Definition 1 *A* **bad** *program is one whose programming style is so poor that its opacity forces the reader to rewrite it from scratch, rather than going to the trouble to understand it and/or debug it.*[1]

Let's look at a few code examples[2] from published books of excellent authors. I have the highest respect for these authors, which is why it pains me so much to see

[1] Comments are useful adjuncts to a well-written program, but if they are *required* to understand the program, then it is often a bad program (or a bad programming language).

[2] We only have space here to discuss 'small' examples, inviting a 'programming-in-the-small' criticism. Most of my comments apply also to 'programming-in-the-large.'

Garbage In/Garbage Out

```
FUNCTION Euclid(a,b : INTEGER) : INTEGER;
{Computes GCD(a,b) with Euclid's algorithm}
VAR m,n,r : INTEGER;
BEGIN m:=a; n:=b;
  WHILE n <> 0 DO BEGIN r:=m MOD n; m:=n; n:=r END;
  Euclid:=m
END {Euclid};
```

Figure 1: Riesel's `Euclid` program (from [Riesel85], p. 242).

their bad programs as 'inspiration' to another generation of computer scientists.

Hans Riesel, a pioneer in the use of computers for number theoretic tasks, has an excellent book "Prime Numbers and Computer Methods for Factorization," [Riesel85/94] now in its second edition. Figure 1 is his program `Euclid` for Euclid's greatest common divisor ("gcd") algorithm.

I would give Riesel's `Euclid` program a grade of "B", because while it is not a truly bad program, it certainly isn't nearly as perspicuous as it should be for such a simple algorithm. Riesel takes a recursive algorithm and casts it into an 'iterative' form. To his credit, he makes use of the "structured" capabilities of Pascal for expressing this iteration. However, what could be more transparent than Wirth's elegant `gcd`[3] from [Jensen74], in Figure 2?

```
FUNCTION gcd(m,n : INTEGER) : INTEGER;
  BEGIN
    IF n=0 THEN gcd:=abs(m)
           ELSE gcd:=gcd(n,m MOD n)
  END;
```

Figure 2: Wirth's `gcd` program (from [Jensen74], p. 82).

Wirth's `gcd` program is smaller, uses fewer names, and is easier to understand because it is virtually identical to the list of mathematical rules for computing the gcd function. It doesn't clutter up the external namespace with extra names, and for 'infinite precision' integers ('bignums'), any minimal overhead for the 'recursion' itself will be swamped by the cost of computing bignum `MOD`. If it weren't for Pascal's silly syntax for returning values, it might also be moderately readable![4]

The usual whiners will complain that 'recursion is less efficient than iteration,' because it requires (among other things) the stacking/unstacking of $O(\log(\max(|m|,|n|)))$ stack frames. But since this gcd algorithm is *tail-recursive*—meaning that the value of the recursively-called subroutine is returned as the value of the routine itself—*no stack frames need to be stacked*, and this program **is** iterative. Compilers which can't compile this loop iteratively are *broken*.

Figure 3 is another particularly bad example of a function `Jacobi` from Riesel's book which computes the 'Jacobi symbol' function $\left(\frac{m}{p}\right)$, which tries to tell us when a number m is a quadratic residue modulo an odd prime p. This function is used in the inner loop of a common 'probabilistic' primality test.[5]

This ugly `Jacobi` program is perhaps 13 years old,[6] but even when it was written, Dijkstra's famous "Go To Statement Considered Harmful" paper [Dijkstra68] was already *16 years old*. If this program were submitted as homework in a programming course today, I would hope that it would get a very low grade.

Why does this `Jacobi` example cause me such anguish?

1. This example is from an extraordinarily bright person with an excellent education (a computer pioneer) and outstanding mathematical skills (a co-author with Erdös).

2. It shows that even *16 years* of well-funded screaming by the best and brightest of computer scientists has had *zero* impact on the programming styles of those outside the narrow scope of 'computer science'. One may

[3] I have inserted `abs()` appropriately to produce a non-negative answer. Unfortunately, Wirth's later 'extended' `GCD` program in the same book—p. 157—is extraordinarily ugly.

[4] Although it is possible to write both good and bad programs in any sufficiently powerful language, I do *not* claim that the choice of programming language is irrelevant. For example, Pascal's clumsy syntax, poor operator precedence, and defective definitions of arithmetic primitives make its programs needlessly hard to read and optimize.

[5] The Jacobi symbol function is discussed in exercise 4.5.4#23a of [Knuth81].

[6] These two examples both appear unchanged in the (1994) edition of Riesel's book.

```
FUNCTION Jacobi(d,n : INTEGER) : INTEGER;
{Computes Jacobi's symbol (d/n) for odd n}
LABEL 1,2,3,4;
VAR d1,n1,i2,m,n8,u,z,u4 : INTEGER;
BEGIN
  d1:=d; n1:=abs(n); m:=0; n8:=n1 MOD 8;
  IF odd(n8+1) THEN
    BEGIN writeln(tty,'n even in (d/n) is not allowed');
      GOTO 3 END;
  IF d1<0 THEN
    BEGIN d1:=-d1; IF (n8=3) OR (n8=7) THEN m:=m+1 END;
1: IF d1=0 THEN
    BEGIN writeln(tty,'GCD(d,n)>1 in (d/n) not allowed');
      GOTO 3 END;
  i2:=0;
2: u:=d1 DIV 2; IF d1=u*2 THEN
    BEGIN i2:=i2+1; d1:=u; GOTO 2 END;
  IF odd(i2) THEN m:=m+(n8*n8-1) DIV 8;
  u4:=d1 MOD 4; m:=m+(n8-1)*(u4-1) DIV 4; z:=n1 MOD d1;
  n1:=d1; d1:=z; n8:=n1 MOD 8; IF n1>1 THEN GOTO 1;
  m:=m MOD 2; IF m=0 THEN Jacobi:=1 ELSE Jacobi:=-1;
  GOTO 4;
3: Jacobi:=0;
4: END {Jacobi};
```

Figure 3: Riesel's `Jacobi` program (from [Riesel85], p. 286).

thus hire students that know programming*or* math, but rarely both.

3. There is no evidence that programming styles being used or taught today (1997) are any better (browse the computer shelf of any Barnes & Noble, or worse, the 'examples' given in Microsoft Press books).

Although Riesel's `Jacobi` program's most obvious fault is its complete ignorance of looping constructs—one of the major reasons for Pascal's existence!—it cannot be easily repaired with `WHILE`'s and `REPEAT`'s.

Rather than attempt to fix Riesel's wirth-less program by incremental improvements, let us rather derive a program directly from the rules for the Jacobi symbol [Riesel85/94] [Knuth81].

$$\left(\tfrac{0}{n}\right) = 0, \text{ except } \left(\tfrac{0}{1}\right) = 1 \qquad \left(\tfrac{ab}{n}\right) = \left(\tfrac{a}{n}\right)\left(\tfrac{b}{n}\right)$$

$$\left(\tfrac{1}{n}\right) = 1 \qquad \left(\tfrac{m}{n}\right) = \left(\tfrac{m \bmod n}{n}\right)$$

$$\left(\tfrac{2}{n}\right) = (-1)^{(n^2-1)/8}$$

$$\left(\tfrac{m}{n}\right) = (-1)^{(m-1)(n-1)/4}\left(\tfrac{n}{m}\right), (m, n \text{ odd})$$

The plan of the computation for $\left(\frac{m}{n}\right)$ is straightforward: recognize the boundary conditions and produce correct answers for them, otherwise use recursion to reduce the complexity of the parameters.

The program in Figure 4 is straight-forward and works very well,[7] since it incorporates most of the optimizations suggested by Riesel—e.g., only the low-order few bits of `m`, `n` contribute to the decision regarding the sign of the result. The program is *not* "tail-recursive", though, because it waits until the recursive calls return before inverting them.

It is a pity that computing Jacobi's symbol requires essentially the same steps as computing the gcd, but the standard Jacobi computation throws away the gcd information. We are thus led to a '`Jcd`' algorithm (Figure 5) that produces $\left(\frac{m}{n}\right)$ when $\gcd(m,n) = 1$ and $\pm\gcd(m,n)$ otherwise. `Jcd(m,n)` costs no more to compute than `Jacobi(m,n)`, and the Jacobi information can be trivially recovered from the `Jcd(m,n)` result.

[7] We ignore `m` < 0, since a non-recursive driver can trivially map this case to `m` ≥ 0 using the 5th rule above.

```
FUNCTION Jacobi(m,n : INTEGER) : INTEGER;
{Computes Jacobi's symbol (m/n) for m>=0, odd n>0.}
  BEGIN
    IF n=0 THEN Jacobi:=1
    ELSE IF m=0 THEN Jacobi:=0
    ELSE IF odd(m) THEN
           IF (m MOD 4=3) AND (n MOD 4=3)
           THEN Jacobi:=-Jacobi(n MOD m,m)
           ELSE Jacobi:= Jacobi(n MOD m,m)
    ELSE IF (n MOD 8=3) OR (n MOD 8=5)   {Compiler does CSE, right?}
         THEN Jacobi:=-Jacobi(m DIV 2,n)
         ELSE Jacobi:= Jacobi(m DIV 2,n)
  END {Jacobi};
```

Figure 4: New, improved `Jacobi` program.

```
FUNCTION Jcd(m,n : INTEGER) : INTEGER; {m>=0, odd n>0.}
  BEGIN
    IF m=0 THEN Jcd:=n
    ELSE IF odd(m) THEN
           IF (m MOD 4=3) AND (n MOD 4=3)
           THEN Jcd:=-Jcd(n MOD m,m)
           ELSE Jcd:= Jcd(n MOD m,m)
    ELSE IF (n MOD 8=3) OR (n MOD 8=5)
         THEN Jcd:=-Jcd(m DIV 2,n)
         ELSE Jcd:= Jcd(m DIV 2,n)
  END {Jcd};
FUNCTION Jacobi(m,n : INTEGER) : INTEGER; {odd n>0.}
  BEGIN Jacobi:=1 DIV Jcd(n+(m MOD n),n) END;
```

Figure 5: Program to simultaneously compute gcd and Jacobi.

Now by mirroring the code, we can trivially make a tail-recursive `Jcd` which keeps the sign of the result "in the program counter" (Figure 6). A good compiler will allocate only one stack frame for this function, and all of the state will be kept in the registers and the program counter. We can thus achieve transparency and efficiency in the same program.

Since more efficient gcd and Jacobi algorithms are known [Meyer96], this note is not intended to be the last word on Jacobi algorithms, but an inspiration for how to think about clear and elegant programming.

References

[Dijkstra68] Dijkstra, E.W. "Go To Statement Considered Harmful." *Comm. of the ACM* 11, 3 (March 1968), 147-148.

[Jensen74] Jensen, K., and Wirth, N. *PASCAL User Manual and Report, 2nd Ed.* Springer-Verlag, New York, 1974, ISBN 0-387-90144-2.

[Knuth81] Knuth, D.E. *Seminumerical Algorithms, Second Edition.* Addison-Wesley, Reading, MA, 1981, ISBN 0-201-03822-6. Problem 4.5.4#23a.

[Meyer96] Meyer, S.M., and Sorenson, J.P. "Efficient Algorithms for Computing the Jacobi Symbol". Cohen H. (*ed.*) *Proc. ANTS-II, 2nd Intl. Symp. Alg. Num. Th.*, Springer LNCS 1122, 1996, pp. 225-239, `http://www.butler.edu/~sorenson/`.

[Riesel85/94] Riesel, Hans. *Prime Numbers and Computer Methods for Factorization.* Birkhäuser, Boston,

```
FUNCTION Jcd(m,n : INTEGER) : INTEGER; {m>=0, odd n>0.}
  FUNCTION negJcd(m,n : INTEGER) : INTEGER; {m>=0, odd n>0.}
    BEGIN
      IF m=0 THEN negJcd:=-n
      ELSE IF odd(m) THEN
             IF (m MOD 4=3) AND (n MOD 4=3)
             THEN negJcd:=   Jcd(n MOD m,m)
             ELSE negJcd:=negJcd(n MOD m,m)
      ELSE IF (n MOD 8=3) OR (n MOD 8=5)
           THEN negJcd:=   Jcd(m DIV 2,n)
           ELSE negJcd:=negJcd(m DIV 2,n)
    END {negJcd};
  BEGIN
    IF m=0 THEN Jcd:=n
    ELSE IF odd(m) THEN
           IF (m MOD 4=3) AND (n MOD 4=3)
           THEN Jcd:=negJcd(n MOD m,m)
           ELSE Jcd:=   Jcd(n MOD m,m)
    ELSE IF (n MOD 8=3) OR (n MOD 8=5)
         THEN Jcd:=negJcd(m DIV 2,n)
         ELSE Jcd:=   Jcd(m DIV 2,n)
  END {Jcd};
```

Figure 6: Tail-recursive `Jcd` program.

1985, ISBN 0-8176-3291-3. Code is online at
`ftp://ftp.nada.kth.se/Num/riesel-comp.tar`.

Henry Baker has been diddling bits for 35 years, with time off for good behavior at MIT and Symbolics. In his spare time, he collects garbage and tilts at windbags. This column appeared in ACM Sigplan Notices 32,3 (Mar 1997), 27-31.