The Story of Mel, A Real Programmer, *Annotated*

Story by Ed Nather

Annotated by Erik Brunvand
Department of Computer Science
University of Utah, SLC, Utah 84112
elb@cs.utah.edu, www.cs.utah.edu/ elb

October 14, 1996

1 The Story of Mel, A Real Programmer

This was first posted to Usenet on May 21, 1983. [1]

A recent article devoted to the macho side of programming
made the bald and unvarnished statement:

Real Programmers write in FORTRAN. [2]

Maybe they do now,
in this decadent era of
Lite beer, hand calculators, and "user-friendly" software [3]
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes, [4]
Real Programmers wrote in machine code. [5]
Not FORTRAN. Not RATFOR. Not, even, assembly language [6][7][8].
Machine Code.
Raw, unadorned, inscrutable hexadecimal numbers. [9]
Directly. [10]

Lest a whole new generation of programmers
grow up in ignorance of this glorious past,
I feel duty-bound to describe,
as best I can through the generation gap,
how a Real Programmer wrote code. [11]
I'll call him Mel,
because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp., [12]
a now-defunct subsidiary of the typewriter company.
The firm manufactured the LGP-30, [13]
a small, cheap (by the standards of the day) [14]

drum-memory computer, [15]
and had just started to manufacture
the RPC-4000, a much-improved, [16]
bigger, better, faster – drum-memory computer.
Cores cost too much, [17]
and weren't here to stay, anyway.
(That's why you haven't heard of the company,
or the computer.) [18]

I had been hired to write a FORTRAN compiler [19]
for this new marvel and Mel was my guide to its wonders.
Mel didn't approve of compilers.

"If a program can't rewrite its own code", [20]
he asked, "what good is it?"

Mel had written,
in hexadecimal,
the most popular computer program the company owned.
It ran on the LGP-30
and played blackjack with potential customers
at computer shows. [21]
Its effect was always dramatic.
The LGP-30 booth was packed at every show, [22]
and the IBM salesmen stood around
talking to each other.
Whether or not this actually sold computers
was a question we never discussed.

Mel's job was to re-write
the blackjack program for the RPC-4000.
(Port? What does that mean?) [23]
The new computer had a one-plus-one
addressing scheme, [24]
in which each machine instruction,
in addition to the operation code
and the address of the needed operand,
had a second address that indicated where, on the revolving drum,
the next instruction was located. [25]

In modern parlance,
every single instruction was followed by a GO TO! [26]
Put that in Pascal's pipe and smoke it. [27]

Mel loved the RPC-4000
because he could optimize his code: [28]
that is, locate instructions on the drum
so that just as one finished its job,
the next would be just arriving at the "read head" [29]
and available for immediate execution.
There was a program to do that job,
an "optimizing assembler", [30]

but Mel refused to use it.

“You never know where it’s going to put things”, [31]
he explained, “so you’d have to use separate constants.” [32]

It was a long time before I understood that remark.
Since Mel knew the numerical value
of every operation code, [33]
and assigned his own drum addresses,
every instruction he wrote could also be considered
a numerical constant. [34]
He could pick up an earlier “add” instruction, say,
and multiply by it,
if it had the right numeric value. [35]
His code was not easy for someone else to modify. [36]

I compared Mel’s hand-optimized programs [37]
with the same code massaged by the optimizing assembler program, [38]
and Mel’s always ran faster.
That was because the “top-down” method of program design
hadn’t been invented yet, [39]
and Mel wouldn’t have used it anyway.
He wrote the innermost parts of his program loops first, [40]
so they would get first choice
of the optimum address locations on the drum. [41]
The optimizing assembler wasn’t smart enough to do it that way. [42]

Mel never wrote time-delay loops, either, [43]
even when the balky Flexowriter [44]
required a delay between output characters to work right. [45]
He just located instructions on the drum
so each successive one was just past the read head
when it was needed; [46]
the drum had to execute another complete revolution
to find the next instruction.
He coined an unforgettable term for this procedure.
Although “optimum” is an absolute term,
like “unique”, it became common verbal practice
to make it relative:
“not quite optimum” or “less optimum”
or “not very optimum”.
Mel called the maximum time-delay locations
the “most pessimum”.

After he finished the blackjack program
and got it to run
(“Even the initializer is optimized”, [47]
he said proudly),
he got a Change Request from the sales department. [48]
The program used an elegant (optimized) [49]
random number generator [50]

to shuffle the “cards” and deal from the “deck”,
and some of the salesmen felt it was too fair,
since sometimes the customers lost.
They wanted Mel to modify the program
so, at the setting of a sense switch on the console, [51]
they could change the odds and let the customer win.

Mel balked.
He felt this was patently dishonest,
which it was,
and that it impinged on his personal integrity as a programmer,
which it did, [52]
so he refused to do it.
The Head Salesman talked to Mel,
as did the Big Boss and, at the boss’s urging,
a few Fellow Programmers. [53]
Mel finally gave in and wrote the code,
but he got the test backwards, [54]
and, when the sense switch was turned on,
the program would cheat, winning every time.
Mel was delighted with this,
claiming his subconscious was uncontrollably ethical,
and adamantly refused to fix it.

After Mel had left the company for greener pa$ture$,
the Big Boss asked me to look at the code
and see if I could find the test and reverse it.
Somewhat reluctantly, I agreed to look.
Tracking Mel’s code was a real adventure. [55]
I have often felt that programming is an art form,
whose real value can only be appreciated
by another versed in the same arcane art; [56]
there are lovely gems and brilliant coups
hidden from human view and admiration, sometimes forever,
by the very nature of the process.
You can learn a lot about an individual
just by reading through his code,
even in hexadecimal.
Mel was, I think, an unsung genius.

Perhaps my greatest shock came
when I found an innocent loop that had no test in it.
No test. None. [57]
Common sense said it had to be a closed loop,
where the program would circle, forever, endlessly. [58]
Program control passed right through it, however,
and safely out the other side. [59]
It took me two weeks to figure it out.

The RPC-4000 computer had a really modern facility

called an index register. [60]
It allowed the programmer to write a program loop
that used an indexed instruction inside;
each time through,
the number in the index register
was added to the address of that instruction,
so it would refer
to the next datum in a series. [61][62]
He had only to increment the index register
each time through.
Mel never used it.

Instead, he would pull the instruction into a machine register, [63]
add one to its address,
and store it back. [64]
He would then execute the modified instruction
right from the register. [65]
The loop was written so this additional execution time
was taken into account —
just as this instruction finished,
the next one was right under the drum's read head,
ready to go.
But the loop had no test in it. [66]

The vital clue came when I noticed
the index register bit, [67]
the bit that lay between the address
and the operation code in the instruction word, [68]
was turned on — [69]
yet Mel never used the index register,
leaving it zero all the time. [70]
When the light went on it nearly blinded me.

He had located the data he was working on
near the top of memory — [71]
the largest locations the instructions could address —
so, after the last datum was handled,
incrementing the instruction address
would make it overflow. [72]
The carry would add one to the
operation code, changing it to the next one in the instruction set:[73]
a jump instruction. [74]
Sure enough, the next program instruction was
in address location zero, [75]
and the program went happily on its way.

I haven't kept in touch with Mel,
so I don't know if he ever gave in to the flood of
change that has washed over programming techniques
since those long-gone days.

I like to think he didn't.
In any event,
I was impressed enough that I quit looking for the offending test,
telling the Big Boss I couldn't find it.
He didn't seem surprised.

When I left the company,
the blackjack program would still cheat
if you turned on the right sense switch,
and I think that's how it should be.
I didn't feel comfortable
hacking up the code of a Real Programmer.

2 Annotations

[1] The Arpanet was the first computer network. That is, the first time that different computers were connected together so that information could be shared between them without using a separate medium like paper tape or magnetic disks. The Arpanet was born with five sites in 1969 (before video games (Pong didn't appear until 1972), before hand-held calculators (1972), home VCRs (1975), and personal computers (1976-1977)) as a way to let researchers funded by ARPA (the defense department's Advanced Research Projects Agency) collaborate. The original five sites on the Arpanet were: Bolt, Beranek, and Newman (BBN), a Cambridge Mass. based company that built the original Arpanet Interface Message Processors (IMP) that were used as the special-purpose computers that passed data on the Arpanet, UCLA, Stanford Research Institute (SRI), University of California at Santa Barbara, and the University of Utah.

The first public demonstration of the Arpanet came in 1972. By then the Arpanet has expanded to 37 nodes across the country. The public demonstration was at the International Conference on Computer Communications and was a great success.

As the Arpanet grew, other networks were developed that used the technology developed on the Arpanet. One of these, the USENET, was a collection of computers connected by phone lines but without the expensive IMPs that made the Arpanet run. USENET was slower, but anyone could connect if you had a phone connection to your computer. USENET came into being around 1979. Although there were special-interest email lists on the Arpanet almost since its inception (the SF-lovers list for science fiction fans was one of the first), it was on the USENET that the idea of interest-specific electronic bulletin boards (bboards) or newsgroups really blossomed. It was on just such a newsgroup that The Story of Mel, a Real Programmer was first posted on May 21, 1983.

Although I first saw this story without attribution, it turns out that the author is known. It is Ed Nather, then of the University of Texas who wrote the story in response to an earlier article in Datamation Magazine.

[2] The article "Real Programmers Don't Use Pascal", by Ed Post, was published in Datamation Magazine in 1983. It is an article written as a humorous piece in response to the recent development of the computer programming language Pascal, and the concept of "structured programming" that went with the Pascal language. At the time of the introduction of Pascal, most scientific programmers used FORTRAN as a programming language.

FORTRAN, which stands for FORmula TRANslation, is a language that had its beginnings in the late 1950's as a mechanism for describing scientific expressions at a higher level than the direct machine instructions of computers. It was, and still is, designed to get the maximum computation speed from a given program. This makes it somewhat "unfriendly" in being able to program more symbolic or structured applications.

With a language like Pascal, the intent was to define a richer set of programming structures that could deal easily with non-numeric data such as text, strings, symbols, etc. Also, Pascal has a different view of how the control flow of the program should be described. Rather than being able to jump from any point in the program to any other point (instructions that perform this type of control change are called "goto" statements because they go to another part of the program), Pascal requires that the control be structured into a fairly strict hierarchal organization.

The main point of the Datamation article was to poke fun of Pascal as an academic language where "real programmers" would use a more fundamental language like FORTRAN which is harder to use, faster, and therefore more macho.

A not-completely-appropriate analogy might be to liken FORTRAN to a race car where the intent is speed, but the car is uncomfortable and difficult to drive, and trying to make it suitable for daily driving off the racetrack would require a lot of work, and Pascal to a standard sedan of some sort where the ride is more comfortable for daily driving, but the performance on the race track would suffer, and the features of the car that make it comfortable are what get in the way of getting all the performance that might be possible from the raw engine parts.

[3] Electronic hand calculators first appeared in 1971, with the influential HP-35 pocket calculator arriving soon afterwards in 1972. This signalled the end of the slide-rule era in engineering.

The term "user-friendly" is used in quotations here in, I believe, a couple of different meanings. One is that software that is called "user-friendly" very often isn't. Oftentimes the extra window dressing put onto a piece of software to make it user-friendly serves mostly to make it harder to use. The other meaning is a reference to the "real programmer" ethic. One on-line collection of real-programmer-isms says something to the effect of "Software is hard to write, it should be hard to use." To programmers of this ethic, adding extra code simply to make the program easier to use is wasted effort. People who value the contribution of the code should have to spend the effort to learn how to use it.

[4] "Drums" refers to drum memories which are magnetic data storage devices much like the familiar hard disks of today but shaped like cylinders instead of platters. In the 50's and 60's the state of the art in magnetic encoding of data didn't allow for very dense storage on the magnetic surface of the medium. So, in order to increase the surface area of the storage medium, to increase the speed at which the magnetic material passed by the read head of the device, and to keep the speed at which the heads encountered the magnetic medium constant, these devices were shaped like cylinders with the magnetic coating on the outside surface of the cylinder and the cylinder rotating along its long axis at very high speeds. Consider the early phonographs which were wax cylinders versus the now-becoming-obsolete vinyl record. The considerations are analogous to the drum versus disk question.

Vacuum tubes are electronic devices used to amplify electronic signals and to act as electronically controlled switches. Digital computers are constructed at their most basic fundamental building blocks as electronically controlled switches. The very first electronic computers used relays for these switches (in the 1940's). Vacuum tubes were used for the switches from the late 40's to the late 50's. The first commercial computer to use transistors for the switching elements was the IBM 1401 in 1959 starting the end of the tube era. Individual transistors gave way to integrated circuits in which hundreds, and eventually millions, of transistors were fabricated on a single silicon substrate, or chip. Integrated circuits have been used from the mid-60's to the present.

[5] Machine code refers to the lowest level of program instructions. Machine codes are the instructions actually interpreted by the hardware of the computer. Every computer has a machine-code instruction set of operations that the machine knows how to do. These instructions are almost always simple add and subtract operations on internal computer registers, simple load and store operations on the computers memory (used to bring data from the memory into the internal registers), and a few basic control-flow operations that test values in the internal registers of the computer and decide whether to alter where the next machine instruction is fetched from in memory or not based on that test.

Machine code instructions are usually represented as strings of 1's and 0's that are interpreted by the binary logic of the computer. For example, an machine code instruction used to add the contents of two registers and put the result in a third might look like:

```
1010 0011 0100 0001
```

Where the `1010` part would be the indicator that this operation is to add two numbers, the `0011` is 3 in binary indicating that the first argument should come from register 3, the `0100` is 4 in binary signifying that the second argument should some from register 4, and the `0001` is 1 in binary which indicates that the result of the addition should be stored into register 1. This is an extremely simple, made up example. A real machine code instruction might have many more bits, and be quite a bit more difficult to explain.

The purpose of so-called high-level languages is to be able to express an addition in a more convenient text form and have the compiler translate that form into the machine code for you. As you can imagine, programming directly in machine code would require a special type of talent.

[6] FORTRAN, as discussed in a previous note, is an early computer language used for scientific programming. Although FORTRAN was one of the earliest, if not the earliest, attempt at a high-level language, it is still used today (albeit in a many-times-updated form) for scientific applications.

[7] Rational FORTRAN (RATFOR) is a dialect of FORTRAN that provides a richer set of control-flow possibilities. RATFOR is not a new language, rather it is a translator that translates Rational FORTRAN into FORTRAN. It allows the programmer to use features not found in the normal FORTAN language, and then translates those features into standard FORTAN automatically.

[8] Assembly Language is a term used to describe the machine code of a particular computer, but as text rather than just bits. For example, consider the machine code for addition in the previous note. An assembly language version of that instruction might be:

```
ADD R3 R4 R1
```

Note that this is essentially the same machine language instruction, but text has been used instead of the actual bits that the machine sees. Programs written in the assembly language of a particular machine are translated into machine code by a program called an assembler.

[9] Hexadecimal means base 16. Binary is a base 2 number scheme because there are only two symbols used: 1 and 0. Decimal numbers are a base 10 scheme because there are 10 symbols used: 0 1 2 3 4 5 6 7 8 9. After 9 numbers are represented using positional notation where the higher order symbols represent multiplying by the base. 15 for example is 1*10 + 5*1.

Hexadecimal is base 16 which means that numbers are represented using 16 different symbols: 0 1 2 3 4 5 6 7 8 9 a b c d e f. After counting to f, the same positional notation is used. 15 in hexadecimal notation (or hex for short) is 1*16 + 5*1 or 21 in base 10.

Hex is used in computer notation as a convenient way of expressing bit values. If you group a binary number into groups of four bits there are exactly 16 possible values for each of those groups. You can easily express each group of four bits as a single hex digit. For example, the previous machine code example was, in binary:

```
1010 0011 0100 0001
```

In hex, this same number would be expressed as:

`A341`

where each hex digit corresponds to a group of four bits in the binary word. Hexadecimal is widely used as a compact way of describing bit encodings of things like machine codes.

[10] The process of writing a program directly in machine code would be to write hex digits that correspond directly to each desired machine code bit-string for each instruction you want the machine to operate. Since it takes many many machine code instructions to execute even the simplest program, this would be tedious to say the least.

Even now tools that extract the raw unformatted data from a memory or from a disk are likely to return a file full of "raw, unadorned, inscrutable hexadecimal numbers." However, the number of programmers who can write programs by writing those numbers directly are few.

[11] To a programmer, the word "code" refers to the text of the program you are writing. It can be used as a noun. The "source code" is the program text in the language that the programmer is using (i.e. FORTRAN). The "object code" is the machine code produced usually by the operation of the compiler on the source code. Generically the code is the program in some form. It can also be used as a verb. "To code" is to write a program. If a programmer tells you he is "coding up an application" he is writing a program to perform a specific task.

[12] The Royal McBee Company is a real company. I have found out little about the company except that the machines named in this story are real machines that were built in the late 1950's and early 1960's.

[13] The LGP-30 computer was first sold in 1959. There were a number of these machines in use in various military installations from 1959 on. The Army Ballistic Missile Agency (now called NASA) for example had 10 LGP-30's in 1961.

[14] The LGP-30 contained 113 vacuum tubes and cost $50,000 in 1959.

[15] Recall the definition of a magnetic drum memory: a rotating cylinder with a magnetic coating on the outside and read and write heads that move across the rotating outer surface. A drum-memory computer is one where the main memory of the machine is a drum of this sort. There was no RAM like you would find in your PC at home, all the programs and data were stored on the drum. Imagine running all your programs from a floppy disk with no RAM on the machine at all and the floppy providing your system memory and your file storage. The result would be a much slower computer than you might be used to.

[16] The following table shows the addition and multiply times for the LGP-30 and RPC-4000 computers. These are taken from records of the computers used by defense contractors in the 1960's. Note that the RPC-4000 computer makes a significant improvement in add times, but has the same multiply and divide time as the older LGP-30. In contrast, your PC at home is likely to have identical add and multiply times of about 10 nanoseconds, or .01 microseconds.

ADD TIME MICROSECONDS	MULTIPLY TIME MICROSECONDS	DIVIDE TIME MICROSECONDS	SYSTEM
8,000	17,000	17,000	LGP 30
500	17,000	17,000	RPC 4000

[17] Core memory is a type of random-access memory that stores bits by changing the magnetic field of a small magnetic doughnut. These torus-shaped magnets (or cores) can be magnetized or De-magnetized by passing a current through a wire that is strung through the center of the core. By passing a current through a different wire, also strung through the core, you can sense whether the core is magnetized or not. Core memory was faster, denser, and more robust than magnetic drums, although initially more expensive. Magnetic core memories were used in computers from the early 1960's through the 1970's.

[18] Core memory ended up being so much faster than magnetic drums and disks that it quickly became the main-memory of choice for computers from the early 60's on. Initially expensive, core rapidly became more cost effective than drums. Companies that didn't make the switch soon enough were not likely to survive. By the same token, semiconductor memory in the late 70's offered the same dilemma. It was faster, smaller, and denser, but originally much more expensive. Some companies didn't believe that semiconductor RAM would catch on. You haven't heard of these companies either.

[19] A compiler is a program that takes the original program text (source code) and translates it to machine code so that the computer can understand it. Initially compilers simply translated the program statements into the direct machine-code versions. As compiler technology advanced, however, the source programs could get further and further away from the machine code (become more "high level" languages) and the job of the compiler to translate these more abstract expressions into the simple machine code that the machine can actually execute became more and more complicated. Modern compilers are very complex programs indeed.

[20] One of the great leaps of understanding in the early days of computing is attributed to John Von Neumann who, along with others, realized that a computer could use the memory that was then thought to be useful only for storing the data, to also store the program that it would execute. Before the advent of the "stored program" computer, the program would be communicated to the machine by setting huge numbers of switches on vast panels that controlled the operation of the machine. Once the switches were set, the machine would execute the indicated operations on the data that was stored in the machine's memory. Needless to say, switching which program was being run was a serious undertaking.

The advantage of the stored program approach is that the program, or set of instructions, for the computer is stored in the memory along with the data. That way the same mechanism that allows the computer to retrieve data from the memory can be used to retrieve the next instruction to execute.

Once you have the program stored in the machine's memory, that program can be thought of as stored instructions, or just as a different type of data. That is, you can have the program that is running modify the program data in a way that modifies how subsequent instructions are executed. This is known as "self-modifying code" because the program modifies itself by writing data in the section of memory that holds the program. Because of the small memories that early machines had, this was a common programming practice. It has since fallen out of favor because it makes programs very difficult to understand and debug. It's difficult to tell if a program is doing the right thing if the program itself might change after writing different instructions into the program data. Compilers, in general, forbid self-modifying code by specifically not generating such code in the machine code produced by the compiler. If you want to write this type of code, you have to do it by hand. Caveat Programmer.

[21] Computer shows were (and still are) large conventions where manufacturers of computer hardware and software show off their latest and greatest products to potential customers. In the early days the highlights of these shows were as likely to be new hardware designed by a couple scruffy teen age hackers as from a big company like IBM, or like Royal McBee. The original Apple II computer that essentially started the home computer revolution was introduced at the West Coast Computer Faire in 1977.

[22] Typical computer shows are organized in large convention centers where each company rents some space and puts up a booth. The size and placement of the booth are usually good indicators of the size and placement of the company in the computer world. However, the star of any particular computer show is likely to be a small booth tucked into a corner somewhere with a ground-breaking, but undiscovered product.

[23] What does "to port" mean when talking about computer programs? To port a program is to translate it to another language, another system, or in some way modify the program to run under another set of circumstances. If you have a program that runs on a Macintosh, for example, it would have to ported to run on an IBM PC. Porting is assumed to involve modifying only small parts of the program, as little as necessary to get the program to run on the new system.

Computer programs are often designed to be portable by isolating the machine-specific part of the code to one well-defined part of the program. The theory is that to port the program to a new system, only that machine-specific part will need to be modified. In practice, this theory is often wrong and there are many subtle changes that need to be made in other parts of the program.

[24] An addressing scheme to a computer is the mechanism by which the computer figures out the location in memory (the address) of the next instruction to be executed in the program. The address of the next instruction is usually either the next sequential location in memory or an instruction somewhere else in memory where the address of the "somewhere else" is encoded in the current instruction. In order to decode the "somewhere else" address from the machine code you would need to know the addressing scheme of the machine.

[25] I don't have any examples actual LGP-30 code, but based on this description, an ADD instruction might look something like:

```
ADD 100 104
```

where the 100 is the memory location of the argument to the addition ("add the contents of memory location 100 to the current sum held in the internal register of the computer"), and the 104 is the address of the next instruction to execute ("When you are finished with the addition, find the instruction at memory location 104 and execute that instruction next").

[26] A "goto" instruction is one that changes where the program will find its next instruction. Most programming languages (unlike the LGP-30 machine language) execute instructions in the same sequence in which they are written in the program. That is, when the machine finishes one instruction, it fetches the next sequential instruction in the machine's memory and executes that one. The goto is an instruction that forces the execution to jump, or "go to" an instruction somewhere else in the memory that is not in sequence.

Influential computer science pioneer (and legendary figure himself), Edsgar Dijkstra considered the goto statement to be a bad idea because they make understanding the orderly flow of control in the program more difficult. (clearly Dijkstra is not a hacker) This was published in a famous (to computer scientists) letter: "GOTO Statement Considered Harmful," Letter of the Editor, Communications of the ACM, March 1968, pp. 147-148.

[27] The Pascal language, developed by Nicklaus Wirth in the mid 1970's, had as its goal to support exactly the type of "structured programming" that Dijkstra was advocating. Not only does Pascal not have a "goto" statement, but it forces the programmer to carefully structure the control flow of the program so that it can be more easily understood and reasoned about.

The first major reference to Pascal by its creator is: Wirth, N., *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

[28] As discussed later in the story, while "optimum" is an absolute descriptor meaning the best that can possibly be, the verb "to optimize" to a programmer has come to mean "to improve." The process of optimizing code is to take a computer program and improve it by making it faster, or smaller, or both.

[29] A drum, like a hard disk, is a magnetic storage medium. That is, the bits that are stored on the drum are encoded as magnetized spots on a magnetic coating on the surface of the drum. In order to retrieve the information stored in this way, the drum needs to be able to read the data. The "read head" is the piece of the drum that senses the magnetic spots and reports the values to the computer. It "reads" the values on the disk in much the same way that a tape recorder reads the magnetic information on a cassette tape and reports them to an audio amplifier.

Drums, like disks, are rotating storage devices. That is, the drum rotates at high speed below the read and write heads. The read head is restricted to reading only spot that is currently directly under the read head. So, in order to read a specific spot on the drum, the read head must wait patiently for the desired spot to rotate around and come under the head. Although the drums might have spun quite quickly, electricity moves much faster. Put another way, even though the time it takes for the disk to rotate one complete revolution seems very short, it is long compared to the speed of the computer processor. In a computer where the drum is the main memory, a poorly optimized program would spend most of its time waiting for the disk to rotate to the right spot.

[30] An assembler is a program that takes symbolic assembly language (like `ADD 100 104`), and converts it to the machine code equivalent that the computer can actually read (like `0011 111110100 1111101100`). An optimizing assembler is an assembler that also tries to improve the program by analyzing the structure of the program ad using the analysis to remove unneeded instructions, replace wasteful sequences with more efficient ones, and in the case of a drum computer, locate the instructions on the drum in an efficient way by modifying the location of the instructions in the program.

[31] One of the things an assembler does is take the symbolic assembly code and locate it somewhere specific in memory. Depending on where the assembler decides to place the code, all the addresses in the assembly program will have to be modified to take account the exact location. By "...where it's going to put things" the story refers to where the instructions will actually be located in terms of the locations on the drum.

[32] "Constants" in a computer program are simply numbers that will come in handy to the rest of the program. These numbers are typically stored in known locations so the programmer can use them as the need requires. For example, if incrementing a value is a common operation, it might make sense to store a 1 at a known location so that you could always increment a value by adding the contents of that special location.

[33] In a machine instruction (or more exactly, an assembly instruction) like `ADD 100 104`, the "ADD" is also called the "operation code" because it defines the operation that the instruction will perform.

[34] Operation codes, like the "ADD" in the instruction `ADD 100 104` are, themselves, coded as a series of 1's and 0's so that the computer can read them. Because these are just strings of numbers, they could also, if you wanted to, be considered a number. For example, if the operation code for ADD was `0011`, this could also be read as the binary number 3.

[35] In the previous example, the operation code for ADD is 0011, which is also the number 3 in binary. So, if you wanted to multiply some value in the program by 3, you could conceivably multiply by the operation code for ADD since that operation code is, coincidentally, also 3.

[36] You can imagine how difficult this would be to understand if you weren't looking for it...

[37] Hand-optimized means that the improvements to the code were done by the programmer and not by any automatic means like using an optimizing assembler.

[38] The verb "to massage" in reference to a computer program means to modify the code in some way. The connotation is that the structure of the program is changed by pushing, pulling, and molding the program code.

[39] "Top down" design refers to a system design philosophy that starts by looking a the overall picture (the top of the design) and working your way down to the specifics. This is a "divide and conquer" method of designing large systems where large pieces of the design are split into smaller and smaller pieces until each of the individual pieces are small and simple enough to build. For software, this method of design would involve starting with the big picture of the whole program and then refining that idea into smaller and smaller units before even starting to write the code.

[40] In contrast to the "top-down" approach, there is the "bottom-up" approach. In this method of designing large systems, you start with the smallest pieces: the ones that you already know how to do well. These pieces are then assembled into larger and larger units until you have the entire design.

In this case the smallest units are "the innermost loops" of the program. Loops are a generic name for a programming language construct that repeats sections of the code some number of times. Suppose you were writing a program to add a monthly bonus onto each employee's paycheck. You would write a program loop that would repeat the "add bonus" process on each employee's record. Now suppose that in each employee's record your program would have to loop again, perhaps to repeat the bonus process for each month of the year (this is a very generous company). The "repeat for each month of the year" loop would be an "inner loop" because it is a looping structure inside a larger looping structure. If this hypothetical inner loop repeated its function 12 times (one for each month), and there are 100 employees in your company, this simple inner loop would execute 1200 times during one run of the program.

It is this property of inner loops, that they account for a large fraction of the total run time of a program, that makes them the first target for program optimization. A general rule of thumb in computer programming is that 90% of the execution time for a generic program is accounted for by only 10% of the written instructions. This is because a small number of inner loops execute over and over again. The other 90% of the code typically runs only a few times during the programs execution and so does not account for much of the total run time.

[41] Remember the rotational latency problem on the drum memory. By placing the next instruction at a place on the drum such that when it was time to execute that instruction the read head was exactly in the right spot, the program would run much much faster.

[42] This anthropomorphism of the assembler program ("The optimizing assembler wasn't smart enough") is very common in the programming world. It is very common to talk about a computer program in terms that make it sound like it is thinking, making mistakes, or otherwise behaving like a human.

[43] One property of even the early computers like the LGP-30 and RPC-4000 machines is that they are much faster at computing answers than the output devices (like printers) are at printing them. Modern solutions to this problem are to send the information to be printed all in one very quick burst to the printer where the information is stored while the slow output device manages to deal with it. In the early days, however, it was common to write computer programs so that they program slowed itself down enough to send a character at a time to a slow printer. The mechanism to slow down the program is called writing a "delay loop." A delay loop is simply a looping structure in the program that doesn't do any computation, but instead has as its sole purpose to waste time. After looping around doing nothing for a while, enough time will be wasted so that another character can be sent to the slow output device.

[44] A Flexowriter was a very popular early computer printer and input device made by Friden. It was essentially a computer controlled typewriter that could also produce and read paper tape. Paper tape is an input/output medium that looks like a long thin (about an inch) roll pf paper with holes punched it it. A programmer might use the Flexowriter to typein a program and have that program put on a paper tape. The paper tape could then be fed into the computer to enter the program into the computer. Alternatively, the computer could produce the results of its computation to the Flexowriter and have those results stored on a roll of paper tape.

[45] The interface to the Flexowriter was for the computer to send individual characters to the Flexowriter one after the other. The problem was that because the Flexowriter had to move physical objects in order to punch holes in the paper tape, it was very slow compared to the computer. If the computer send characters to the printer too quickly, they were lost (or "dropped on the floor" as a programmer would say). Output procedures in programs would often use delay loops to slow things down enough for devices like the Flexowriter.

[46] Remember the rotational problem with the drum memory. If locating an instruction so that it is under the read head just when you want it makes the program run faster, then locating the next instruction so that it was just past the read head when you went looking for it means that it would take another entire rotation of the drum before the correct location appeared at the read head. This rotation is a long as far as the computer is concerned.

[47] Recall the definition of "inner loops" of a program, and the maxim that 90% of the run time of a program is account for by 10% of the instructions. It doesn't really make sense to worry about optimizing the other code since even if it were optimized to nothing at all, the most difference it would possibly make is a 10% difference in run time. Unless you're a Real Programmer, that is. Then you feel that all code deserves to be given attention and so even the initializer should be given the benefit of optimization.

[48] A Change Request, sometimes called an ECO for Engineering Change Order, is a mechanism used by engineering companies to specify and track modifications to the systems they produce. These Change Requests follow a chain of command, and allow large engineering groups to collaborate without losing track of is going on. To an engineer, a change request from the sales department is not likely to be a meaningful engineering issue.

[49] "Elegant" is a widely used term of praise and respect by programmers. If a solution to a problem is simple, efficient, or otherwise aesthetically pleasing to a programmer, it would be given the tile of elegant.

[50] Generating random numbers is a generic problem that s used in all sorts of applications. Games, in particular, require random numbers so that when they set up the initial conditions they are not the same each time you play the game. A computer program to play cards, for example, would us a random number generator to simulate shuffling the deck prior to dealing the cards.

In practice, generating truly random numbers is very difficult. Mathematicians could tell you a lot about what it means for a set of numbers to be truly random and why computer generated "random" numbers have all sorts of non-obvious regularity, but for purposes of shuffling a deck of cards, program generated random numbers are completely adequate.

[51] Older computers typically had a console on the body of the machine where internal status information of the machine was displayed. There might also be a number of switches on the console that could be used to modify the state of the machine. In addition to the switches dedicated to particular actions, there would also be some switches that the program running on the machine to read to determine the value during program execution. These switches, called sense switches, could be used for input to the program. A program could, for example, get to a point in a program, and sense the value of a switch on the console to determine what to do next.

[52] The Hacker Ethic is a code of conduct defined by hackers in their pursuit of computer knowledge. Simply put, it is that information should be free and available to all who so that they can use to to further their own exploration of the system. A hacker's program is an expression of his/her mastery of the system and as such, should be as elegant as possible (Even where in order to be elegant the programmer makes use of tricks so obscure that non-hackers can't understand what's going on. The beauty is still there for the real hacker). The idea of purposely making a program deceitful would definitely be considered by some as a violation of the hacker ethic.

[53] It is well established in hacker lore that company higher-ups such as the Head Salesman and the Big Boss would know nothing about engineering, programming, and definitely nothing about the hacker ethic. The Dilbert cartoon series by Scott Adams has exploited this knowledge in an way that is both extremely humorous and possibly extremely close to the mark for most engineers. Scott Adams says that he gets many of his ideas for new comic strips from people working for big companies who send him ideas based on their actual workday experiences.

The Fellow Programmers referred to in this passage are clearly not Real Programmers!

[54] "The test" referred to is the process of testing the value of the sense switch to see if it is in the on or off position.

[55] Although there are many stories about legendary programmers that circulate amongst programmers, it is thought by some that although they are admired and idolized, it's probably a good thing that there aren't too many of them around. The tricks that make a programmers code legendary typically make the code itself very hard to understand by anyone else.

[56] Programming is often described in terms that relate to magic. Computers themselves are often considered as magical devices. People not used to how computers operate often describe the result of a program as magical somehow. Programmers also use these terms for their best programming tricks. If you were to look at the comments (non-program-text annotations) in the source of a hackers program listing you might see a comment about how the next routine involves magic, or if it is especially obscure, deep-magic.

In addition, programmers have since the early days used magical terms to describe themselves and their responsibilities. Terms like "wizard" and "guru" are commonly applied to programmers of special ability. To say someone is a "UNIX wizard" is to say they have deep knowledge of the UNIX system (a computer operating system liked by hackers) that allows them to make the computer do magical things.

[57] Loops in a program are used to repeat a portion of the code a number of times. Each time the program executes the code in the loop a decision must be made of whether to loop again or whether this was the last iteration and the program can continue onto different activities. This is the test that is referred to in this part of the story. If a loop had not test it would have to loop forever. The test is what indicates that it is time to stop looping.

[58] A program loop that has no test is known as an "endless loop" or "infinite loop" to a programmer.

[59] That is, the program managed to check the loop's exit condition, find it changed, and exit the loop, even though no exit test was ever performed according to the code.

[60] An index register is a feature of a computer processor that relates to the machine's addressing scheme (see note [24]). Using an index register in an addressing scheme involves adding the value in the index register to the value in the instruction for all memory references. If the index register is incremented automatically by the computer, the programmer could walk through sequential memory locations by simply setting the index register to the first location and then, with no further manipulation of the address by the program, look at sequential memory locations on each memory reference. This turns out to be very a very helpful addition to a basic addressing scheme.

[61] The singular of data is datum.

[62] It is a common technique to store data in the computer's memory so that data that are related to each other are also closely spaced in memory. A list of names, for example, might be placed in consecutive memory locations. Walking through the list of names would involve operating on each datum in the list by looking at consecutive memory locations.

[63] A machine register is a temporary storage location that is inside the computer processor itself. All the data that a program uses is held in the main memory of the computer (in the LGP-30 and RPC-400 case, this would be the drum, in the case of your PC this would be the RAM). In order to operate on this data the computer brings it into a machines register in its CPU (central processing unit). The program typically operates on the data in the machines registers until the operation is complete, and then writes the updated values back to the main memory.

[64] That is, write the updated value back to the drum to the location from which it came originally.

[65] That is, instead of getting the instruction from the drum which would be the normal case.

[66] That is, he's walking through a list by executing a single instruction, but each time through the loop, the instruction itself is modified to point to a different location. There is no test that says when to finish this loop.

[67] Single bits are often used as flags to indicate which features are enabled or being used in a particular program. Computers generally have a number of these bits, sometimes called a Processor Status Register that is kept internally in the computer and accessible only to the operating system.

[68] Although I don't have the exact instruction format for the either the LGP-30 or RPC-4000, this indicates that the bit that controls the use of the index register is in between the operation code bits on the left, and the address of the data on the right. The layout is something like:

```
operation-code index-bit address
```

[69] Bits have only two states: on and off. Also called 1 and 0.

[70] Having the index register contain the value 0 means to add 0 to every address before it's executed. That is, use the index register, but since it's adding 0, it never has any effect on the actual address.

[71] Memory in a computer is addressed by number. Each number corresponds to a different location in memory. These addresses typically start at 0 and go to whatever the highest number is for a memory of that size. For example, if a memory had 1024 locations, they would be numbered 0 to 1023. The lower numbered locations are considered the bottom of memory, and the highest numbered locations are considered the top of the memory.

[72] Arithmetic overflow is caused when the result of an operation is larger than the maximum size allowed for the result. Consider if you wanted to use only numbers that could be expressed in two decimal digits. You would be restricting yourself to numbers between 0 and 99 because 100 must use three digits. Now, if you performed addition on two legal numbers, say 55 and 62, you would get overflow because the answer, 117, is too large to fit in the two-digit limit. Because computers have fixed-bit-width arithmetic units, overflow is an issue that must be considered carefully.

[73] Another way of looking at overflow is to consider the extra digit of an overflowed number as a carry to the next operation. Suppose you are adding 55 and 68. First computer 5+8=13. The 3 is given as the first digit, and the 1 is carried to the next operation. Now add 5+6+1 to get the next digit and you get 12. Again, the 2 is the digit and the 1 is carried to the next operation. If the operation is fixed at two digits long, the 1 is a carry-out from the operation which overflowed its size limit.

In this case, the carry out of the addition was allowed to be added into the next part of the instruction word. Looking at Note 68 you can see that if this were to be allowed to happen, you would add the carry value to the operation-code. Because the operation code is just another stream of bits, this would be the same as adding one to the operation code, and thus changing which operation code was being indicated.

[74] A jump instruction is one that transfers program control to a different part of memory. When a program jumps it starts fetching instructions from a different location than it would if the instruction had not been a jump.

[75] Consider that an arithmetic overflow in the right part of the instruction word has caused the carry bit that changed the ADD instruction into a jump. Because we were also told that the program was adding one to the contents of the ADD instruction to walk through data in a sequential series, it must be the case that adding 1 caused the overflow. If adding one caused the overflow, then the remaining bits in the right part of the word must be 0 because in binary numbers the number one larger than 111 is 1000 for whatever size numbers you are using. So, the resulting instruction after overflow must have been a jump instruction with an address of 0, telling the program to start executing its next instruction from location 0.