LECTURE 4

HISTORY OF COMPUTING - SOFTWARE

As I indicated in the last Lecture, in the early days of computing the control part was all done by hand. The slow desk computers were at first controlled by hand, for example multiplication was done by repeated additions, with column shifting after each digit of the multiplier. Division was similarly done by repeated subtractions. In time electric motors were applied both for power and later for more automatic control over multiplication and division. The punch card machines were controlled by plug board wiring to tell the machine where to find the information, what to do with it, and where to put the answers on the cards (or on the printed sheet of a tabulator), but some of the control might also come from the cards themselves, typically X and Y punches, (other digits could, at times, control what happened). A plug board was usually specially wired for each job to be done, and in an accounting office the wired boards were usually saved and used again each week, or month, as they were needed in the cycle of accounting.

When we came to the relay machines, after Stibitz's first Complex Number Computer, they were mainly controlled by punched paper tapes. Paper tapes are a curse when doing one-shot problems - they are messy, and gluing them to make corrections, as well as loops, is a trouble, (because, among other things, the glue tends to get into the reading fingers of the machine!). With very little internal storage in the early days the programs could not be economically stored in the machines, (though I am inclined to believe that the designers considered it).

The ENIAC was at first (1945-6) controlled by wiring as if it were a gigantic plugboard, but in time Nick Metropolis and Dick Clippenger converted it to a machine that was programmed from the ballistic tables, which were huge racks of dials into which decimal digits of the program could be set via the knobs of the decimal switches.

Internal programming became a reality when storage was reasonably available, and, while it is commonly attributed to von Neumann he was a consultant to Mauchly and Eckert and their team. According to Harry Huskey internal programming was frequently discussed by them <u>before</u> von Neumann began the consulting. The first, at all widely available discussion (after Lady Lovelace wrote and published a few programs for the proposed Babbage analytical engine), was the von Neumann Army reports that were widely circulated, but never published in any refereed place.

The early codes were <u>one address</u> mainly, meaning each in-

struction contained an instruction part and the address where the number was to be found or sent to. We also had two address codes, typically for rotating drum machines, so that the next instruction would be immediately available once the previous one was completed - the same applied to mercury delay lines, and other storage devices that were serially available. Such coding was called minimum latency coding, and you can imagine the trouble the programmer had in computing where to put the next instruction and numbers (to avoid delays and conflicts as best possible), let alone in locating programming errors (bugs). In time a program named SOAP (symbolic optimizing assembly program) was available to do this optimizing using the IBM 650 machine itself. There were also three and four address codes, but I will ignore them here.

An interesting story about SOAP is that a copy of the program, call it program A, was both loaded into the machine as a program, and processed as data. The output of this was program B. Then B was loaded into the 650 and A was run as data to produce a new B program. The difference between the two running times indicated how much the optimization of the SOAP program (by SOAP itself) produced. An early example of self-compiling as it were.

In the beginning we programmed in absolute binary, meaning that we wrote the actual address where things were in binary, and wrote the instruction part also in binary! There were two trends to escape this, octal, where you simply group the binary digits in sets of three, and hexadecimal where you take four digits at a time, and had to use A,B,C,D,E,F for the representation of other numbers beyond 9, (and you had, of course, learn the multiplication and addition tables to 15).

If, in fixing up an error, you wanted to insert some omitted instructions then you took the immediately preceding instruction and replaced it by a transfer to some empty space. There you put in the instruction you just wrote over, added the instructions you wanted to insert, and then followed by a transfer back to the main program. Thus the program soon became a sequence of jumps of the control to strange places. When, as almost always happens, there were errors in the corrections you then used the same trick again, using some other available space. As a result the control path of the program through storage soon took on the appearance of a can of spaghetti. Why not simply insert them in the run of instructions? Because then you would have to go over the entire program and change all the addresses that referred to any of the moved instructions! Anything but that!

We very soon got the idea of reusable software, as it is now called. Indeed Babbage had the idea. We wrote mathematical libraries to reuse blocks of code. But an absolute address library meant that each time the library routine was used it had to occupy the same locations in storage. When the complete library became too large we had to go to relocatable programs. The necessary programming tricks were in the von Neumann reports, which were never formally published.

The first published book devoted to programming was by Wilkes, Wheeler, and Gill and applied to the Cambridge, England EDSAC, (1951), and I, among others, learned a lot from it, as you will hear in a few minutes.

Someone got the idea that a short piece of program could be written that would read in the symbolic names of the operations (like ADD) and translate them at input time to the binary representations used inside the machine (say 01100101). This was soon followed by the idea of using symbolic addresses - a real heresy for the old time programmers. You don't now see much of the old heroic absolute programming (unless you fool with a hand held programmable computer and try to get it to do more than the designer and builder ever intended).

I once spent a full year, with the help of a lady programmer from Bell Telephone Laboratories, on one big problem coding in absolute binary for the IBM 701 which used all the 32K registers then available. After that experience I vowed that never again would I ask anyone to do such labor. Having heard about a symbolic system from Poughkeepsie, IBM, I ask her to send for it and to use it on the next problem, which she did. As I expected, she reported that it was much easier. So we told everyone about the new method, meaning about 100 people, who were also eating at the IBM cafeteria near where the machine was. About half were IBM people and half were, like us, outsiders renting time. To my knowledge only one person - yes, only one - of all the 100 showed any interest!

Finally, a more complete, and more useful, Symbolic Assembly Program (SAP) was devised - after more years than you are apt to believe during which most programmers continued their heroic absolute binary programming. At the time SAP first appeared I would guess that about 1% of the older programmers were interested in it - using SAP was "sissy stuff", and a real programmer would not stoop to wasting machine capacity to do the assembly. Yes! Programmers wanted no part of it, though when pressed they had to admit that their old methods used more machine time in locating and fixing up errors than the SAP program ever used. One of the main complaints was that by using a symbolic system you would not know where anything was in storage - though in the early days we supplied a mapping of symbolic to actual storage, and believe it or not they later lovingly pored over such sheets rather than realize that they did not need to know that information if they stuck to operating within the system - no! When correcting errors they preferred to do it in absolute binary address.

FORTRAN, meaning formula translation, was proposed by Backus and friends, and again was opposed by almost all programmers. First, it was said that it couldn't be done. Second, if it could be done, it would be too wasteful of machine time and capacity. Third, even if it did work, no respectable programmer would use it - it was only for sissies!

The use of FORTRAN, like the earlier symbolic programming, was very slow to be taken up by the professionals. And this is typical of almost all professional groups. Doctors clearly do not follow the advice they give to others, and they also have a high proportion of drug addicts. Lawyers often do not leave decent wills when they die. Almost all professionals are slow to use their own expertise for their own work. The situation is nicely summarized by the old saying, "The shoe maker's children go without shoes." Consider how in the future, when you are a great expert, you will avoid this typical error!

With FORTRAN available and running, I told my programmer to do the next problem in FORTRAN, get her errors out of it, let me test it to see that it was doing the right problem, and then she could, if she wished, rewrite the inner loop in machine language to speed things up and save machine time. As a result we were able, with about the same amount of effort on our part, to produce almost 10 times as much as the others were doing. But to them programming in FORTRAN was not for real programmers!

Physically the management of the IBM 701, at IBM Headquarters in NYC where we rented time, was terrible. It was a sheer waste of machine time (at that time $300 per hour was a lot) as well as human time. As a result I refused later to order a big machine until I had figured out how to have a monitor system - which someone else finally built for our first IBM 709, and later modified it for the IBM 7096.

Again, monitors, often called "the system" these days, like all the earlier steps I have mentioned, should be obvious to anyone who is involved in using the machines from day to day; but most users seem too busy to think or observe how bad things are and how much the computer could do to make things significantly easier and cheaper. To see the obvious it often takes an outsider, or else someone like me who is thoughtful and wonders what he is doing and why it is all necessary. Even when told, the old timers will persist in the ways they learned, probably out of pride for their past and an unwillingness to admit that there are better ways than those they were using for so long.

One way of describing what happened in the history of software is that we were going from absolute to <u>virtual machines</u>. First, we got rid of the actual code instructions, then the actual addresses, then in FORTRAN the necessity of learning a lot of the insides of these complicated machines and how they worked. We were buffering the user from the machine itself. Fairly early at Bell Telephone Laboratories we built some devices to make the tape units virtual, machine independent. When, and only when, you have a totally virtual machine will you have the ability to transfer software from one machine to another without almost endless trouble and errors.

FORTRAN was successful far beyond anyone's expectations because of the <u>psychological</u> fact that it was just what its name implied - <u>for</u>mula <u>tran</u>slation of the things one had always done

in school; it did not require learning a new set of ways of thinking.

Algol, around 1958-60, was backed by many worldwide computer organizations, including the ACM. It was an attempt by the theoreticians to greatly improve FORTRAN. But being logicians, they produced a logical, not a humane, psychological language - and of course, as you know, it failed in the long run. It was, among other things, stated in a Boolean logical form which is not comprehensible to mere mortals (and often not even to the logicians themselves!). Many other logically designed languages which were supposed to replace the pedestrian FORTRAN have come and gone, while FORTRAN (somewhat modified to be sure) remains a widely used language, indicating clearly the power of psychologically designed languages over logically designed languages.

This was the beginning of a great hope for special languages, POL's they were called, meaning problem oriented languages. There is some merit in this idea, but the great enthusiasm faded because too many problems involved more than one special field, and the languages were usually incompatible. Furthermore, in the long run, they were too costly in the learning phase for humans to master all of the various ones that they might need.

In about 1962 LISP language began. Various rumors floated around as to how actually it came about - the probable truth is something like this: John McCarthy suggested the elements of the language for theoretical purposes, the suggestion was taken up and significantly elaborated others, and when some student observed that he could write a compiler for it in LISP, using the simple trick of self compiling, all were astounded, including, apparently, McCarthy himself. But he urged the student to try, and magically almost overnight they moved from theory to a real operating LISP compiler!

Let me digress, and discuss my experiences with the IBM 650. It was a two address drum machine, and operated in fixed decimal point. I knew from my past experiences in research that floating point was necessary (von Neumann to the contrary) and I needed index registers which were not in the machine as delivered. IBM would some day supply the floating point subroutines, so they said, but that was not enough for me. I had reviewed for a Journal the EDSAC book on programming, and there in Appendix D was a peculiar program written to get a large program into a small storage. It was an interpreter. But if it was in Appendix D did they see the importance? I doubt it! Furthermore, in the second edition it was still in Appendix D apparently unrecognized by them for what it was.

This raises, as I wished to, the ugly point of when is something understood? Yes, they wrote one, and used it, but did they understand the generality of interpreters and compilers? I believe not. Similarly, when around that time a number of us realized that computers were actually symbol manipulators and not just number crunchers, we went around giving talks, and I saw

people nod their heads sagely when I said it, but I also realized that most of them did not understand. Of course you can say that Turing's original paper (1937) clearly showed that computers were symbol manipulating machines, but on carefully rereading the von Neumann reports you would not guess that the authors did - though there is one combinatorial program and a sorting routine.

History tends to be charitable in this matter. It gives credit for understanding what something means when we first do it. But there is a wise saying that "Almost everyone who opens up a new field does not really understand it the way the followers do". The evidence for this is, unfortunately, all too good. It has been said that in physics no creator of any significant thing ever understood what he had done. I never found Einstein on the special relativity theory as clear as some later commentators. And at least one friend of mine has said, behind my back, "Hamming doesn't seem to understand error correcting codes!" He is probably right; I do not understand what I invented as clearly as he does. The reason this happens so often is that the creators have to fight through so many dark difficulties, and wade through so much misunderstanding and confusion, that they cannot see the light as others can, now that the door is open and the path made easy. Please remember, the inventor often has a very limited view of what he invented, and that some others (you?) can see much more. But also remember this when you are the author of some brilliant new thing; in time the same will probably be true of you. It has been said that Newton was the last of the ancients and not the first of the moderns, though he was very significant in making our modern world.

Returning to the IBM 650 and me. I started out (1956 or so) with the following four rules for designing a language:

1. Easy to learn
2. Easy to use
3. Easy to debug (find and correct errors)
4. Easy to use subroutines

The last is something that need not bother you as in those days we made a distinction between "open" and "closed" subroutines which is hard to explain now!

You might claim that I was doing top-down programming, but I immediately wrote out the details of the inner loop to check that it could be done efficiently, (bottom-up programming) and only then did I resume my top-down, philosophical approach. Thus, while I believe in top-down programming as a good approach, I clearly recognize that also bottom-up programming is at times needed.

I made the two address, fixed point decimal machine look like a three address floating point machine - that was my goal - A op B = C. I used the ten decimal digits of the machine (it was a decimal machine so far as the user was concerned) in the form

A address Op. B address C address

xxx x xxx xxx

How was it done? Easy! I wrote out in my mind the following loop, Figure 4-1. First, we needed a current address register, CAR, and so I assigned one of the 2000 computer registers of the IBM 650 to do this duty. Then we wrote a program to do the four steps of the last Lecture. (1) Use the CAR to find where to go for the next instruction of the program you wrote (written in my language, of course). (2) Then take the instruction apart, and store the three addresses, A, B, and C, in suitable places in the 650 storage. (3) Then add a fixed constant to the operation (Op) of the instruction and go to that address. There, for each instruction, would be a subroutine that described the corresponding operation. You might think that I had, therefore only ten possible operations, but there are only four three address operations, add, subt, mult, and div, so I used the 0 instruction to mean "go to the B address and find the further details of what is wanted". Each subroutine when it was finished transferred the control to a given place in the loop. (4) we then added 1 to the contents of the CAR register, cleaned up some details, and started in again, as does the original machine in its own internal operation. Of course the transfer instructions, the 7 instructions as I recall, all put an address into the CAR and transferred to a place in the loop beyond the addition of 1 to the contents of the CAR register.

An examination of the process shows that whatever meaning you want to attach to the instructions must come from the subroutines that are written corresponding to the instruction numbers. Those subroutines define the meaning of the language. In this simple case each instruction had its own meaning independent of any other instruction, but it is clearly easy to make some instructions set switches, flags, or other bits so that some later instructions on consulting them will be interpreted in one of several different ways. Thus you see how it is that you can devise any language you want, provided you can uniquely define it in some definite manner. It goes on top of the machine's language, making the machine into any other machine you want. Of course this is exactly what Turing proved with his Universal Turing Machine, but as noted above, it was not clearly understood until we had done it a number of times.

The software system I built was placed in the storage registers 1000 to 1999. Thus any program in the synthetic language, having only 3 decimal digits could only refer to addresses 000 to 999, and could not refer to, and alter, any register in the software and thus ruin it; designed in security protection of the software system from the user.

I have gone through this in some detail since we commonly write a language above the machine language, and may write several more still higher languages, one on top of the other, until we get the kind of language we want to use in expressing our problems to the machine. If you use an interpreter at each stage, then, of course, it will be somewhat inefficient. The use

of a compiler at the top will mean that the highest language is translated into one of the lower languages once and for all, though you may still want an interpreter at some level. It also means, as in the EDSAC case, usually a great compression of programming effort and storage.

I want to point out again the difference between writing a logical and a psychological language. Unfortunately, programmers, being logically oriented, and rarely humanly oriented, tend to write and extol logical languages. Perhaps the supreme example of this is APL. Logically APL is a great language and to this day it has its ardent devotees, but it is also not fit for normal humans to use. In this language there is a game of "one liners"; one line of code is given and you are asked what it means. Even experts in the language have been known to stumble badly on some of them.

A change of a single letter in APL can completely alter the meaning, hence the language has almost no redundancy. But humans are unreliable and require redundancy; our spoken language tends to be around 60% redundant, while the written language is around 40%. You probably think that the written and spoken languages are the same, but you are wrong. To see this difference, try writing dialog and then read how it sounds. Almost no one can write dialog so that it sounds right, and when it sounds right it is still not the spoken language.

The human animal is not reliable, as I keep insisting, so that low redundancy means lots of undetected errors, while high redundancy tends to catch the errors. The spoken language goes over an acoustic channel with all its noise and must caught on the fly as it is spoken; the written language is printed, and you can pause, back scan, and do other things to uncover the author's meaning. Notice that in English more often different words have the same sounds ("there" and "their" for example) than words that have the same spelling but different sounds ("record" as a noun or a verb, and "tear" as in tear in the eye, vs. tear in a dress). Thus you should judge a language by how well it fits the human animal as he is - and remember I include how they are trained in school, or else you must be prepared to do a lot of training to handle the new type of language you are going to use. That a language is easy for the computer expert does not mean that it is necessarily easy for the non-expert, and it is likely that non-experts will do the bulk of the programming (coding if you wish) in the near future.

What is wanted in the long run, of course, is that the man with the problem does the actual writing of the code with no human interface, as we all too often have these days, between the person who knows the problem and the person who knows the programming language. This date is unfortunately too far off to do much good immediately, but I would think that by the year 2020 it would be fairly universal practice that the expert in the field of application does the actual program preparation rather than have experts in computers (and ignorant of the field of application) do the preparation.

Unfortunately, at least in my opinion, the ADA language was designed by experts, and it shows all the non-humane features that you can expect from them. It is, in my opinion, a typical Computer Science hacking job - don't try to understand what you are doing, just get it running. As a result of this poor psychological design, a private survey by me of knowledgeable people suggests that although a Government contract may specify that the programming be in ADA, probably over 90% will be done in FORTRAN, debugged, tested, and then painfully, by hand, be converted to a poor ADA program, with a high probability of errors!

The fundamentals of language are not understood to this day. Somewhere in the early 50's I took the then local natural language expert (in the public eye) to visit the IBM 701 and then to lunch, and at desert time I said, "Professor Pei, would you please discuss with us the engineering efficiencies of languages." He simply could not grasp the question and kept telling us how this particular language put the plurals in the middle of words, how that language had one feature and not another, etc. What I wanted to know was how the job of communication can be efficiently accomplished when we have the power to design the language, and when only one end of the language is humans, with all their faults, and the other is a machine with high reliability to do what it is told to do, but nothing else. I wanted to know what redundancy I should have for such languages, the density of irregular and regular verbs, the ratio of synonyms to antonyms, why we have the number of them that we do, how to compress efficiently the communication channel and still leave usable human redundancy, etc. As I said, he could not hear the question concerning the engineering efficiency of languages, and I have not noticed many studies on it since. But until we genuinely understand such things, - assuming, as seems reasonable, that the current natural languages through long evolution are reasonably suited to the job they do for humans - we will not know how to design artificial languages for human-machine communication. Hence I expect a lot of trouble until we do understand human communication via natural languages. Of course, the problem of human-machine is significantly different from human-human communication, but in which ways and how much seems to be not known nor even sought for.

Until we better understand languages of communication involving humans as they are, (or can be easily trained) then it is unlikely that many of our software problems will vanish.

Some time ago there was the prominent "fifth generation" of computers that the Japanese planned to use, along with AI, to get a better interface between the machine and the human problem solvers. Great claims were made for both the machines and the languages. The result, so far, is that the machines came out as advertised, and they are back to the drawing boards on the use of AI to aid in programming. It came out as I predicted at that time (for Los Alamos), since I did not see that the Japanese were trying to understand the basics of language in the above engineering sense. There are many things we can do to reduce "the

software problem", as it is called, but it will take some basic understanding of language as it is used to communicate understanding between humans, and between humans and machines, before we will have a really decent solution to this costly problem. It simply will not go away.

You read constantly about "engineering the production of software", both for the efficiency of production and for the reliability of the product. But you do not expect novelists to "engineer the production of novels". The question arises, "Is programming closer to novel writing than it is to classical engineering?" I suggest yes! Given the problem of getting a man into outer space both the Russians and the Americans did it pretty much the same way, all things considered, and allowing some for espionage. They were both bound by the same firm laws of physics. But give two novelists the problem of writing on "the greatness and misery of man", and you will probably get two very different novels (without saying just how to measure this). Give the same complex problem to two modern programmers and you will, I claim, get two rather different programs. Hence my belief that current programming practice is closer to novel writing than it is to engineering. The novelists are bound only by their imaginations, which is somewhat as the programmers are when they are writing software. Both activities have a large creative component, and while you would like to make programming resemble engineering, it will take a lot of time to get there - and maybe you really, in the long run, do not want to do it! Maybe it just sounds good. You will have to think about it many times in the coming years; you might as well start now and discount propaganda that you hear, as well as all the wishful thinking that goes on in the area! The software of the utility programs of computers has been done often enough, and is so limited in scope, so that it might reasonably be expected to become "engineered", but the general software preparation is not likely to be under "engineering control" for many, many years.

There are many proposals on how to improve the productivity of the individual programmer as well as groups of programmers. I have already mentioned top-down and bottom-up; there are others such a head programmer, lead programmer, proving that the program is correct in a mathematical sense, and the waterfall model of programming to name but a few. While each has some merit I have faith in only one that is almost never mentioned - think before you write the program, it might be called. Before you start, think carefully about the whole thing including what will be your acceptance test that it is right, as well as how later field maintenance will be done. Getting it right the first time is much better than fixing it up later!

One trouble with much of programming is simply that often there is not a well defined job to be done, rather the programming process itself will gradually discover what the problem is! The desire that you be given a well defined problem before you start programming often does not match reality, and hence a lot of the current proposals to "solve the programming problem" will fall to the ground if adopted rigorously.

The use of higher level languages has meant a lot of progress. One estimate of the improvement in 30 years is:

assembler:machine code	=	2:1	x2
C language:assembler	=	3:1	x6
time share:batch	=	1.5:1	x9
UNIX:monitor	=	1.5:1	x12
System QA:debugging	=	2:1	x24
Prototyping:top down	=	1.3:1	x30
C++:C	=	2:1	x60
Reuse:redo	=	1.5:1	x90

so we apparently have made a factor of about 90 in the total productivity of programmers in 30 years, (a mere 16% rate of improvement!). This is one person's guess, and it is at least plausible. But compared with the speed up of machines it is like nothing at all! People wish that humans could be similarly speeded up, but the fundamental bottleneck is the human animal as it is, and not as we wish it were.

Many studies have shown that progammers differ in productivity, from worst to best, by much more than a factor of 10. From this I long ago concluded that the best policy is to pay your good programmers very well but regularly fire the poorer ones - if you can get away with it! One way is, of course, to hire them on contract rather than as regularly employed people, but that is increasingly against the law which seems to want to guarantee that even the worst have some employment. In practice you may actually be better off to pay the worst to stay home and not get in the way of the more capable (and I am serious)!

Digital computers are now being used extensively to simulate neural nets and similar devices that are creeping into the computing field. A neural net, in case you are unfamiliar with them, can learn to get results when you give it a series of inputs and acceptable outputs, without ever saying how to produce the results. They can classify objects into classes that are reasonable, again without being told what classes are to be used or found. They learn with simple feedback that uses the information that the result computed from an input is not acceptable. In a way they represent a solution to "the programming problem" - once they are built they are really not programmed at all, but still they can solve a wide variety of problems satisfactorily. They are a coming field which I shall have to skip in this series of Lectures, but they will probably play a large part in the future of computers. In a sense they are a "hard wired" computer (it may be merely a program) to solve a wide class of problems when a few parameters are chosen and a lot of data is supplied.

Another view of neural nets is that they represent a fairly general class of stable feedback systems. You pick the kind and amount of feedback you think is appropriate, and then the neural net's feedback system converges to the desired solution. Again, it avoids a lot of detailed programming since, at least in a simulated neural net on a computer, by once writing out a very

general piece of program you then have available a broad class of problems already programmed and the programmer hardly does more than give a calling sequence.

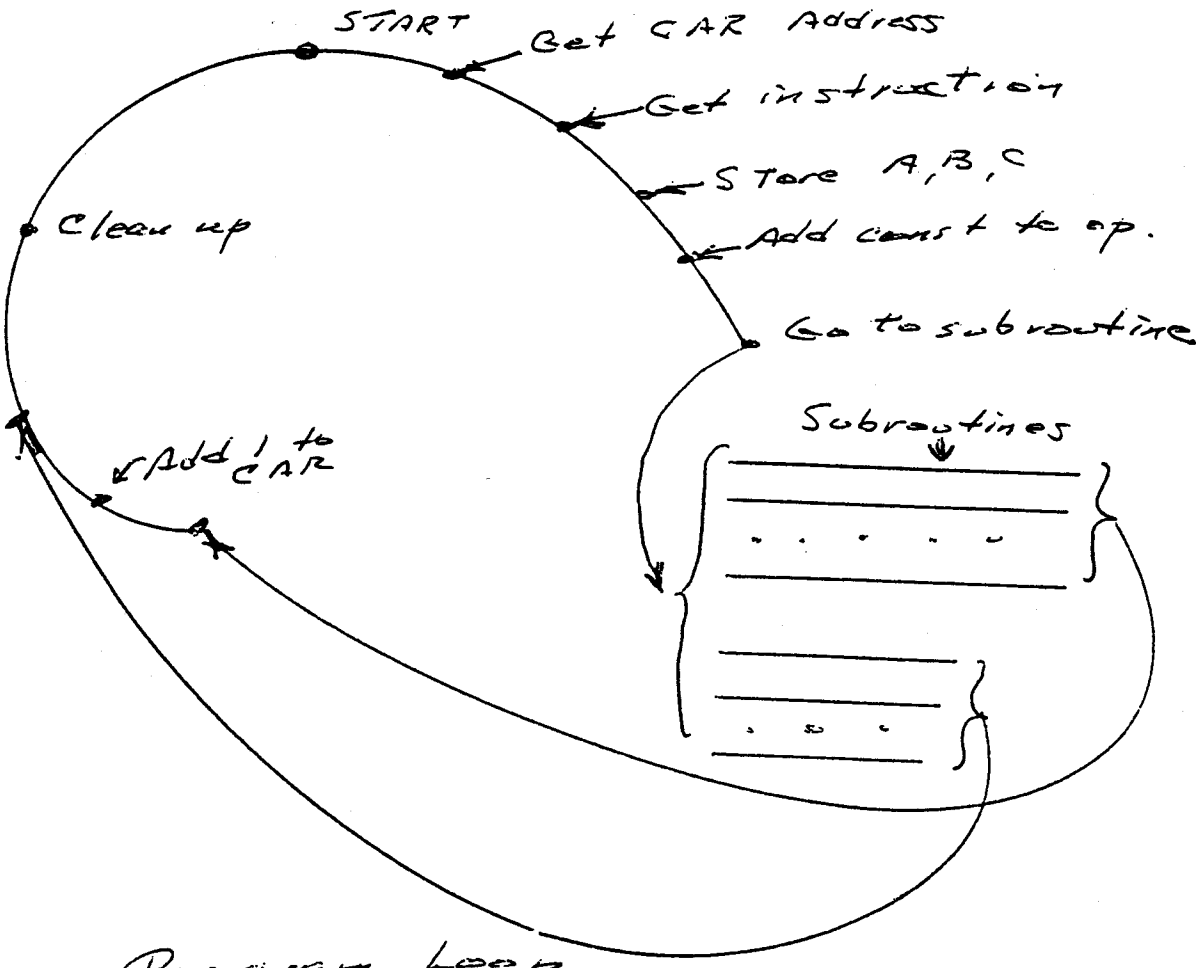
What other very general pieces of programming can be similarly done is not now known - you can think about it as one possible solution to the "programming problem".

In the Lecture on hardware I carefully discussed some of the limits - the size of molecules, the velocity of light, and the removal of heat. I should summarize correspondingly the less firm limits of software.

I made the comparison of writing software with the act of literary writing; both seem to depend fundamentally on clear thinking. Can good programming be taught? If we look at the corresponding teaching of "creative writing" courses we find that most students of such courses do not become great writers, and most great writers in the past did not take creative writing courses! Hence it is dubious that great programmers can be trained easily.

Does experience help? Do bureaucrats after years of writing reports and instructions get better? I have no real data but I suspect that with time they get worse! The habitual use of "governmentese" over the years probably seeps into their writing style and makes them worse. I suspect the same for programmers! Neither years of experience nor the number of languages used is any reason for thinking that the programmer is getting better from these experiences. An examination of books on programming suggests that most of the authors are not good programmers!

The results I picture are not nice, but all you have to oppose it is wishful thinking - I have evidence of years and years of programming on my side!



Program Loop

Figure 4-1