[*Proceedings of the National Symposium on Machine Translation*, UCLA February 1960]

Session 10: PROGRAMMING

FLEXIBILITY VERSUS SPEED

A. F. R. Brown
Georgetown University

During the description of work in progress at Georgetown, I briefly described the programming system used on French-to-English translation, which has been christened the "simulated linguistic computer". Though I would like to say a little more about it now, I did not want this paper to appear too much like that report, and so I looked around for something mildly controversial to say. Fortunately, there is at least one person prominent in machine translation who feels that a simulation system is bound to be bad, except in certain trivial cases, because it wastes the computer's time in a most shameful manner. At least one of my colleagues at Georgetown feels very much the same way; or asks why, at least, some kind of compiler could not turn the linguistic statements into a rationalized computer program in advance of run time, rather than interpret them one by one at run time. At the moment I believe that only an interpretive program can offer the flexibility needed for a good translation system, and the speed that such a method appears to lose was not really available in the first place.

"Simulated linguistic computer" may be an over-ambitious name, so I will reduce it to SLC, which can be taken to stand for anything at all. At any rate, the SLC with a 32, 000-word memory on the IBM 704 begins each cycle of translation by reading about 2, 000 words of the input text. This looks like a small batch to process with one reading of the dictionary, but it means that no tape sorting or merging has to be done, and the output from the dictionary lookup routine can be stored in the upper half of the memory for the real translation program to find and work on afterwards. The two programs, lookup and translation, alternately load themselves from the system tape once per cycle, and the dictionary tape is read once per cycle. Program loading consumes about 12 seconds, and reading the dictionary consumes about 30 seconds per 10, 000 entries. But if the system uses an IBM 709 computer, for which the program is now being checked out, the dictionary reading time covers almost all the computing done during the lookup. A minor advantage of the small batch is that the system can run with only 4 tapes: input, output, programs, and dictionary. Indeed, the programs and the dictionary could be combined on a single tape with no loss of speed.

The translation program handles the dictionary output one sentence at a time, organizing the memory in such a way that two or more of 128 "item locations" are occupied, and arranged in a ring-shaped list structure. Naturally these locations have numbers for the computer to use as their addresses, but they cannot be addressed by number in the SLC system. After all, the fact that a word is number 10 in a sentence is usually not of much interest to a linguist. And after a few rearrangements have been carried out, the linguist would be lucky if he could say with certainty what numbered position the word now occupied. The first word and the last word have important locations, and can readily be found; and the linguist can move leftward or rightward, word by word, or move directly to some word that he has previously identified structurally and flagged. So a single command, in the macro-coding system, contains at least a macro-operation code, two addresses for control transfer after execution, and an indication of what item in the sentence is to be tested or altered or moved. This indication is not in the form of a numerical address for the item, but rather an indication of whether the item previously current, or the item at which the operation began, or the item next on the right or left of either of these, is to become current. If the command requires constants of some kind for its execution, it will also include the address of the first constant, and a count of the number of constants to be used.

What may be called a "linguistic operation" is made up of a group of these commands, with some of their necessary constants. The operation is initiated by an instruction either taken from the dictionary along with the rest of a word-entry, or supplied automatically for every sentence. The most important operations are held in core memory, and the less frequently used ones can be left in the dictionary, accompanying the instructions that may initiate them. So there is practically no limit to the number of linguistic operations that can be coded into the system. Of course, the more material there is in the dictionary, the longer it takes to read, and the slower the system is, but I think dictionary reading will in fact prove to be a rather small fraction of the over-all running time for the system on the IBM 709, as it certainly is in the IBM 704 version.

In principle, then, the SLC system provides a linguist with almost complete freedom in coding linguistic operations to meet specific problems, and deciding in what order the operations should be carried out. An operation can do something as trivial as translating a two-word idiom, or as fundamental as finding the subject and predicate of a sentence. The latter function might actually have to be coded as a series of integrated operations, because there is a limit to the size of any one operation.

Now we come to the question of how much speed is sacrificed by this kind of system, compared to an equivalent system coded directly for the physical computer rather than for a hypothetical simulated computer. This is hard to answer without doing the experiment, and no-one will ever do that experiment. However, in the existing IBM 704 program, about 500 linguistic commands per second are executed; very roughly, the ratio is about 80 computer orders per linguistic command. A command can be as simple as "go to the end of the sentence" or as complicated as "go to the item next on the left of the item where this operation began, and add the following English equivalent which it already contains". On the whole, I did not think the interpretive routines do much worse than double the number of IBM 704 orders that have to be obeyed to do any particular job. When the IBM 709 program is checked out, I hope the rate of simulation will turn out to be 1, 000 or 1, 500 linguistic commands per second, because of the more powerful order code, as well as the avoidance of many wasteful techniques in the original IBM 704 program. But this will not affect the equal division of the running time between subroutines that accomplish things and interpretive routines that merely make programming easier.

Is it worthwhile, then, to force the computer to waste half its time during the translation phase of every cycle? I think it is certainly worthwhile, as long as one is in the research stage of MT, and it is at least arguable that for high-quality translation the waste of computer time cannot be avoided until much bigger core memories are available. In other words, the time may not be wasted at all. A set of interpretive routines can make it possible to code operations much more concisely, and so to fit an immensely complicated algorithm into core memory.

Consider the idioms in the French language involving the word plus. In the SLC system it is possible to code one operation, containing about 100 IBM 704 words, which handles most of them. The operation can be self-contained; there it sits on two or three pages of coding paper, instead of being scattered around in several levels of the program. Even though the operation is self-contained, it can be coded so that different parts of it are executed at different times in the translation process. Each 36-bit word in the operation is a test or an alternation of some linguistic feature in a sentence and, as such, intelligible to a linguist with no programming experience. One must admit that it saves the linguistic coder's time and effort to have the details of the IBM 704 operation handled for him by an interpretive routine that he need not think about. And as long as one is in the phase of testing linguistic formulations, it certainly saves computer time. An error in an IBM 704 program probably causes the computer to hang up, and hunting for the error is then somewhere between interesting and infuriating. An error in the SLC coding probably causes no more than a worse translation; many kinds of miscoding can be detected and bypassed, and looping can be detected and terminated automatically. But if the computer does hang up occasionally, the operator merely transfers control to a routine that dumps the offending sentence and begins normal processing of the next one. If the switch for monitoring was on, there will be a complete record of every step taken, and a post-mortem dump.

To defend the thesis that some kind of macro-programming and simulation is probably essential, let me compare an MT system with, for example, the FORTRAN system. With FORTRAN, the user can program an elaborate calculation in a convenient language. But this program will not be interpreted step by step while the calculation is being done. Instead, the FORTRAN system will read the user's program, understand it, so to speak, and convert it fairly intelligently into a computer program that is not too much less efficient than what a good programmer might have done without the aid of FORTRAN. So here we have a convenient language for writing programs, and a system for translating this language efficiently into computer instructions. Is this what we should aim for in machine translation programming? No. Sentences in a natural language cannot properly be compared with sets of data to be processed by a compiled FORTRAN program. A much better analogy would be between a sentence in a natural language and a program written in FORTRAN statements. A short program in FORTRAN is, after all, an imperative sentence; it would make a more complicated grammatical diagram than an English sentence, but the rules for diagramming FORTRAN are simpler and fewer than those needed for diagramming English sentences. If this is a plausible analogy, then an MT system with an input of source language sentences is in somewhat the same position as the FORTRAN system with an input of programs in FORTRAN language, between 20 and 30 statements in each. And we do not expect the same kind of speed from the FORTRAN compiler system that we expect from a program which has been compiled by FORTRAN. Machine translation of languages is a little like compiling FORTRAN programs into SAP symbolic programs without ever assembling and running the SAP programs.

Let me pursue another line of argument, based more particularly on the kind of general attack on translation which the SLC was designed to implement. In the SLC system, if there were no general linguistic instructions supplied automatically with every sentence, and if no entry in the dictionary contained any linguistic instructions, then a word-for-word translation would be printed out immediately following dictionary lookup. Everything after the word-for-word stage is done by operations that are initiated by linguistic instructions. The order in which these are executed, in any one sentence, is established by examining the 9-bit priority number included in every instruction. The program looks for the instruction with the lowest priority number in the sentence, interprets it, deletes it from the sentence, and then executes it. When all the instructions have been executed, they have all been deleted, and when no instructions remain in the sentence, the translation output begins.

It seems to me that this kind of procedure can be carried out only by some kind of interpretive system. Until a sentence in the source text is presented to the program, there is no way of knowing what linguistic operations will have to be done for that sentence, or in what order. So a compiler system would have to compile an efficient program for each sentence individually. This would be like tooling up a factory for mass production, making one automobile, changing the model, tooling up for mass production again, making one automobile, changing the model, and so on. An interpretive program, or a simulator, is inefficient in the same way as building automobiles by hand; but in the peculiar situation of machine translation it is probably the least inefficient method available.

A computer programmer thinks of simulation as being very wasteful of machine time. This is true for, say, simulation of an IBM 650 by an IBM 704. An "add" instruction in an IBM 650 program would at most give rise to an "add" instruction in the corresponding IBM 704 program, if the 650 program were translated into the 704 program by a human being or by a good compiler system. An interpretive routine, on the other hand, would need five or ten 704 instructions to pick up the 650 instruction, look at it to see that it says "add", see what memory location it refers to, add the content of that location to the content of the pseudo-accumulator, and store the sum in the pseudo-accumulator.

Obviously, a crude 704 program for simulating a 650 will run 5 or 10 times as slowly as a computer with the speed of the 704 and the logic of the 650. Just so, a 704 simulating an imaginary linguistic computer will run at least 20 times as slowly as the imaginary computer would if its components were in the same class as those of the 704. But this is a meaningless objection today to the simulation method. The 704 simulating the imaginary linguistic computer does take 200 microseconds to achieve what a straight 704 program might do in 100 microseconds, and it is this 2-to-l ratio that has to be justified, not the 20-to-l ratio which refers to a non-existent computer. And it can be justified. What can be packed into 10, 000 words of core storage in macro-coded operations could be recoded so that every linguistic operation ran as fast as the 704 could possibly make it run. The operations would now occupy 100, 000 or 200, 000 words on a magnetic tape, and there goes the speed. Furthermore, checking out these 100, 000 or 200, 000 words of 704 coding would be a nightmare. Or could a selection of the 10, 000 words of macro-coded operations be made at run time, and compiled into a faster 704 program? Yes, but run time begins all over again each time a new sentence in the source language is presented to the translation program; and compiling for every sentence would take many times as long as the running of the compiled programs.

The only other way of getting rid of the stigma that seems to attach to simulation and interpretive methods is to compress the translation system until it can be written as an efficient program for a real computer, and fitted into that computer's core memory. At least this means sacrificing flexibility for speed in what seems to me an intolerable way. And as long as we aim at something like the translations that human beings produce, I think it is hopeless to aim at a translation algorithm which is that simple.