Electronics Research Laboratory
University of California
Berkeley, California

THE MACHINE GENERATION OF TRANSLATORS FOR PROGRAMMING LANGUAGES

by

W. H. Wattenburg

This research was supported in part by the Air Force Office of Scientific Research under Grant AF-AFOSR 292-63.

January 13, 1964

TABLE OF CONTENTS

I. INTRODUCTION

A programming language becomes of value for the final specification of computer programs only when a translator or translators for that language are constructed. The construction of practical translators (assemblers, interpreters, compilers) is the main software concern of the computer system designer. The construction of translators for sophisticated programming languages has become a field of its own and formal procedures have been developed just as in every other area of computer system design. The object of this paper is to summarize the formal procedures for constructing translators which utilize a computer to generate substantial portions of the translators. The internal organization and logic of translators (e.g., compiling algorithms) are not the subject of this paper.

A general, but highly idealized solution to the problem of constructing translators for a variety of programming languages and machines was proposed in Ref. 1. The solution involved the use of a hypothetical universal computer-oriented language called UNCOL. The use of an UNCOL would allow all translators to be constructed so that they produced the UNCOL language as output. Hence, each new machine would require only an UNCOL to machine language translator. Using a "bootstrap" procedure all existing translators which produced UNCOL output could be implemented on the new machine as soon as an UNCOL to machine language translator were available. Although a generally accepted UNCOL has not been developed, the concept of an intermediate language has been useful. Bootstrap methods have been successfully applied in several major translator projects.

The term, "bootstrap" procedure was originally used to describe a method by which a translator is used to translate its own source language description; however, the term is now commonly used to describe any procedure which uses an existing programming language and translator to construct a new translator. The essence of any

"bootstrap" procedure is that it is an attempt to minimize the amount of programming required and the amount of human effort repeated in each new translator project. This is the objective of the methods discussed in this paper, and in that sense they can be called bootstrap procedures. Each of the procedures described has proved to be of value in the actual construction of one or more translators.

II. NOTATION

A. PROGRAMMING LANGUAGES

The capital letter L, appropriately subscripted, will be used to represent a programming language. The subscripts m and n will commonly be used to denote an executable machine language. For example, FORTRAN and COBOL might be represented by L_F and L_C; IBM 1401 machine language by L_m (m defined in the context to be the IBM 1401).

B. TRANSLATORS

Symbolic machine language assemblers, macro expanders, compilers, and a variety of other translators all perform the task of translating programs written in one language into equivalent programs written in another language. In the following paragraphs the word translator may refer to any of these various translator programs.

Three external characteristics of a translator which are necessary to identify it are:

1) the language it accepts as input
2) the language it produces as output
3) the language in which it is described (the language used to program the translator).

Item 3 is the consideration most often overlooked. A translator is just a program and, of course, it can be described in any number of

different programming languages. But it is just this fact which provides many practical ways of generating translator programs. In fact, the possible ways of constructing a given translator are so numerous that it is necessary to use a formal notation in order to adequately describe and demonstrate the construction procedures which are the subject of this paper. Attempts to describe more than the simplest of these procedures without the use of a formal notation usually result in confusion and misunderstanding.

C. TRANSLATION OPERATIONS

The symbols for programs and translators defined below are identical to those introduced in Ref. 1. Bratman later introduced a diagram[2] for describing translation operations. The operational notation defined below allows a concise description of multi-level translation operations.

Let

$$ {}_{i}P^{m}_{j} \qquad (1) $$

represent a program P, written in language L_m, which accepts inputs i and produces outputs j. P^m will represent any program written in language L_m.

Let

$$ {}_{k}T^{m}_{\ell} \qquad (2) $$

represent a translator program T, written in language L_m, which translates programs written in language L_k into equivalent programs written in language L_ℓ.

Note that ${}_{k}T^{m}_{\ell}$ could also be written as

$$ {}_{k}T^{m}_{\ell} = {}_{pk}P^{m}_{p\ell}. \qquad (3) $$

A translator ${}_{k}T^{m}_{\ell}$ is just a program written in language L_m which

accepts inputs P^k (programs written in language L_k) and produces outputs P^ℓ (equivalent programs written in language L_ℓ); hence we have Eq. (3) above.

Let

$$ {}_iP^m_j \left\{ i \right\} \rightarrow j \tag{4} $$

denote the execution of program P^m on the set of inputs i to produce the outputs j. This notation will be most useful for describing translation operations such as

$$ {}_kT^m_\ell \left\{ P^k \right\} \rightarrow P^\ell \tag{5} $$

since ${}_kT^m_\ell$ is just a program which accepts inputs P^k and produces outputs P^ℓ.

Note that the operational notation of Eqs. (4) and (5) has physical significance only when the language L_m which describes the operators ${}_iP^m_j$ and ${}_kT^m_\ell$ is an executable machine language for some existing machine; that is, when the operation can be performed by some existing machine. This condition will be true for all cases described in the next section which summarizes several useful applications of the bootstrap method of translator construction.

III. TRANSLATOR CONSTRUCTION

A. BASIC BOOTSTRAP PROCEDURE

The first translators constructed were written in the machine language of a particular machine because it was generally the only method available. These first translators were mostly symbolic machine language translators (assemblers). Once a symbolic machine language translator became available for a particular machine, most programs were written in

the symbolic language, including the translator itself. This was really the first application of the "bootstrap" procedure. The process can be summarized as follows:

Let L_1 represent the machine language of a particular machine and L_2 the symbolic machine language. The first step was to write the translator program ${}_2T_1^1$, which is a symbolic machine language to machine language translator written in machine language. The next step was to write the translator in the symbolic language, i.e., construct ${}_2T_1^2$, and then translate it into an executable program ${}_2T_1^1$ by

$$ {}_2T_1^1 \left\{ {}_2T_1^2 \right\} \longrightarrow {}_2T_1^1 . \tag{6} $$

At first glance this may seem a useless exercise, but in fact it is a very practical manuever. The translator ${}_2T_1^2$ is described in a higher level language and is generally much easier to modify and expand than is ${}_2T_1^1$. In fact, a translator ${}_3T_1^2$ is usually programmed for the second step where L_3 is an expanded language which includes L_2. It is easy to show that one application of the original translator still produces an executable program:

$$ {}_2T_1^1 \left\{ {}_3T_1^2 \right\} \longrightarrow {}_3T_1^1 . \tag{7} $$

In this case two purposes have been served: 1) the translator has been written in a more sophisticated programming language, L_2, and 2) the translator has been expanded to accept a larger language L_3 which usually includes L_2.

The above examples demonstrate the "bootstrap" procedure. It is important to note the essentials of the bootstrap process described in the previous paragraph. First, the language L_2 need only be sufficient for describing the translator ${}_3T_1^2$; it need not be sufficient for general programming. Second, the language L_3 need not include

L_2; indeed it could be entirely different. And finally, ${}_3T_1^2$ is just a program—it could just as easily have been ${}_3T_4^2$ where L_4 is some language other than our original machine language. This immediately presents a great many possibilities, the most important of which are described below.

The basic bootstrap step described by Eq. (7) can be continued with a series of translators ${}_4T_1^3$, ${}_5T_1^4, \ldots {}_nT_1^{n-1}$. If each L_n includes L_{n-1} we can be assured that the last translator can be translated into an executable program. Each step can be used to perform the next step as follows:

$$\begin{aligned} {}_2T_1^1 \quad & \left\{{}_3T_1^2\right\} \longrightarrow {}_3T_1^1 \\ {}_3T_1^1 \quad & \left\{{}_4T_1^3\right\} \longrightarrow {}_4T_1^1 \\ {}_{n-1}T_1^1 \quad & \left\{{}_nT_1^{n-1}\right\} \longrightarrow {}_nT_1^1 , \end{aligned} \tag{8}$$

or written another way:

$${}_2T_1^1 \quad \left\{{}_3T_1^2 \quad \left\{{}_{n-1}T_1^{n-2} \quad \left\{{}_nT_1^{n-1}\right\}\ldots\right\}\right\} \longrightarrow {}_nT_1^1 . \tag{8a}$$

Each of the translators ${}_kT_1^{k-1}$ can be translated into an executable program ${}_kT_1^1$ by the previous translator ${}_{k-1}T_1^1$. Notice, however, that each ${}_kT_1^{k-1}$ can also be translated by ${}_kT_1^1$ to produce itself, i.e.,

$${}_kT_1^1 \quad \left\{{}_kT_1^{k-1}\right\} \longrightarrow {}_kT_1^1 \tag{9}$$

since each language L_k included the language L_{k-1}. Stated in words: Each of these translators can translate its own description into itself since it is described by a language which is a subset of the language it translates.

Most existing symbolic machine language translators (assemblers) were constructed by the procedure summarized by Eqs. (8) and the end results have the property displayed by Eq. (9). That is, the final version of a typical symbolic assembler is a program written in the symbolic machine language that is translated into machine language by the symbolic assembler. Let us call this symbolic program (translator) ${}_3T_1^2$, that is, it translates L_3, the symbolic machine language, into L_1, the machine language, and it is described in language L_2 which is a symbolic language identical to or included in language L_3. Of course, this symbolic description of the translator itself must have been translated into an executable program by a previous translator [see Eqs. (8)] to produce an executable form of the symbolic assembler, ${}_3T_1^1$. Hence, two forms of such translators always exist, one called the source language description (${}_3T_1^3$) and another called the object language description (${}_3T_1^1$). The latter is the executable form, and the two have the property:

$$ {}_3T_1^1 \quad \left\{ {}_3T_1^3 \right\} \longrightarrow {}_3T_1^1 . \tag{10} $$

A number of translators (compilers) for sophisticated programming languages have been rapidly constructed by using this same basic bootstrap procedure and proceeding in the steps described above. The resulting compilers have become known as "self-compiling compilers,"[3,4] that is, the compilers are described (programmed) in the source languages they translate, and the two forms of each compiler obey Eq. (10).

For example, the first NELIAC compiler for the Remington Rand M460 computer was constructed by the basic bootstrap procedure.[3] The first "hand coded" translator accepted only a very limited subset of the NELIAC language. But this subset was sufficient to describe a translator for the complete language and the final M460 NELIAC compiler was programmed in NELIAC. The M460 compiler was then used to construct NELIAC compilers for numerous other machines by the methods described in the next section.

B. MULTIPLE BOOTSTRAP PROCEDURES

None of the examples of the basic bootstrap procedure described above involved translators which produced object programs for machines other than the machine which performed the translation, i. e., the end result was always a translator ${}_{L}T^{m}_{m}$ where m was a machine language. There is obviously no reason why a translator ${}_{L}T^{n}_{m}$ could not be constructed where n and m are different executable machine languages. In fact, such translators are the most valuable at our present stage of computer development where new computers and new programming languages are appearing faster than appropriate translators can be constructed. The possible uses of such "hybrid translators" are numerous but the two most common problems for which they are useful are:

1) construct a translator ${}_{A}T^{n}_{n}$ for a new machine n when a translator ${}_{B}T^{m}_{m}$ exists and the machine m is available

2) construct a translator which produces object programs for machine m but is executed by machine n. In this case the hybrid translator ${}_{L}T^{n}_{m}$ is the desired result. This situation usually comes about when the machine m is not available or not capable of performing the translation operation. It will be obvious that the solutions given below for Problem (1) embody a solution to Problem (2). Hence, only Problem (1) will be discussed.

The first problem is continually encountered. Whenever a new machine n appears there occurs the problem of constructing new translators for it. First, consider the most common variation of this problem: the language L_A is identical to L_B. The desired translator ${}_{A}T^{n}_{n}$ can be constructed in the following steps.

1) Construct a translator ${}_{A}T^{A}_{n}$, that is an L_A translator which will produce programs for the new machine n but is described (programmed) in language L_A.

2) Perform the translation

$$ {}_{A}T^{m}_{m} \qquad \left\{ {}_{A}T^{A}_{n} \right\} \longrightarrow {}_{A}T^{m}_{n} \tag{11} $$

which yields an L_A translator which runs on the existing machine m and produces executable programs for the new machine n. The first such program to be produced and executed on the new machine should rightfully be the translator itself, so...

3) Perform the translation

$$ {}_{A}T^{m}_{n} \qquad \left\{ {}_{A}T^{A}_{n} \right\} \longrightarrow {}_{A}T^{n}_{n} \tag{12} $$

which yields the desired L_A translator for the new machine n.

The obvious difficulty in this seemingly painless procedure is the construction of the translator ${}_{A}T^{A}_{n}$ in step 1. This might seem as difficult as simply programming the ${}_{A}T^{n}_{n}$ directly in machine language. Note that the required ${}_{A}T^{A}_{n}$ program is described (programmed) in language L_A, not a machine language. This fact can mean a substantial savings in programming cost. The remainder of the work (steps 2 and 3) is done by the computer m.

Further comments are in order about the bootstrap method as compared to the direct construction of ${}_{A}T^{n}_{n}$. First, there is no guarantee that the language L_A will be convenient for programming the translator ${}_{A}T^{A}_{n}$; however, if L is sufficient for general purpose programming and it is sufficiently useful to justify constructing an L_A translator, it is probably sufficient for programming ${}_{A}T^{A}_{n}$. This has proved to be true for most of the current general purpose programming languages. Second, there are several rewards provided by the bootstrap method in addition to the immediate objective ${}_{A}T^{n}_{n}$. The most important of which is that a source language (L_A) description (${}_{A}T^{A}_{n}$) of the translator ${}_{A}T^{n}_{n}$ is available for future reference. The importance of adequate documentation of logical systems is well known. Also, the translator

can be maintained (corrected, modified) and/or expanded by the basic bootstrap procedure. Finally, the ${}_{A}T^{A}_{n}$ might be modified to produce a ${}_{A}T^{A}_{p}$, where p is a new machine, much easier than writing the program ${}_{A}T^{A}_{p}$ or ${}_{A}T^{p}_{p}$ from scratch. This modification may involve changing only the command generators of ${}_{A}T^{A}_{n}$ so that it will produce command for p, i.e., become ${}_{A}T^{A}_{p}$. Hence, if we had started with the assumption that the original ${}_{A}T^{m}_{m}$ translator was the product of a bootstrap process and ${}_{A}T^{A}_{m}$ existed, we could have summarized the programming problems with the previous sentence. At any rate we are back where we started.

So far we have only considered the case of Problem (1) where L_A was identical to L_B. The bootstrap method can be useful when L_B is quite different from L_A as long as they are both sufficient for describing translators. In this case the previous solution must be modified as follows:

1) construct ${}_{B}T^{A}_{n}$ and ${}_{B}T^{B}_{n}$ (the same translator written in the two languages L_A and L_B).

Perform the translations

2) $${}_{A}T^{m}_{m} \quad \left\{ {}_{B}T^{A}_{n} \right\} \longrightarrow {}_{B}T^{m}_{n} \tag{13}$$

3) $${}_{B}T^{m}_{n} \quad \left\{ {}_{B}T^{B}_{n} \right\} \longrightarrow {}_{B}T^{n}_{n} \, . \tag{14}$$

The solution given above for the case L_A different from L_B requires the programming of two translators in step 1; however, this requires little more effort than required for one since the logic is identical for both translators. Of course, this solution degenerates to the original one when L_A is equal to L_B.

Finally, consider the case that L_A and L_B differ but have a common portion, say L_C sufficient to describe the translator for L_B.

The solution is identical to the $L_A = L_B$ case with L_C used as the describing language instead of L_A. The steps are:

1) construct ${}_BT_n^C$.

2) Perform the translation

$$ {}_AT_m^m \quad \left\{ {}_BT_n^C \right\} \longrightarrow {}_BT_n^m \quad . \tag{15} $$

3) Perform the translation

$$ {}_BT_n^m \quad \left\{ {}_BT_n^C \right\} \longrightarrow {}_BT_n^n \quad . \tag{16} $$

The translations in steps 2 and 3 can be performed since it was assumed that L_C is included in both L_A and L_B. This last case is a very practical one. It has often been the case in translator projects using the bootstrap method that a subset of the programming language was sufficient to describe the various translators.

C. APPLICATIONS

The important step in the solutions for the three cases of Problem (1) above is the programming of a hybrid translator(s) which runs on one machine and outputs the machine language of another machine. The remaining steps are performed by a machine. The practical value of these translator construction procedures can be best judged by the results. For example, among others eight operational NELIAC compilers[4, 5] for eight different machines have been constructed to date using the original M460 compiler and the steps summarized in Eqs. (15) and (16). These compilers have also been used to construct a number of compilers for other languages.[6, 7]

The concept of an intermediate language[1] between sophisticated programming languages and binary machine languages has been valuable in

several cases of translator construction for restricted classes of computers. In fact, if one looks within most any compiler, he can recognize the use of some intermediate form between the input programming language and the output binary machine language. However, the explicit use of an intermediate language and multiple bootstrap procedures have been very valuable in several translator projects. The CLIP and JOVIAL translators constructed by the System Development Corporation[8] make use of intermediate languages. A translator project recently completed by Mendicino, Storch, and Sutherland at the U. C. Lawrence Radiation Laboratory, Livermore will be described as the final example, in this section. This project was a particularly significant and complete application of both the bootstrap method and the intermediate language concept.

The translator constructed was for the FORTRAN language and the IBM 7090 and CDC 3600 computers. The initial objective was to use the existing IBM FORTRAN compiler for the IBM 7090 (which was programmed in symbolic machine language) to construct a FORTRAN compiler for the CDC 3600. For this purpose an additional statement was added to the language which allowed direct manipulation of subfields within words and BCD information. An intermediate language was defined which could easily be translated into the machine languages of the IBM 7090 and the CDC 3600.

Let us use L_F for FORTRAN, L_D for the intermediate language, L_{7090} for IBM 7090 machine language and L_{3600} for CDC 3600 machine language. Their first step was to program the translators ${}_F T_D^F$ and ${}_D T_{3600}^F$. Then using the IBM 7090 FORTRAN compiler, ${}_F T_{7090}^{7090}$, they performed the operations

$$ {}_F T_{7090}^{7090} \qquad \left\{ {}_F T_D^F \right\} \longrightarrow {}_F T_D^{7090} \tag{17} $$

and

$$ {}_F T_{7090}^{7090} \qquad \left\{ {}_D T_{3600}^F \right\} \longrightarrow {}_D T_{3600}^{7090} \,. \tag{18} $$

The two results of Eqs. (16) and (17) yielded a means of translating FORTRAN into CDC 3600 machine language using the IBM 7090. Obviously, the next step was to bootstrap the compiler over to the CDC 3600. This required two operations. First they used the result of Eq. (17) to bootstrap itself into an intermediate language form:

$$ {}_{F}T_{D}^{7090} \qquad \left\{ {}_{F}T_{D}^{F} \right\} \longrightarrow {}_{F}T_{D}^{D} \; . \tag{19} $$

Then this was translated into CDC 3600 machine language using the result of Eq. (18):

$$ {}_{D}T_{3600}^{7090} \qquad \left\{ {}_{F}T_{D}^{D} \right\} \longrightarrow {}_{F}T_{D}^{3600} \; . \tag{20} $$

The next problem was to produce an intermediate language translator which would run on the CDC 3600 and produce CDC 3600 machine language. This also required two steps:

$$ {}_{F}T_{D}^{7090} \qquad \left\{ {}_{D}T_{3600}^{F} \right\} \longrightarrow {}_{D}T_{3600}^{D} \tag{21} $$

and

$$ {}_{D}T_{3600}^{7090} \qquad \left\{ {}_{D}T_{3600}^{D} \right\} \longrightarrow {}_{D}T_{3600}^{3600} \; . \tag{22} $$

The results of Eqs. (20) and (22) together form a FORTRAN to CDC 3600 translator which runs on the CDC 3600. The resulting FORTRAN compiler for the CDC 3600 has been an operational compiler used at Livermore for the last six months. It is also worth noting that the intermediate translators [Eqs. (17) and (18)] which run on the IBM 7090 were used to compile major FORTRAN programs for the CDC 3600 before it was possible to execute the compiler [Eqs. (20) and (22)] on the CDC 3600.

One obvious reward in addition to meeting their initial objective at minimum cost (less than a man-year of effort) is that they now have a

FORTRAN translator, ${}_{F}T^{F}_{D}$, which can be easily modified to produce an appropriate output for the next new machine.

The bootstrapping operations performed in this project are similar to those proposed by the SHARE COMMITTEE[1] using their UNCOL. However, the approach taken in the above described project differs in one very practical aspect. All programming, namely ${}_{F}T^{F}_{D}$ and ${}_{D}T^{F}_{3600}$, was done in the FORTRAN language rather than in the intermediate language. This necessitated extra machine runs but saved months of programming and debugging time by not having to hand program ${}_{F}T^{D}_{D}$ and ${}_{D}T^{D}_{3600}$.

IV. CONCLUSIONS AND COMMENTS

The common characteristic of the methods summarized in this paper is that they all make use of previous programming effort (existing translators) in the construction of new translators. In this sense there are attempts to stand on the shoulders of those who have gone on before us—rather than step on their toes! The particular formal procedures described in this paper have all been used with success for the construction of operating, state-of-the-art compilers. Of course, a great number of permutations of the bootstrap method alone have been ignored because of lack of space. It is hoped that the reader will find the operational notation defined in Sec. II and used in Sec. III useful for investigating translation and/or translator construction operations in general. Indeed, it should be obvious that the process of translator generation and the operational notation described in previous sections could be generalized in several directions. However, there is one concept, which we shall call executability for want of a better word, which must be considered in any generalization that is to be valuable for explaining and predicting physical phenomena.

In the previous sections the operational notation ${}_{a}T^{m}_{b}\{\ldots\}$ has been used (and been meaningful) only when the language L_m was an

executable machine language. This is a necessary constraint if the operational notation is to be physically meaningful. ${}_aT_b^c$ as defined in Sec. II represents a program; ${}_aT_b^c\{\ldots\}$ represents a program plus machine which together perform a translation operation defined by ${}_aT_b^c$ and executed by machine c. One might dismiss the differentiation between program and machine by saying that ${}_aT_b^c$ represents a finite state machine and the means by which it accomplishes the transitions between states is of no interest. This reasoning leads to the conclusion that there is no significant difference between two translators, say, ${}_aT_b^n$ and ${}_aT_b^c$ since they are equivalent finite state machines. But in fact ${}_aT_b^n$ may be an executable program for machine n (and actually represent a finite state machine) whereas ${}_aT_b^c$ may be no more than a passive definition (in language L_c) of a finite state machine. Hence, the concept of executability is of paramount importance in considering a generalization of translator construction which will be physically meaningful.

Interesting possibilities are presented by having two forms of a translator: ${}_aT_b^n$, which represents a physical machine, and ${}_aT_b^c$, which is only a passive description of the translator in a language L_c but which can be translated into L_n by other machines. These possibilities go far beyond the highly constrained manipulations possible with existing computers and translators for computer languages. For instance, the explanation of biological evolution by interpreting genetic transformations as "bootstrap" translation operations appears promising. This subject is being pursued.

REFERENCES

1. "The problem of programming communication with changing machines," Report of the SHARE Ad-Hoc Committee on Universal Languages, Part I: Communications of ACM, No. 8, Vol. 1; August 1958; Part II: Communications of ACM, No. 9, Vol. 1; Sept. 1958.

2. H. Bratman, "An alternate form of the UNCOL diagram," Communications of the ACM, No. 3, Vol. 4; March 1961.

3. H. D. Huskey, M. H. Halstead, and R. McArthur, "NELIAC - a dialect of ALGOL," Communications of ACM, Vol. 3, No. 11, pp. 463-468; 1960.

4. K. S. Masterson, Jr., "Compilation for two computers with NELIAC," Communications of ACM, Vol. 3, No. 11, pp. 607-611; 1960.

5. M. H. Halstead, Machine-Independent Computer Programming, Washington, Spartan Books; 1962.

6. J. B. Watt, and W. H. Wattenburg, "A NELIAC generated 7090 1401 compiler," Communications of the ACM; Feb. 1962.

7. Niklaus Wirth, "A generalization of ALGOL," Communications of the ACM, Vol. 6, No. 9, see Appendix, p. 554; Sept. 1963.

8. Donald Englund, and Ellen Clark, "The CLIP translator," Communications of the Association for Computing Machinery, Vol. 6, No. 1, p. 19; Jan. 1961.