The disposition of the parentheses is computed by numbering the multiplication signs consecutively. If n is divisible by 2^k but not by 2^{k+1}, then the nth multiplication sign is preceded by k right parentheses, and followed by k left parentheses. If the last multiplication sign is numbered m, then the entire expression is surrounded by k parentheses, where $2^k > m$. The extension to negative integral exponents is obvious. The rewritten expressions are compiled in the normal manner, the equivalent subexpressions being automatically recognized.

An operational translator would require additional tests at several points to detect symbol strings not allowed by the language. Such tests are omitted here for the sake of clarity in the flow charts.

ACKNOWLEDGMENT

The author is indebted to Arthur Anger, presently at Harvard University, for many helpful criticisms and suggestions, and for coding the algorithm on the UNIVAC 1105.

REFERENCES

1. ERSHOV: *Programming Programme for the BESM Computer.* Pergamon, 1959.
2. WESGTEIN, J. H. From formulas to computer oriented language. *Comm. ACM 2* (Mar. 1959), 6–8.
3. ARDEN, B., and GRAHAM, R. On GAT and the construction of translators. *Comm. ACM 2* (July 1959), 24–26.
4. KANNER, H. An algebraic translator. *Comm. ACM 2* (Oct. 1959), 19–22.
5. SAMELSON, K., and BAUER, F. L. Sequential formula translation. *Comm. ACM 3* (Feb. 1960), 76-83.

A Syntax Directed Compiler for ALGOL 60[*]

Edgar T. Irons[†]

Princeton University, Princeton, N. J.

Although one generally thinks of a compiler as a program for a computer which translates some object language into a target language, in fact this program also serves to *define* the object language in terms of the target language. In early compilers, these two functions are fused inextricably in the machine language program which is the compiler. This fusion makes incorporation into the compiler of extensions or modifications to the object language extremely difficult.

This paper describes a compiling system which essentially separates the functions of *defining* the language and *translating* it into another. Part 1 presents the meta-language used to define the object language in terms of the target language. This meta-language is an extension of the syntax meta-language used in the ALGOL 60 report which allows specification of meaning (in terms of the target language) as well as of form. This succinct definition allows modifications to the form or meaning of the object language to be incorporated easily into the system, and in fact makes the original specification of the object language a reasonably easy task. Part 2 is a description of the program which utilizes a direct machine representation of the meta-linguistic specifications to effect a translation.

Before proceeding to a description of the meta-language we wish to demonstrate heuristically that the proposed meta-language does suffice to specify a translation for any language it can describe. If one proposes to translate language A into language B, it is necessary to have some kind of description of language A in terms of language B. More specifically, one must be able to describe the alphabet of A in B, and must have a set of rules for assigning meaning in B to various possible structures which can be formed in A by concatenating the characters of A's alphabet. The set of rules might be called the syntax of language A, if one considers definitions (in the usual sense of the word) to be merely additional rules of syntax. A translation process might then be to start with the beginning symbols of the string to be translated and to assign meaning and a new syntactic name to symbol groups as they fall into the several syntactic structures. Having thus formed a new set of syntactic elements, the next step is to modify the meanings or amplify them according to the new structures into which these syntactic elements fall. If one considers the characters of the alphabet to be syntactical units themselves, the two steps in the process are indeed indentical. Evidently the only restriction necessary to make such a description uniquely specify a language is that there be a unique syntactic structure for any possible finite string of symbols in the language.

*This report was supported, in part, by the Office of Naval Research under Contract Nonr-1858(22). Reproduction, translation, publication, use and disposal in whole or in part by or for the United States Government is permitted.

† The work leading to this report was done while the author was employed principally by Princeton University and later by the Communications Division of the Institute for Defense Analyses, Princeton.

Given that the syntactic description meets this uniqueness requirement, a translation using the description can be effected by fitting already discovered syntactic units (starting with the syntactic units which are the basic symbols of the language) into the syntactic structure to produce a new set of larger syntactic units, and assign meanings to these new units according to the meanings of the original units. This translation algorithm is then repeated on the new set of larger syntactic units.

Since several syntactic units are coalesced effectively into one each time the translation algorithm is performed, the process will converge to exactly one syntactic unit—a "program". *Also* the meaning of that unit will have been discovered. In essence, the process is to diagram the string of symbols according to syntax *and* simultaneously to keep track of the meaning associated with each structure. It therefore suffices to define language A in terms of B by listing a series of "definitions", each definition listing:

1. A string of syntactic units in A.
2. One syntactic unit of A which is the equivalent of the string.
3. The meaning (described in language B) associated with the syntactic structure or, if meaning has already been assigned to some of the syntactic units of 1, *modifications* or *amplifications* of these meanings.

When human beings translate one language to another, we would generally consult a subset of rules like these, namely a dictionary, in order to assign basic meanings to concatenated characters of the alphabet. The rules for assigning meaning (or modifying meanings) of larger syntactic units are usually kept in the head of even the amateur translator. But nevertheless, the rules must be referenced to complete any translation and, if we are to ask a computing machine to translate for us, we must be able to make these rules explicit.

The Meta-Language

We now give a system for stating explicitly a set of translation rules or specifications which can be referenced by a program on a digital computer to effect a translation.

The translation specifications consist of a series of sentences, each one consisting of a syntax formula followed by a string of symbols designating the semantics of that syntax formula. The sentences have the following form:

Let S be a syntax unit: either a metalinguistic variable or a symbol of the input language.

Let P denote a semantic unit: either a symbol of the output language or a designator of a string of such symbols.

Each sentence of the specifications then has the form:

$$\underbrace{\mathrm{S\ S\ S\ S \cdots S\ S}}_{\text{Components}} =:: \underbrace{\mathrm{S}}_{\text{Subject}} \Rightarrow \underbrace{\{\mathrm{PPPPPPPP \cdots\ P}\}}_{\text{Definition}}$$

The syntax unit S following the metasymbol $=::$ in any sentence is the "subject" of the sentence, and the syntax units to the left of $=::$ are the "components" of the sentence. The string PPPP $\cdots$ PP between the metasymbols { and } is the "definition" of the sentence. Specifically, P may have one of the following three forms:

1. Any (p) symbol of an output language. The output language alphabet may contain any symbols, but when that alphabet does contain the symbols

$$\{\mathcal{P}\ '\ \pounds\ [\![\ ]\!]\ ;\}$$

special conventions will hold in the cases described below.

2. An output string designator of the form

$$\mathcal{P}\underbrace{{}^{''\cdots'}}_{\text{may be empty}}{}_{n}\ \underbrace{[\![\mathrm{p} \leftarrow \mathrm{PP\cdots P}\ ;\ \mathrm{p} \leftarrow \mathrm{PP\cdots P}\ ;\ \cdots\ ;\ \mathrm{p} \leftarrow \mathrm{PP\cdots P}]\!]}_{\text{may be empty}}$$

where P and p are defined as above, and n is an integer designating a particular string.

If a string designator (2) is of the simple form $\mathcal{P}_n$ it denotes the string which is the *definition* of the sentence whose *subject* is the $\underline{n}$th *component* to the left of $=::$ in the sentence containing the designator $\mathcal{P}_n$.

If the string designator is of the form

$$\mathcal{P}_n\ [\![\mathrm{p} \leftarrow \mathrm{PP} \cdots \mathrm{P}\ ;\ \cdots\ ]\!]$$

it denotes the same string but with substitutions made as indicated; namely, with the symbol p replaced at every occurrence by the symbols PPPPP $\cdots$ PP, these substitutions being made one after the other from left to right. Examples are given below.

3. An output string function designator of the form

$$\pounds\underbrace{{}^{''\cdots'}}_{\text{may be empty}}{}_{n}\ \underbrace{[\![\mathrm{PPP\cdots P}\ ;\ \mathrm{PP\cdots P}\ ;\ \cdots\ ;\ \mathrm{PP\cdots P}]\!]}_{\text{may be empty}}$$

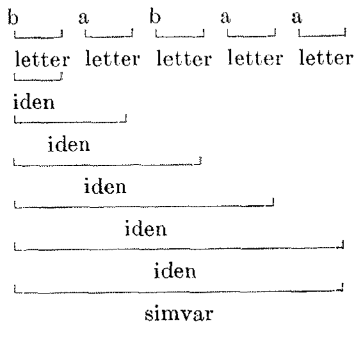
where P is defined as above, and n is an integer designating a particular string function.

The output function designator £, of (3), is used to specify a function of the strings PPPPP $\cdots$ PP enclosed between the brackets following the function designator. The integer n serves to identify a particular function which is relevant to some particular set of syntatcic sentences. They serve to enhance the descriptive ability of the output language, and constitute part of the description of an input language.

Consider as an example of the use of string designators the following five sentences specifying a translation of an input string consisting of some series containing the letters a and b to an output string composed of the letters A, B, x and y, t, m.

$$\begin{aligned}
&\mathrm{a} =:: \mathrm{letter} \Rightarrow \{\mathrm{A\ x}\}\\
&\mathrm{b} =:: \mathrm{letter} \Rightarrow \{\mathrm{B\ t}\}\\
&\mathrm{letter} =:: \mathrm{iden} \Rightarrow \{\mathcal{P}_1\}\\
&\mathrm{iden\ letter} =:: \mathrm{iden} \Rightarrow \{\mathcal{P}_2\ \mathcal{P}_1\ [\![\mathrm{t} \leftarrow \mathrm{m}]\!]\}\\
&\mathrm{iden} =:: \mathrm{simvar} \Rightarrow \{\mathcal{P}_1\ [\![\mathrm{x} \leftarrow \mathrm{y}]\!]\}
\end{aligned}$$

The unique diagram of the input string babaa is



and the meaning of the final syntactic unit "simvar" is

$$\text{BtAyBmAyAy}$$

An example of function $\pounds$ might be one whose value is a string of the characters 0 1 2 3 4 5 6 7 8 9, concatenated to represent the *number* of symbols in the parameter string of the function on any use. If we identify this function by the integer 1 and change the last syntactic statement on the previous page to

$$\text{iden} =:: \text{simvar} \Rightarrow \{\pounds_1 [\![\wp_1]\!]\}$$

the meaning of the string of our example would now be merely the characters

$$10$$

The metasymbol $'$ serves as a left metaparenthesis counter to allow the output language to contain the metasymbols of the description, so that an input language may be described in terms of—and hence translated into—the meta language. This convention enables a translator to modify the set of translation specifications it is currently using according to the particular input string it is examining. This enabling convention depends on the use of the symbols { and } as metaparentheses. If in any definition of a syntactic sentence the number of $'$ following any occurrence of $\wp$ or $\pounds$ is *not* equal to

$$(\text{the number of } \{\text{s to the left}) - 1$$

then the symbol $\wp$ or $\pounds$ and its associated symbols

$$' \; [\![\; ; \leftarrow \;]\!]$$

will be treated as symbols of type 1 rather than in the way described above. The metasymbols { and } are always treated as symbols of type 1, when they occur in the string of a definition.

If the last sentence of the descriptions of the example were changed to

$$\text{iden} =:: \text{simvar} \Rightarrow \{\wp_1 =:: \text{realtype} \Rightarrow \{\wp_1' [\![x \leftarrow y]\!] \pounds_1\}\}$$

the translation of the string babaa would be:

$$\text{BtAxBmAxAx} =:: \text{realtype } \{\text{BtAyBmAyAy}\pounds_1\}$$

The Translator

The translator is a program which operates on a set of specifications already described and on a string of symbols in the object language for those specifications. It produces a string of symbols in the output language. The translator itself is completely independent of either language, and may operate on *any* object language which can be described by the specifications of part 1.

The heart of the translator is a program which "diagrams" a string of input symbols by referencing the specifications. The output of this program is a linked list connecting various of the definitions together.

In the operation of the translator, this output list serves as input to a second program which forms the output string from the indicated sequence of definitions.

The diagramming program, given below in ALGOL, is programmed as a recursive procedure. Essentially, given the name of a syntactic unit and the index of a symbol in the input string, the program will try to form the syntactic unit from the longest possible string of symbols following the one indicated. If it is possible to form the requested syntactic unit, the program will place in its output string a linked list indicating the definitions of the syntactic units which compose the requested syntactic unit and supply as an output parameter the location in the output string of this list. If it is not possible to form the requested syntactic unit from the indicated string, the failure is reported.

In the machine implementation of this ALGOL program the translation specifications are stored in a semi-linked list, represented in the ALGOL program by the three vectors STAB, STC, and TRAN. This list is constructed in the memory of the machine from the string of symbols which are the specifications. In the ALGOL program, as in the machine representation, the syntactic units and symbols of both languages are represented by integers.

In the tabling of the translation specifications, the numerical representation of the leftmost syntactic unit of any sentence is taken as the index of the integer vector TRAN. The value of any element of TRAN is the index of the element of STAB which is the syntactic unit following the first in some sentence. If the first syntactic unit is the leftmost one of more than one sentence, the value of STC[TRAN[syntactic unit]] is the index of the syntactic unit following the first in its second occurrence. The linkages are continued from the second element in the same way. The *definition* follows the syntactic unit which is the subject of the sentence. In the tabling, the subject is treated in the same manner as the components, the fact that it *is* the subject being indicated by the brace which follows it. The symbols =:: and ⇒ are ignored. The following example illustrates the composition of the three vectors. The five sentences:

SVAR	+	SVAR	=:	TSUM	⇒	$\left\{\begin{matrix}\text{LDA } \wp_1 \\ \text{ADD } \wp_2\end{matrix}\right\}$
SVAR	+	TERM	=:	TSUM	⇒	{ ··· }
SVAR	−	SVAR	=:	TSUM	⇒	{ ··· }
SVAR	−	TERM	=:	TSUM	⇒	{ ··· }
SVAR	−	TRIM	=:	TSUM	⇒	{ ··· }

would be linked as follows:

```
SVAR    +    SVAR          TSUM          {LDA···  }
        |     ↓
        |    TERM          TSUM          {℘₁···   }
        ↓
        −    SVAR          TSUM          {        }
              ↓
             TERM          TSUM          {        }
              ↓
             TRIM          TSUM          {        }
```

Thus SVAR may be followed by + or − while + may be followed by TERM or SVAR, etc. This same information is stored in the machine representation by the three vectors STC, STAB, and TRAN. STAB consists of elements of the tree ordered by exhausting one horizontal line, then adding the next branch from the last junction passed, etc. STC lists, in the position corresponding to one element of the tree, the location of the next alternate *element* which could have been reached from the junction which led to this element. TRAN has one entry for each syntactic name. The value of the entry is the index of STC and STAB where the tree for that name begins. The composition of the vectors for the above example would be:

	TRAN		STC		STAB
SVAR	I1	I1	I4	I1	+
TERM		I2	I3	I2	SVAR
			0		TSUM
+			0		{
			0		}
−		I3	0	I3	TERM
			0		TSUM
			0		{
			0		}
		I4	0	I4	−
		I5	I6	I5	SVAR
			0		TSUM
			0		{
			0		}
		I6	I7	I6	TERM
			0		TSUM
			0		{
			0		}
		I7	0	I7	TRIM
			0		TSUM
			0		{
			0		:
			0		}

where In is the integer which is the index of the adjacent component.

In order to determine when indeed the longest string of input symbols meeting the requirements of the requested syntactic unit have been found, the diagramming routine makes use of a "precedence matrix". This matrix, SUCCR, is constructed in its elementary form while the syntax tables are constructed, and is then extended to form the complete matrix. If SUCCR [p, q] is true, then syntactic unit p is the first element of a sentence whose subject is either q or is the first element of a sentence whose subject is either q or · · · and so on.

STAB, STC, TRAN, and SUCCR are considered to be global to the procedure DIAGRAM. Other global parameters of DIAGRAM are the vectors INPUT and OUTPUT and their indices, j and k respectively, these vectors being the input string of symbols and the output string.

The output string consists of positive and negative integers. If an element is positive, it is the index of an element of STAB which begins a definition in some sentence. Each such positive element will be followed by n − 1 negative integers, these being the negative of indices of other *positive* elements of the output string which are in turn links to other definitions. n is the number of components of the sentence in question, and the jth element of OUTPUT after any positive element is the link to the definition of the jth component to the left of the subject of the sentence, whose definition begins in STAB at the spot marked by the positive element. This string of integers is then very much like an assembler macro notation, and in fact is translated to the final output string in a quite similar manner. Of course the symbol substitution and invocation of compiling functions of the definitions must be done as the integer string is unravelled.

In the operation of DIAGRAM, the parameter i marks the spot in STAB which is currently of interest. DIAGRAM first examines all the components of sentences following the component discovered one level up in the recursion, to determine whether the elements at the current location in the input string will form any of these components. If the input string meets the requirements of one of the components, the successors of this component are specified as the ones to be examined in the next call of DIAGRAM. If it does not, DIAGRAM then specifies—in order—the successors of the subjects following the component discovered on the last level of recursion as the next components to be considered, until either one of the subjects leads to a correct path, or until the list of subjects is exhausted. If the list is exhausted before a correct path is found, exit is made through ERROR to pass back the information that the path in question did not lead to success.

Since each step forward through the syntax table causes another recursion of DIAGRAM, it is not possible to eliminate a path through the table until all possible (according to the input string) paths have been examined, or until the requested syntactic unit has been formed. Hence if the paths from one structure to another in the table are unique (that is, if the syntax allows a unique diagramming of any input string) the sentences may be tabled in any order. If the paths are not unique, DIAGRAM gives perference to the first sentences in the table. Note, however, that the order of the sentences may have some effect on the efficiency of the diagramming process.

Other local parameters of DIAGRAM are SW, a Boolean variable which indicates that a syntactic unit has been discovered (though not necessarily that it encompasses the longest possible set of elements in the input string), and the constant LEFTBRACE ({), which indicates that the syntactic unit preceding it is a subject. (The metasymbol ⇒ is ignored in the tabulation.)

Implementation

The system outlined here has been instrumented at Princeton University on a Control Data Corporation 1604 computer to translate Algol 60 in its entirety to an assembly language used with the CDC-1604. The program, not including input-output transliteration but including routines to table the syntax symbols, requires about 800 machine instructions. The tables themselves require about 10000 symbols in the memory of the machine. The system operates in three phases; the first invokes DIAGRAM to output syntax descriptions pertinent to the particular Algol program (for example, to define identifiers). These

```
  procedure DIAGRAM (i, GOAL, PARAM, ERROR);
value i, GOAL;  integer PARAM;  label ERROR;  comment i is the starting position in the syntax table STAB.  GOAL is the
  requested syntax unit. If this unit is discovered the index of the appropriate definition string is placed in the output string, and the
  negative index of the output string is assigned to PARAM. If not the procedure exists via ERROR;
begin integer J, K, I, OTCEL;
boolean SW;
J := j  ;  K :=k  ;  I := i  ;
START:  if STAB [i + 1] ≠ LEFTBRACE then
          begin j := j+1  ;
                SW := if INPUT [j] = STAB [i] then true else false;
                if SUCCR [INPUT [j] , STAB [i]] then begin DIAGRAM (TRAN [INPUT [j]] , STAB [i] , OTCEL, NOGO) ;
                                                        go to CONTINUE end
                                    else begin NOGO: if SW  then go to CONTINUE
                                                             else begin RETRACE:  j = J  ;
                                                                                 k := K end end end START  ;

i := STC [i]  ;
if i = 0 then go to START  ;
i := I  ;
NEWSTART: if STAB [i] = LEFTBRACE then begin OTCEL := i+2  ;
                                       if STAB [i] = GOAL then begin PARAM := −k  ;
                                                                     SW := true end
                                                    else SW := false;
                                       if SUCCR [STAB [i] , GOAL] then begin DIAGRAM (TRAN [STAB [i]] ,
                                                                                GOAL, PARAM,
                                                                                NOPATH)  ;
                                                                           go to FOUND end
                                                         else NOPATH: if SW then go to FOUND
                                                                                 end NEWSTART  ;

i := STC [i]  ;
if i = 0 then go to NEWSTART  ;
j := J  ;
k := K  ;
go to ERROR  ;
CONTINUE: DIAGRAM (i+1, GOAL, PARAM, RETRACE)  ;
FOUND: OUTPUT [k] := OTCEL  ;
k := k+1 end
```

FIG. 1

descriptions are added to the syntax tables used for the second phase, which invokes DIAGRAM to output the assembly language program. The third phase is the assembly of the code. The compiler will output approximately 300 assembly language instructions per second on the CDC-1604. Although sufficient experimental data are not as yet available to permit definitive statements, we estimate that the compiled program is about 85 percent as efficient as a hand translation except where array references are used in the ALGOL program.

Thunks

A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations*

P. Z. Ingerman

University of Pennsylvania, Philadelphia, Pa.

Introduction

This paper presents a technique for the implementation of procedure statements, with some comments on the implementation of procedure declarations. It was felt that a solution which had both elegance and mechanizability was more desirable than a brute-force solution. It is to be explicitly understood that this solution is *one* acceptable solution to a problem soluble in many ways.

Origin of Thunk

The basic problem involved in the compilation of procedure statements and declarations is one of transmission of information. If a procedure declaration is invoked several times by several different procedure statements,

*This work was supported by the University of Pennsylvania, Office of Computer Research and Education, and the U. S. Air Force under Contract AF-49(638)-951.