RADC-TR-66-54
Final Report

AUTOMATIC PROGRAMMING TECHNIQUES

P. Gilbert
D. M. Gunn
C. L. Schager
Teledyne Systems Corporation

TECHNICAL REPORT NO. RADC-TR-66-54
July 1966

This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC (EMLI), GAFB, N.Y. 13440.

Rome Air Development Center
Research and Technology Division
Air Force Systems Command
Griffiss Air Force Base, New York

When US Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded, by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacturer, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

BLANK PAGE

AUTOMATIC PROGRAMMING TECHNIQUES

P. Gilbert
D. M. Gunn
C. L. Schager

This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC (EMLI), GAFB, N.Y. 13440.

AFLC, GAFB, N.Y., 2 Sep 66-119

FOREWORD

This final technical report was prepared by P. Gilbert, D.M. Gunn, and C.L. Schager of Teledyne Systems Corporation, 12525 South Daphne Avenue, Hawthorne, California, under contract AF 30(602)-3330, Project Number 4594. The RADC Project Engineer is William G. McClellan, EMIRD.

This document may be further distributed by any holder only with the specific prior approval of RADC (EMIRD), Griffiss AFB, New York.

This report has been reviewed and is approved.

Approved: William G McLellan
WILLIAM G. MCLELLAN
Project Engineer

Approved:
ROBERT J. QUINN, JR., Colonel, USAF
~~Chief~~, Intel and Info Processing Div.

FOR THE COMMANDER:
IRVING J. GABELMAN
Chief, Advanced Studies Group

ABSTRACT

This report is the third of a set of three reports documenting work in the area of automatic compiler generation. The first two reports described the theoretical basis for such a system. This report documents an operating system embodying the concepts described in the first two volumes. The system described in this report allows a programmer to write in FORTRAN, ALCOL, or JOVIAL and produce object code for either the DCD 1604B or the UNIVAC 1218. The system described can be expanded to incorporate other machines or languages.

BLANK PAGE

TABLE OF CONTENTS

Note: This report consists of 2 volumes; the table of contents given applies only to volume 1.

BLANK PAGE

SECTION I

OVERVIEW

A. INTRODUCTION

The Air Force has had a continuing interest in facilitating the production of compilers for procedure-oriented languages, and specifically in generation of these compilers automatically (by computer program). The reasons for such interest include desire for easy accommodation of new languages and/or new machines, as well as desire for availability of standardized forms of all procedure-oriented languages. The economic and technological benefits which may be derived if these goals are achieved are many and obvious.

We have, through successive studies culminating in this contract, characterized and then developed a comprehensive system for compiler generation, based on some novel concepts and processing techniques [1, 2, 3]. A prototype version of the system is now undergoing final checkout and refinement.

This report attempts to give a comprehensive picture of the system, its reasons, and its methods. We will stress the general methods and techniques which are used, rather than the details of program steps. The remaining portions of this action will characterize the problem of compiler generation, outline our overall solution, and note the results which have been achieved. Further sections of the report will discuss in detail the various components of the solution.

B. OBJECTIVES AND REQUIREMENTS OF COMPILER GENERATION

The notion of compiler generation has been attractive from a theoretical viewpoint for some years: there are no theoretical bars of any kind, and the very existence of compilers itself suggests the possibility of automatic generation. By 1961 much relevant work had been done [4, 5, 6, 7], and these results (and in particular the methods of Irons) have provided a foundation for present compilation techniques.

However, the achievement of automatic compiler generation entails the solution of further problems. Any method for compiler generation, if it is to work at all, must by its very nature involve the characterization of languages, and of the processing rules and mechanisms used to effect translations. To be successful, such a method should give assurance of adequacy in several respects. Adequacy questions to be dealt with concern

> Completeness: languages to be accommodated will embody differing grammatical structures, so that the grammatical notions contained in a generation schema must be adequate to properly treat a wide class of languages (as well as guarantee proper treatment for the known languages). Also, a sufficient variety of internal (to the compiler) processing mechanisms is required, as well as an adequate schema for production of output code.
>
> Uniqueness of interpretation: every algorithm expressed in a procedure-oriented language must have a unique translation in machine language, so that ambiguity of interpretation is not permissible. A proper generation schema will thus provide guarantees that the compilers which are generated cannot embody such ambiguities.

Finiteness of computation: every compiler must incorporate both a decision algorithm (as to program grammaticalness and legality) and a translation algorithm. To ensure that generated compilers will fulfill these functions, a generation schema should provide assurance that all computations (grammatical analysis, internal processing, production of output code) performed for any input program whatsoever, must terminate in a finite amount of time. That is to say, there should not exist any input program which could cause a generated compiler to "loop".

The usual practical objectives must also be considered -- namely, the size and speed of both generated compilers and compiled object programs. Some loss of efficiency in these regards could result from the use of compiler generation and such efficiency loss would be tolerable only if kept within reasonable bounds.

Finally, the concept of compiler generation itself imposes implicit requirements. If several compilers are generated for the language L, then obviously the languages accepted and processed by these compilers should be identical (else the phrase "generating compilers for L" is meaningless). The characterization of L embedded in each of the compilers should be truly machine-independent (in order to guarantee that the identity of L is maintained from compiler to compiler).

C. OVERALL METHOD OF SOLUTION

The argument above can be carried further. If the characterization of a language L embedded in each of several compilers is indeed machine-independent, then surely

> The processing algorithms within the compilers which interpret this characterization should also be machine-independent, and
>
> a logical separation should be maintained between the characterization of L and the machine language for which the compiler .s intended; that is, L should not be characterized directly in terms of the machine language.

Both of these conditions are required if the machine-independence of L is to be guaranteed. If the second of these is met, then L must be characterized in terms of a "buffer language"; that is, a machine-oriented but machine-independent computation language.

The arguments of the last two paragraphs cannot be iterated indefinitely, for if machine-independence is required at each successive step, then the longed-for machine language can never be encountered. But our most vital concern is to ensure the invariance of language structure and processing rules from one compiler of L to another, especially if L is a language having a large number of rules of formation (such as ALGOL or FORTRAN). For this reason, the use of an intermediate language is very desirable.

Further, suppose that the intermediate language which is used has very few rules of formation, so that programs in this language consist of simple sequences of "instructions", each "instruction" consisting of an "operation" followed by one or more operands and signifying a small and clearly-defined computation (e. g. ADD(X)). It is then simple to maintain the structure of this language from machine to machine, to translate from this language to a machine language in machine-independent fashion, and to achieve (excluding for the moment machine-dependent optimizations) simple sequences of machine language instructions for each intermediate language instruction. Thus for all practical purposes the intermediate language is machine-independent.

Our overall solution to the compiler generation problem can now be quickly summarized. First, a logical separation between procedure-oriented languages and machine language is maintained by the use of an intermediate language which we call BASE language (see Figure 1). Correspondingly, transformation from a procedure-oriented language L to a machine language M takes place in two steps, which we call

1. Compilation from L to BASE
2. Translation from BASE to M

These steps are shown in Figure 2.

BASE language consists of a set of operations in "macro" or "functional" notation; i. e. a BASE operation has the form OP (T_1/X_1, T_2/X_2, ..., T_n/X_n), where OP signifies the operation (e. g. ADD), T_i the i^{th} operand type (e. g. variable, data, label) and X_i the i^{th} operand itself.

Each compiler (L ⟶ BASE) corresponding to Figure 2 performs its processing via machine-independent algorithms, and is in effect generated by the use of a programming system which we call the POLYLINGUA System. Conceptually this system consists of three items:

1. Compiler Model program - a table-directed processor,

2. An abstract notation (or more precisely, a set of notations) which allows precise specification of language structure and processing rules, and

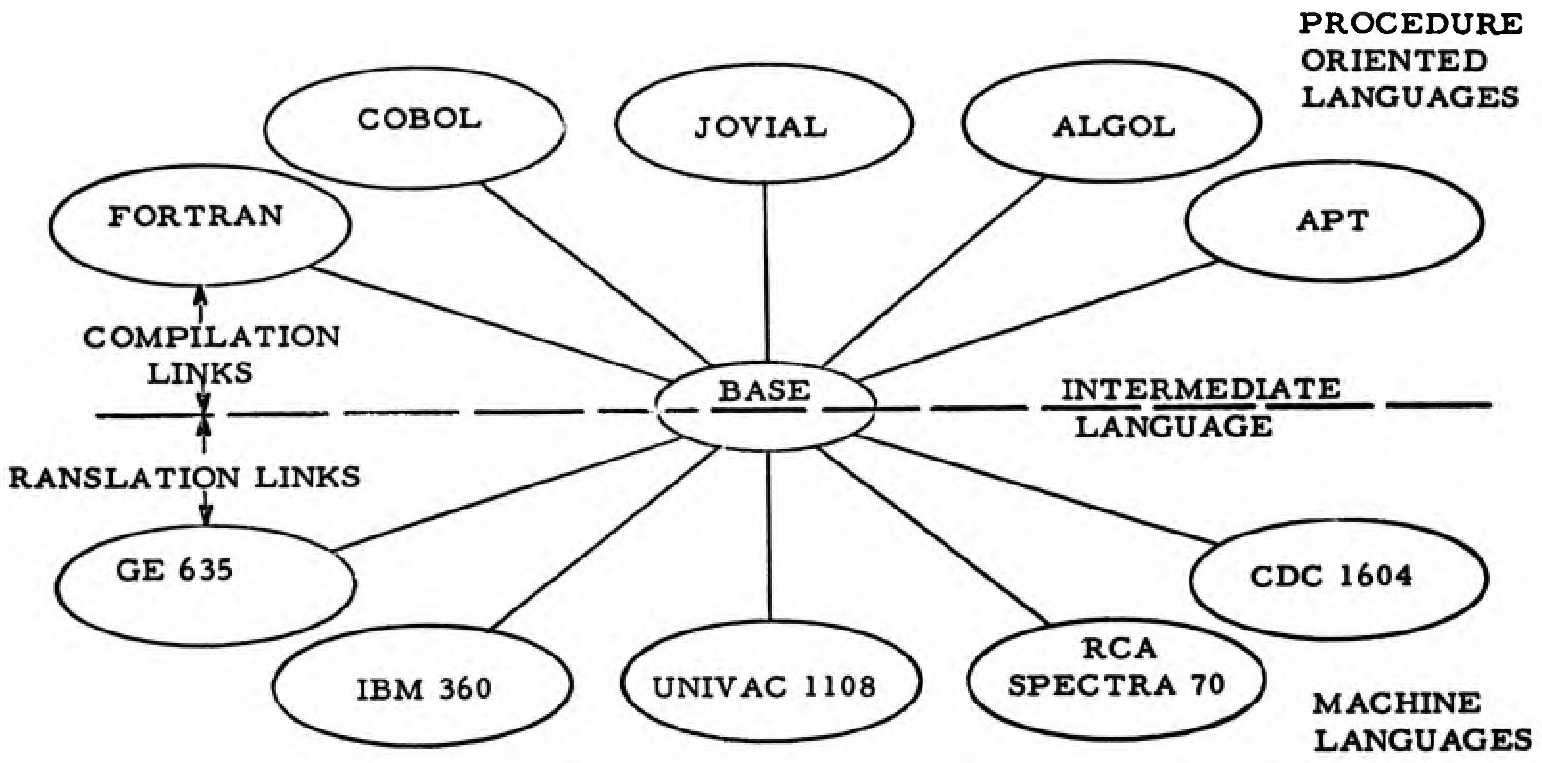


Figure 1. Logical Separation of Procedure-Oriented Languages and Machine Languages

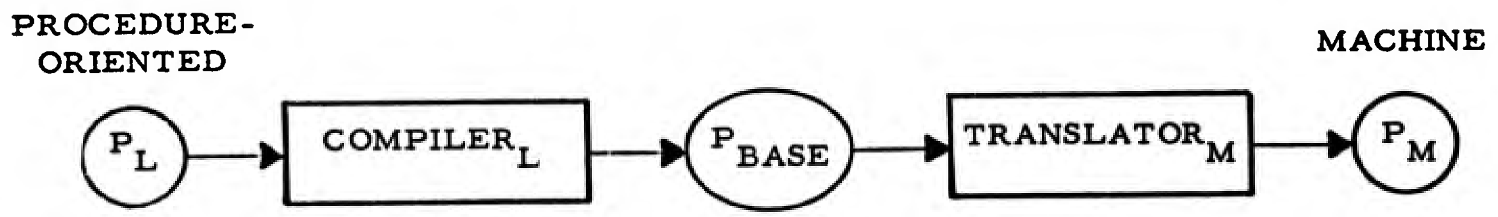


Figure 2. Compilation System Corresponding to Figure 1

3. A Generation System program, which converts abstract language specifications into data tables for use with the Compiler Model, after performance of various formal checks on the specifications.

As a first step in the production of a compiler (L→BASE), an abstract specification is written for the language L (see Section III for details), in terms of the system data base described in (IIIC), so as to enable precise reference to data entities which will be treated by the compiler. Next, the specification of L is processed through the generation system, and thus converted to a set of data tables which is called a "tape" of L. When this tape is combined with the compiler model, so as to direct the operation of that program, a compiler (L→BASE) is formed. This sequence of operations is illustrated in Figure 3.

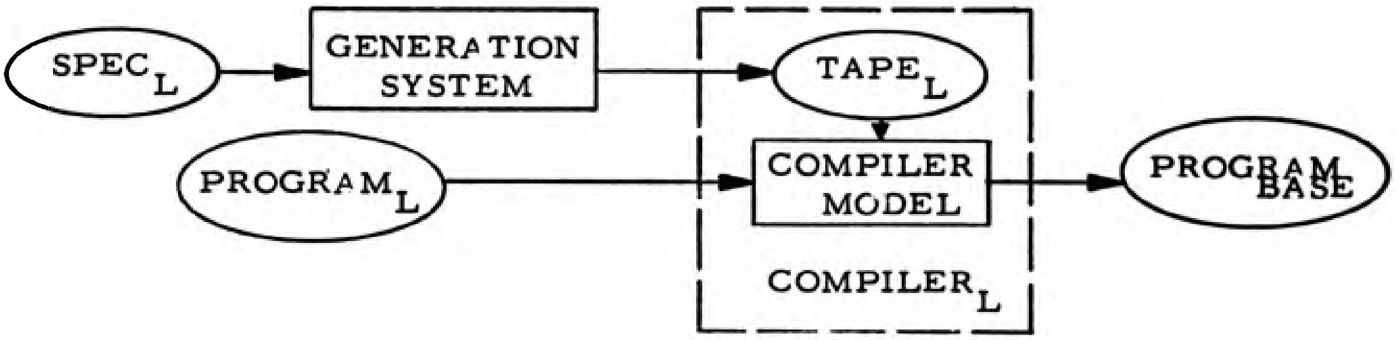


Figure 3. Sequence of Operations in Generation of a Compiler (L→BASE)

Each translator (BASE ⟶ M) is in concept machine-dependent. However, for the class of machines typified by the IBM 7094, the CDC 1604, and so on, translators are constructible which perform their processing entirely via machine-independent algorithms, machine dependencies being completely confined to the contents of certain data tables. Thus in practice, for this class of machines, it is possible to regard each translator (BASE ⟶ M) as the conjunction of a machine-independent "translator model" program and a "machine specification" consisting of a set of data tables. These does not seem to be any bar against extension of this translator construction method for more complex classes of machines.

Thus our solution conforms to the requirements we have developed for compiler generation systems. Transformation from a procedure-oriented language L to a machine language M is performed in two steps: first via a machine-independent compiler (L ⟶ BASE), which is generated by program, and then via a translator (BASE ⟶ M) which may (for a wide class of machines) be generated also. Uniqueness of interpretation is intrinsic to the syntactic analysis technique embedded in each compiler, and sufficient checks to ensure finiteness of analyses [1, 3] are made during processing of the language specification by the generation system. It appears that our technique will have excellent efficiency.

What class of procedure-oriented languages is treated by our solution? With regard to grammatical structure, we do not know of any restriction. The syntactic model upon which the system is based [3] can accommodate as a language any recursive set (i. e. any set for which "sentencehood" or "membership" is decidable); the particular variation of the model actually used within the system is at least as inclusive as (and in fact, appears to be more inclusive than) the context-sensitive ("type 1") phrase structure grammar of Chomsky [8]. Language specifications have been prepared for ALGOL, FORTRAN and JOVIAL, with no grammatical difficulty. Further, the system accommodations for compiler internal processing functions and for code production have proved quite ample for these three languages. Finally, production of translators is dependent only on the BASE language, and appears to offer no conceptual difficulty.

Thus our system, in its present form, will accommodate a very wide class of languages. Its limitations with regard to efficient treatment of "object time" code will result in part from the design of the BASE language (specifically, inclusion or exclusion of various operations). Limitations on the set of "compile time" functions performable by generated compilers will be a direct result of the set of "internal functions" operations. The current set of BASE language codes may not be optimal, and the set of "compile time" functions which may some day be necessary cannot be foretold; hence care has been taken to provide ease of expansion in both respects.

D. OBJECTIVES AND RESULTS OF THIS STUDY

The present study is the third of a series. The first of these was purely exploratory: an initial specification was made of our system for compiler generation, and basic notions were developed, including the incorporation of formal checks within a generation system to ensure properties such as finiteness of analysis.

The second study was a small effort, designed to determine more conclusively the feasibility of the method chosen by means of two projects:

1. Writing of a generation system for use on the Univac 1103A at RADC, and
2. Writing of a specification for the JOVIAL language.

The results of this study served to prove the worth of a generation system to show the adequacy of the definitional scheme and to allow development of the notations used in language specifications.

The overall objective of the present study has been to attain a final development of our system, and in doing so, to develop as completely as possible the compilation system shown in Figure 4. Specifically, the objectives of this study were

1. to write the Compiler Model program,
2. to write two translators, for the CDC 1604 and the Univac 1218, and
3. to write and test language specifications for ALGOL and FORTRAN.

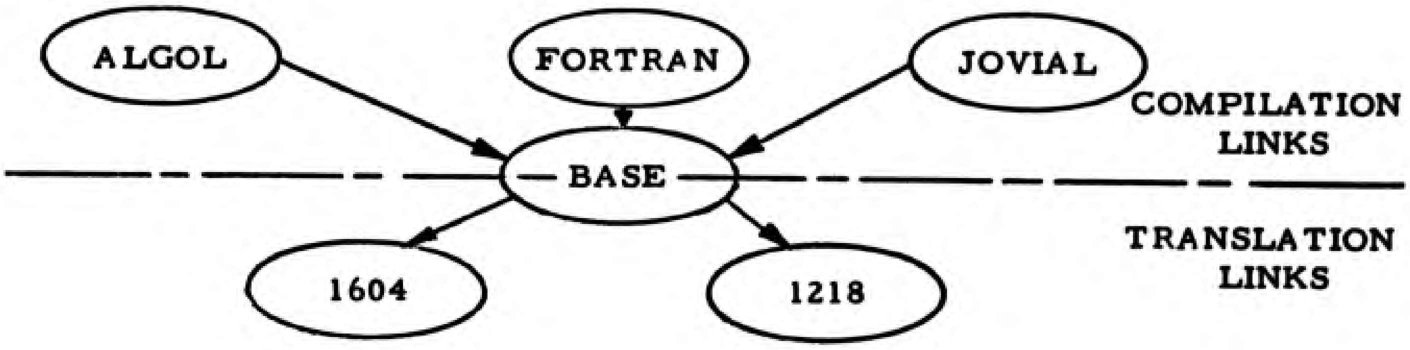


Figure 4. Capabilities of Desired System

In this effort stress was placed on the development of general methods and principles, rather than on details which would be necessary to provide production compilers. It has been felt throughout these studies that such attention to method would be repaid a thousandfold, via the resulting efficiency of the programs, in the event of success. Accordingly, considerable effort was spent in re-examination and redesign of the functions of the compiler model, resulting in a short, general, and reasonably efficient program. Extensive effort was spent in deriving the translator formulation noted previously.

As an extra effort (not called for in the study), a set of table builder programs was written to replace the generation system developed in the second study. We had hoped instead to develop a very general operation system, consisting of the compiler model and an auxiliary processor driven by a set of specifications delineating the specification notations themselves. In fact, all of the parts necessary have been written, but we have not had time to perform checkout of the merged system.

At this moment, the compiler model program, translator for the 1604, and the ALGOL specification are all simultaneously undergoing final checkout. Although these still contain "bugs", the compilation system has already compiled several programs which have been able to assemble and then execute. The principle of operation is thus proven, and it is only a matter of time until complete error-free operation is obtained.

SECTION II

THE BASE LANGUAGE

In the system of Section I-C, a logical separation between procedure-oriented languages and machine language is maintained by the use of an intermediate "macro" or "functional" language. The intermediate language is called BASE, and the transformation from a procedure-oriented language into a machine language M is said to be accomplished in two steps (cf. Figures 1 and 2, page I-6):

1. Compilation (L → BASE) aid then
2. Translation (BASE → M).

The interposition of BASE as a "buffer language" allows a mutual independence of the processes (L → BASE) and (BASE → M). Thus one translator (BASE → M) suffices for all L, and vice versa. Further, the operations of BASE can potentially be defined to have arbitrarily desired meaning, since

a. BASE is the sole link between compilation and translation,

and b. The translation process need conform only with the definitions of BASE operations and those of machine instructions.

In other words, a translator is precisely an interpretation of desired BASE specifications. The potential power and complexity of operations which might comprise BASE are proscribed only by the format in which these codes may be specified. The particular codes and interpretations given here are not as important as the use throughout of a uniform and non-restrictive notation, because of which unlimited freedom and versatility are obtainable within BASE. Evolution of BASE operation types or interpretations will always be accommodatable by appropriate changes within a BASE → M translator.

The format of BASE operations is modeled after that of a function of n variables, $F(X_1, \ldots, X_n)$. Each operation consists of a three-letter operation code followed by some number of arguments, each argument consisting of an operand type specifier and the operand itself. Typical symbolic notations (cf. Section III-C) are

$FFF(S_1/X_1)$ for a one-argument operation and

$GGG(S_1/X_1, S_2/X_2, \ldots, S_n/X_n)$ for an n-argument operation,

where S_i signifies the i^{th} operand type specifier and X_i the i^{th} operand itself.

By convetion, every operation has at least one argument (which need not have meaning), and otherwise every operation has an appropriate (possibly variable) number of arguments, as required. Within the system, an operation having more than one argument has the form of a sequence of one-argument subfunctions. The dummy operation code ARG, which serves merely to carry one argument, is used to accomplish this "decomposition", which results in the identity

$$GGG(S_1/X_1, \ldots, S_n/X_n) \equiv GGG(S_1/X_1)\,ARG(S_2/X_2) \ldots ARG(S_n/X_n)$$

The code ARG is also a permissible three-letter operation code, so that the operation GGG may be written symbolically in either of the above forms. Availability of the ARG code allows the composition of a sequence for GGG as the result of a number of processing steps. Thus the arguments for an operation may be specified all at once (using either notation above), or an appropriate number of arguments may be composed during subsequent processing. Note that a variable number of arguments is obtainable for any operation using this mechanism.

The operand types used in BASE operations are:

Operand Type	Mnemonic
Program variable name	V
Program label	V
Program data constant	D
Temporary register	T
Compiler-defined constant	C
Compiler-defined label	L
Property	P

By "program variable name", we mean a name defined within a procedure-oriented language program (e. g., in item declarations, such as XYZ in real XYZ). Labels in such programs are conceptually defined in the same way and thus are given the same operand type. A "program data constant" is a number (e. g. , 123. 45E6) or a symbolic constant (e. g. , 4H(ABCD)) which appears in a procedure-oriented language program. A standard procedure is used to treat such constants. The string of symbols comprising the constant is transmitted to the translator and also assigned an identity number (say N); thereafter, that constant may be reference in any BASE operation as the operand D/N. "Temporary registers" are of course registers within an object program which are used to store data temporarily. "Compiler-defined constants" are integers defined by the compiler, as for example, the first operand of a GTO code. "Compiler-defined labels" are labels defined for inclusion into the object program; such labels are usually implicit in the program under compilation, as for example, in an ALGOL or JOVIAL "IF statement".

The treatment of "property operands" differs from that of other operand types. When this operand type is used, the argument actually desired is a "property" of the operand given, rather than the operand itself. Specifically, the desired argument is a machine-dependent number which will exist in the object program and which is known to the translator, but cannot (because of machine-dependence) be known to the compiler. An example of such an argument is "the number of machine words corresponding to a given data structure" (e. g. , an entry of a table). The "property" operand type allows "naming" of such a parameter by the computer and subsequent substitution by the translator of the appropriate number for the given name.

A simple convention is used to interpret property operands: the operand field associated with the P type contains a numeric code indicating the property desired; and the operand for which this property is desired follows as the argument of an ARG code. Additional arguments may be necessary to further define the property desired; in general as many extra arguments as necessary are simply inserted into the string of operation codes whenever this operand type is used.

The following properties are at present provided for:

Code	Property
0	Machine address of the given item, or of the origin word of the given data structure
1	Number of machine words corresponding to the named data structure
2	Length of a dimension of the given data structure. This property requires an additional argument, specifying the dimension whose length is desired.

Typical usages and their interpretations are given below.

Usage	Interpretation
ADD (V/ X)	Add the variable X
ADD(P/ 0, V/ X)	Add the machine address or origin address of the variable X
ADD(P/ 2, V/ X, C/2)	Add the length of dimension 2 of the data structure X

The table following this discussion gives the BASE codes which are used at present, together with their interpretations and conventions. First, a short discussion of the input/output operation INO is in order, both to illustrate the general use of arguments throughout BASE and also to explain the philosophy which has been adopted with regard to input/output.

Every procedure-oriented language assumes the existence of an overall environment in which object programs are to operate. In particular, each language assumes the provision of certain specific input/output facilities to its object programs. In other words, a language-specific set of subroutines must be presumed to exist at object time to accomplish the desired input/output processing. Accommodation of a set of procedure-oriented languages thus requires a range of such subroutines of varying types, each pertinent to one or more languages.

Modern operating systems, such as the COOP Monitor, normally furnish such subroutines; and in any case the subroutines are required to be available at object time. Accordingly, the INO operation is in reality simply a subroutine call having an arbitrary number of arguments; the first argument is a numeric code corresponding to the desired subroutine, and succeeding arguments are as required by the desired subroutine. In this way any number of subroutines can be accommodated.

As a specific illustration of this method, consider the FORTRAN statement: WRITE (6, 100) A, B. This statement is normally implemented via 3 subroutines, which might be described a follows:

Arbitrary Name	Arbitrary Num. Code	
WRITE 1	4	Initialize all WRITE subroutines and data tables
WRITE 2	5	Transfer one list elements to WRITE subroutines for processing
WRITE 3	6	End WRITE processing as necessary

Aside from a trivial variation in the handling of the constants 6 and 100, the sequence of BASE operations which would appear for the above FORTRAN statement is:

Base Sequence		Corresponding CODAP Sequence		
INO (C/4)	Initialize	+	ENA	+ 6
ARG (C/6)	WRITE		ENQ	. . 100
ARG (C/100)	Routines	+	RTJ	Q8QINGOT
		-	0	0
INO (C/5)	Transfer	+	RTJ	Q8QGOTTY
ARG (V/A)	List	-	0	0
	Element	+	0	0
		-	1	A
INO (C/5)	Transfer	+	RTJ	Q8QGOTTY
ARG (V/B)	List	-	0	0
	Element	+	0	0
		-	1	B
INO (C/6)	End WRITE	-	RTJ	Q8QENGOT

The operation INO (C/4, C/6, C/100) simply specifies a call on WRITE 1, with the arguments 6 and 100, and so on. The names by which the 1604 COOP Monitor actually calls these routines are Q8QINGOT, Q8QGOTTY, and Q8QENGOT so that the BASE sequence is almost trivially translatable to the corresponding CODAP sequence also shown above.

OPERATION CODES AND CONVENTIONS

Category	Code	Operands	Significance
General	ARG	One	Dummy operation which carries one argument.
Arithmetic and Boolean Operations	CLR	None (i.e. has one operand which is ignored). Nominal form is CLR (C/0)	New computation follows (clears "accumulator")
	STO	One	Store results of computation thus far
	ADD	One	Add to computation thus far
	SUB	One	Subtract from computation thus far
	MPY	One	Multiply
	DIV	One	Divide computation thus far by indicated argument.
	DVI	One	"Integer divide" - quotient is truncated to nearest integer.
	EXP	One	Exponentiate computation thus far by indicated argument.
	CHS	None; nominally CHS(C/0)	Change sign of computation thus far
	ABS	None; nominally ABS (C/0)	Take absolute value (set sign "+")
	BOR	One	Boolean "OR"
	AND	One	Boolean "AND"
	NOT	None; nominally NOT(C/0)	Take Boolean complement of computation thus far.
Symbolic Operations and Operations on Parts of Items	EXB	Three operands; first is variable name	Extract bits from variable given by first operand. Second operand indicates starting point (from "left") of bits to be extracted; third operand indicates number of bits to be extracted.

	EXC	Three operands; first is variable name	Extract characters from variable given by first operand. Second operand indicates starting point (from "left") of characters to be extracted; third operand indicates number of characters to be extracted.
	PTB	Three operands; first is variable name	Put bits in variable given as first operand. With this exception, other operands have same significance as in EXB.
	PTC	Three operands; first is a variable name	Put characters in variable given as first operand. With this exception, other operands have same significance as in EXC.
Input/ Output	INO	Variable number as required - see discussion	Calls input/output subroutine indicated by first operand.
Transfer of Control	GTS	One	Call to subroutine indicated VPR and/or NPR operations follow, giving parameters as required by subroutine.
	GTO	First operand is an integer constant; second operand is a label	Go to instruction indicated by second operand. First operand is a code indicating the condition under which the transfer takes place:

Code Value	Conditions for Transfer
0	(unconditional)
1	< 0
2	≤ 0
3	= 0
4	≥ 0
5	> 0
6	≠ 0

	HLT	None; nominally HLT(C/0)	Halt

Data Declarations	ITM	Nine operands; first is variable name, and the rest are integers	Declares the first operand as an item. The ARG codes which carry the rest of the operands give the following information: ARG_1 ↔ item type: n = 0 unspecified 1 floating 2 fixed 3 bit string (length = 1↔Boolean) 4 Hollerith string 5 STC string 6 complex 7 double precision ARG_2 ↔ number of bits or characters (0 = word size) ARG_3 ↔ number of components ARG_4 ↔ For ARG_2, = 3, 4, or 5: 1↔length fixed, 0↔length variable For ARG_1 = 2: 1↔unsigned, 0↔signed ARG_5 ↔ For ARG_1 = 1 or 2: 0↔truncate, 1↔round ARG_6 ↔ {0↔+, 1↔-} ARG_7 ↔ For ARG_1 = 2: bits of precision For ARG_1 = 1: number of bits in exponent (0↔machine format) ARG_8 ↔ word within entry for table items (0↔not specified) ARG_9 ↔ If ARG_8 ≠ 0: bit origin within word

ITA	Two operands: first is variable name, second is integer signifying item type.	Declares the first operand as an item; type is given by the second operand. This is an abbreviation of the ITM declaration as follows: $ITA(V/X, C/n_1) \equiv ITM(V/X, C/n_1, C/0, C/1, C/0, C/0, C/0, C/0)$
TBL	Five operands	First operand is declared as a table. Succeeding ARG codes give information as follows: ARG_1 ↔ layout: 0 ↔ parallel, 1 ↔ serial ARG_2 ↔ packing density: 0 ↔ none 1 ↔ medium(machine-based) 2 ↔ dense ARG_3 ↔ Maximum number of entries ARG_4 ↔ Size definition: 0 ↔ variable, 1 ↔ rigid
FIL	Ten operands; first gives variable name	Declares first operand as a file. Succeeding ARG codes give information as follows: ARG_1 ↔ numeric code signifying device nmae ARG_2 ↔ file use: 0 ↔ unspecified 1 ↔ input 2 ↔ output 3 ↔ scratch ARG_3 ↔ recording mode: 0 ↔ binary, 1 ↔ hollerith ARG_4 ↔ number of records ARG_5 ↔ 0 ↔ number of records is variable 1 ↔ number of records is fixed ARG_6 ↔ number of bits or character/logical record ARG_7 ↔ 0 ↔ number of bits/chars is variable 1 ↔ number of bits/chars is fixed

			ARG$_8$ ↔ number of bits/characters/physical block ARG$_9$ ↔ 0 ↔ number of bits/chars is variable 1 ↔ number of bits/chars is fixed
	SEE	One	Declares operand as a reference name. This code may be used in place of second and following arguments of TBL, FIL, ITM codes, or may follow a DIM code. The attributes desired are declared identical to those of the SEE operand.
Data Structure	SEQ	Variable number, as required	All operands are variable names. The first operand is declared as naming a data structure which consists of the sequence (in the given order) of the data entities given in the following arguments. If the first argument is C/0, then no name is given, but object program space is declared to be allocated sequentially as specified by the arguments.
	OVR	Variable number, as required	Appears following SEQ declarations. Same as SEQ, except that the space allocations declared are to be overlaid over those of the preceding SEQ declaration, beginning at the same place in memory as the previous SEQ declaration.
Dimensions, Indices, Subscripts and Switches	DIM	One operand	Begins a dimension declaration for the stated operand. The dimension declaration consists of alternating LOB and UPB codes (as many as required to specify the proper number of dimensions), the ith LOB (or UPB) code followed by a sequence giving an arithmetic expression for the lower bound (or upper bound) of the ith dimension.

		The declaration is terminated by an EDI code. A SEE code may appear as an alternate form of declaration.
EDI	One operand	Declares termination of the dimension declaration of the stated variable.
INX	Variable number, as required	Declares set of index values for the variable which is the first argument. Values are given in order by succeeding arguments for the dimensions of the variable.
SCR	One operand	Declares operand as a "subscript" for the following computations.
ESC	One operand	Declares end of "subscript" connotation for indicated variable.
LOB	One operand	If operand is a constant, this is lower bound value (see DIM discussion). Otherwise the operand is ignored, and a sequence of codes follows giving an arithmetic expression for the lower bound.
UPB	One operand	Declares upper bound. Similar to LOB.
SWI	One operand	Declares operand as a "switch name", and begins switch declaration. Switch points are given by succeeding codes; see below.
SWP	One operand	If operand is a program variable (V type), then this variable is declared as a switch point for the switch in whose declaration this code appears. Otherwise, the switch point is defined by the sequence of codes following the SWP (this sequence must terminate with a GTO code).
ESW	One operand	Ends switch declaration of indicated variable.

Declaration of Constants	DTA	One operand	Declares the operand as the name of a program data constant. Succeeding SYM or STC codes each carry one character of the named constant.
	SYM	One operand	Operand is a 6 bit Hollerith character.
	STC	One operand	Operand is a character in 6 bit Standard Transmission Code.
	INI	One operand	Precedes one initial value of the stated variable. Succeeding SYM or STC codes each carry one character of the initial value.
	ENI	One operand	Ends sequence of initial values for dated operand.
Codes Pertaining to Subroutines	SBR	One operand	Declares operand or a label for the subroutine whose body follows.
	ESB	One operand	Ends the indicated subroutine body
	OSR	One operand	Declares operand as label of optional subroutine: optional subroutines are included in the code output when called:
	NPR	One operand	Declares operand as a "name parameter" of a subroutine. NPR and VPR codes carry formal parameters which appear in the declaration of a subroutine and also carry actual parameters appearing in subroutine calls.
	VPR	One operand	Declares operand as a "value parameter" of a subroutine. See discussion of NPR.
Program Structure	IDT	One operand	Declares operand as "program identity". If used, this code must begin the BASE program.
	TRM	One operand	Terminates the program.
	LBL	One operand	Declares operand as a l .bel for the operations which follow. If two or more LBL codes appear in sequence, the labels declared are all assumed to be equivalent.
	BEG	None	These two codes are used as segmentation markers to denote segmentable code blocks.
	END	None	

SBC	None	Subcomputation: The codes between any two SBC markers are denoted as a subcomputation.
DIR	None	Precedes a "direct code" sequence the characters of which are carried with the SYM codes.
EDR	None	Ends a "direct code" sequence.

SECTION III

COMPILATION TO BASE LANGUAGE

A. INTRODUCTION TO THE POLYLINGUA SYSTEM

This section will begin discussion of the POLYLINGUA system, the programming system which is used to generate compilers which translate from procedure-oriented languages to BASE language. It should be recalled that the POLYLINGUA system consists of two programs —

> a Compiler Model, which is a table-directed processor, and
>
> a Generation System which produces tables for use by the Compiler Model

— in conjunction with an abstract notation (actually a set of notations) which is used to write abstract specifications of language structure and processing rules.

The sequence of steps in the generation and use of a compiler, shown in Figure 3, begins with the writing of an abstract specification of the language under consideration, using the notations provided for that purpose. Each specification consists of portions delineating

> alphabetic symbols of the language
> syntax
> internal processing functions
> code produced
> diagnostic messages

Processing of this specification through the generation system yields a set of data tables (called a "tape"), which when combined with the compiler model, forms a (machine-independent) compiler from the language in question to BASE language. In use, the operation of the compiler is directed by the contents of the tape.

The workability of this system is due, not to specific features contained in the various programs, but to the emphasis placed throughout on consistency of means for abstract definition, consistency of means of interpretation used in the compiler model, and especially on the relation between these. Thus the abstract notations used in language specification are an integral part of the system: the compiler model in operation carries out the intent of the abstract notations; and the precise result of such notations is determined by the details of compiler model operation.

The notations of language specification refer to an assumed universe of data, called the system data base. In turn, the compiler operates on physical representations of these entities. For this reason our presentation will discuss first the system data base, then the specification notations and use of the conversion programs which serve as an interim generation system. In Section IV, operation of the compiler model will be discussed in detail.

B. SYSTEM DATA BASE

The universe of data in which the compiler model operates is called the system data base. As noted above, entities in the data base are referred to by the specification notations, and operated on by portions of the compiler model. We will indicate here the entities which comprise the data base, in sufficient detail to allow further discussion of the notations and compiler model.

The compilation process consists in concept of the syntactic analysis of a given string (the "input" or "program" string), and consequent construction of a second string (the "output" or "code" string). Both of these strings are regarded as of unbounded length, and as being contained on some suitable but unspecified physical medium. In construction of the code string, data is retrieved from the input, is processed, and appears in or determines segments of the code string.

From a syntactic viewpoint, the input string is a sequence of syntactic types or constructs, which are grouped by the analysis into larger and larger units. Thus the input string is viewed abstractly as a sequence of number pairs, one representing a syntactic type and the other being an associated reference number or datum, as required.

The code string is a simple sequence of "operations", each operation consisting of a number triplet representing respectively an operation code, an operand type, and an operand.

Also within the system data base are data entities pertinent to the compiler's internal processing — retrieval of this data, processing of the data, and construction of the code string. There are three such entities:

1. a set of "function registers" F_i, each such register assumed to contain a positive integer
2. a "property table" of positive integers P_i (j). At times in the processing, various symbol strings are "defined" to this table (i. e. entered into the table), and the jth entry in the table corresponds to the jth such string. The integers P_i (j) are called "properties" of the jth entry; P_o (j) is also called the "syntactic classification" of the jth entry; and the defined symbol string itself is called the "name" of the jth entry.
3. a set of "symbol registers" S_i, each containing some string of symbols.

The following discussion gives further details on use of these data entities.

C. SPECIFICATION NOTATIONS

We indicate here the notations used in the various portions of a language specification. Many examples of these notations are to be found in the specifications of ALGOL and FORTRAN contained in Volume II of this report. We will illustrate our discussion using a mythical language, aptly called EXAMPLE, the specification of which is shown in Figure III-1. EXAMPLE has no purpose or meaning other than illustration.

As shown in Figure III-1, a specification is in general composed of five portions, as follows:

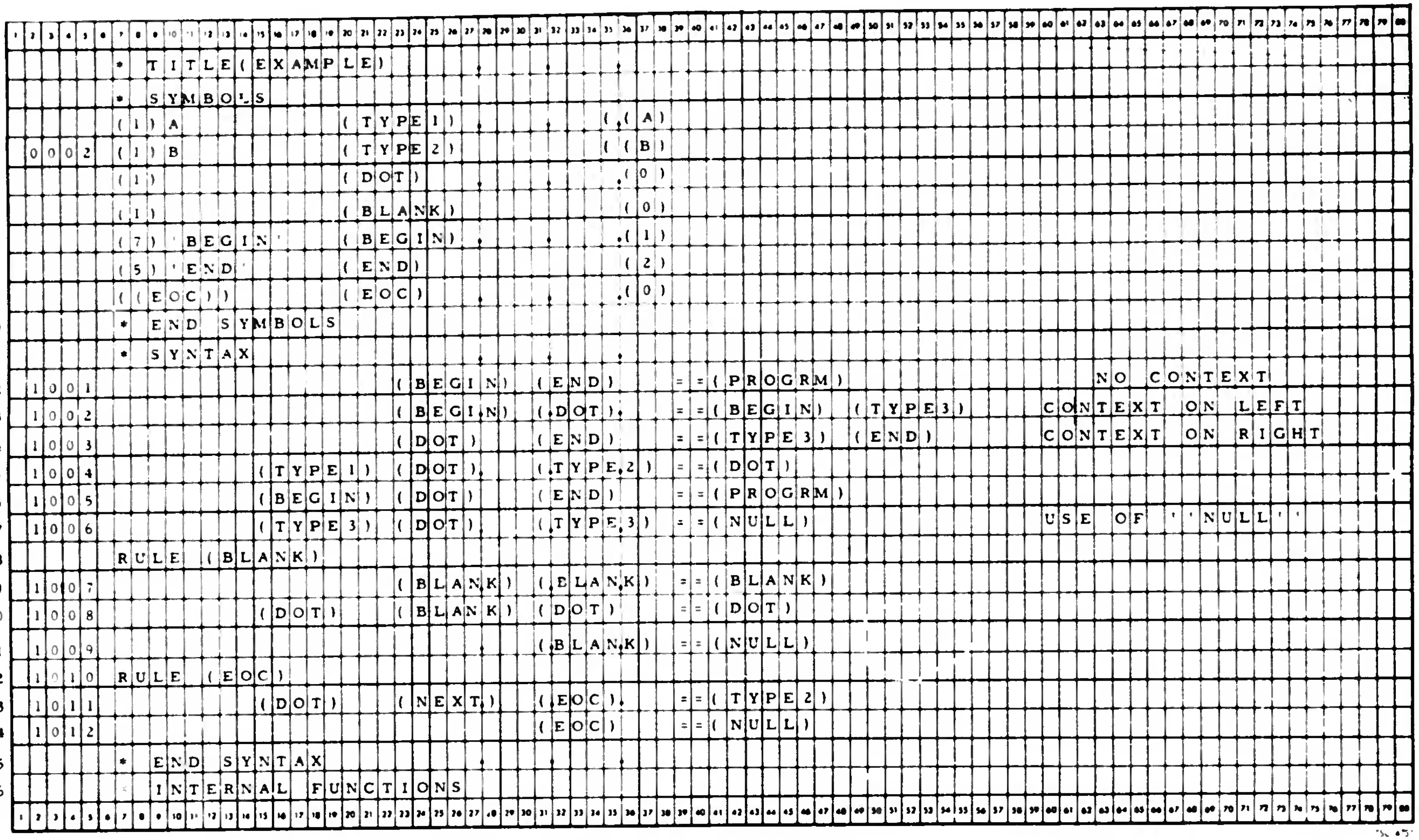
Symbols: a list of the alphabetic symbols and reserved words which constitute the primitive syntactic types of a language; for each of these both a syntactic type and a datum are specified.

Syntax: a set of syntactic productions of P_k (where k is a label affixed to the production).

Internal Functions: sets of S_k of "internal operations"; each set S_k corresponds to the production P_k having the same label affixed.

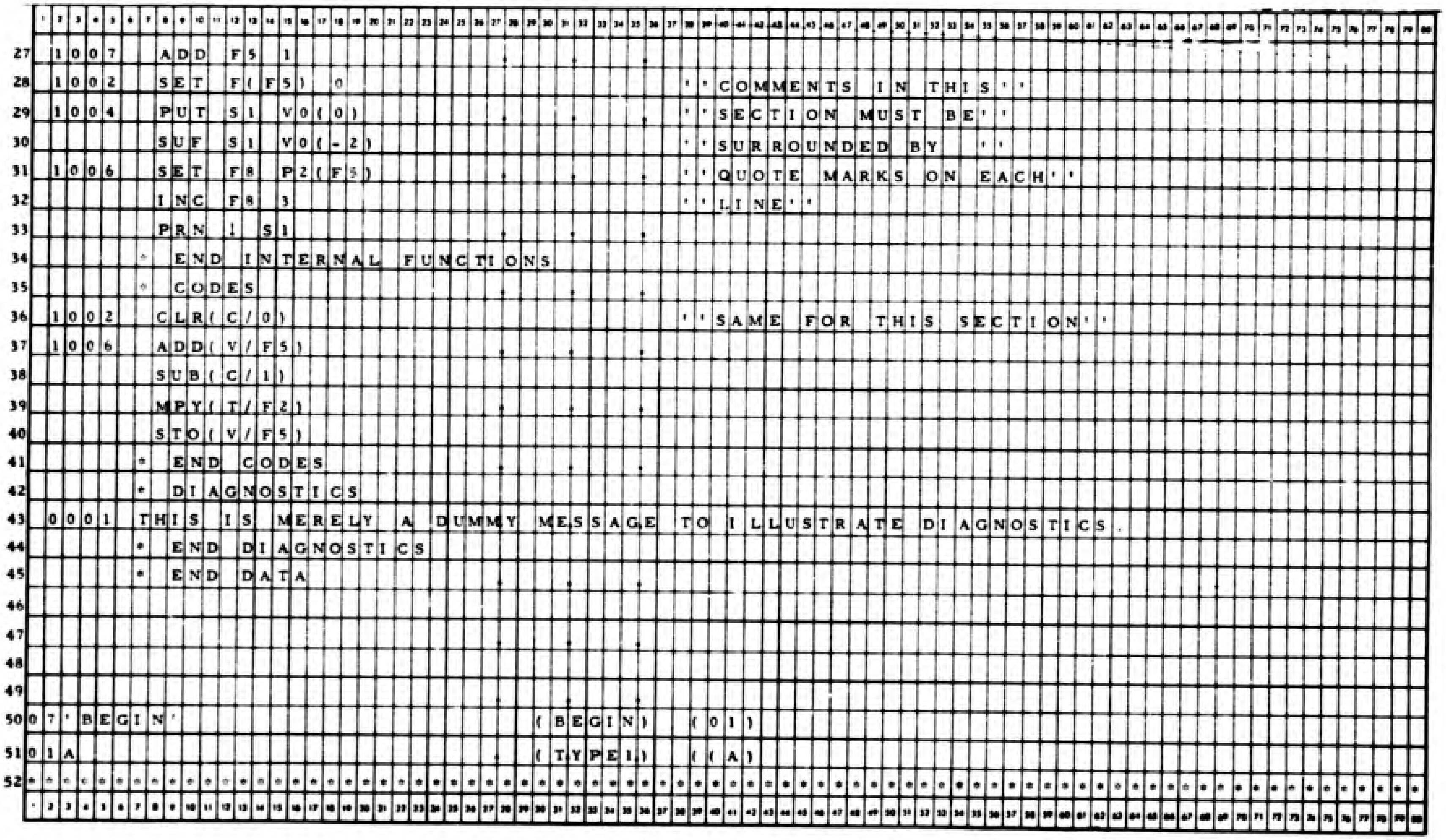
Codes: blocks B_k of "output code"; each block B_k corresponds to the production P_k having the same label affixed.

Diagnostics: a set of messages for diagnostic output.



```
      * TITLE(EXAMPLE)
      * SYMBOLS
      (1)A         (TYPE1)        ((A)
 0002 (1)B         (TYPE2)        ((B)
      (1)          (DOT)           (0)
      (1)          (BLANK)         (0)
      (7) 'BEGIN'  (BEGIN)         (1)
      (5) 'END'    (END)           (2)
      ((EOC))      (EOC)           (0)
      * END SYMBOLS
      * SYNTAX
 1001                  (BEGIN) (END)  ==(PROGRM)               NO CONTEXT
 1002                  (BEGIN) (DOT)  ==(BEGIN)  (TYPE3)    CONTEXT ON LEFT
 1003                  (DOT)   (END)  ==(TYPE3)  (END)      CONTEXT ON RIGHT
 1004         (TYPE1)  (DOT)   (TYPE2)==(DOT)
 1005         (BEGIN)  (DOT)   (END)  ==(PROGRM)
 1006         (TYPE3)  (DOT)   (TYPE3)==(NULL)              USE OF ''NULL''
      RULE (BLANK)
 1007                  (BLANK) (BLANK)==(BLANK)
 1008         (DOT)    (BLANK) (DOT)  ==(DOT)
 1009                          (BLANK)==(NULL)
 1010 RULE (EOC)
 1011         (DOT)    (NEXT)  (EOC)  ==(TYPE2)
 1012                          (EOC)  ==(NULL)
      * END SYNTAX
      * INTERNAL FUNCTIONS
```

Figure III-1. A Sample Language Specification



```
 1007  ADD F5 1
 1002  SET F(F5) 0                   ''COMMENTS IN THIS''
 1004  PUT S1 V0(0)                  ''SECTION MUST BE''
       SUF S1 V0(-2)                 ''SURROUNDED BY  ''
 1006  SET F8 P2(F5)                 ''QUOTE MARKS ON EACH''
       INC F8 3                      ''LINE''
       PRN 1 S1
      * END INTERNAL FUNCTIONS
      * CODES
 1002  CLR(C/0)                      ''SAME FOR THIS SECTION''
 1006  ADD(V/F5)
       SUB(C/1)
       MPY(T/F2)
       STO(V/F5)
      * END CODES
      * DIAGNOSTICS
 0001 THIS IS MERELY A DUMMY MESSAGE TO ILLUSTRATE DIAGNOSTICS.
      * END DIAGNOSTICS
      * END DATA




07'BEGIN'                    (BEGIN) (01)
01A                          (TYPE1) ((A)
********************************************************************************
```

Figure III-1 (Cont). A Sample Language Specification

Symbols

The figure shows a typical deck structure. The symbol portion of this specification (lines 3 through 9) contains all of the usual "symbol cards". Each symbol delineation may be preceded by a label, as in line 4, if desired; the labels have no intrinsic meaning. The number within parenthesis indicates the number of characters under consideration; these characters follow immediately after the right parenthesis. Next, the symbolic name enclosed within parenthesis is the name of a "syntactic type" or "construct". The number or character following this is the associated datum:

> ((A) signifies a datum consisting of the hollerith character "A"
> (2) signifies a datum consisting of the number "2"

Thus line 3 specifies that a symbol A appearing in an actual input program is to be represented in the conceptual input string by the syntactic type TYPE1 having an associated datum consisting of the hollerith character "A", while line 8 specifies that the symbol sequence 'END' appearing in an actual input program is to be represented in the conceptual input string by the syntactic type END having an associated datum of the number 2.

Both a syntactic type and a datum must be given for each symbol or sequence of symbols delineated within the symbol section. As shown in line 6, the symbol "blank" may be thus delineated. The symbol ((EOC)) is a special symbol which denotes a physical "end of card", this may also be delineated. All alphanumeric symbols not appearing in the symbol section are assumed to be delineated by the syntactic type NULL and a datum of 0.

Syntax

The syntax portion (which appears on lines 12 through 24 of Figure III-1) consists of a group of unsequenced productions, followed by ordered sequences of productions called "Rules". Each production must be labeled in the manner shown. Productions in the unsequenced group may not contain any construct for which a Rule exists. The productions within a Rule are tested for satisfactory match in the order in which they appear (see [1], pp. 107 and 108).

Special significance is placed on the construct names NULL and NEXT, which may be used only in the manner described here. NULL signifies the "empty" or "vacuous" construct; it may appear only on the right side of a production, as the replacement of a sequence of other constructs. Replacement by NULL is equivalent to deletion from the string.

The name NEXT may be used only on the left side of a production. It signifies "any construct in this position". Thus the sequence given in line 22 is equivalent to any of

(DOT) (DOT) (EOC)
(DOT) (TYPE 1) (EOC)
(DOT) (END) (EOC)

A production containing NEXT may appear only within a Rule.

Internal Functions

As Figure III-1 shows, the internal functions portion consists of sets of "operations", each in general having two operands. Each such operation specifies a specific manipulation of some data entity within the system data base. A labeled operation (e. g. , line 29) and all succeeding operations until the labeled operation -- constitute a set associated with the production having the same label. Thus the set of lines 29 and 30 is associated with the production labeled 1004. In principle a set of internal functions is specified for each production in the syntax section (with the intent that "performance" of the production during compiler operation causes performance of the associated set of internal functions); if no such set is specified, then the associated set is simply a null set.

The set of operations, and the data classes with which they deal, are exhibited in Figures III-2 and III-3. Figure III-2 shows the possible operand types, categorized into operand subclasses. Note that in general operands may be specified in any of three ways --

Directly

Indirectly, as the value of a specified function register (e. g. , F_{F_5})

As data associated with some syntactic construct, which construct is specified via its position in an associated syntactic production.

Figure III-3 shows the complete set of operation types. The notations used in the column "Form of Operation" in this figure have the format

OPCODE OPERAND SUBCLASS OPERAND SUBCLASS

where the operand subclasses are those of Figure III-2. The appearance of two or more operand subclasses within brackets -- e.g., $\begin{bmatrix} A \\ B \end{bmatrix}$ -- signifies that either operand subclass may be used. The operations are discussed in detail in Section IV.

A syntactic type may be used to designate a numeric value (either xx of class B, or the second operand of the STV operation). Thus, for example, the operation STV F5 ((TYPE3)) would result in the numeric value corresponding to TYPE3 being used to set F_5. A syntactic type, when used in this manner, must be enclosed within double parentheses as shown.

Operation Category	Operand Sub-class	Operand Symbolic Notation	Meaning of Notation
Numeric Operations	A	Fxx	The (xx)th function register Fxx i. e. , F_i, where $0 \leqslant i \leqslant 99$.
		F(Fxx)	F_{F_j}, where $0 \leqslant j \leqslant 99$.
		Px(Fxx)	The (x)th property of the (Fxx)th variable - the value of Fxx specifies the desired entry within the "property table". I. e. , $P_i\ (F_j)$, where $\begin{cases} 0 \leqslant i \leqslant 3 \\ 0 \leqslant j \leqslant 31 \end{cases}$
		Px(±xx)	$P_i\ (J_C)$, where $0 \leqslant i \leqslant 3$. J_C specifies the desired entry within the property table. The argument ±xx specifies the construct C via its position in the associated syntactic production, and J_C is the datum associated with C.
	B	xx	A positive integer $\leqslant 99$
		Scope	The current scope "begin number" - See detailed discussion of scope in Section IV.
		N(Sxx)	"Number conversion" of the contents of Sxx. See discussion below of Sxx, and detailed discussion of Number Conversion.
		N(S(Fxx))	Number conversion of S_{Fxx} - see comments above.

Figure III-2. Classes of Operand Types

Operation Category	Operand Sub-class	Operand Symbolic Notation	Meaning of Notation
Symbolic Operations	C	Vx(±xx)	The x^{th} transform or "value" associated with a syntactic construct. The argument ±xx specifies the construct via its position in the associated syntactic production.
	D	Sx	The $(x\ )^{th}$ symbol register Sx i.e., S_i, where $0 \leq i \leq 3$. S_i is assumed to contain a (possibly null) string of non-blank symbols, left-justified. The rest of S_i is assumed to be "blank".
		S(Fxx)	The symbol register S_{Fxx}.
Numeric and Scope Tests	E	xxx	A three-digit number
		Fxx	As above
		Px(Fxx)	As above
		Scope	The current SCOPE "begin number" -- see detailed discussion of SCOPE

Figure III-2. Classes of Operands (Cont)

Figure III-3. Internal Function Operation Types

Operation		Form of		
Category	Mnemonic	Operation	Result	Comments
Numeric Operation	SET	SET $A_1 \begin{bmatrix}A_2\\B\end{bmatrix}$	$\begin{bmatrix}A_2\\B\end{bmatrix} \rightarrow A_1$	Set F_i or $P_i(j)$ to a desired value.
	INC	INC $A_1 \begin{bmatrix}A_2\\B\end{bmatrix}$	$A_1 + \begin{bmatrix}A_2\\B\end{bmatrix} \rightarrow A_1$	Increase F_i or P_i (j) by a desired value.
	DEC	DEC $A_1 \begin{bmatrix}A_2\\B\end{bmatrix}$	$A_1 - \begin{bmatrix}A_2\\B\end{bmatrix} \rightarrow A_1$	Decrease F_i or P_i (j) by a desired value.
	MPY	MPY $A_1 \begin{bmatrix}A_2\\B\end{bmatrix}$	$A_1 \cdot \begin{bmatrix}A_2\\B\end{bmatrix} \rightarrow A_1$	Multiply F_i or P_i (j) by a desired value.
	DIV	DIV $A_1 \begin{bmatrix}A_2\\B\end{bmatrix}$	$A_1 / \begin{bmatrix}A_2\\B\end{bmatrix} \rightarrow A_1$	Divide F_i or P_i (j) by a desired value.
	EXP	EXP $A_1 \begin{bmatrix}A_2\\B\end{bmatrix}$	$A_1^{\begin{bmatrix}A_2\\B\end{bmatrix}} \rightarrow A_1$	Exponentiate F_i or P_i (j) by a desired value.
	STV	STV A xxx	xxx $\rightarrow$ A	Set F_i or P_i (j) to the value xxx which is $\leqslant$ 999
Symbolic Operations	PUT	PUT $D_1 \begin{bmatrix}C\\D_2\end{bmatrix}$	$\begin{bmatrix}C\\D_2\end{bmatrix} \rightarrow D_1$	Put the desired symbol or symbols into S_i.
	PRE	PRE $D_1 \begin{bmatrix}C\\D_2\end{bmatrix}$	$\begin{bmatrix}C\\D_2\end{bmatrix} D_1 \rightarrow D_1$	Prefix the symbols of S_i by the desired symbol or symbols

Figure III-3. Internal Function Operation Types (Cont'd)

Operation Category	Mnemonic	Form of Operation	Result	Comments
Symbolic Operations (Continued)	SUF	SUF $D_1 \begin{bmatrix}C\\D_2\end{bmatrix}$	$D_1 \begin{bmatrix}C\\D_2\end{bmatrix} \rightarrow D_1$	Suffix the symbols of S_i by the desired symbol or symbols
	RML	RML D $\begin{bmatrix}A\\B\end{bmatrix}$		Remove from the left (prefix) of S_i, or from the right (suffix) of S_i, xx symbols or the number of symbols specified in Fxx.
	RMR	RMR D $\begin{bmatrix}A\\B\end{bmatrix}$		
Scope Operations	SSI	SSI		Set SCOPE "interior"
	SSE	SSE		Set SCOPE "exterior"
Property Table Manipulation, Searches, and associations with syntactic input string	DEF	DEF D xxxx		Assign a new property table entry to the name in the given S_i. Set F_0 to the new entry number. Set P_0 (F_0) to xxxx.
	ASO	ASO ±xx $\begin{bmatrix}A\\B\end{bmatrix}$		"Associate" the value in Fxx or Px (Fxx) with the construct in position ±xx. i. e., set the construct's associated datum to the value in Fxx or Px (Fxx).

Figure III-3. Internal Function Operation Types (Cont'd)

Operation Category	Mnemonic	Form of Operation	Result	Comments
Property Table Manipulation, Searches, and associations with syntactic input string (Continued)	RTV	RTV A ±xx		"Retrieve" the datum "associated" with the construct in position ±xx, and set Fxx or Px (Fxx) to this value.
	SER	SER Sx		Search property table for entries having name in Sx.
		SER ±xx		Search property table for entries having same name as is associated with construct in position ±xx.
		SER ALL		Search all property table entries.
	DO	DO $\begin{bmatrix}A\\B\end{bmatrix}$		Do operations succeeding this and preceding next ENS operation a number of times specified in the operand.
	ENS	ENS		End search sequence. (A search sequence is the sequence of operations preceded by SER or DO and succeeded by ENS.
	IDF	IDF ±xx $\begin{bmatrix}A\\B\end{bmatrix}$		"Identify" the construct in position ±xx as specified by A. i.e., set the construct in position ±xx, to the numeric value of the given Fxx or Px (Fxx).

Figure III-3. Internal Functional Operation Types (Cont'd)

Operation Category	Mnemonic	Form of Operation	Result	Comments
Numeric and scope Tests	TST	TST $\underbrace{xx}_{n}$ $\underbrace{xx}_{n}$		A test consisting of n consecutive test operations (see below) follows this operation. If the test is "successful", the k operations following the test are performed. If the test is not successful, the k operations following the test are <u>skipped.</u>
	AGR	AGR $E_1 \begin{bmatrix}A\\B\end{bmatrix}$	Test $E_1 > E_2$?	A test consists of a sequence of those operations and/or the scope tests which follow, and no others. The "success" of a test is a boolean function of the individual tests; the first letter of the mnemonic indicates whether the individual result is to be "or"ed or "and"ed into the total result.
	OGR	OGR $E_1 \begin{bmatrix}A\\B\end{bmatrix}$		
	AEQ	AEQ $E_1 \begin{bmatrix}A\\B\end{bmatrix}$	Test $E_1 = E_2$?	
	OEQ	OEQ $E_1 \begin{bmatrix}A\\B\end{bmatrix}$		
	ANQ	ANQ $E_1 \begin{bmatrix}A\\B\end{bmatrix}$	Test $E_1 \neq E_2$?	
	ONQ	ONQ $E_1 \begin{bmatrix}A\\B\end{bmatrix}$		
	ASI	ASI $E_1 \begin{bmatrix}A\\B\end{bmatrix}$	Test if E_1 is "interior" to E_2	
	OSI	OSI $E_1 \begin{bmatrix}A\\B\end{bmatrix}$	Test if E_1 is "interior" to E_2	

Figure III-3. Internal Functional Operation Types (Cont'd)

Operation Category	Mnemonic	Form of Operation	Result	Comments
Miscellaneous	PRS	PRS $\begin{bmatrix}\text{Mess.}\\ \text{No.}\end{bmatrix}\begin{bmatrix}\text{Sx}\\ \text{Name (Fxx)}\\ \text{Name }(\pm\text{xx})\end{bmatrix}$		Print the message whose number is given, together with the specified symbol string. Set error flag to indicate that error has occurred during compilation.
	PRN	PRN $\begin{bmatrix}\text{Mess}\\ \text{No.}\end{bmatrix}\begin{bmatrix}\text{Sx}\\ \text{Name (Fxx)}\\ \text{Name }(\pm\text{xx})\end{bmatrix}$		Same as above, except no error has occurred.
	CAP	CAP $\underbrace{\text{xxxx}}_{n}$		Call auxiliary processor. The number n specifies that the nth subprocessor is desired.

Codes

Sets (or "blocks") of code are arranged in roughly the manner as the internal function sets, and with roughly the same intent -- each set is associated with the production having the same label, and the block is output upon "performance" of the associated production.

The operation codes and format are those discussed in Section II. However, the argument field does not usually contain an actual argument, but rather indicates (using the symbology of the numeric operand subclasses A and B of Figure III-2) the numeric value which is to be used as the argument. Thus MPY(T/F2) specifies a multiply operation on a temporary register, and the number which is output as the actual argument is to be taken from F2. As an exception to the nomenclature of subclasses A and B, the symbology R(±xx) is used to signify the datum associated with the construct in position ±xx.

A number of operation "pseudo-codes" are used, which do not signify BASE language operations, but rather have special meanings. The code ARG is a dummy operation code used only to carry an argument. For example, the sequence

$$\begin{array}{l} XXX(V/A_1) \\ ARG(V/A_2) \end{array}$$

is precisely equivalent to $XXX(V/A_1, V/A_2)$. The code MAC is used to designate a macro block -- a block which has the macro name as label. Thus if

MAC (BLOCK1)	and	(BLOCK1)	$ADD(V/A_1)$
			$SUB(V/A_2)$
			$MPY(T/A_3)$

are given in a code specification, the three-operation sequence is output when the MAC code is encountered. MAC codes may not occur within macro blocks.

The codes CON and VAL are used to provide conditional code output. The code CON designates a "condition value", the argument field specifying the source of this value. As shown below,

CON(C/F5)
VAL(C/A_1)
•
•   Block performed if condition satisfied (i. e., $F_5 \leqslant A_1$)
•
VAL(C/A_2)
•
•   Block performed if condition satisfied (i. e., $F_5 \leqslant A_2$)
•
VAL(C/A_3)
End of Block

a CON operation is succeeded by a sequence of subblocks, each such subblock beginning with a VAL operation. The entire sequence shown above is called a " conditional block". If the "condition value" is $\leqslant A_1$, which is given in the first VAL operation, then the block following that VAL operation is output, and no other block within the conditional block is output. If the condition is not satisfied, then the next VAL operation is tested, and so on. The constants A_i are all $A_i \leqslant 7$, and it is required that the condition value be satisfied. If a VAL operation appears as the last operation in a code block, then no code is output. For a sequence of two VAL operations, e.g.,

```
VAL(C/3)
VAL(C/4)
```

no code is output if the first of these satisfies the conditions. A conditional block may not be embedded within another conditional block, and obviously, if a conditional block is part of a larger block it may occur only at the end of that larger block.

Examples of these features occur in the ALGOL and FORTRAN specifications.

Diagnostics

The cards specifying diagnostic messages are of fixed format. The message numbers appearing in columns 2 through 5 are used in the PRN and PRS internal function operations to refer to the messages. These numbers indicate position of the messages within a print table, and should have the smallest values possible. The message portion of the card consists of columns 7 through 78.

Table Builder Formats

The example of Figure III-1 is written using the "reference" notation which will be implemented via the generation system. The language accepted by the table builder now in operation differs from this in two simple respects:

1. A control card of the form shown in line 54 is used in place of each sequence of control cards shown.

2. Symbol cards are of fixed field format; lines 52 and 53 show the fixed field equivalent of lines 7 and 3 respectively. The left parenthesis of the syntactic type must occur in column 30; the left parenthesis of the datum must occur in column 39. A numeric datum must have two digits.

D. OPERATION OF THE POLYLINGUA SYSTEM

Current operation is illustrated in Figure 3-4. The Polylingua system operates under the COOP MONITOR as three separate jobs.

The first job consists of the table builders (BLDA, BLDB) which process a language specification deck on logical unit 9, and output a binary specification tape (SPECTAPE) on logical unit 8, as well as a listing of the input with interspersed error messages on logical unit 6.

The second job is the compiler model (CM1) which processes a source program from (INTAP) logical unit 5 in a manner specified by the language specification of (SPECTAPE) logical unit 8. Output is BASE language on (CODTAP) logical unit 7 and a listing of the input on (LOGTAP) logical unit 6 with interspersed error messages. Optional output is a syntax trace and listing of BASE language on (LOGTAP). The optional output is obtained by setting jump switch 1 ON during execution of the job.

The third job consists of one of the translators (TO4 or T18). Its input is CODTAP on logical unit 7 and output is symbolic machine code CODAP1 or TRIM III on logical unit 2. After execution of TO4, CODAP1 is called on with input from logical unit 2 and output relocatable binary on unit 56. After execution of T18 the symbolic tape is taken to the UNIVAC computer and processed by TRIM III.

E. OPERATION OF THE JOVIAL COMPILER

1. Operation

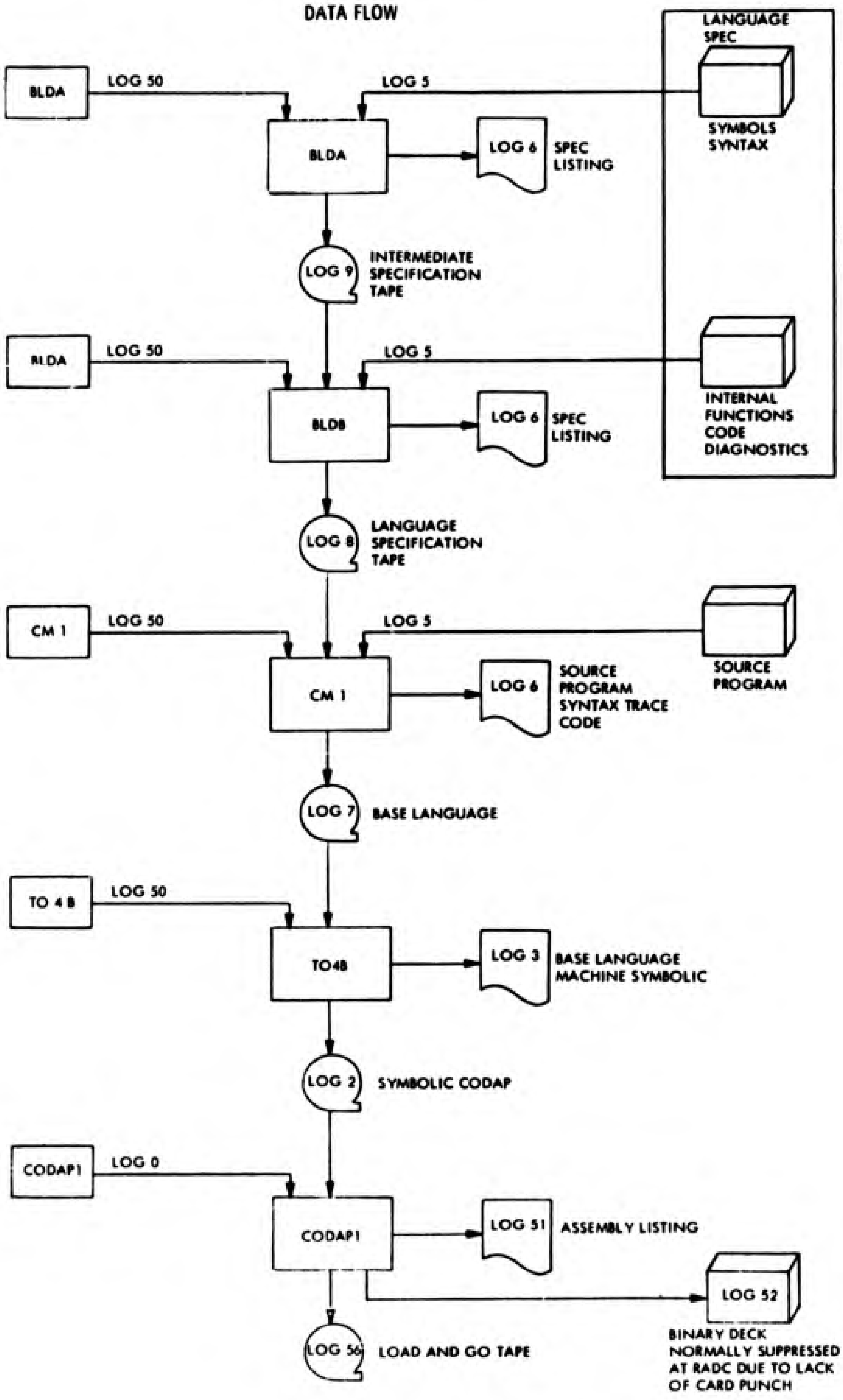
a. Mount COOP MONITOR on channel 3 unit 1

b. Auto-load

c. Mount jovial compiler on unit specified on console typewriter for logical unit 10

d. Mount scratch tapes on logical units 6, 8, 56, 57

POLYLINGUA SYSTEM
DATA FLOW



T10417

e. Set jump key 2

f. At completion of job save tape 56.

Label "load and go for program name"

F. HARDWARE AND SYSTEM SOFTWARE REQUIREMENTS

1. Hardware

a. CDC 1604, 1604A, or 1604B main frame with 32 k core

b. Card reader

c. Line printer

} Can be replaced by additional tape drives when off-line media conversion equipment available

d. Min. 4 tape drives for operation, at least 6 tape drives are needed for maintenance

2. Software

a. Coop monitor

(1) The monitor must initialize available CORE to ZERO instead of RTJ ERROR*

(2) I-O drivers for the specific model of I-O devices attached

(3) Interrupt processing for the specific model of main frame

(4) CODAP1 assembler

(5) Object time subroutines supplied with JOVIAL compiler

b. JOVIAL compiler

3. Documentation required for operation and maintenance

a. COOP monitor operators manual	CDC	509
b. COOP Monitor programmers guide	CDC	60050800
c. CODAP1	CDC	510
d. CODAP1 system revisions	CDC	60081000
e. Library routines	CDC	60051600
f. JOVIAL	SDC	TM-WD/988/200/00

The following additional documents were used during development

g.	COSY	CDC	60081100
h.	FORTRAN-63	CDC	60052400
i.	ALDAP	CDC	60083400
j.	COBOL	CDC	60052100b
k.	Documentation aids	IBM	H20-0177-0

The following four pages illustrate tape unit assignments and deck structures.

Logical Assignments Made to Physical Input/Output Units

Units: Input/Output Function Physical Unit Log Unit No. Assigned by COOP JOBS: Segments in order of execution	Input/Output Units					
	Coop Monitor 606 0	Std. In 405 50	Std. Out 1612 51	Scratch 606 56	Scratch 606 57	606 ---
Job 1						
BLDA Load	I 0	I 50	O 51			
Execute		I 5	O 6			O 9
BLDB Load	I 0	I 50	O 51			
Execute		I 5	O 61			I 9; O 8
Job 2						
CMI Load	I 0	I 50	O 51			
Execute		I 5	O 6		O 7	I 8
Job 3						
T 04B Load	I 0	I 50	O 51			
Execute			O 3		I 7	O 2
CODAP Load	I 0					
Execute				O 56	S 57	I 2
OBJECT Load	I 0		O 51	I 56		
Execute			O 51			

NOTE: I ⟷ Input
O ⟷ Output
S ⟷ Scratch

```
00000000011111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890

9BEGIN JOB C01  MC/CA/YR
9COOP,ACCOUNT NUMBER,PROGRAMMER NAME,
9C/8/S/9/E/5#50/6#51,
930,1000C,S,TABLE BUILDER BLDA-BLDB EXECUTION.
9EXECUTE,,50,1.
          BINARY DECK OF BLDA
9
********************************************************************************
          @@SYMBOLS SECTION@@
********************************************************************************
          @@SYNTAX SECTION @@
********************************************************************************
9REWIND,9.
9EXECUTE,,50,1.
          BINARY DECK OF BLDB
9
********************************************************************************
          @@INTERNAL FUNCTIONS SECTION@@
********************************************************************************
          @@CODE SECTION@@
********************************************************************************
          @@DIAGNOSTIC MESSAGE SECTION@@
********************************************************************************
9REWIND,8.
          END OF FILE CARD
```

Control cards for Table Builders

* 9 in Column 1 represents 9/7 multiple punch.

```
00000000001111111111222222222233333333334444444444555555555566666666667777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890

9BEGIN JOB 002  MO/DA/YR
9COCP,ACCOUNT NUMBER,PROGRAMMER NAME,
9I/8/S/56/57/E/5#50/6#51/7#57,
930,10000,5,COMPILER MODEL CM1 EXECUTION.
9EXECUTE,,50,1.
       BINARY DECK OF CM1
9
       @@SOURCE PROGRAM IN LANGUAGE X@@
       END OF FILE CARD
9BEGIN JOB 003  MO/DA/YR
9COCP,ACCOUNT NUMBER,PROGRAMMER NAME,
9S/56/57/2/E/7#57,
930,10000,5,TRANSLATOR EXECUTION AND ASSEMBLY OF OUTPUT WITH CODAP1.
9REWIND,ALL.
9EXECUTE,,50,1.
       BINARY DECK OF T04 OR T18
9
9REWIND,ALL.
9CODAP1,INPUT#2,LIST,EXECUTE.
9EXECUTE,,56.
       DATA FOR OBJECT PROGRAM, IF ANY
       END OF FILE CARD
9END MONITOR INPUT
```

CONTROL CARDS FOR COMPILER MODEL AND TRANSLATOR

* 9 in Column 1 represents 9/7 multiple punch.

```
00000000011111111112222222222333333333344444444445555555555666666666677777777778
1234567890123456789C123456789012345678901234567890123456789012345678901234567890

9BEGIN JOB 004  MO/DA/YR
9COOP,ACCOUNT-NUMBER,PROGRAMMER NAME,
9I/10/S/6/8/56/57/E/2#50/3#56/4#57/5#57/7#51,
945,30000,,JOVIAL COMPILATION AND ASSEMBLY.
9EXECUTE,,10.
*COMPILE  NAME                                           1
         START
         @@ PROGRAM DECK@@
         TERM $
9REWIND,ALL.
9CODAP1,INPUT#6,LIST,EXECUTE.
         END OF FILE CARD
9END MONITOR INPUT
```

CONTROL CARDS FOR JOVIAL COMPILER

* 9 in Column 1 represents 9/7 multiple punch.

SECTION IV

THE COMPILER MODEL

A. PROGRAM ORGANIZATION AND TERMINOLOGY

All processing performed by the Compiler Model (Program CM1) is table-directed, and its organization reflects this orientation. It is strictly divided into five subprograms:

SYNPR - syntax processor
IFTPR - internal functions processor
CODPR - code processor
CNLPR - control processor
INOPR - input/output processor

Of these, the programs SYNPR, IFTPR, and CODPR comprise the "inner loop" of analysis, processing, and code output (see Figure IV-1). The other processors are infrequently called by these, usually when information transfer is required for continuation of processing.

In turn, the operation of the "inner loop" programs is specified by a set of language data tables:

SYNTAB - syntax table
DIRTAB - director table
IFTTAB - internal functions table
CODTAB - code table

Thus the programs themselves are concerned, not with the totality of the logical processes they perform, but rather with efficient handling of the data entities pertinent to these logical processes.

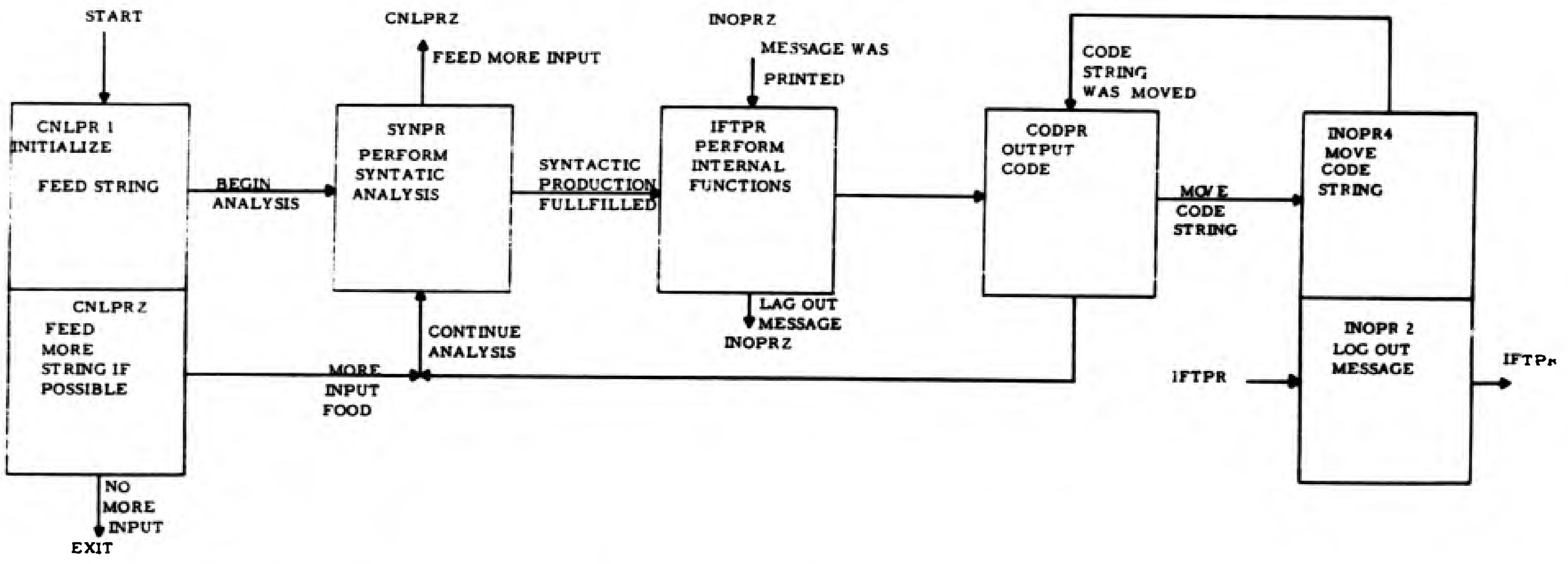


FIGURE IV - 1 : ORGANIZATION OF COMPI LER MODEL PROGRAM

The "innermost" loop within CMl is the syntactic analysis performed by SYNPR. Communication between SYNPR and the other "inner loop" programs (IFTPR and CODPR) is provided by the "director table" (DIRTAB).SYNPR performs the syntactic analysis of the input "construct string" (CONCOR) via a matching process. When the i^{th} syntactic production is matched, a <u>syntactic replacement</u> (a construct which replaces one or more constructs of CONCOR) is called for. Also called for in general are performance of the i^{th} set of internal functions, and output of the i^{th} set of code operations.

All of these actions are specified via the i^{th} entry of DIRTAB. Each entry of this table has the format:

LEFTNO	RITNO	REPCON	IFTLOC	CODLOC
← Syntactic Replacement →			← Internal Functions →	← Code Processing →

The items LEFTNO and RITNO give the positions (within the syntactic production) of the constructs which are to be replaced; REPCON represents the replacement construct (to be placed at position RITNO. IFTLOC specifies the first entry of the portion of IFTTAB which is now to direct the processing performed by IFTPR. Similarly, CODLOC specifies the first entry of the portion of CODTAB which will be processed by CODPR to become output code.

Thus, once the i^{th} production has been matched, the i^{th} entry of DIRTAB completely specifies the ensuing processing performed by the inner loop until resumption of the analysis matching process.

B. SYNTAX PROCESSOR

<u>Organization</u>

The syntax processor (SYNPR) is used to perform syntactic analysis of the input "construct string." This string is represented for processing purposes by a table, designated the CONCOR table. Each entry of the table consists of two entries:

CONSTR - to represent a "construct" or "syntactic type"
REFNO - an associated "reference number" or datum

The syntactic analysis is performed by matching the CONSTR items within CONCOR against similar items in SYNTAB (the syntax table), then - for a successful match - replacing one or more entries of CONCOR by a single entry. Thus the primary processes performed by SYNPR are the referencing and handling of CONCOR, and the referencing of SYNTAB.

SYNPR is conceptually divided into two subprograms:

RECOG, which performs the matching or "recognition" process, and
REPLAC, which performs the replacement procedure

As shown in Figure IV-2, REPLAC is represented by two separate subprograms, REP1 and REP2. This arrangement is logically necessary because operations within the internal functions processor (IFTPR) and the code processor (CODPR) must refer to positions within CONCOR pertaining at the time of matching the production (i. e. , before a replacement is made). Thus REP1, which operates immediately after RECOG, places the "replacement construct" REPCON as directed by DIRTAB, but does not in any other way change CONCOR. Then, upon return to SYNPR for continuation of analysis, after operation of IFTPR and CODPR, the rest of the replacement procedure is performed by REP2.

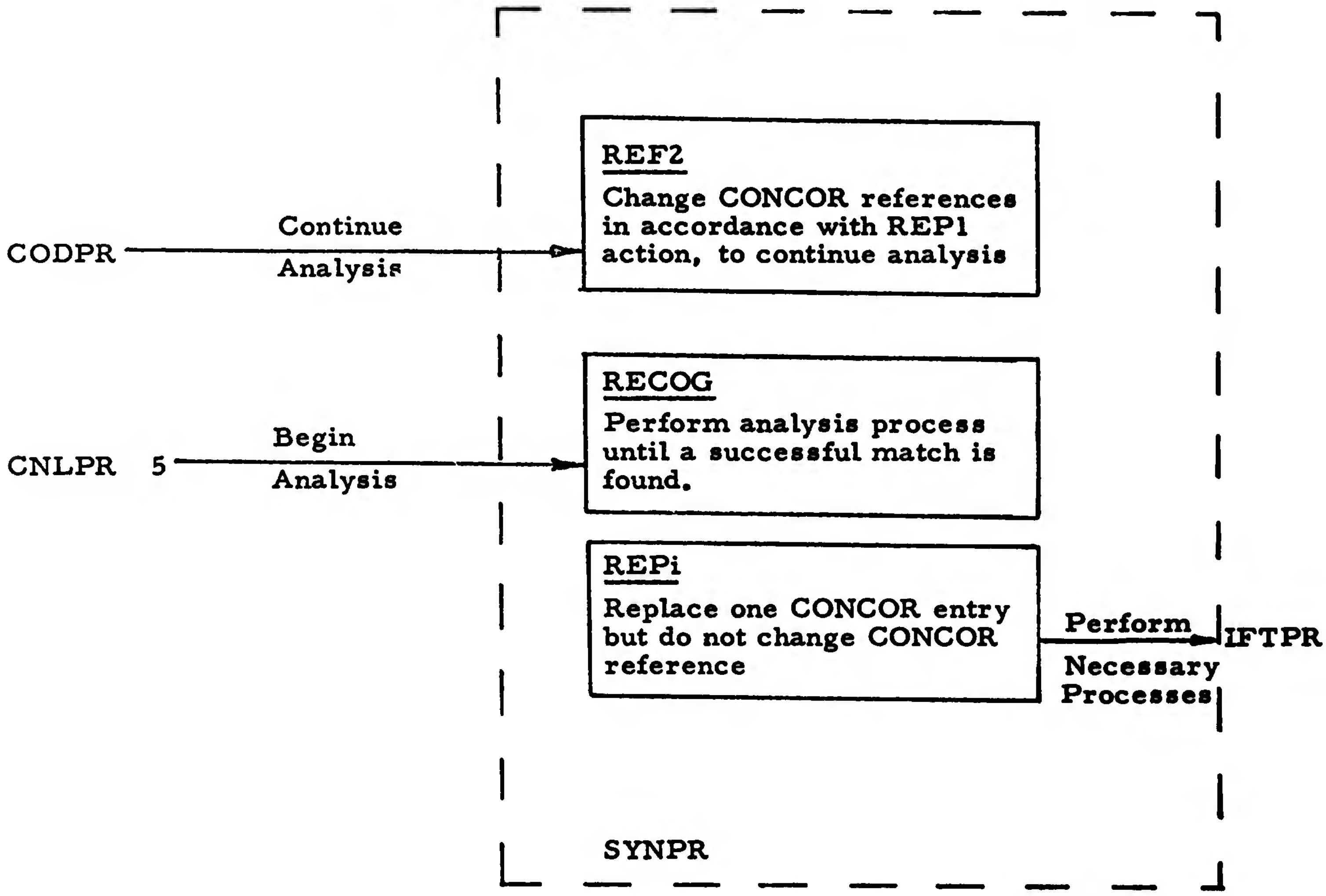

CODPR
Continue
Analysis
REF2
Change CONCOR references in accordance with REP1 action, to continue analysis
CNLPR 5
Begin
Analysis
RECOG
Perform analysis process until a successful match is found.
REPi
Replace one CONCOR entry but do not change CONCOR reference
Perform
Necessary
Processes
IFTPR
SYNPR

Figure IV-2. SYNPR Organization

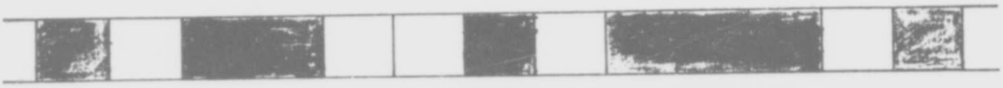
Referencing and Handling CONCOR

CONCOR is conceptually a sequence of "symbols", any of which is referenced by its position within the table. This conceptual model is analogous to a sequence of marbles laid in an inclined trough, where each marble is referred to by its position. Then, the syntactic replacement process is analogous either to changing the color of a marble (which has no effect on position) or removal of some marbles -- and in this last case, other marbles inevitably roll down the incline to fill the gap, so that positions of some marbles are "automatically" changed.

The actual implementation of CONCOR cannot be this simple. Each CONCOR entry must correspond not merely to a "relative symbol position", but to an actual core memory location (or locations). Thus the core memory allotted to CONCOR is rigid in format, rather than flexible as with the conceptual model, so that the "nulling out" process performed during replacement creates "holes" in CONCOR.

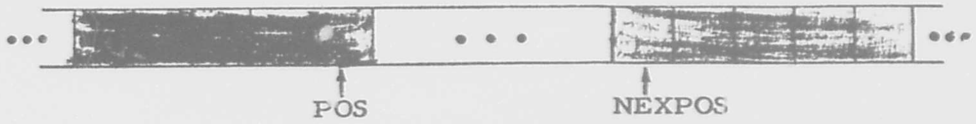
The appearance of CONCOR before processing is begun can be represented graphically as follows:

where each shaded box is a CONCOR entry. However, after some processing, CONCOR might have the appearance:



where each white box is a "hole" or "null symbol" (written Λ).

To achieve speed by eliminating the buildup of long strings of Λ's between non-null entries, CONCOR is maintained as illustrated below:



This method "cuts" CONCOR into two pieces:

1. The string of <u>non-null</u> entries ending at the entry number POS; this is the part of the string which has already been examined.

2. The string of <u>non-null</u> entries beginning with the entry number NEXPOS; this part of the string has not yet been examined. In fact, the "scan position" is always at NEXPOS.

There is of course a third (but insignificant) piece — the portion of CONCOR between POS and NEXPOS, which consists solely of Λ's.

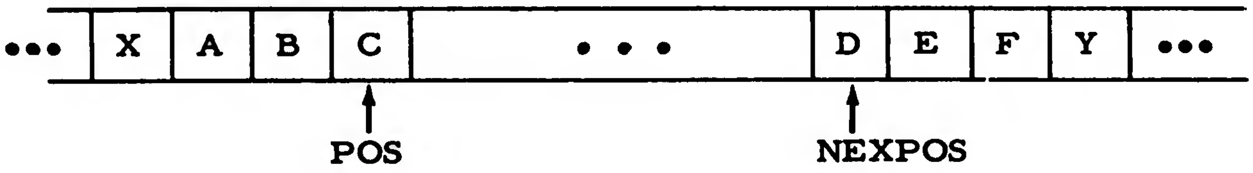
Processing is begun with a small initial gap between POS and NEXPOS. Thereafter, REP2 maintains CONCOR so as to

a. Pack all Λ's into the center

b. Maintain the scan position at NEXPOS.

As a result, nulling out is commonly accomplished by decreasing POS and/or increasing NEXPOS; entries in the "dead" portion need not actually be nulled out, since they are never examined.

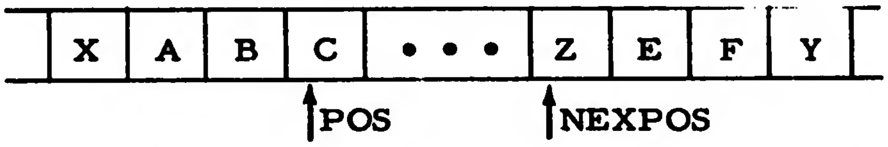
We can now show the actual effect on CONCOR of various syntactic productions. Suppose the string under examination is



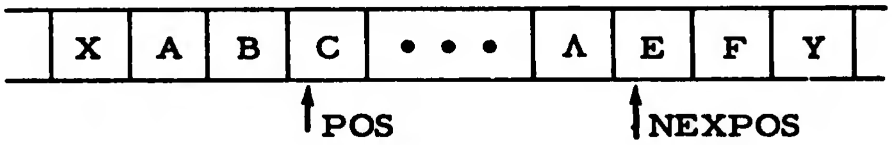
Then each syntactic production below results in the string to the right of it.

<u>Syntactic Production</u>   <u>Resulting String</u>

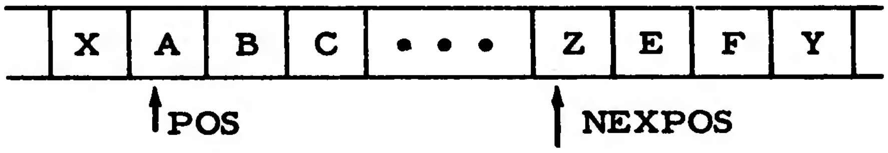
$D \longrightarrow Z$

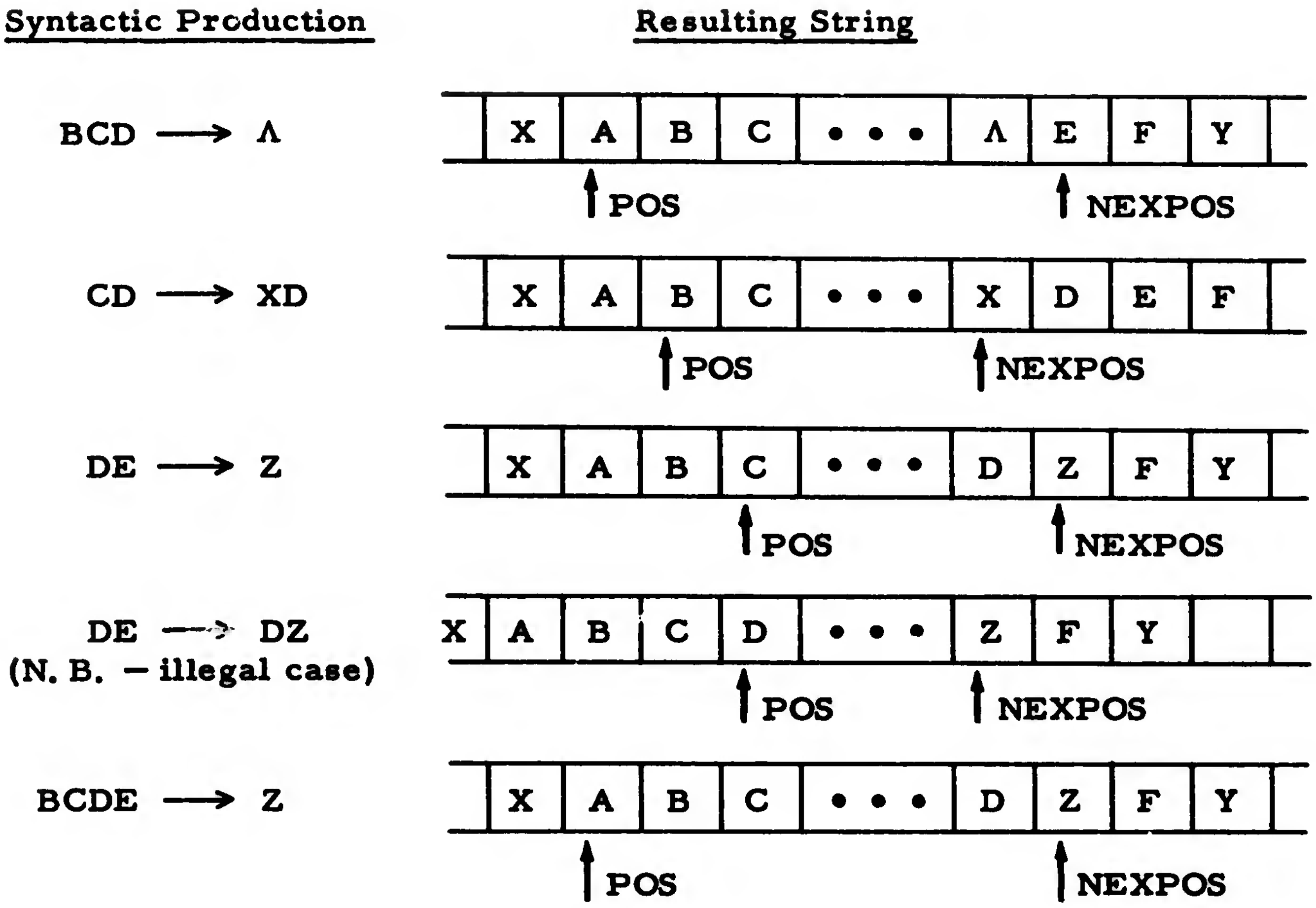


$D \longrightarrow \Lambda$



$BCD \longrightarrow Z$





Referencing of entries by their relative position within a syntactic production (as in REP1, REP2, and IFTPR) is always done relative to NEXPOS, which is "position 0". Entries to the "left" or "beginning" have negative positions (e. g. POS is "position (-1)"), and entries to the "right" or "end" have positive positions. For example, in the string used in all the above examples, A ⟷ position(-3), E ⟷ position(+1), and D ⟷ position 0.

The "Ends" of CONCOR

A final detail in the handling of CONCOR is the detection of the "ends" of the string. Both a "permanent end of string" (PES) and a "temporary end of string" (TES) must be detected. Although the input string (CONSTG) is conceptually of unlimited length, the actual processing of this string is done in the (fixed) CONCOR area which may necessitate segmentation of the string. On the other hand, the string might not be sufficiently long to fill CONCOR.

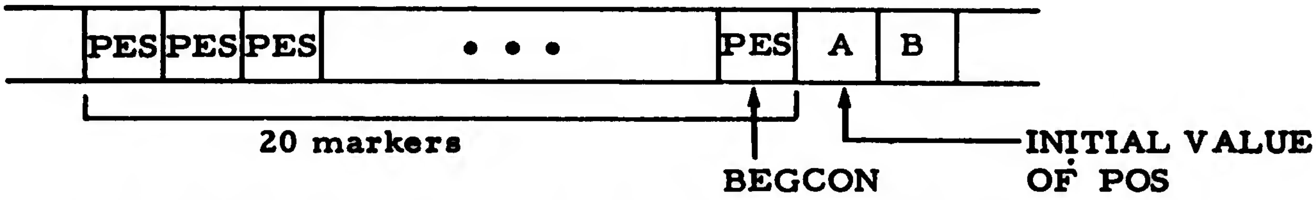
A simple scheme is used to satisfy these requirements. Markers (i.e. reserved values of CONSTR) are used to denote PES and TES; every program string is bracketed between PES markers, and TES markers are used to delineate segments of CONSTG within CONCOR. Thus, the CONCOR portion of CONSTG is always bounded by "end markers". If CONSTG is contained entirely within CONCOR, the markers on both sides are PES markers. If CONCOR is a "leftmost" ("rightmost") portion of CONSTG, then it is bounded on the right (left) by TES markers; if a "middle" portion, it is bounded on both sides by TES markers.

Two conditions must be insured:

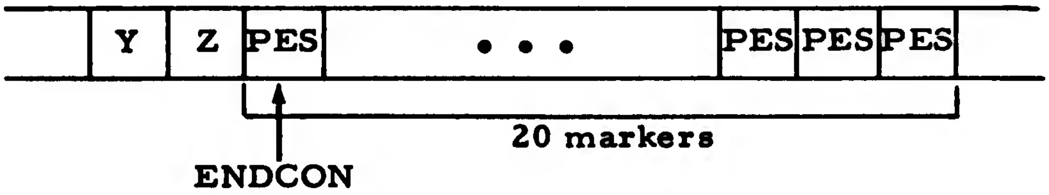
a. That TES markers can not interfere with the syntactic analysis (causing, for example, the appearance of (say) A B TES rather than A B C, and so on), and

b. That PES markers cannot be bypassed (as, for example, by a production involving the reserved construct NEXT).

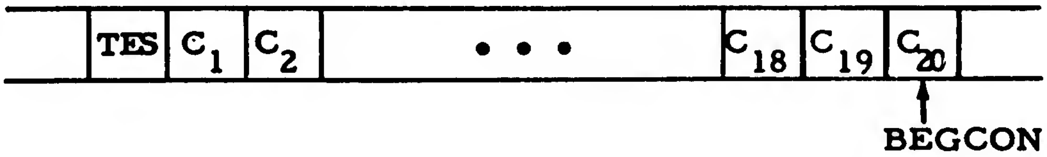
In the implementation of this scheme, a single marker (CONSTR = 0) is used to denote both PES and TES; for TES, the corresponding value of REFNO is 0, while for PES this value is 1. The marker is used in conjunction with two positions of CONSTG, denoted BEGCON and ENDCON. First, every program string is preceded by twenty (20) PES markers, and succeeded by twenty PES markers. The position BEGCON specifies the "beginning" of that portion of CONSTG which is in CONCOR, and similarly, ENDCON specifies the "end" position. In the vicinity of PES markers, BEGCON and ENDCON are assigned as follows:

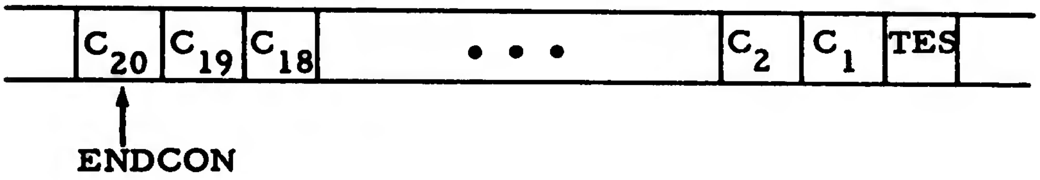


and the initial values of POS is set to the position following BEGCON.



Only one TES marker is used for each TES bound. In this case, the values of BEGCON and ENDCON are assigned as follows:





Each time the scan position changes, the value of NEXPOS is tested against ENDCON; if NEXPOS $\geq$ ENDCON, an "end" has been reached. Also, each time the scan position moves to the left, the value of POS is tested against BEGCON; if POS $\leq$ BEGCON, an "end" has been reached.

The Syntax Table - Structure and Interpretation

a. General

Syntactic analysis is performed primarily by matching sequences of CONCOR symbols (i. e. , constructs) against their similar sequences in SYNTAB. In addition, analysis is occasionally dependent on the values of "function registers". Also, the order in which CONCOR subsequences (in the vicinity of the scan position) are examined is dependent on the construct which is in the scan position.

All of these features are accommodated within a table of uniform format, hence this format and its interpretation are of necessity complex. The basic structure of SYNTAB is that of a linked set of tables and subtables, each entry specifying a ling to another table. A "primary table" is used, which contains one entry for each construct,*

*The primary table is of course the first n entries in SYNTAB.

and which is thought of as being aligned with the CONCOR symbol in the scan position (NEXPOS). The matching procedure is always begun by reference to the entry corresponding to the construct in scan position; this reference determines for the given scan position the interpretation of all subtables to be examined.

b. Typical Subtable Configuration

The set of possible interpretations of primary table entries is:

(a) No match is possible for this scan position.

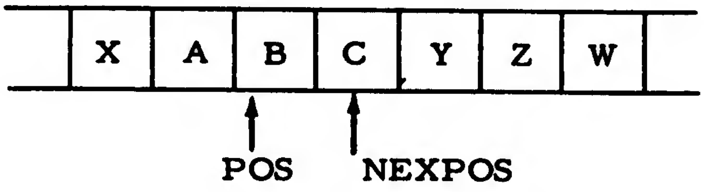
(b) A match has already been obtained.

(c) Matching will proceed successively to the "left", starting at POS.

(d) Matching will proceed successively to the "right", beginning at NEXPOS + 1.

(e) Matching will begin at POS, and then in directions given in further entries.

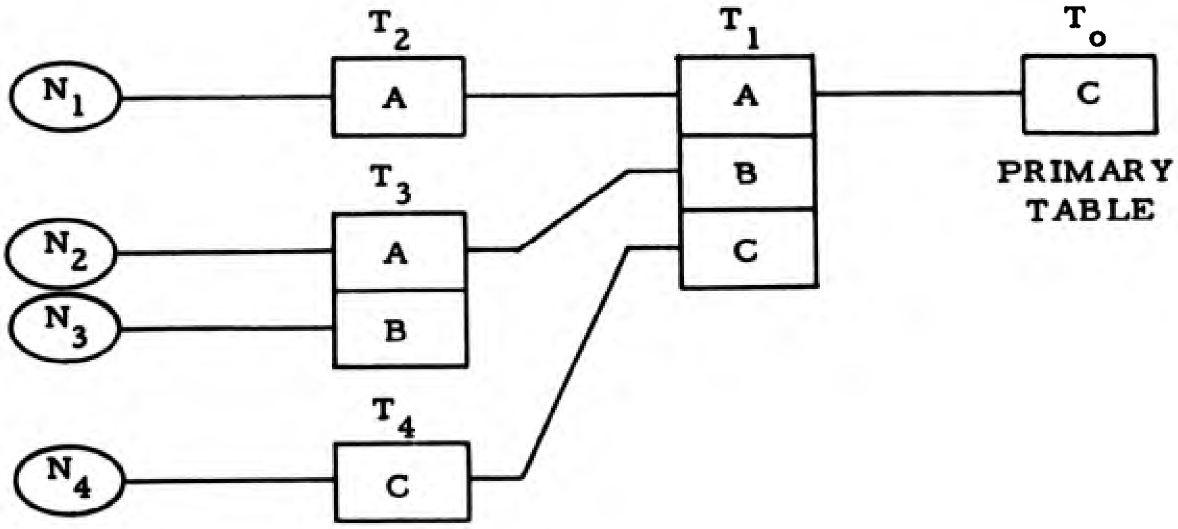
Clearly, the handling of primary table entries is atypical: match with the "scanned" symbol is automatic, and the entry sets the conditions for further matching. To illustrate more usual handling, and also a typical subtable configuration, suppose that the CONCOR string



is under consideration. Then if the construct C leads to interpretation (c) above, and if the sequences in SYNTAB are

$$\begin{cases} A\ A\ C \\ A\ B\ C \\ B\ B\ C \\ C\ C\ C \end{cases}$$

the subtable configuration below is obtained.



In this example, the primary table entry designates the subtable T_1, and similarly every entry designates either a subtable T_i or (if a complete sequence is matched) a "definition number" N_i. Basically, the entries of a subtable are matched successively against the appropriate CONCOR symbol, each successful match resulting in examination of a further subtable, until either

(a) no further match is possible (the subtable is exhausted), or

(b) a complete sequence is matched.

Analysis in this case leads to the "definition number" N_2.

For the interpretation (d), this process would be identical except for the direction in which sequences are checked. The interpretation (a) would signify that no subtables exist, whereas (b) would signify that no subtables were necessary, since the primary table entry itself had resulted in a complete sequence match.

c. Configuration for Syntactic Functions

The sequences of the above example correspond to syntactic productions of the form shown on pages 35 and 36. Another form of syntactic production must now be discussed which form utilizes the "syntactic function". We have noted that analysis sometimes depends on the value of a "function register"; we now present both a notation and an interpretation of SYNTAB to accommodate this feature.

All productions thus far discussed have the basic form

$$\psi \longrightarrow \phi$$

where ψ and ϕ are symbol strings; once ψ has been "recognized", ϕ is uniquely determined. In certain cases, ϕ is defined as a function Φ, defined for some numeric variable Z, so that the corresponding syntactic production may be written as

$$\psi \longrightarrow \Phi\ (Z),$$

or may be written $\Psi \rightarrow \Phi$ provided all "function names" are required to be unique.

The "recognition" of ψ now results in testing of the value of Z, and one of several strings is chosen as ϕ on the basis of this value.

For example, we might have

$$XYZ \longrightarrow \Phi\ (Z) \quad ,$$

where

$$\Phi\ (Z) = \begin{cases} XYA \text{ for } 0 \leq Z \leq 3 \\ Q \text{ for } 4 \leq Z \leq 6 \\ ZYZ \text{ for } Z = 7 \\ X \text{ for } Z \geq 8 \end{cases}$$

so that recognition of XYZ results in either XYA, Q, ZYZ, or X, depending on the current value of Z. Our system use function registers F_i to represent numeric variables, and only the values of such registers are permitted as arguments Z of syntactic functions. We may thus use the more descriptive notation (F_Z) to represent a syntactic function.

Syntactic functions are accommodated in SYNTAB as follows. First, the construct value of 1 is reserved to signify a syntactic function in SYNTAB;[*] we use the notation FCN to denote this value. Such entries appear only in subtables (e.g. T_1, T_2, etc.), but not in the primary table. Each FCN entry designates a subtable, which designates

(a) the function register which is to act as the numeric variable, and

(b) a set of bounding values, each value designating a "definition number".

To illustrate this use, consider the examples

$$\begin{cases} P \longrightarrow \Phi\,(F14) \\ XYZ \longrightarrow \Psi\,(F22) \end{cases}$$

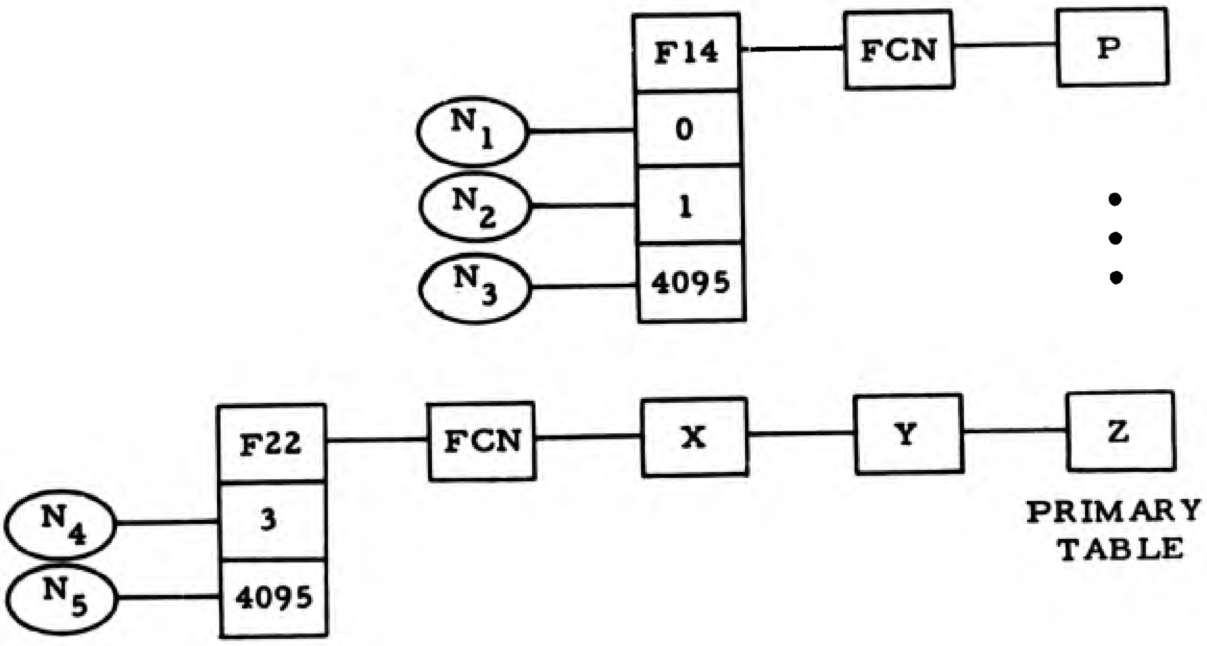
where

$$\Phi\,(F14) = \begin{cases} Q \text{ for } F14 = 0 & (N_1) \\ R \text{ for } F14 = 1 & (N_2) \\ S \text{ for } F14 \geq 2 & (N_3) \end{cases}$$

$$\Psi\,(F22) = \begin{cases} XYA \text{ for } 0 \leq F22 \leq 3 & (N_4) \\ Z \text{ for } F22 \geq 4 & (N_5) \end{cases}$$

These examples result in the subtable configuration below

[*] For any language, all values of CONSTR in CONCOR entries are ≥ 2, with of course the exception of TES and PES markers having the reserved value 0.



(Note: 4095 is the largest possible function register value.)

In each case the FCN entry (construct value = 1) denotes a subtable whose first entry specifies the function register F_i which is involved. Each successive entry designates a value V_j, and corresponding definition number N_j. The V_j are arranged in strictly increasing order (i.e. $V_j < V_{j+1}$), and the last value given is always 4095 (the largest possible value of F_i). To evaluate the syntactic function, the entries are examined in succession. If $F_i \leq V_j$, then the designated N_j is taken, otherwise the (j+1)st entry is examined. Since the last value is 4095, the relation must be satisfied for some entry.

d. Configuration for "Rule Sets"

We have mentioned previously that the order in which CONCOR substrings are scanned may depend on the construct in scan position. Correspondingly, interpretation (e) on page IV-13 allows scan direction to be table-directed. This mechanism is intended to accommodate "rule sets" (i.e., ordered sets of productions), which are discussed briefly in the 1962 report (pp. 107, 108).

From the viewpoint of CM1, a rule set for a symbol S is comprised of a set of sequences, all containing the symbol S in scan position, which are to be tested for a match in a specified order. For example, the following rule set for X:

RULE (X)

(1) WXH $\longrightarrow$ P

(2) XY $\longrightarrow$ Q

(3) WX $\longrightarrow$ R

(4) X $\longrightarrow$ Z

specifies that the symbol to the right of X is to be matched against Y in (2), before the match to the left for W in (3) is made. Thus, if the CONCOR substring containing X is in fact WXY, the match completed will be (2) rather than (3); if this substring is in fact WXH, the match completed will be (1).

The direction of scan may thus be either to the left or to the right or both in the same production. More important, it is necessary to accommodate the situation typified by (3) and (4), in which

(a) further symbols to the left or right are checked for match, and then

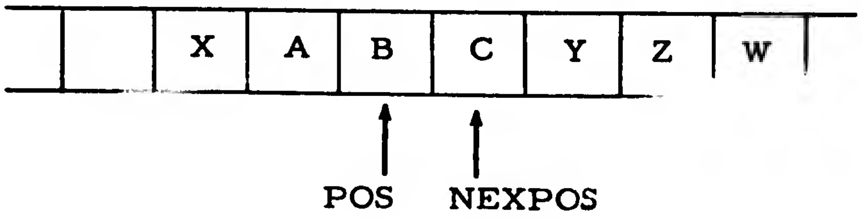
(b) if none is obtained, then a match is complete, whatever symbol had previously been checked.

Thus, (3) checks for WX; but if this is not present, X is nevertheless matched. As another example, consider

(5) ABCX ⟶ L
(6) BCX ⟶ S
(7) CX ⟶ M
(8) X ⟶ F

as an alternative rule set for X. In (5), if B is matched, then (5) will be matched if an A is to the left, and otherwise (6) will be matched; and so on for (6), (7), and (8). First three symbols, then two symbols, then one symbol, then no symbols to the left of X are to be checked.

A combination of mechanisms is used to accommodate rule sets. First, the construct value of 0 is reserved to represent a special reserved construct name: NEXT. The construct NEXT is assumed to match any construct! Thus the appearance in CONCOR of



is matched by any of

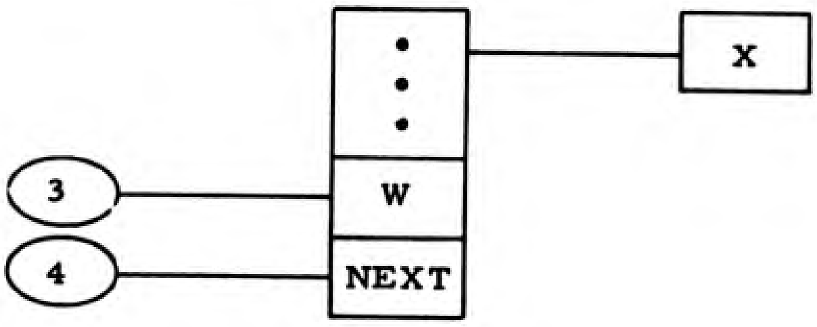
$$\left\{\begin{array}{cccc} A & B & C & \\ \text{NEXT} & B & C & \\ A & \text{NEXT} & C & \\ & B & C & \text{NEXT} \end{array}\right.$$

appearing in SYNTAB.

Productions (3) and (4) may now be written

$$\left\{\begin{array}{lccl} (3) & W & X & \longrightarrow R \\ (4) & \text{NEXT} & X & \longrightarrow Z \end{array}\right. ,$$

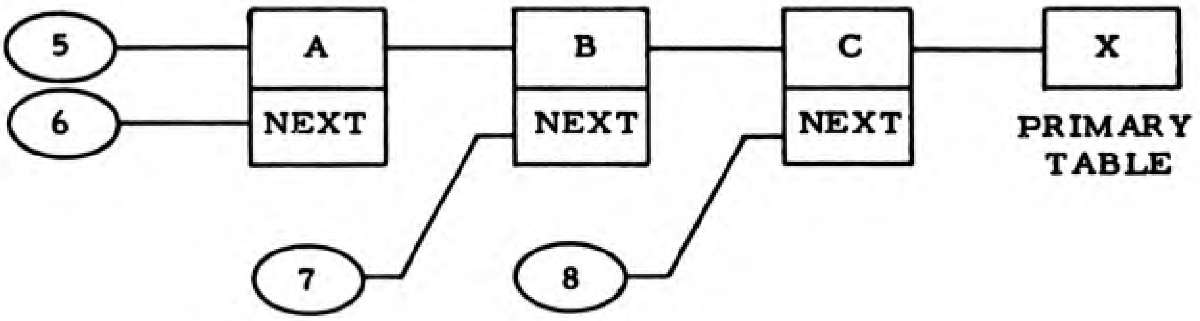
and simply accommodated by the table structure



Similarly, the alternative example may be rewritten

$$\left\{\begin{array}{lccccl} (5) & A & B & C & X & \longrightarrow L \\ (6) & \text{NEXT} & B & C & X & \longrightarrow S \\ (7) & & \text{NEXT} & C & X & \longrightarrow M \\ (8) & & & \text{NEXT} & X & \longrightarrow F \end{array}\right.$$

and accommodated by the table structure



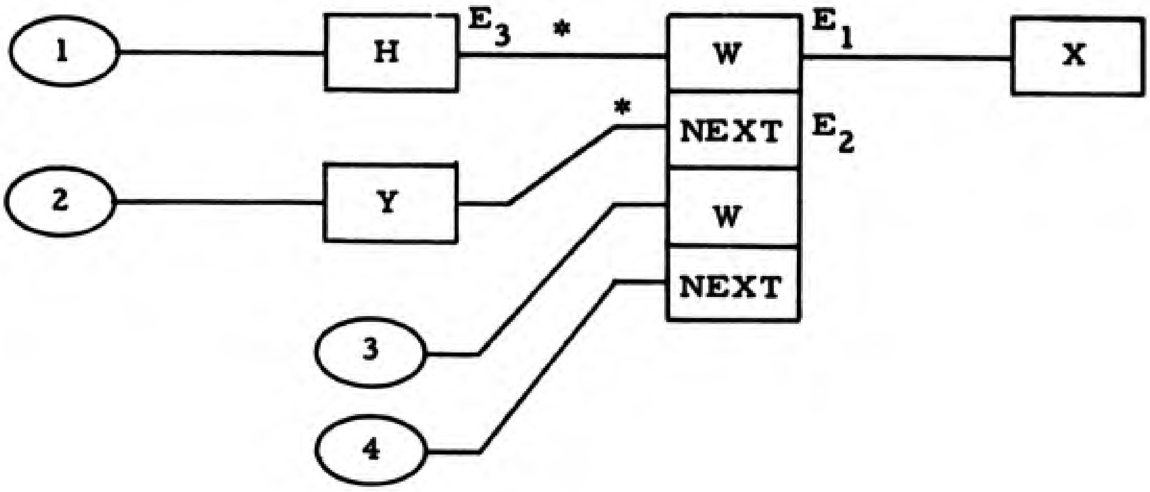
Interpretation (e), which is used for rule sets, operates as follows: scanning is always assumed to begin toward the left, and to continue toward the left until SYNTAB indicates a change of direction. When a direction change is specified, scanning of subtables then proceeds toward the right. Each subtable entry E is checked

(1) for a match with CONCOR

(2) for the values which signify FCN or NEXT.

The action taken for FCN has already been discussed. If either of the other conditions is met, the subtable designated by the entry E is considered next. Prior to such consideration, however, note is taken of the entry E^* which succeeds E; if no match is completed in consideration of the subtables of E, the matching process returns to consideration of E^*.

To illustrate these points, we exhibit the table configuration for our first example:



A (*) on a subtable linkage indicates a change in scan direction from left to right. The matching process begins at entry E_1 ; if the symbol to the left of X is a W , E_3 is then considered. However, if the symbol to the right is not an H , then entry E_2 must next be considered, in order to test for the sequence of (2).

Interpretation (e) of page IV-13 is the most general interpretation of SYNTAB, and could suffice for interpretations (c) and (d). These interpretations are used merely for the sake of processing speed, since (e) is inherently much slower.

In comparison with (e), interpretations (c) or (d) have the following characteristics:

(1) Matching scan is in one direction only.

(2) If the entry E is matched, and the subtable designated by E is considered, then (except for the process of (3) below) the entry E^* which succeeds E is never considered. Thus, once any subtable has been exhausted with

no match produced, the matching process for that scan position is terminated by the process of:

(3) The last entry of the subtable designated by the primary table entry (e. g. , E_5 above) is checked for the values FCN or NEXT. An FCN entry is processed as in section (c), while a NEXT entry designates a "definition number" N.

Each entry of the SYNTAB table is composed of the following items:

TABCON = a number representing a construct/ syntactic type

DONE = indicates completion of match (= 1 if match is complete)

RITE = indication of scan direction (for interpretation (e) only) = 1 if scan direction is to change to the right

LINK 1 = number of entries in subtable whose first entry is given in LINK 1.

C. INTERNAL FUNCTIONS PROCESSOR

Organization

The internal functions processor (IFTPR) is used to perform the operations described in Section III C, via direction by the internal functions table (IFTTAB). These operations are performed primarily on three data entities:

FTAB - A set of "function registers", each register containing a positive integer < 4096.

PTAB - The "property table", a table of positive integers < 4096. The i^th^ entry of PTAB has 4 properties $P_o(i)$, $P_1(i)$, $P_2(i)$, $P_3(i)$, and an associated name contained in the PINTAB table.

STAB - A set of "symbol registers", each containing some string of symbols,

IFTPR is organized into discrete subprograms, one for each of the different types of operations. At the beginning of each operation, control is transferred to the appropriate subprogram by the use of a program switch which is indexed by operation code. In the following we will outline, for each type of operation, the specific details and important methods.

Numeric Operations

Function registers and properties are all assumed to be positive integers <4096. Accordingly, all operations are performed modulo 4096. As example, if $F_1 = 2050$ and $F_2 = 2055$, then

INC F_1 F_2 results in $F_1 = 9$, since $4105 = 9 \pmod{4096}$

DEC F_1 F_2 results in $F_1 = 4091$, since $-5 = 4091 \pmod{4096}$.

For the operation DIV, the integer part of the quotient is taken, so that for $F_1 = 8$ and $F_2 = 3$, the operation DIV F_1 F_2 results in $F_1 = 2$.

The operation STV, which sets a function register or property to a given value, allows the use of numeric values $\leqslant 999$ as the second operand.

Two special types of operands may be used in numeric operations; namely SCOPE and N(Sxx), the latter is the "number conversion" of Sxx. The use and significance of these operands is discussed later.

Symbolic Ooerations

Each symbol register S_i (i = 0, 1, 2, 3) consists of 120 consecutive cells, each cell capable of containing one six-bit character. Each cell is either "blank" or "nonblank"; S_i is said to be empty if all of its cells are "blank".

Symbolic orperands are basically of two types - contents of symbol registers, or single six-bit characters. By "contents of S_i", we mean the (possibly null) string of nonblank characters which is left-justified in S_i. A single character is referred to either as the ith "transform" or "value" $V_i(C)$ associated with a given construct position, since single characters are obtained only from the CONCOR string. Every character is assumed to have a number of transforms which may be defined; the 0th transform is the character itself. These transforms are contained in a "value table" (VALTAB); if the character is regarded as a 6-bit number n, then the VALTAB entry corresponding to V_i is given by $64i + n$.

The PUT operation places the desired character (s) in S_i, left-justified. The operations PRE (for "prefix") and SUF (for "suffix") are both concatenation operations, resulting in the "attachment" of two strings. For SUF, a string is attached at the "end" of the initial string, while for PRE, a string is attached at the "beginning" of the initial string. As an example, suppose S_1 = PQRS and S_2 = WXYZ. Then SUF S_1S_2 would result in S_1 = PQRSWXYZ, while PRE S_1S_2 would result in S_1 = WXYZPQRS.

The RML and RMR operations remove or detach from S_i, a given number of characters. Characters are removed from the "left" (prefix) or "right" (suffix) of the symbol register. For example, if S_1 = PQRS, then RML Sl 3 results in S_1 = S, while RMR Sl 3 results in S_1 = P.

It should be noted that symbol registers may contain no more than 120, and no less than 0 (nonblank) characters. If S_i is empty, we write $S_i = \Lambda$. Recall that for any symbol string α, we must have $\alpha\Lambda = \Lambda\alpha = \alpha$. Then if S_1 = PQR and $S_2 = \Lambda$ we will have:

Operation	Result in S_1
PUT S_1 S_2	Λ
PRE S_1 S_2	PQR
SUF S_1 S_2	PQR

The effect of the 120-character limitation is to "lose" characters from the right or suffix part of the string. Let $S_1 = a_1 \ldots a_n$ and $S_2 = b_1 \ldots b_k$; the results of PRE and SUF operations on these are defined as follows:

Operation	Result in S_1	
	For $n + k \leqslant 120$	For $n + k > 120$
PRE S_1 S_2	$b_1 \ldots b_k a_1 \ldots a_n$	$b_1 \ldots b_k a_1 \ldots a_{(120-k)}$
SUF S_1 S_2	$a_1 \ldots a_n b_1 \ldots b_k$	$a_1 \ldots a_n b_1 \ldots b_{(120-n)}$

Number Conversion

One type of operand which may appear in numeric operations is the "number conversion" of a symbol register, shown in the table of operand types as N(SXX). Use of such an operand implicitly specifies conversion of the contents of a symbol register to a number in function register format (i. e. a 12 bit positive integer). This implicit operation allows CM1 to make use in its analysis of numbers furnished by the input program itself.

A number of conventions are rigidly adhered to in use of number conversion:

1. The symbol register contents to be converted are assumed to constitute a positive integer, for some radix.
2. The radix r, $2 \leq r \leq 63$, which is to be used in conversion is the value found in F_1 at the time of occurrence of the operand N(SXX).
3. Use of the number conversion operation always results in $F_1 = 10$ at the end of the operation.
4. Each (non-blank) six-bit character of SXX is assumed to be a digit of the number in the given radix system.
5. The conversion operation thus computes, modulo 4096, a positive integer in internal format equivalent to the integer specified in the radix r. The result of conversion is used as specified by the numeric operation (e. g. SET, ADD, etc.).

Scope and Scope Operations

A scope mechanism is provided within IFTPR. Each scope value corresponds to a particular portion of the "program string" CONCOR, and is written as a number pair $(B_i, E(B_i))$, corresponding to the portion of CONCOR between the i^{th} "begin marker" B and its associated "end Marker" E. A "begin marker" by the operation SSE. Scopes are processed by IFTPR using the integer items SCOPE AND ENDM, and the table SCPTAB.

The i^{th} entry in SCPTAB represents the number of the end marker associated with the i^{th} begin marker. The value of SCOPE represents, at any time, the entry number corresponding to the first available entry in which an endmarker number may be written (that is, SCOPE represents the "current scope"). The item ENDM represents the number of the last end marker which has appeared.

SCOPE, ENDM, and all entries of SCPTAB are initially set to zero. All further manipulation of these entries is performed only in response to the SSI and SSE operations, as follows:

SSI - Set the value of SCOPE to the smallest integer
I >SCOPE such that the I^{th} entry of SCPTAB is O.

SSE - Increase the value in ENDM by 1.
Set the SCPTAB entry indicated by SCOPE to the new value of ENDM
Set the value of SCOPE to the largest integer
I < SCOPE such that the I^{th} entry of SCPTAB is O.

The item SCOPE may be used as an operand in numeric operations. The basic reason for this usage is to allow PTAB properties to be set to SCOPE, for eventual tests on SCOPE of various identifiers. A special convention applies to properties and function registers which represent SCOPE values, namely all numbers which represent SCOPE values are ≥ 2048.

Therefore, whenever SCOPE is specified as an operand, the actual quantity which must be used as an operand is precisely SCOPE + 2048.

PTAB Manipulation and Searches - Association with CONCOR

The operations discussed here are concerned essentially with the formation and later "detection" of associations between PTAB entities and syntactic entities (constructs in CONCOR). Specifically, the processes performed by these operations are:

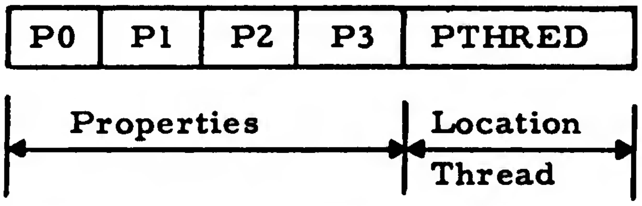
1. "definition" of a given symbol string -
   a. entry of the string into PINTAB
   b. reservation of a new PTAB entry to correspond to the new PINTAB entry
   c. setting of the "syntactic class" (P_o) of the new PTAB entry.

2. "association" of a PTAB entry with a construct in CONCOR (as well as "retrieval" of this association from CONCOR).

3. search to determine if a given symbol string has been assigned a PTAB entry (and if the properties in this entry satisfy certain conditions).

4. "identification" of a construct in CONCOR as specified by a function or property value (i.e. setting of CONSTR to this value).

Before discussing the operations, we will indicate, in more detail the structure of the PTAB and PINTAB tables. An entry of PTAB has the format below,



and the items have the following significance:

P0 = the "zeroth" property, i.e. syntactic class

P1 = the "first" property

P2 = the "second" property

P3 = the "third" property

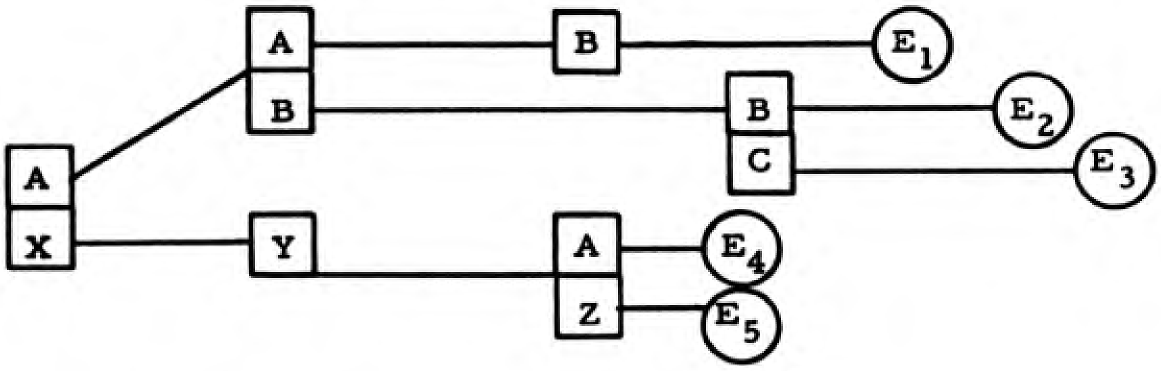
PTHRED = location thread; specifies the entry number of the next entry associated with the same name in PINTAB.

All PTAB entries are assumed to be zero initially. It should be noted that in many languages names need not be unique, so that a given name may be associated with each of a number of PTAB entries. For speed in searching PTAB, the entries associated with a given name are "threaded" together, and the PINTAB entry for that name contains the number of the first entry.

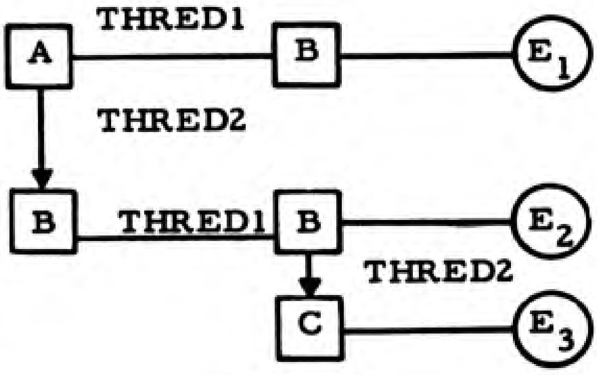
The PINTAB table contains the set of names associated with PTAB, and is used as an index by name to PTAB. Thus, PINTAB contains a set of names, and for each name, a number referencing one of the entries associated with that name. PINTAB is conceptually structured as a linked set of tables and subtables, each entry specifying a link to another subtable. As an example, consider the set of names

$$\left\{\begin{array}{lcl} \mathrm{AAB} & \longleftrightarrow & E_1 \\ \mathrm{AAB} & \longleftrightarrow & E_2 \\ \mathrm{ABC} & \longleftrightarrow & E_3 \\ \mathrm{XYA} & \longleftrightarrow & E_4 \\ \mathrm{XYZ} & \longleftrightarrow & E_5 \end{array}\right.$$

which is conceptually accommodated by the PINTAB structure:



Linkages between table entries are indicated via two sets of threads denoted THRED1, and THRED2. THRED1 specifies the first entry of the subtable associated with the element, while THRED2 specifies the next entry in the subtable of which the element is a member. Reproducing part of our diagram, we have



PTAB entry numbers are specified via an item denoted ENTNO.

The individual operations can now be specified.

DEF - Operates on a symbol string contained in the designated symbol register, as follows:

1. A new PTAB entry is reserved, and the entry number corresponding to this entry is set into F_o.

2. The string is entered into PINTAB, and referenced to the new TPAB entry as necessary (see discussion on use of PINTAB).

3. The syntactic class of the new entry (i.e. $P_o(F_o)$) is set to the given number.

ASO - Set the CONCOR item REFNO ($±XX$) (i.e. the REFNO item for the given construct position) to the value specified. The primary intent of this operation is to allow the setting of REFNO to the value in F_o, thereby "associating" the particular construct with the PTAB entry referred to by F_o.

RTV - Set the specified function or property to the value found in REFNO ($±XX$), i.e. to the REFNO value for the specified construct position. The intent here is to retrieve associations formed as above.

IDF - Set the CONSTG item CONSTR ($±XX$) to the value of the specified function or property. The primary intent here is to allow identification of a given construct position with a PTAB entry, by setting the construct to the syntactic class P_o of the entry.

SER - Search PTAB for all entries having the name specified. For each such entry:

1. Set the entry number into F_o

2. Perform the set of internal operations following the search operation, and ending with an ENS operation.

A specified set of operations may thus be performed for every entry having a given name. Conditions on other properties of the entry may be imposed in conjunction with the name test, by use of the TST operation to be discussed.

ENS - End search sequence. No operation is performed; this code merely acts as a marker to delineate the end of the set of operations performed for an entry meeting the search condition.

The name used for comparison in the searching of PTAB may be given in one of two ways:

a. As the contents of a symbol register

b. Implicitly, as the name associated with a given construct position.

For a name given as contents of a symbol register, a search of PINTAB is first performed for this name. This search will be successful if and only if PINTAB contains a string identical in all respects with that in the symbol register - a string of the same length, all of whose characters "match" those of the string in the symbol register. If PINTAB contains such a string, then a PTAB entry number will be specified for this string. If PINTAB does not contain such a string, then there will not be any PTAB entries satisfying the condition - execution of the search, and of all operations contingent on it, is immediately fulfilled by skipping all of these and proceeding to the next operation following these.

If the PINTAB search is successful, the PTAB entry number given is (potentially) the first of a list of numbers, each succeeding number to be found in the PTHRED item of the entry being considered. All of the entries which will be considered will meet the name criterion. Entries are considered as follows: The contingent operations are performed for the specified entry number. After this is done, the PTHRED item is tested; if zero, there are no more PTAB entries to be considered. If PTHRED $\neq$ 0, then PTHRED specifies the next entry number.

Assignment of a new entry for a given name also requires a search of PINTAB. If the search is not successful, then the name is added to PINTAB, and the new entry number referenced. If the name is in PINTAB, then all the entries having that name are examined until the one having PTHRED = 0 is found. Then, this PTHRED is set to the new entry number, PTHRED of the new entry being set to zero.

If a search name is given as specified being identical with that of a given construct, the PTHRED items of PTAB must be examined to find the beginning of the list having that name. For all entry numbers smaller than the one specified, the PTHRED items are examined to see if they are equal to the one specified. If none such is found, the one specified is the beginning of the list having the same name; if one is found, the process is repeated for that entry number.

Tests

The operations described here provide a means of testing for conditions on numeric and scope items - which is to say, testing for conditions on the values of function or property registers. Contents of symbol registers may not be tested by this means.

The underlying purpose of these operations is to allow three types of manipulation:

1. Testing for conditions of "legality" of the language, and setting an "error flag" if such a condition is not met.

2. Effectively rendering PTAB searches conditional on tests of numeric properties.

3. Conditional performance of operations.

Two types of operations are used in performance of testing: a "delineation" operation TST, and a set of "conditional" operations (each of which tests for a certain condition). The TST code is used to delineate the set of conditional operations which actually comprise the test, and also the set of operations whose performance is dependent on outcome of the test.

Every test delineated by a TST operation is comprised of a set of "condition" operations, each of which tests for a given numeric or scope condition. The set of numeric conditions used is .

GR ↔ >

EQ ↔ =

NQ ↔ ≠ ;

Scope conditions will be discussed later in this section. The set of results obtained from such a set of operations defines a test "value";

1 ↔ successful

0 ↔ unsuccessful

Each conditional operation has such a value, and these values are logically "anded" or "or'ed" together, as specified by the operation. Thus, each condition operation specifies

a. A numeric condition on functions or properties

b. Desired manipulation of the test value -- whether "and" or "or" is desired.

The total test value is thus an ordinary Boolean function of the individual test values, and is computed in this fashion.

The individual operations can now be presented.

<u>TST</u> - Perform the sequence of condition operations which comprise the specified portion of IFTTAB. Compute a test value as specified by these operations. If the final test value is 1 (test is successful), perform the operation following the TST code; if the final value is 0, skip the following operation.

AGR - Test if $E_1 > E_2$ (see table of operation types); "and" the test result.

OGR - Test if $E_1 > E_2$; "or" the test result.

AEQ - Test if $E_1 = E_2$; "and" the test result.

AEQ - Test if $E_1 = E_2$; "or" the test result.

OEQ - Test if $E_1 = E_2$; "or" the test result.

ANQ - Test if $E_1 \neq E_2$; "and" the test result.

ONQ - Test if $E_1 \neq E_2$; "or" the test result.

ASI - Test if E_1 is "interior" to E_2; "and" the test result

OSI - Test if E_1 is "interior" to E_2; "or" the test result.

Miscellaneous Operations

This section discusses the "print" operations PRS and PRN, which are intended to "log out" messages and the operation CAP.

PRS - Print or log out:

a. Program Line Number

b. Message whose number is specified

c. Symbol string in Sxx, or
Number in Fxx, or
Name associated with a given PINTAB entry.

Note that a search through PTAB (similar to that when a name is implicitly given) and also a "backward" search through PINTAB are required to obtain a name.

Also: set an "error flag" to indicate that an "illegal condition" for this language has occurred.

PRN - Print out as for PRS; do not set error flag, since an illegal condition has not occurred.

CAP - Call (transfer control to) the "auxiliary processor" AUXPR. The intent of this operation is to allow use of the subprograms of CM1 for purposes other than compilation and to facilitate expansion of the set of internal operations. The number given as NUM2 is used as a numeric index for an AUXPR switch, to choose the correct auxiliary routine desired.

D. CODE PROCESSOR

The code processor performs the simple function of transferring operation codes from the code table (CODTAB) to the code string (CODCOR), at the same time substituting proper arguments. A simple macro-block capability is incorporated, and CODPR maintains a "push-down list" of "return points" in order to expand these properly.

CODPR also outputs conditional blocks, as discussed in Section III-C. The mechanism for this is extremely simple. A CON code causes evaluation of the specified condition. If the value of the succeeding VAL code successfully compares with the condition value, the block following the VAL code is output. Otherwise, the VAL code itself indicates the location of the next VAL code.

SECTION V

TRANSLATION FROM BASE LANGUAGE TO MACHINE LANGUAGE

A. INTRODUCTION

This section will discuss the methods of construction of the translation (BASE→1604), which is denoted T04B, and the translator (BASE→1218), which is denoted T18. It has been possible to construct these translators to perform their processing via machine-independent algorithms, indicating machine dependencies by the contents of a data table. Thus each translator consists essentially of a basic program (which might be called a "translator model") in conjunction with a set of machine-dependent data.

Both of the translators produce, instead of machine binary language, instructions in the formats respectively of the two assembly languages. This procedure was followed for several reasons: first, both a "BASE-to-binary" translator and an assembler must construct dictionaries, and later assign machine addresses for the dictionary entries. Also, current assemblers produce code in consonance with their standard operating systems, and may even produce code sequences to optimize use of machine facilities. Hence production of assembly code appeared to be more desirable than production of binary code. It should be possible ultimately to bypass the first pass of the assemblers, using only the assembler second pass, but thus far this has not been attempted.

It has been felt that BASE language operations, besides being couched in macro notation, could be directly translated by a macro assembler, and preliminary investigation had shown that macros for this purpose could in fact be written. However, it was found that present macro assemblers lack certain crucial facilities - the ability to set and use attributes of data variables, and retrievable auxiliary data storage - without which such macros become very tedious, lengthy and inefficient. Consequently, the translator model algorithms have been programmed directly in JOVIAL, and it is felt that the efficiency of these algorithms is much higher than the potential efficiency of a macro processor.

B. ORGANIZATION AND DESIGN

Inputs to the translator model are 256-entry blocks of BASE language code, each entry containing an operation code, operand type, and operand. As noted previously, the output from translation is assembly code.

The translator is organized into a main program (which consists primarily of a set of program switches), major subroutines GET and INSTR, and some small subroutines.

The program switches within the main program are based on the type of BASE language operation under consideration, and (when necessary) on operand type and accumulator type. Data-defining operation codes cause formation of dictionary entries, and corresponding data definitions in assembly code format are output at the end of translation. Several of the operations (e.g. SBR) require the use of a retrievable auxiliary data storage. Such use is indicated at the appropriate program switch points. Decision within the translation model to output code and/or form a dictionary entry and/or store information in auxiliary storage is always made via the same mechanism - transfer within the program to a "switch point" corresponding to the operation, operand or accumulator type under consideration. This mechanism is of course independent of the specific operations which are considered. The GET procedure obtains successive entries of BASE language code when initiated by the main program, and lists these entries on a "debug listing".

The actual code conversion - from machine-independent form to machine-dependent assembly language - is performed within the procedure INSTR. To perform this conversion, two tables containing machine-dependent information:

FORMAT table - each entry is a particular sequence of output characters, into which the procedure inserts data operands (also in the form of character sequences)

DRCTOR table - "director table"; each entry of this table specifies

(1) a particular character sequence within the format table

(2) number of data operands

(3) for each operand, a position within the desired character sequence.

The number of operands which may be specified in a DRCTOR entry is at present limited to 5. If desired, the translator output can be preceded or succeeded by assembler control cards.

It whould be noted that this mechanism is completely general for the case of conversion to assembly format. This is because assemblers accept only a limited number of different formats, which can be specified using the FORMAT table. The mechanism allowing insertion of specific operands then guarantees that any legal assembler format can be genenated with appropriate operands.

SECTION VI

REFERENCES

1. Gilbert, Hosler, and Schager. Automatic Programming Techniques, RADC-TDR-62-632, Final Report for Contract AF 30 (602)-2400.

2. Gilbert, Hosler, and Earnest. Automatic Programming Techniques. (Phase I), RADC-TDR-63-563, Final Report for Contract AF 30 (602)-2924.

3. P. Gilbert. " On the Syntax of Algorithnic Languages," to appear in Journal of the ACM.

4. P. Naur (Editor). "Report on the Algorithmic Language ALGOL 60", Comm. of the ACM (May 1960).

5. E. T. Irons. "A Syntax-Directed Compiler for ALGOL 60", Comm. of the ACM (January 1961).

6. S. Ginsburg and H. Gordon Rice. "Two Families of Languages Related to ALGOL", Journal of the ACM (July 1962).

7. A. E. Glennie. "On the Syntax Machine and the Construction of a Universal Compiler", Technical Report No. 2, July 1960, Carnegie Institute of Technology, Computation Center.

8. N. Chomsky. "On Certain Formal Properties of Grammers", Information and Control, vol. 2 (1959), pp. 137-167.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Teledyne Systems Corp
12525 So. Daphne Ave
Hawthorne, Calif.

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE
AUTOMATIC PROGRAMMING TECHNIQUES

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final Report

5. AUTHOR(S) *(Last name, first name, initial)*

6. REPORT DATE
July 1966

7a. TOTAL NO. OF PAGES
112

7b. NO. OF REFS

8a. CONTRACT OR GRANT NO. AF30(602)3330

b. PROJECT NO. 4594

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
None

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
RADC TR-66-54

10. AVAILABILITY/LIMITATION NOTICES
This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC (EMLI), GAFB, NY, 13440.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Rome Air Development Center (EMIRD)
Griffiss Air Force Base, New York 13440

13. ABSTRACT

This report is the third of a set of three reports documenting work in the area of automatic compiler generation. The first two reports described the theoretical basis for such a system. This report documents an operating system embodying the concepts described in the first two volumes. The system described in this report allows a programmer to write in FORTRAN, ALGOL, or JOVIAL and produce object code for either the CDC 1604B or the UNIVAC 1218. The system described oan be expanded to incorporate other machines or languages.

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
COMPILERS						
PROGRAMMING LANGUAGE						
FORTRAN, ALGOL, JOVIAL						
AUTOMATIC PROGRAMMING						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriat), enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specifi: reporting period is covered.,

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as dry, month, year; or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBEK OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBEF: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through

_______________________________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through

_______________________________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through

_______________________________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional

UNCLASSIFIED

Security Classification