A Base For The Definition Of
Computer Languages
(Richard K. Bennett)

SIGNATRON, Inc.
research and consulting
area code 617 • TEL. 862-3365
MILLER BUILDING • 594 MARRETT ROAD • LEXINGTON, MASSACHUSETTS 02173

A BASE FOR THE DEFINITION OF COMPUTER LANGUAGES

by

Richard K. Bennett

October, 1967

This research was supported in part under Contract F44620-68-C-0007 by the Air Force Office of Scientific Research, Office of Aerospace Research, United States Air Force.

TABLE OF CONTENTS

	Page
ABSTRACT	
1.0 INTRODUCTION	1-1
2.0 BACKGROUND	2-1
2.1 Definitions	2-1
2.2 Evolution of Software and Languages	2-2
2.2.1 Machine Languages and Assemblers	2-2
2.2.2 User Languages and Compilers	2-3
2.3 Current State of Software and Languages	2-5
2.4 Background of the Present Research	2-6
2.5 The Need	2-8
3.0 TECHNICAL APPROACH	3-1
3.1 The Language Base	3-3
3.1.1 Characters and Symbols	3-3
3.1.2 Reading vs. Writing Rules	3-4
3.1.3 Symbol Delimitation	3-6
3.1.4 Symbol Meaning	3-7
3.1.5 Quote Forms	3-8
3.1.6 Syntactic Statement Forms	3-8
3.1.7 Argument Interpretation	3-11
3.1.8 Recursion	3-12
3.2 Definitional Capabilities	3-12
3.2.1 New Data Types	3-13
3.2.2 Structure/Element Reference	3-14
3.2.3 Constants	3-14
3.2.4 Complexity Level	3-15
3.2.5 New Converstion Routines	3-15
3.2.6 New Operations for Existing Operators	3-16
3.2.7 New Operators	3-16
3.2.8 Controlled Diagnostics	3-17
3.3 The Software Base	3-17
3.3.1 Structure of Software Base	3-19
3.3.2 Coding the Translator	3-21
3.3.3 Hand-Translating the Translator	3-21
APPENDIX I	A1-1
APPENDIX II	A2-1
BIBLIOGRAPHY	B - 1

ABSTRACT

This presentation investigates and describes certain promising ideas for the design of computer languages and translators. By isolating and utilizing the fundamental elements of computer languages and the fundamental mechanisms of assemblers and compilers, this approach provides a very flexible and powerful base for defining languages and their translators.

Throughout the approach, the user's needs are emphasized in a studied and analytical manner. The approach seeks to give the user facilities that really help him and place at his command many devices and facilities which today are available only in separate, special-purpose languages.

The purpose of this approach is to achieve simplification, unification, and standardization of existing languages so they may all be built on one universal base and may all be handled by one general translator. At the same time, this universal base and its translator will encourage the orderly and rapid development of many new powerful languages tailored to special areas. These new languages would greatly reduce the programming effort required in these areas, could help solve the programming bottleneck, and could further the development of some of the really challenging applications of the digital computer.

SECTION 1

INTRODUCTION

A basic approach to the design of computer languages and their translators is presented. This approach will give programmers, at all levels, greater power than present systems and should unify the structure of languages to a point where one general translator would handle all language requirements.

This method is based on the identification of certain fundamental elements common to all programming languages and translators (i.e., assemblers and compilers). This approach was developed, over the past several years, as the outcome of dealing with the practical production of large, varied, and complex programs and was further refined by specific research into this area.

An attempt at achieving flexibility, power, and extendability has been and continues to be the motivating factor of our work on programming languages and translators. For example, DECAL (for the PDP-1 compiler) — the earliest implementation of this point of view — provided the user with the ability to define new operators. This feature was not tacked on; it was the method used in building the original operators of the compiler/assembler.

The essence of the study in question is to provide the user with a universal base for computer languages, so that any number of extendable languages may later be created. In the following text, this base is referred to as BUILD (Base for Uniform Language Definitions).

BUILD is not itself a language, but rather a set of ground rules and primitives upon which a class of languages can be defined. The definitional process is much the same to the user as the process of defining a set of subroutines.

BUILD is backed up by a general 'software base' which encompasses all of the program mechanisms which are currently in use. These mechanisms include the manipulation of

strings, structured lists, partial words, logical information, tables, arithmetic expressions, etc. By using suitable definitions, these mechanisms are combined to create the capability to translate any language built on the language base.

By attacking the man-to-machine communication problem at the fundamental level, the definitional capabilities are provided in a simple and natural way. Statement syntax has been simplified to a point where the definer in general does not have to think about it. He specifies only the names of his new operators or functions, what they are to do, and the parameters they require. In defining the new operators, he may call on any previously defined operators and facilities, whether defined by him or others.

It is believed that this basic approach to language and translator design is the best way to attack the general programming problem, that it promises greater power and utility for computer users of all types and levels than provided by current approaches, and that it can lead the way to the solution of today's software confusion. When this approach is put into general use and programmers gain experience with its capabilities, they will build up a library of language facilities that will enable them to write large-scale programs in a fraction of the time it now takes. The programming bottleneck would be broken and the door opened to more rapid progress on sophisticated computer applications. The BUILD approach lays a sound basis for an orderly advancement towards achieving more powerful facilities.

SECTION 2

BACKGROUND

2.1 Definitions

It may be helpful at the outset to define several terms which are used in a special way in this presentation.

Language: Any programming or user language which is used to communicate from man to machine.

Language Base: A set of rules for defining a class of languages.

Translator: A computer program which accepts as input a communication in a "language," interpreting the meaning of the communication, and carrying out the actions requested. The usual task of a translator is to produce a binary version of a computer program which will perform the functions indicated in the input. An assembler, a compiler, and an interpreter are special cases of a translator.

too broad (every program is a "translator")

interpreters are different in some very basic ways (but maybe we are ignoring even these)

Character: The basic element of input.

String: A sequence of characters.

Symbol: An entity represented by a (short) string of similar characters, for example: ABC2, or **.

Mechanism: The data-manipulative device created by a particular programming technique, for example: a symbol-table search.

Expression: A sequence of symbols representing operators and operands, together with parentheses to indicate intended groupings. The operators are generally binary, with the operator appearing between the two operands, according to the so-called "in-fix" notation. Operator-operand association is controlled by rules of "precedence" and parentheses. An expression is essentially the FORTRAN or ALGOL arithmetic expression, generalized to include non-arithmetic operators and non-numerical operands.

2.2 Evolution of Software and Languages

In order to place the current investigation in its proper perspective it may be useful to review the background and evolution of programming languages and their translators.

2.2.1 Machine Languages and Assemblers

In the early days of the digital computer, programmers coded their programs in numerical form. In the process of writing their programs, however, they found it helpful to use mnemonic symbolic codes for the machine instructions and data addresses. After a program was finished, the data addresses were assigned values by counting memory words, and the program was hand translated to numerical words acceptable to the machine.

It was soon recognized that this translation was a job for the computer itself, and assemblers were born. The first assembler writer was undoubtedly a programmer who wearied of the job of hand-translating his programs. Little attention needed to be given to language considerations. The assembler simply read a sequence of symbolic machine operation codes, each followed by a symbolic or numeric address.

It was soon found that information other than instructions was required to control the assembly process and to introduce data at assembly time. This was accomplished by including in the 'language' symbolic 'pseudo-operations,' which were written just like machine operations, but which did not produce binary output, but rather caused some assembly-time function (such as resetting the simulated location counter). The 'syntax' of the pseudo-operation statements was generally made the same as for ordinary (machine) operations.

Not much attention was given to the details of language design. As assemblers were written for different computers there seemed to be little reason to be concerned with the form of the language. The rules of input followed what the assembler writer thought would be easiest to implement and/or easiest to use. Each writer had a different view, and consequently a variety of different and often conflicting rules resulted.

For example, some assemblers used a variable-field card mat, with the fields separated by a blank or comma. Others used a fixed-field format. Still others determined the parts of the instruction word implicitly — for example: caa2 meant "clear and add" (ca) the word at location "a2" (no separation between "ca" and "a2" was required). The use of the space or blank varied widely. In some assemblers it was ignored; in others, it was not permitted. In between these extremes, the blank could serve as a symbol delimiter, but be otherwise ignored.

2.2.2 User Languages and Compilers

Somewhere in the mid-fifties, the interpretation and translation of algebraic equations was tried and proved feasible. Underlying the experiment was the concept that computer users should be able to write their programs in a language similar to their usual notations, rather than in the strange language of the computer.

FORTRAN was the first such language for engineers and mathematicians, which was made generally available. One of the obvious decisions of the committee designing it was to avoid the mistake of some of the earlier assembler designers in subordinating the user's interest to the effort of implementation. Thus the design of the FORTRAN language was undertaken primarily to help the user with little regard to how this language would be later translated.

FORTRAN, however, has proved beneficial in several important ways. First, it has reduced costs of mathematical programming by at least a factor of two and, in some cases, even ten. Second, it has resulted in a much wider use of the computer by the scientific community. Third, it has established the importance of user-oriented languages.

Unfortunately, the FORTRAN language has some serious design flaws which limit its usefulness and, more important, its extendability. Some of these flaws are widely recognized and have been eliminated in subsequent languages (but not in later versions of FORTRAN). Others are not generally recognized and have been perpetuated.

FORTRAN compilers (that is, software programs to translate user-programs written in the FORTRAN language) have been written using a variety of techniques. The first FORTRAN compiler was written by IBM for their 704 at a cost of about 25 man-years effort and 200 hours of computer time. It consisted of about 25,000 instructions.

FORTRAN compiler writers have found some of the syntactic complexities of the language difficult and awkward to cope with. Nevertheless, compiler-writing has progressed to the point where FORTRAN compilers can now be turned out typically with less than one man-year of effort.

Despite the general success and acceptance achieved by FORTRAN, many users have found that it was not suitable for their problems. Due to the lack of extendability in the FORTRAN language, these users were forced to develop their own languages and compilers or to use those developed by others.

Over the years many languages were designed and translators (compilers) built for them on one or more computers. Each of these languages has certain features and characteristics which distinguish them from one another, but most of them overlap in capability. In general, in spite of occasional similarities, these languages are incompatible with each other and encompass a variety of conflicting rules. Furthermore, the design of most of the languages has resulted in complex and awkward syntactic structures.

Consider an example illustrating the difficulties encountered when the statement structure is syntactically complex.

```
DO 15   I = 1,15
```

This FORTRAN statement is a well-formed statement. However, is the following well-formed?

```
DO 15   I = 1.15
```

(The difference is that the comma has been replaced by a period.) Surprisingly enough, this second statement is also well-formed. By removing the blanks, which FORTRAN totally ignores, it is seen that the second statement is the arithmetic assignment statement

DO15I = 1.15

The syntactic complexities here require the translator — or the user for that matter — to read as far as the comma or period before it can be decided whether the statement is a DO or an assignment statement. Thus, with this structure, even a small change can cause confusing interpretation, and can also make language extension extremely difficult.

2.3 Current State of Software and Languages

The many dozens of assembly and compiler languages developed during the last few years constitute today a "tower of Babel" problem, which appears to be getting worse rather than better. Each user must pick among the available languages on the particular machine he is using. In almost every case he must select a language which is fixed. If he does not select FORTRAN, he may not find the language available on other machines, should he wish to transport his program.

In general, the user must stick with one language for his program. Occasionally, the user is permitted to combine both assembly-language and FORTRAN or COBOL subroutines at load time. But the user cannot combine the capabilities of the several languages which may be available on his computer; their compilers are distinctly separate programs.

We can illustrate what this means to a programmer or computer user. Let us take, for example, string substitution. The user should be able to say in effect: Wherever you see "xyz", substitute in its place "rx5,(". An experienced programmer will recognize that such an ability is of proven value in special

areas. In fact, several string manipulation languages (e.g., SNOBOL, TRAC) and their compilers have been designed and built. However, this ability should be generally available to all programmers and users, not as a separate language, not as an imbedded language, but as a basic capability, which they can use together with all the other facilities they now have or would like.

2.4 Background of the Present Research

The BUILD approach is an outgrowth of earlier work in assemblers and compilers. This work was directed to providing more powerful programming tools and to improving the design of compilers.

Skeleton Compiler Approach

The early approach to compiler design involved the construction of a skeleton which possesses the ability to read input, handle symbols, and recognize basic operators. These operators permit the definition of machine instructions and of other operators. Thus, one can construct a compiler from its own input.

The resulting compiler can be extended at any time by continuing the definitional process. Also, one compiler can be replaced with another by substituting a second set of operator definitions.

Although the approach has since evolved to a point where the skeleton concept is no longer considered important, the use of definitional operators has been continued and extended.

Action Operators

The basic method of defining and extending the compiler is the use of Action Operators. An Action Operator (AO) is a symbol which, when encountered by the compiler, causes an action to take place immediately. The action occurs at compile time and is thus a compiler action.* Since the action occurs as a result of a symbol in the input program, AO's provide a very general method

*The pseudo-operations (assembler-commands) in an assembler are essentially built in Action Operators.

of compiler construction. AO's are defined very simply. The compiler is given the command to define an AO by calling the AO definer (itself an AO) and giving it the name of the new AO, together with the coding of an associated routine. This routine will later be executed upon every encounter of the name of the new AO.

DECAL

The practical application of this skeleton approach occurred in 1960 with the construction of DECAL[1] for the Digital Equipment Corporation's PDP-1 by the author. DECAL is a one-pass combined assembler and algebraic compiler which retains the open-ended capability of the skeleton.

The skeleton approach made it possible to write a powerful compiler with a very modest effort. Only the "skeleton" of 1197 instructions was coded by hand. From this skeleton, DECAL was written in its own language using Action Operators.

Use of BE-FAP Macros

The method employed in BE-FAP[2] to realize its macro facility is character string substitution. The great power of this macro facility was used to extend the capabilities of BE-FAP, when writing a large-scale simulator. The conditional assembly and symbol concatenation features of BE-FAP permitted the realization of major new facilities within the FAP language. By this means he was able to construct compile-time threaded lists and provide for partial-word manipulation.

The simulator employed these new facilities to great advantage. The program was far more flexible and error-free than would have been possible without these facilities. Furthermore, the facilities were so constructed as to produce code as efficient as hand coding.

However, it was found that, as powerful as the string substitution mechanism may be to construct new facilities, some facilities cannot be practically realized by this method, and others can be realized only with great effort and at great cost in compile time. For instance, in realizing the threaded lists, macros were nested as deep as fifty levels, and the compilation of the simulator required over one-half hour of IBM 7090 computer time.

In most cases, Action Operators would have realized these functions much more efficiently and with less programming effort.

SET

SET[3] was an extension of the experience with DECAL and BE-FAP, applied to the IBM 7094. To the basic capabilities of DECAL, SET added the string substitution facility of BE-FAP, in a completely generalized form. In addition, major improvements were made in the processing of symbols by "type," and a very general and powerful approach to the handling of arguments was introduced. Also, several general facilities were added to provide for the use of threaded lists, tables, fields (partial words), and dispatches. These facilities were integrated with the basic translator mechanism.

2.5 The Need

The need today is for a complete reappraisal of the language and software situation. Old methods must be critically reviewed to determine their good and bad parts, and new methods must be developed. In viewing the old, it is clear that user languages have done much to help the user. However, it is also clear that they have at the same time introduced a great deal of confusion. The existing languages and compilers certainly have overlapping capability, and much can and should be done to simplify and unify them.

There are two general approaches in vogue today which purport to be improving the situation. One of these is to build

translators to translate all of the many existing languages. The second one is to design a universal language (e.g., PL/I)[4] which will satisfy the needs of all users, hopefully for all time.

Unfortunately, neither of these approaches is in fact solving the problem. The first approach perpetuates the problem by retaining the existing set of languages, with their attendant flaws, complexities, and incompatibilities. At best this approach tries to accommodate the problem — it does not clarify the picture nor really help the user. It does, however, recognize the need for more than one language.

The second approach does not recognize this need. It fails to realize that the great variety of existing and future applications requires a corresponding variety of languages. One language, no matter how good, cannot hope to accommodate all needs.

The need today is to develop a new approach which could simplify, unify, and standardize existing languages, so that they could all be handled by one general translator, and at the same time provide for an even wider variety of languages than exists today.

SECTION 3
TECHNICAL APPROACH

A careful analysis of the devices and mechanisms which give existing languages, assemblers, and compilers their power has resulted in an understanding of the fundamentals of these processes. The BUILD approach draws upon this understanding to establish a language base and a software base, on which any number and variety of languages and translators may be built.

The languages constructed upon this base possess a sufficient richness and variety that they can encompass all of today's programming requirements, while at the same time providing the framework for accommodating as yet undefined needs. Since all the languages would be built on the same base — the same set of rules — the languages would be consistent and compatible. And since the base would be general, a host of new languages could be readily defined to cover many areas not properly serviced today by existing languages.

The BUILD Approach vs a Universal Language

The BUILD approach differs radically from that of developing a universal language. The universal language provides just one language, which has a fixed set of capabilities. In constrast, the BUILD approach establishes only the rules for the languages (and even these can be changed). Thus the BUILD approach allows, and actually encourages, the creation of an unlimited number of user languages.

The best way to see the difference between these two approaches is to compare language facilities to subroutines. The BUILD approach, which establishes a base for language definition, is like an extendable library for subroutines. The library is a framework in which subroutines can be stored and made available. The library may be originally equipped with one or more sets of subroutines, but the user can also add his own.

In contrast, the universal language approach is like a fixed library of subroutines. Although the set of subroutines may be made as complete as its designers know how, the library is fixed and the user cannot add to it.

Language Definition

Now, this comparison goes further than merely illustrating the openness of the BUILD approach. A language built on this base is actually defined by the set of subroutines (or algorithms) which perform (or specify) the functions of that language, for the purpose of translation. Another set of subroutines in the library would translate another language. The user can introduce his own set to give him his own language or can add to a given set to extend one of the existing languages.

The library would be originally equipped for translating several standard languages, some of which would look much like the languages in current use. Later in this section we give an example of how FORTRAN could be re-cast to permit its simple translation.

Relation of Languages and Translators

This view, that a language is defined as the set of algorithms which translate it, reveals the fundamental relation between languages and their translators. A language is defined as a set of algorithms, and its translator consists of those subroutines which implement these algorithms. In this way, the elements of a language base are related to the mechanisms of the translator and are really two sides of the same coin. This fact can be visualized by considering the names of the mechanisms as the verbs in a language, or conversely, the verbs in a language as the names of the mechanisms which translate the language.

In the sections that follow, languages and translators will be discussed separately, although they are merely different views of the same process.

3.1 The Base for Language Definition

The BUILD approach to the design of a particular computer language is to view this design as a small part of the much larger problem — the design of all computer languages. This approach starts with a set of general rules and definitional capabilities, which serve as a base for such languages. (Current practice, in contrast, is to design each language separately, with separate — and often conflicting — rules and conventions.)

Briefly, the base consists of the following components and rules:

1. A character set.
2. Symbols as groups of characters.
3. Symbol-delimiting rules.
4. Assignment of symbol meaning.
5. Quote forms.
6. Syntactic statement forms.
7. Argument interpreting rules.
8. Recursion.
9. Definitional capabilities.

These rules are described in detail in the following subsections.

3.1.1 Characters and Symbols

The purpose of a (computer) language is to provide the user with a precise means of unambiguously stating his commands and introducing his data to the computer complex. The English language is not good for this purpose, since it is not precise and since it cannot be translated by computer. However, there are certain aspects of natural languages which are useful in developing human-engineered artificial languages. The principal ones are the use of the alphabetic characters and the use of 'words.' A 'word,' it should be noted, consists of a particular group of characters, which is assigned a meaning that is independent of the meaning of the constitutent characters.

In BUILD, the first task is to establish the character set. The user would have the ability to introduce new characters. This will permit using the language base in any computer environment. The user could group and employ the characters as he wishes, although preferred groupings are recommended.

The second task is to establish the rules for forming the 'words.' We should state here that instead of the term 'word,' we use the term 'symbol,' in keeping with the terminology of assembly languages. Therefore we will speak of the rules of symbol formation.

3.1.2 Reading vs. Writing Rules

At this point, it would be well to pursue a question which may well lie at the root of the philosophical difference between our approach of building languages on a language base and the current practice of designing languages from a supposedly user's point of view.

We were just talking about symbol formation — that is, how one forms a symbol as he writes his program. Later, we will turn this around and talk about symbol delimitation, that is — how the translator delimits symbols, as it reads the input character string. The question is what is the relationship between reading rules and writing rules, and which should be treated as basic.

This same question comes up later when we discuss statement syntax. Should we develop the rules from the point of view of reading or of writing — and what is the difference.

It is believed that the reading rules whould be treated as basic, since in the last analysis the user must write in such a way that the resulting text has the desired meaning when it is read and interpreted according to the reading rules.

If we treat the reading rules as bisic, we will design the rules on this basis and end up with simple reading rules. For example, we might say that symbols are delimited by blanks and by characters in classes different from that of the characters

in the symbol. As another example, we might say that if a symbol is of the type, Subroutine, it will be checked to see if it expects an argument list, and if so, the next Quote* will be interpreted as an argument list.

Giving the reading rules makes it easy for the user to determine exactly what his text means. Similarly, the translator rules — being identical to the reading rules — will be simple, and the translator will be fast. Since the language rules and the translator rules are now identical, the link between a language and its translator becomes easier to see and understand.

In contrast, if we treat the writing rules as basic — in the name of being user-oriented — we find the picture inverted. When reading, we must look at all writing rules to see which ones apply. In some existing languages, we may have to follow different writing rules for some distance until all but one fails to fit what was written. The FORTRAN example in Section 2 is a good example of this. Here the writing rules are almost in conflict — that is, it is almost possible to be able to write two different statements with different meanings to look identical We do not know what was meant, in the statement in the example, until we hit the comma (,) or period (.).

Starting with a set of writing rules, then, makes it difficult and confusing to interpret the text. It is even possible to design a language which has inconsistent writing rules. At best, translation is time wasting, and the design of translators becomes a matter of controversy. Different people build translators in different ways, with different properties, for the same language. Also, a new set of translators must be built for each new language.

However, by working with the rules for reading, we achieve simplicity of expression and interpretation from both the user's and the translator's point of view. The resulting simplicity permits the unification of many existing forms of expression, including the integration of the assembly and compiling functions, the call of open subroutines (macros) and closed

---

*The term Quote is defined later.

subroutines, and the call of run-time* and translate-time* functions.

3.1.3 Symbol Delimitation

The most important task in establishing a solid base for language design is to delineate a simple, symmetrical, and general approach to symbol delimitation. By symbol delimitation we mean the breaking up of a character string into a sequence of symbols.

In considering the rules for symbol delimitation, we have emphasized the following factors:

1. Human. The rules should result in text that looks natural to the eye. The symbols (words) that the computer uses should be the same ones that the eye sees.
2. Simplicity. The rules should be simple and consistent.
3. Symmetry. The rules should apply generally to all characters.
4. Precision. The rules should be precise, with no possible chance of ambiguity.
5. Flexibility. The rules should permit change within wide limits.

These factors have led to a simple set of rules (given in the Appendix) which permit the delimitation of a symbol in the input string by reading ahead no farther than one character beyond the end of the symbol. Thus, the precision requirement of Point 4 is achieved, for if we had to read ahead any distance (as in the FORTRAN example of Section 2) in order to delimit a symbol, we open the door to ambiguity.

It should be pointed out that these rules provide essentially what most programmers would consider to be the normal interpretation of their text. We have simply codified and generalized the rules which are followed by most assemblers and compilers (FORTRAN excepted).

---

*The terms 'run-time' and 'translate-time' are defined later when discussing the translator.

3.1.4 Symbol Meaning

The "meaning" of a symbol is divided into two parts: Type and Specification. The Type might be Variable, Operator, Subroutine, etc. (The Type, Variable, is further broken down into the various data types, such as: Integer, Real, Complex, Double-Precision, Array, etc.) The Specification is the further information which is specific to the particular symbol in question. For some types of symbols, it might be a numerical equivalence; for others it may include a considerable amount of descriptive information.

Some of the symbol types may appear as the first, or principal, symbol in a statement of the prefix* notation form. For these symbols the Type, together with the Specification of the particular symbol, specifies the syntax of the remainder of the statement. For example, a symbol of type Subroutine would carry in its Specification the information as to the number of arguments expected and their names, modes, and 'default' values. (A 'default' value is the value to be used if the user 'defaults' by not giving a value for an argument, when calling the subroutine.) This information would then determine the syntax of the remainder of the statement — in this case, the argument list.

To achieve the desired clarity, we have found it important to require that the first symbol in a statement be pre-declared. In this way, there is no question as to the type and syntax of the statement, either from the user's or the translator's point of view. Thus each principal symbol carries a definite meaning and establishes the syntax of the remainder of the statement which it introduces. This is in contrast to the currently popular approach where it is often necessary to analyze the syntactic structure of a statement in order to properly interpret it.

---

* The prefix notation syntactic form is discussed later.

Pre-declaration has become a controversial subject. To many, it is an unnecessary requirement. However, pre-declaration is necessary for complete generality and precision in a language, but special rules can be introduced into special purpose languages to relax the requirement. In this way the user could have automatic declaration of symbols, say, where it would save him time and yet would have precision of expression when he needs it.

3.1.5 Quote Forms

There exist several instances where a piece of the character string must be marked for the purpose of quoting, or grouping, the contained material. For example, character strings must be quoted for macro skeletons, macro arguments, subroutine arguments, and subroutine argument lists. Grouping (bracketing) is required for expressions in an algebraic statement and for statements in a compound statement. Current usage generally employs different forms to accomplish these several purposes. The Quote Form described here has been developed to accomplish all these purposes with one unified form.

To provide the user with ease for simple cases and generality for more complex ones, three alternate forms of quotation are provided to serve the purposes for all quotation and grouping (bracketing). The user may pick the one which best suits his needs in each case. A precise definition of a Quote and of the three forms is given in Appendix 2.

3.1.6 Syntactic Statement Forms

It is not possible, either now or in the foreseeable future to use unrestricted English as a computer language. Since we must restrict ourselves, it would be well to start with simple syntactic forms and gradually progress to more complex forms.

As we ascend the scale of complexity we should always ask critically what the additional complexity buys for the user. However, by and large, languages have been designed with little regard for their syntactic structure. Linguists have concentrated on the resulting complex syntactic structures, for the understandable reason that syntactically-complex computer languages do in fact already exist.

Let us look at some simple statement forms. The simplest form is:

Action Argument Argument ... (1a)

The "Action" is the name of an action, function, etc. Or it may be a word or symbol which directly implies the action. For example, ØRG, BSS, DO, IF, GOTO. An argument (parameter) may be a number, symbol, word, expression, string, list, etc. Or an argument may be a complete statement. This latter possibility produces a recursive definition, implying unlimited nesting.

Since the arguments may in general be complex in structure, let us separate them with commas:

Action Argument, Argument, ... (1b)

We can also put a comma between the Action and the Arguments and enclose the whole statement in parentheses, or we could just enclose the parameters in parentheses:

(Action, Argument, Argument, ...) (1c)

Action (Argument, Argument, ...) (1d)

The above are several manifestations of the "prefix" notation form. All of these are employed today some place in the many compilers and assemblers. They all state an action to be performed, together with the parameters which make specific an otherwise general action.

Analysis shows that the simple statement form (1d) above (or any of its listed variants) can actually accommodate almost all requirements that exist today. The principal exception is the <u>expression</u>, which requires the so-called "in-fix" notation:

Argument Action Argument (2)

where the action is an operator and the arguments are the operands, which may be complete statements, of the same form. This definition is recursive, permitting the usual algebraic equation form (assignment statement) of FORTRAN, ALGOL, etc. Operator precedence and parentheses rules control the association of operands with operators.

The two statement forms (1) and (2) might be mixed. As an example, the Function in FORTRAN is a prefix form, used as an argument of an in-fix form.

Because complex syntactic structures are in such general use, some people may fear that simple syntactic forms may be inadequate to specify all operations. However, the simple statement forms (1) and (2), recursively nested, permit the convenient expression of all types of statements, including compiler facilities, assembler facilities (pseudo-ops), the introduction of data, on-line commands, executive commands, user commands, etc. Of course, more-complex syntactic forms can accomplish the same functions. But it is by no means clear that more-complex syntactic forms actually benefit the user. As a matter of fact, no case is known where the two simple form (1) and (2) would not provide the user with a facility as good as or better than that provided by the use of a more complex syntactic form.

To illustrate how the BUILD approach would utilize the basic forms to simplify existing languages, consider the FORTRAN DO statement. In present FORTRAN, one would write

```
      DO 15 I = 2, 17, 3
      Statement 1
      Statement 2
15    Statement 3
```

in order to iterate the three statements for successive values of I, ranging from 2 through 17, in steps of 3. In our approach we would simplify or standardize the syntax so that the user would write

```
DO (I, 2, 17, 3)
(Statement 1
 Statement 2
 Statement 3)
```

to accomplish the same results as above. Here we have used the prefix form for stating the main action and its four parameters.

The 'range' of the DO statement is one Quote, or statement (in this case, compound). This method of specifying the 'range' is neater and more precise than giving the statement number (15), as FORTRAN presently requires.

3.1.7 Argument Interpretation

BUILD provides extensive facilities for argument interpretation. Each argument may have a name, a default value, and an argument-interpretation mode.

Argument names are provided so that the user may identify arguments by name instead of by their position in the argument list. This feature makes it unnecessary for the user to remember

the order of the argument list and it is also quite valuable in those cases where a user may be giving one or two arguments out of a list of many. This feature is optional.

Default values are provided so that the user may omit stating those arguments which have standard values.

Argument interpretation modes are provided to allow various types of arguments. These modes include run-time expressions (as in compiler languages), translate-time expressions (as in assembly languages), structured lists (trees), strings, symbols, and several others.

When the user defines a new facility, he specifies the mode of each of its arguments. In addition, he gives each argument a name and the default value. During definition, the user may omit any information, in which case standard modes, etc. will be used.

3.1.8 Recursion

Complete recursive capability will be assumed for all BUILD rules. (By recursion is meant the process wherein one type of expression may be nested within that same type of expression, or where a subroutine may call itself.)

3.2 Definitional Capabilities

The definitional capabilities are inherent in the BUILD approach. It will be possible to create even the "built-in" functions with the basic definitional capability. This approach, in addition to being neat and self-consistent, allows the user to modify or delete existing functions, as well as add new ones.

To employ any of the definitional capabilities, the user would call the appropriate defining operator. The information and format will be designed to be simple and natural to the

user, consistent with the flexibility and power desired. A "default" value will be provided for the 'standard' case, so that the user need give only information which is significant to him. The user will be able to change these default values, if he desires a different case to be standard.

3.2.1 New Data Types

The user will be able to introduce new data types. When considering the presently used and contemplated forms of data it becomes apparent that there are two factors involved. The greatest flexibility will be obtained by considering and handling these two factors separately.

Data Structures

The first factor is the structure of the data, such as arrays, triangular matrices, simple lists, structured lists (trees), etc. In addition to the built-in data structures the user would have the ability to define new structures. Each data structure would have a name for later reference, and rules for relating location in the structure to location in memory.

Data Types

The second factor is the type of the data. Data types will include those which can be represented by a fixed number of words of storage. For example, INTEGER might require one word; REAL, two; COMPLEX, four; DPCOMPLEX (double-precision), eight; etc.

The language base would provide the user with the means of defining a new data type by, for example, giving it a name and stating the number of storage words required. In addition, the user would specify the relative complexity of the new type (see below).

Association of Types and Structures

In addition to the simple data types, such as INTEGER, described above, any data type should be able to be associated with any structure to produce a new data type, such as INTEGER ARRAY.

3.2.2 Structure/Element Reference

A syntactic problem requiring investigation arises when the user wishes to be able to refer to both the elements of a data structure and the data structure as a whole. For instance, the user may wish to add matrices or elements of matrices. He might like to say

A + B

to mean the addition of matrices A and B, and to say

A(I,J) + B(K,L)

to mean the addition of their specified elements.

The method of distinguishing the two cases shown above (by whether or not subscripts appear) is not syntactically simple and may therefore introduce unanticipated later complications.

Another approach would be to introduce one (or the other) of the two forms by an operator. Our approach to symbol delimitation and definition allows the operator to be either alphabetic or composed of special characters.

For example, the matrix as a whole could be referred to by, say, //, which according to our symbol delimiting rules (see Appendix 1) is a proper symbol. Thus, the first addition would be written:

//A + //B

3.2.3 Constants

After defining a new data type, the user should be able to state how a constant of this type will be recognized and how it will be mapped into a storage cell of proper size. Constants should be handled when they appear both in expressions

and in data-storing statements (c.f. the DATA statement in FORTRAN IV).

3.2.4 Complexity Level

When defining a new variable type, the user would, as suggested above, indicate its relative level of complexity. The purpose of the complexity level would be to order the data types so that mixed expressions could be handled simply and intelligently. For example, take the three types: INTEGER, REAL, and COMPLEX, in that order of complexity. In a mixed expression, the less complex type would be converted to the more complex. Thus, in the addition of an INTEGER and a REAL, the INTEGER would be converted to REAL and "real" addition would be performed.

3.2.5 New Conversion Routines

Mixed-mode expressions would be handled in a general way. The user would provide for the conversion of data from one type to another by defining those conversion routines which he wishes included in the compiler. In general, he would want to include at least two for each new data type. The first would convert from the next less-complex type to the new type; the second, from the new to the next more-complex type. In this way, the chain of conversions would be complete, going from any type to any more-complex type. Thus any combination of types could be used in mixed-mode expressions.

In addition, the user may wish to give conversion routines in the other direction, that is, from more complex to less complex (say, REAL to INTEGER). If he does, then the assignment statement could be handled with the less-complex type on the left side of the equal sign.

If the user does not equip a particular conversion — either because he forgets to or because he does not consider it meaningful — the translator would give a diagnostic when it encounters a mixed-mode expression which requires that particular conversion.

In addition to equipping conversions between data types of

adjacent levels of complexity, the user may wish to equip (for the sake of efficiency) other paths. For example, he may wish to give the conversion from INTEGER-to-COMPLEX. The compiler then would not have to perform the conversion in the two steps: INTEGER-to-REAL followed by REAL-to-COMPLEX.

3.2.6 New Operations for Existing Operators

The user should be able to define new operations for existing operators, that is, he should be able to equip an existing operator so that it can handle new types of operands.

The user would give the operator and the name of the operand types. Then he would state the operations in terms of any previous capability he had available, including existing variable types and operators.

After a user defines a new data type he should define the operations for that type of operand for all operators of interest. If he later uses the new data types with operators not so equipped, the compiler would give a diagnostic.

The meaning of the binary (in-fix) operators is dependent on the operand types. In general, the operator may have a different meaning — that is, perform a different operation — for different combinations of operand types. Ordinarily, only operations where the operands are of the same type are provided; the mixed operands would be handled by converting the simpler to the more complex.

The more general capability allows optimization. Thus the operator "+" may be defined for the combinations of operands: (INTEGER, INTEGER), (INTEGER, REAL), (REAL, INTEGER), and (REAL, REAL), if the user wishes to individually handle any or all of the mixed cases.

3.2.7 New Operators

The user should be able to define a new (in-fix) operator by giving its symbol (name) and its precedence. The precedence permits writing

A + B * C

without parentheses. (The "*" carries a higher precedence than the "+").

After the user has defined this operator, he would define the operations for the different operand types which he wishes this operator to handle, as described above under "New Operations for Existing Operators."

3.2.8 Controlled Diagnostics

As indicated above, the compiler should give diagnostics whenever it cannot match operand types with operations; either directly or by successive conversions. In addition, the user should be able to specify any combinations which to him are illegal and therefore should produce diagnostics.

3.3 The Software Base

In the BUILD approach, the base for language definition is backed by a general software base. This software base includes both the translating and operating functions. For the purpose of this discussion, we will treat the software base from the standpoint of its role as a translator. However, it should be remembered that the mechanisms of the translator are general and powerful enough to provide most, if not all, of the capability required for the other software functions.

The software base, in this approach, is a framework in which the basic programming mechanisms are embedded, together with certain special facilities built upon these mechanisms. In addition, it would have built-in definitional capabilities which would permit the user to create whatever facilities he desires.

The basic mechanisms are the primitives and include:

- String manipulation
- Partial-word manipulation

- Threaded-List Facility
- Table Facility
- Dispatch Facility
- Macro Facility
- Subroutine Facility
- Recursive subroutine calls.

These mechanisms form a basis, at a level above the machine instructions, for all programming. Consequently, they would provide for a translator of great power, including the power of existing assemblers and compilers.

The special facilities include those for the definition and manipulation of symbols and the handling of arguments, run-time expressions, and translate-time expressions. Symbols can be introduced to a central symbol table. A symbol can be defined by giving it a meaning, which includes a Type and a Specification. The symbol can later be retrieved, together with its meaning.

The facility which handles arguments is called the Argument Facility. It includes the machinery to set up argument descriptors, when defining a new function, and to use these descriptors to control the process of interpreting the arguments, when the new function is called.

The definitional capability would allow the definition of new machine instructions, new pseudo-operations (assembler commands), of new statements, new macros, new subroutines, new data types, new operators, and new operations.

This definitional capability, together with the primitives and special facilities, would back up the language base. As the user defines new functions, statements, etc., he would be, in effect, defining a new language. The language would thus be defined in much the same manner as a user would define macros to an assembler or subroutines to a compiler.

Of course, standard languages could be defined once and for all, to be made available to the user automatically. He could view these as fixed or as extendable, according to whether or not the definitional capability was removed before making the translator available to him.

3.3.1 Structure of Software Base

The basic structure of the software base is shown in the flow diagram of Fig. 1. The first operation of the loop is to read the next symbol, which is expected to be the first, or principal, symbol of the statement. The rules of symbol delimitation described in Appendix 1 are employed here. In addition, a string substitution mechanism is built in this box to provide a very general macro-like facility.

The symbol is found in the symbol table and a dispatch on its Type is executed. Each symbol Type has its own processing routine. Four examples are shown in Fig. 1. If the symbol is an Action Operator (AO), its arguments (if any) are read and the subroutine corresponding to the AO is executed.

If the symbol is a Subroutine (SB), its arguments are read and a calling sequence is generated for the object program. If the symbol is of Type, Word (WD), one memory word is generated. The WD type includes machine instructions, constants, packed words, etc.

If the symbol is a Variable (VA), the statement is assumed to be of the (nested) in-fix notation form. In this case, the routine which scans and translates arithmetic statements and expressions is called.

This flow diagram, of course, just shows the high-level logic of the input and interpretation process. The mechanisms and facilities are available as subroutines. The logic is simple, flexible, and very powerful, and it conforms with the design principles of the language base.

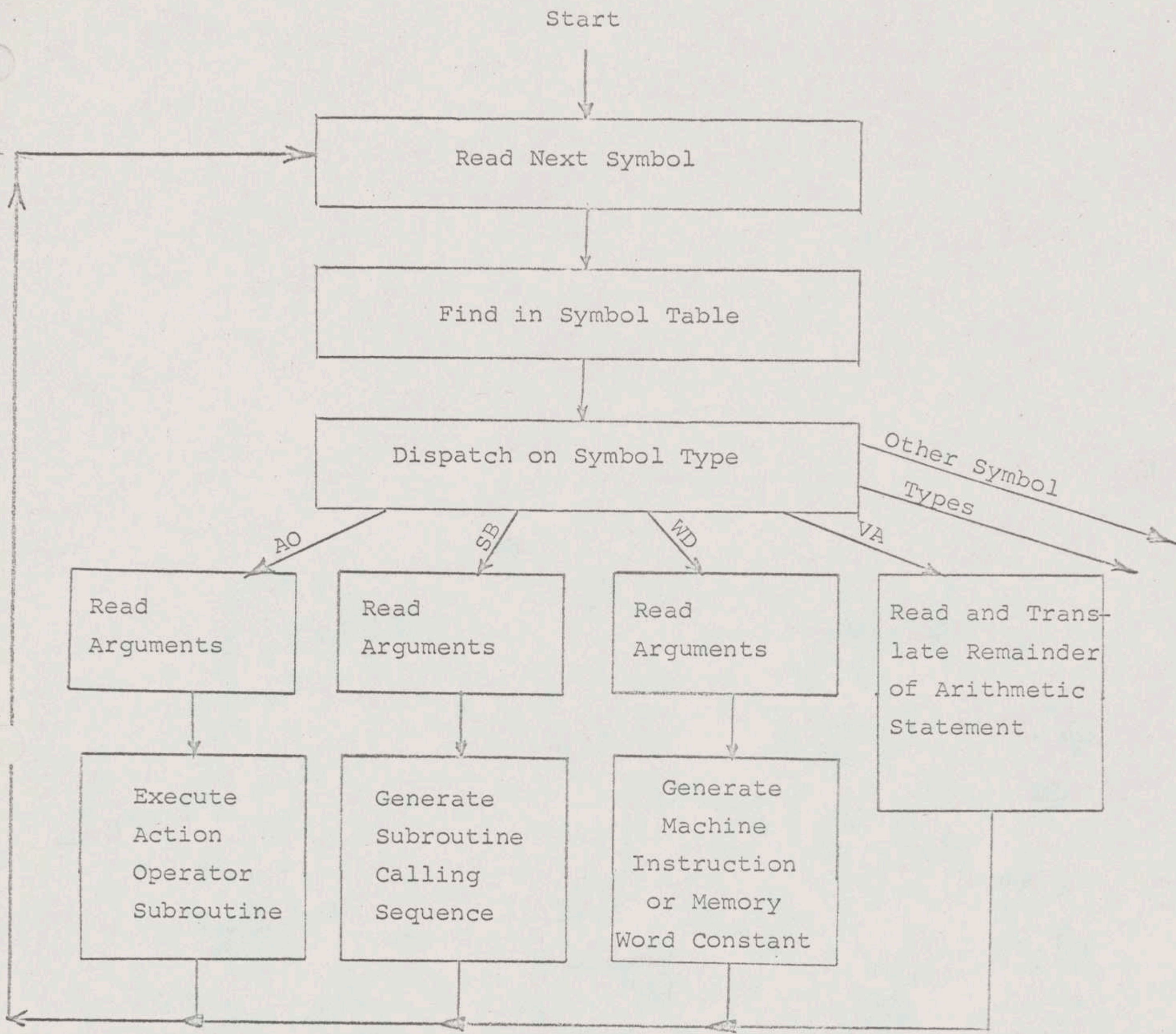

Fig. 1 Simplified Flow Diagram of Software Base

3.3.2 Coding the Translator

The translator — or more precisely, the translator portion of the software base — would be coded in a language defined on the language base. Those primitive operations which must be written in machine language will be coded for a particular computer. Where feasible these primitives will employ conditional-assembly features, based on tests on property lists which will be set up to describe the object computer. To the extent that this approach proves feasible, it will not be necessary to recode the primitives when later going to a different computer.

3.3.3 Hand-Translating the Translator

Since the translator will be written in its own language — that is, in a language defined on the language base — and since no translator presently exists for translating this language, it will be necessary to perform a hand translation of the translator into a language which exists for the computer to be used. This hand translation will probably be done to a machine (assembly) language.

For this task, efficiency, neatness, and flexibility of the resulting machine-language program are not at all important, since, once working, this version will be needed only once to translate the good version of the translator. The good version may then be used for all subsequent translations.

APPENDIX 1

SYMBOL DELIMITING RULES

In these rules, the input character string is considered to be field-free, and the End-of-Card (EC) (or carriage Return) is treated as a character. Each character is assigned a class number solely for the purpose of symbol delimitation. The character classes have no effect on the interpretation of the symbols, with the single exception that a concatenation of numerals is specially treated as a number.

The specific rules used in SET follow:

1. A Self-Delimiting character is one which always forms a one-character symbol. It never combines with any other character, including itself, to form a multi-character symbol.
2. Except for Self-Delimiting characters, characters of the same class concatenate to form a symbol.
3. A symbol is delimited by characters of classes different from the class of the characters composing the symbol.
4. Blanks (spaces) and EC are handled specially in that they delimit symbols for all character classes, but are themselves ignored, except for printing format.

As an example of the use of character classes, the assignment made in SET is given. This set will result in symbol delimitation very close to that performed by the human eye. The characters of the first class are self-delimiting.

1. (,)
2. A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
3. + - * /
4. $ ' . =

For example,

F(AB+C,D)

results in the symbols

F

(

AB

+

C

,

D

)

according to these rules.

APPENDIX 2

RULES OF QUOTES

A Quote (or Quotation) is a particular piece of the input character string, which has been quoted for the purpose of uniform handling. We have developed three methods of quotation which appear adequate for all uses: the normal, simple, and general methods. In all methods, the quote marks used to quote a character string are not themselves part of the Quote.

As long as no ambiguity is introduced, a Quote may contain another Quote, including (of course) the quotation marks of the inner Quote. This capability is recursive to any depth.

The <u>normal</u> method of quotation is to use matched parentheses as quote marks. This method is precise with regard to leading and trailing blanks and is completely general, except that it cannot quote strings containing un-matched parentheses.

The <u>simple</u> method of quotation is to use only a terminating comma. The Quote, then, starts 'here' and continues up to, but not including, the terminating comma. The purpose of this method is to allow arguments to be quotes, without requiring that each argument be enclosed in parentheses. Matched parentheses are permitted within a Quote of this form, so that an algebraic statement would be a Quote, but would need not to be enclosed in parentheses.

Let us consider a few examples of these two quote forms. We may quote "A + B" by either

(A+B)

A+B ,

We would normally use the latter form only in an argument list, such as

(X*Z, A+B, (Y,2))

In this example, we have three Quotes within a larger Quote. Removing the outer parentheses gives the larger Quote

X*Z, A+B, (Y,2)

The three inner Quotes are, of course

X*Z

A+B

Y,2

The next example shows the use of parens (the normal Quote form) to group two statements into a single compound statement.

```
(B = B+1
C = D*F)
```

In FORTRAN, statement grouping is not permitted. In ALGOL, this would be accomplished by the use of the statement parentheses BEGIN and END:

```
BEGIN    B = B+1 ;
         C = D*F
END
```

In our approach, there is no need to have two or more different forms for statement and other grouping purposes and then other forms for quoting purposes. Our one set of Quote forms serves all of these purposes.

The <u>general</u> method of quotation is to use labeled parentheses. The label is any symbol declared to be of Type, <u>Paren Label</u>. This symbol is then used to the left of an open paren and to the

right of a closed paren to create a set of matched parens. For example, to quote "B, (" we would write

L1(B, ()L1

where L1 is the Paren Label. This method is completely general, since the Paren Labels may be arbitrarily-chosen symbols.

In addition to its generality, the labeled paren provides the user with an unlimited number of <u>levels</u> of brackets for appearance or diagnostic purposes. In comparison, ALGOL has only two levels: the first is the BEGIN END statement parentheses and the second includes both [] and (), which are used for different purposes.