COMPUTATION CENTER

Massachusetts Institute of Technology
Cambridge 39, Massachusetts

To: P. M. Morse

From: J. McCarthy

Date: December 13, 1957

SUBJECT: A PROPOSAL FOR A COMPILER

ABSTRACT

This memorandum contains the first version of the first two chapters of a proposal for a compiler. Comments on the points raised so far and complaints about ambiguities are earnestly solicited.

CHAPTER 1

1. Introduction

The purpose of an automatic coding system in scientific computing is to reduce the elapsed time between the decision to make a computation and getting the results. It can make feasible computations which, without it, would be too complicated to undertake.

This report describes a proposed new automatic coding system which I hope will be a sufficient advance over those now available or soon to be available to justify the effort of writing the required translation program. The specifications for the system are presented in sufficient detail for evaluation of its merits, but would be subject to modification in the course of writing the translation program. A number of the ideas to be presented have been suggested by the Fortran system for the IBM 704, the proposed Scat system for the IBM 709, and the Flowmatic system for the UNIVAC. The source language is mainly independent of the machine being used, except that the provisions for referring directly to machine registers and their parts, which we believe must be included in any powerful source language, have been worked out only for the IBM 704.

In what follows, underlined terms are defined by the sentences in which they occur.

1.1 What is an Automatic Coding System

An automatic coding system has two parts. These are

1. a source language in which procedures for solving

problems can be described more conveniently than in machine language, and

2. a translation program which translates programs written in the source language into machine language.

Thus we have to do with three programs: the <u>source program</u> written in the source language, the <u>object program</u> which is the result of translating this program into machine language, and the <u>translator</u> or <u>compiler</u> which does the translating.

Programmers sometimes lose sight of the distinction between a problem and a procedure for solving it; this sometimes causes them to talk about having written a problem in Fortran. The distinction is important in deciding what it is possible to make automatic coding systems do for us. A <u>problem</u> is defined by a procedure for telling whether one has a solution, not by a procedure for getting one. For example, the problem of proving or disproving Fermat's last theorem in one of the known systems of formalized arithmetic is well defined since an alleged proof one way or the other can readily be tested, but there is no known procedure for getting a proof. The artificial intelligence problem is that of getting a procedure which is good at solving problems in general and is much harder than the automatic coding problem which is merely that of translating already formulated procedures from one language to another. The automatic coding problem may admit a fairly satisfactory general solution although we don't expect to achieve a fully general solution in this system.

1.2 <u>What Should a Good Source Language Be Like?</u>

It has often been said that if only we could program the calculator in English, the automatic coding problem would be solved. The English language has features which have not as yet been incorporated in any programming language and which programmers covet, such as a very rich vocabulary and provisions for introducing new terminology; nevertheless it is a priori no more likely that English is very well suited for describing complicated procedures, than it is that English is well suited for describing the theorems of an advanced branch of mathematics or the laws of physics. In fact, English is a very poor language for giving complicated instructions. Some programming systems for business use have been advertised as allowing the programmer to write in English. It seems to me that these claims are somewhat fraudulent. It is, of course, easy to make a system in which the instructions are English sentences. To take an extreme example, we could require the programmer to write "put the number in register 1000 in the accumulator" instead of "CLA 1000". However, to really be able to claim that English is being used as a programming language, one would have to be able to accept any reasonable synonym for a sentence, and even more important one would have to have the facility available in English of being able to define new terminology. One may hazard a guess, that were such a facility available, the programmer would quickly use it to establish a jargon that would look almost as incomprehensible to the uninitiate as the present programming languages.

It might be surmised that perhaps mathematics has already provided us with the symbolic tools necessary to describe procedures. This turns out not to be the case for two reasons. First mathematical symbolism is mainly used for the expression of declarative sentences; programming deals in imperative sentences. Secondly, defining new terminology is almost always carried out informally in the natural language, so that mathematics doesn't give too much help in this important problem.

For this reason, it seems most likely that a special symbolic language will be developed for the expression of procedures which will contain those features of the natural and mathematical languages which are the most valuable. This language will not be dependent on a particular calculator, although it will have facilities for describing calculators and taking special account of their peculiarities. It is not likely that the language will be as easy to learn to use as present computer languages, because one will be able to express in a primitive way concepts which are expressed in a very complicated way in present systems.

One may regard a programming language as a co-ordinate system in the space of procedures. From this point of view, we can see that one of the desirata for a language is that those aspects of a program one would most like to vary are expressed as changes in one or just a few co-ordinates. We shall call the various attributes of a program variables. These variables may, in a given system be divided into four categories: System variables, program variables, program segment variables, and computational variables. A system variable is one which can be changed only by changing the programming system, a program variable one which is set by the programmer and which does not change during the course of the calculation, a program segment variable is one which can be different for different segments of the same program, and a computation variable is one which changes its value during the course of the program. As Perlis emphasizes, a system can be more powerful than another simply by making a system variable in the one system a program variable in another. Some of the most important differences between this system and Fortran can be expressed as saying that certain attributes of a Fortran program which can only be changed by changing the system, are program segment variables in our system. Some of Fortran's program variables are program segment and even computation variables in this system. The simplest examples of this are that the kinds of arithmetic available and with them the meanings of the operation symbols are program segment variables since new kinds of quantity and new meanings for the operations can be defined within the system. The typographical conventions are also program segment variables. The statements themselves which are program variables in Fortran are computational variables here since the program can generate more source language program in the course of operation and can call in the compiler to compile it.

The source language is general enough to express the compiler itself. This will enable the compiler to be written in a sort of bootstrapping way wherein early inefficient versions are used to compile later more efficient ones with added features.

1.3 <u>Features of the Source Language</u>

The most important feature of the source language of this system is the freedom it gives the programmer to define new ways of expressing himself. This ability is provided by several features.

1. A type of statement called the equivalence statement which provides for the introduction of abbreviations for any kind of expression.

2. The translator starts with certain tables giving the relation between statements in the source language and the successive languages through which the translation goes. Much of the translation is accomplished by compiling tables comprising information taken from the source program. Either set of tables can be directly enlarged or altered by suitable source program statements. This of course includes the tables which determine how table alteration instructions are obeyed.

3. The above two features should suffice for most extensions of the language. However, in addition, certain points in the compiling program are accessible to the programmer in the sense that he himself can describe program to be executed at these points under appropriate

conditions, The writing of such program is made easy by providing convenient ways to refer to parts of a statement in various stages of translation and to entries in the tables.

4. The ability to describe a computation by giving final state of the machine in terms of the initial state without having to worry about intermediate changes to the variables used in the computation.

5. An extended set of basic quantities and operations compared to Fortran including fixed-point full words, logical words, and 1-bit quantities which play an especially important role in the system.

6. A direct way of handling propositions and predicates and conditional functions which eliminates much branching in the source program.

7. A large generalization of the concept of subscripted variable where the set of subscripts can be any ordered set and not just the set of integers. Subscripts in expressions can be arbitrary expressions.

8. A way of describing flow apart from the computation statements.

9. The ability to compile statements referring to lists and tables.

10. The ability to define functions and other open and closed subroutines in a powerful way.

11. The ability to refer to the machine registers.

12. The ability to compile statements which modify others.

13. The ability to compile interpreters and interpretive coding.

14. The ability to define one's own typographical conventions including the ability to define what is to be done in cases where nothing is stated. These conventions can be program or program segment variables.

Because the system as a whole has so many features it will not be as quick to learn fully as previous systems. However, simplified subsystems will be available, which will be easier to learn if less powerful.

The library tape of the system can contain not only open and closed subroutines, but also the sets of definitions for introducing new kinds of quantity or for defining simplified subsystems.

1.4 Objectives in Designing the Translator

Given the source language and the computer on which the object programs are to be run, there are a number of desirable properties for the translator. These include:

1. The object programs should be efficient. This system will carry out several kinds of optimization on the program including, taking calculations out of loops when possible, calculating common sub-expressions only once, straight lining parts of tight loops, deciding whether certain quantities should be recalculated or updated, deciding whether tables should be formed of certain auxiliary quantities, and finally, taking advantage of certain special situations.

2. It should be possible to impose constraints on the object program as to where it finds certain variables and what regions of storage it occupies. Other constraints may also help optimization.

3. The time required for compiling should not be excessive. This can be accomplished by putting less effort into optimizing the rarer parts of the program. This compiler will also have facilities for compiling very small programs entirely in high speed storage.

4. It should be possible to make small changes expressed in the source language without recompiling the whole program.

5. It should have good facilities for detecting as many errors as possible in the source program and printing out a complaint about all errors that can be found. If possible, the machine should go on to other work while an error is being corrected and then take up from where it left off rather than starting the compilation from the beginning.

6. It should make a report on the translation which should include the correspondences between the source program and the object program, changes the compiler has made in the source program for optimization purposes, the location of quantities in storage, information about the object program including lists of the instructions referring to particular storage addresses and the times required for all subcomputations for which this can be determined.

7. The compiler should fit into a complete system for operating the machine which should be so designed as to minimize the elapsed time between submitting a request for computation and getting correct results.

1.5 Plan of this Report

The next chapter, chapter 2, describes the kinds of compute staterents allowed in the system. Compute statements are those which cause new values to be computed for certain quantities. The important concept of non-recursive program segment which is a natural unit of program is introduced and discussed.

Chapter 3 discusses the statements which determine the flow of control. These include the conditional branches, indexing over ordered sets, and the algebraic way of describing flow separated from the computations.

Chapter 4 takes up the statements by which the language can be extended. These include a kind of statement called the equivalence statement which makes abbreviations and changes of notation easy, table entry statements which alter the tables used by the compiler in making the translation, and finally the facility for introducing program at strategic places in the compiling process. An example is given of how these facilities can be used to provide new kinds of quantity such as complex numbers or quaternions in terms of which algebraic formulas can then be written.

Chapter 5 takes up the manipulation of symbolic quantities such as algebraic formulas or statements in a compiler. This is important in itself for making the compiler do calculus and other symbolic computations and also because this kind of computation is performed by the compiler itself and hence will be needed in the boot-strapping operation of writing the compiler in the language of the compiler and using the simpler parts to translate the more difficult parts.

Chapter 6 takes up input and output.

Chapter 7 takes up the detailed design of the compiler and the facilities provided for optimizing programs and also the fitting of the compiler in an operator system.

CHAPTER 2

2. Quantities, Symbols, Compute Statements, and Non-Recursive Program Segments

This chapter takes up a kind of statement which is basic in any compiler and which we call the compute statement. Compute statements, which correspond in function to the arithmetic statements in Fortran, are compiled into program which computes new values for certain quantities. An example of a compute statement is

A = B + C/A.

The program compiled from this causes the expression on the right of the equality sign to be computed using the current values of the quantities denoted by the symbols A, B, and C. The result becomes the new value of the quantity denoted by the symbol A.

Before describing compute statements, we first discuss quantities in general, the symbols which are the handles with which we hold them, and the functional expressions (called algebraic expressions in Fortran) in functions, pseudo-functions and operations which describe the computations. The particular importance of propositional quantities is discussed. Finally, we introduce the new concept of non-recursive program segment. For many purposes including common sub-expression optimization by the compiler this is a natural unit of program.

2.1 Quantities

Previous compilers admit a fixed set of kinds of quantity. In particular, Fortran admits two: the floating point number and the integer of 15 bits plus sign. The present compiler admits an arbitrary set of kinds of quantity, since there is a process by which new kinds of quantity can be defined and used. The compiler language will have the important conservative property that the major kinds of expression which can be used with the kinds of quantities originally provided for can also be used with the newly defined kinds of quantity. In particular, functional expressions can be used with all kinds of quantity.

Basic to this compiler will be the two kinds of quantity allowed in Fortran and the full length fixed-point quantity, the full length logical word of the 704, and the one-bit propositional quantity. Other kinds of quantity can be defined in terms of the basic ones or else by giving the programs which define what the operation and function symbols mean when applied to these quantities.

In general, a type of quantity is defined by describing how it is represented in the machine and what operations combine quantities of this type with others of the same type and also with quantities of other types. We give some examples of kinds of quantity which may be used.

1. Multiple precision numbers
2. Complex numbers
3. Quaternions
4. Vectors
5. Clifford numbers
6. Functions represented in some way, either by a table, a formula, or perhaps by a sequence of expansion coefficients. More generally, elements of function spaces.
7. Strings of characters. This kind is especially important since the compiler itself functions by manipulating strings of characters.
8. Lists, described in the manner used by Newell, Shaw, and Simon. We shall have more to say about these later.

Quantities can be objects quite different from numbers such as algebraic and functional expressions, differential equations, shapes, colors, programs (in some particular language) or electrical networks. It is worth while to define a new kind of quantity if enough examples will occur in the program and useful operations can be defined involving quantities of this kind and other kinds. For example, the operations of simplification, substitution and differentiation with respect to a variable may be defined for algebraic expressions. An operation of solution might be defined for a class of differential equations. Operations of combination, identification of variables, and compilation might be defined for programs. Operations of combination might be defined for electrical networks as might operation of solution combining a network with initial conditions.

None of the above kinds of quantity will be explicitly provided for in the system, though once the statements defining them have been made, the definitions can be included in the library tape.

2.2 Symbols

We describe computations involving quantities by expressions in the symbols representing these quantities. The connection between a symbol and the quantity of quantities it represents is determined by conventions which in this compiler are usually program variables, but sometimes program segment variables, and even computation variables. In SAP symbols represent the numbers of storage registers and sometimes program parameters. That this is so is best indicated by the meaning of arithmetic expressions in the symbols. However, the asterisk (*) representing the current value of the location counter in the new SAP is an example of a symbol whose connection with numbers is quite different.

In Fortran a symbol represents the contents of a register except that a symbol used only as an index may never have a fixed home register. The meaning of arithmetic expressions in the symbols bears out this interpretation.

In the course of the later chapters, the reader will see that a symbol may be connected with the quantities it represents in quite a variety of ways.

Typographically, we shall allow sequences of letters and digits beginning with a letter to represent a symbol. We shall not make a restriction on the length of symbols and we will avoid system conventions such as that in Fortran that symbols beginning with I,...,N represent fixed point variables. We will, however, reserve tentatively special symbols for the contents of the machine registers AC, MQ, ILC, IR1, IR2, IR4, SL1, SW1, etc. By "tentatively" I mean that the programmer can reject this usage by an appropriate statement and keep these symbols uncommitted. The conventions defining a duffers' system might contain such a statement in order to keep the duffers uncontaminated by any actual knowledge of the machine. We shall give some examples of the use of the symbols for the machine registers later.

2.3 Algebraic Expressions and Simple Compute Statements

The points we want to make first are best illustrated by giving an example of a simple compute statement which is what Fortran calls an arithmetic statement. In our opinion the Fortran term prejudges the question of what such statements are good for. Our example is

```
A = A + B*C + COS(D)
```

This formula is an imperative to the computer to compile instructions that will replace the value of the quantity A by the result of evaluating the formula on the right side using the current values of the quantities represented by the symbols in it.

What is the advantage to the programmer of being able to write such an expression rather than the sequence of expressions

```
X = COS (D)
Y = B*C
Z = X + Y
A = A + Z
```

especially considering the fact the first thing the compiler does with the original formula is to translate it into something corresponding to the sequence of four elementary formulas? The following are some of the advantages:

1. This is the way non-programmers are used to writing

2. The programmer saves writing a number of characters. This has to be balanced against the fact that the program consisting of a sequence of elementary formulas is more easily changed than the single more complicated formula.

3. The programmer avoids having to invent the auxiliary quantities X, Y, and Z. We regard this last as the most important advantage because experience has shown that it is in the inventing and handling of auxiliary quantities that errors are most often made.

4. There is an additional advantage that the compiler can plan the sharing of temporary storage better than the programmer can.

The ability to make the output of one calculation the input of another without having to give the intermediate result any other name than the name of the calculation that produces it is of use in other than numerical computation. Certainly it is useful in describing symbolic manipulations as we shall show later in this paper, and we believe it will also be useful in data processing.

Algebraic expressions are obtained by combining the symbols representing constants, quantities, operations, and functions together with commas and parentheses as punctuation according to recursive rules which are too familiar to need repetition here. Just as in Fortran we shall use the symbols + - * / and ** to represent the elementary operations of addition, subtraction, multiplication, division, and exponentiation. We shall also want symbols for the elementary Boolean operations, and additional symbols for the elementary Boolean operations, and additional symbols are desirable. We shall also establish as tentative conventions the same seniority rules between the operation symbols. It should be understood that since functional notation is provided for, the operation symbols are a concession to custom; a worthwhile one in terms of the legibility of programs.

The calculations represented by the particular operation and function symbols depend on the kinds of quantity the quantity symbols in the expression represent. However, the first step in compiling a formula which transforms an algebraic expression into a sequence of elementary expressions, is independent of what the operations represent. It is only after this transformation has taken place that the rules established by the programmer which define the operations on his kinds of quantities affect the compilation process by determining the translation of the elementary algebraic statements. The translation rules may have several effects. First they may give rise to sequences of machine operations. Thus A = B + C may give rise to one of the four sequences

```
CLA A               CLA A
ADD B     ADD B     ADD B     ADD B
STO C     STO C
```

depending on the neighboring formulas. Second, a transfer to a subroutine may be compiled. Third, the elementary expressions may be replaced by complex expressions in symbols representing more primitive quantities. We do not discuss how the programmer indicates what kind of quantity a given symbol represents in this section.

2.4 Pseudo-functions

Programming has not yet reached a state where all kinds of calculations can be described with no regard at all for the fact that the machine has a storage which is divided into numbered registers. In this language we provide certain pseudo-functions which allow one to connect numbers with the contents of the corresponding registers. They are called pseudo-functions because while they compose like functions, the value of a pseudo-function of a number depends not merely on the number but also on the contents of the memory of the results of the assembly process. Here are a few such pseudo-functions:

1. CAR   CAR(X) denotes the contents of the address part of register number X. Thus CAR(3) is the 15 bit quantity stored in the address part

of register number 3. We have several pseudo-functions similar to CAR.

2. CDR   contents of the decrement part of register number

3. CWR   contents of the whole of register number

4. BITS(Y,Z,Z) denotes the Y-X+1 bit quantity in bits X through Y of register number Z. (This pseudo-function should be distinguished from the function EXBIT(X,Y,Z) whose value is the Y-X+1 bit quantity consisting of bits X through Y of the 36 bit quantity Z. It is related to the pseudo function BITS by BITS(X,Y,Z) = EXBIT (X,Y,CWR(Z)). Both BITS and EXBIT have their uses.)

Additional pseudo functions of this kind can be defined as system or program variables.

5. LOC(X)   This pseudo-function for those quantities for which it makes sense, gives the address of the first register assigned by the compiler for its storage. In the compiled program it will generally be a constant.

6. NAME(X).   This is mainly useful in input-output statement when X is an index which runs over a list of quantities. Its value is the string of letters used by the programmer to name the quantity. Its use can greatly simplify output statements.

2.5 Propositional Quantities and Functions

A propositional quantity is a one bit quantity generally associated with the truth of falsity of a proposition. The value 1 of the quantity is associated with the truth of the proposition and 0 with its falsity. This system provides a number of operations and functions which can be used to combine propositional quantities with each other and with other kinds of quantity.

First of all, we have the predicates = , < and ≤ which are used to compute propositional quantities from numbers. A predicate is a function which takes on the values "true" and "false" which here are represented by the bits 1 and 0. A typical example of a compute statement involving a predicate is

P = (A = B + C)

which calls for the quantity P to be replaced by 1 if the value of the quantity A is equal to the sum of the values of the quantities B and C. Notice in this statement the character = is used both as a predicate operation and as a symbol for the operation of replacement. We can probably get by with this dual usage, although if there were plenty of character symbols it might be worth while to use something like a left pointing arrow as a symbol for replacement and reserve the = sign for use as a predicate.

Secondly, we have the Boolean operations by which propositions are combined. The symbols for these operations are $\wedge$ for "and", $\vee$ for "inclusive or", $\sim$ for "not", $\oplus$ for "exclusive or", $\supset$ for "materially implies", and even $|$ for "not both". A typical statement using these operations is

$$P = Q \wedge ((A = B) \vee P)$$

Thirdly, for using propositional quantities to compute quantities of other kinds, we have the function IF. An example of compute statement involving the IF-function is

$$A = \mathrm{IF}(P,\ X + Y:\ Q,\ U + V:\ (A = B),\ A + B:\ \mathrm{OTHERWISE},\ R)$$

The execution of this statement causes the variable A to be replaced by X + Y if P is true. If P is not true and Q is true, then A is replaced by U − V. If neither P nor Q is true and A = B, A is to be replaced by A + B. Finally, if none of the preceding predicates is true, A is to be replaced by R

(If the IF and Boolean functions are to be compiled into efficient programs, the usual way of compiling algebraic statements, which involves computing all the arguments of a function before trying to compute its value, cannot be followed. Consider the statement discussed in the previous paragraph. If P turns out to be true, it is unnececessary to compute Q, (A=B), or the quantities corresponding to them. A similar circumstance holds in the case of the previous example. Namely, if Q is false, nothing else need be computed.)

Propositional quantities will play an important role in our later discussion of control statements.

Propositional quantities have not been explicitly used in computation as much as their importance warrants. This is probably because the machine facilities for dealing with them conveniently have not usually been provided.

It may be possible to introduce explicitly some propositional pseudo-functions which occur frequently in informal descriptions of programs. One example is "A has been done already" where A denotes a certain action.

2.6 Non-Recursive Program Segments and Compound Compute Statements

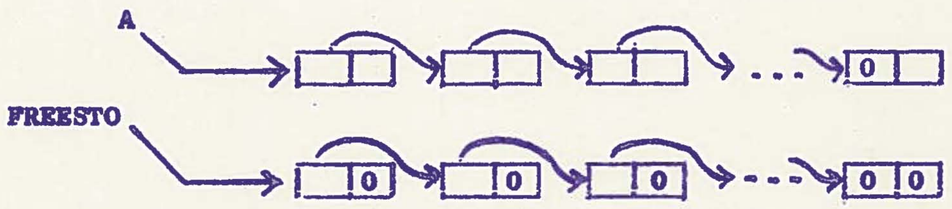
It is frequently possible, when planning a part of a computation, to regard the segment of program as changing the machine from a situation A to a situation B where the difference between the two situations is that certain quantities have new values in situation B. If each of these new values can conveniently be expressed directly in terms of the values of the quantities in situation A we say that we are dealing with a non-recursive program segment. We shall give three examples of non-recursive program segments.

1. A program to interchange the values of two quantities X and Y.

2. A program to perform one step of a prediction operation, in the solution of a system of ordinary differential equations by Milne's method.

3. The following operation with list structures which requires a digression to describe a method of storing lists which has been developed most fully by Newell, Simon, and Shaw in their Information Processing Languages. In that system a list consists of a number of machine words. In each word of the list is the address of the next word of the list as well as a datum. (This assumes that the length of a word is such that a word can contain an address and still have room for a datum.) In addition to the data lists there is a free storage list in which all the registers not filled with data are connected together. The situation is shown in figure 2.6.1 wherein an arrow from a symbol to a register indicates that the value of the symbol is the number of the register. In the case of the 704 we put the address of the next element of a list in the decrement part of a list register and put the datum in the address part. The last item on a list has zero in its decrement part.

Figure 2.6.1



The main advantage of such a way of handling list is when the length of a given list is a computation variable such that it is not feasible to assign enough storage permanently to each list to take care of the largest number of elements it may ever have. In addition it is convenient to insert items in the middle of such a list or to delete items from it.

The program segment we wish to describe dealing with these lists is needed when one wishes to insert an element at the beginning of a list, getting the register for this element from the free storage list.

The programs for the above three examples are all conveniently described by means of a compound compute statement and are given in Figure 2.6.2

Figure 2.6.2

```
1.    X   |Y
      Y   |X
2.    Y0P |A1*Y1+B1*Y1P+A2*Y2+B2*Y2P+A3*Y3+B3*Y3P
      Y1  |Y0
      Y1P |Y0P
      Y2  |Y1
      Y2P |Y1P
      Y3  |Y2
      Y3P |Y2P
3.  FREESTO          |  CDR(FREESTO)
    CDR(FREESTO)     |  A
    CAR(FREESTO)     |  B
    A                |  FREESTO
```

As can be seen from the examples, a compound compute statement consists of two columns. Corresponding to each quantity in the left column is the value it is to assume in the right column. The nomenclature of the quantities and their values are all assumed to be given in terms of the values of these quantities as of the beginning of the execution of the compound statement.

A more elaborate kind of compound compute statement is also allowed in which there are three columns: the quantity to be calculated, a condition, and a value. An example of this is given in figure 2.6.3.

Figure 2.6.3

G1	A	B>0	A+1
		B=0	A
			A-1
	B	(B<0) P	0
	C		C+1
	P	A<0	Q
	NEXT	A>0	G1

The name of the statement is G1. The first line states that if B>0, A is to be replaced by A+1. The second line states that if B=0, A is to be left as is, while in the remaining case, A is to be replaced by A-1. The next line says that if (B<0) P, B is to be replaced by 0. Since there are no other statements made about B it is assumed that if the above condition does not hold, B will be unchanged. The last line illustrates the use of another special symbol: NEXT denotes the next statement to be executed, and in this compound compute statement, we have that if A>0 the present statement G1 is to be executed again. If the condition is not satisfied the physically next statement in the program will be executed next.

This illustrates another possibility which will be more fully explored in subsequent chapters, the concept of the normal procedure. One can set up conventions as to what is normally done in certain situations when the program does not say otherwise. These conventions will be under the control of the programmer.

2.7 Universal Quantifiers

What calculations can be written as non-recursive program segments depends on the richness of the language and in particular on what functions and operations have been defined in the system. To take a trivial example, if square root function has not been defined in the system, then any program segment which requires the extraction of a square root is recursive. Of course, if a square root function is used in a compound compute statement, the method of calculating the square root will be taken for granted and will not be subject to further optimization in the compiling process.

In this section we present another of the concepts of compound compute statement which will enable more program segments to be written in this form. This extension changes the previous three column format to a four column one where in the additional column which is to be written on the left contains an index and a set over which the index is to vary. An example of such a statement is given in figure 2.7.1.

Figure 2.7.1

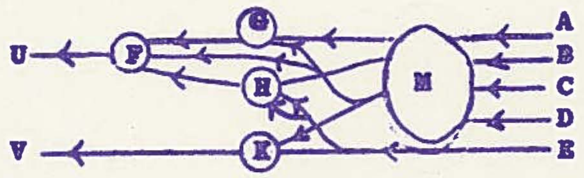
Quantifier	Quantity	Condition	Value
$I \in (1(2)11)$	A(I)		B(I)+C(I)
$J \in L$	A(B(J)+C(J))	B(J) > 5	R(J)*S(J)
	A	B > 0	A+1
$I \in (1 \text{ to } N)$ $K \in (1 \text{ to } M)$	C(I,K)		$\sum_{J=1}^{L}$ A(I,J)*B(JK)

The most obvious domain of variation of an index is a segment of the integers, but others are possible. For example, an index may vary over the elements of a Newell list.

2.8 Multiplet-Valued Functions and Their Composition

It is convenient to be able to use subroutines which take several inputs and produce several quantities as outputs. We shall call such routines multiplet-valued functions. (The multiply valued function in mathematics is something different. There the emphasis is on the ambiguity of the value rather than on the value being an ordered collection of quantities.) The problem of composing multiplet valued functions is best illustrated by the example shown in figure 2.8.1.

Figure 2.8.1



In the figure F, G, H, K, and M represent multiplet valued functions. For example, M has 4 inputs and 3 outputs. The arrows show the flow of data and the diagram represents a multiplet valued function with 5 inputs and 2 outputs which is a sort of composition of F, G, H, K, and M. It is tempting to try to devise a notation to represent this kind of composition and which will include the ordinary composition of functions as a special case, because if we can, we can write