A Description of Mercury Autocode in Terms of a Phrase Structure Language

R. A. BROOKER AND D. MORRIS
University of Manchester

THIS paper is a sequel to two earlier papers (An Assembly Routine for a Phrase Structure Language (Refs. 1 and 2)) which describe what amounts to an autocode for writing autocodes. More precisely, they describe a system whereby the user can specify by means of 'phrase definitions' and 'statement formats' the form of the statements (and their constituent expressions) to be used in the source language program; and by means of 'statement definitions' describe how these are to be translated in the target language. The system is self-expanding and allows the user to define the form and meaning of new statements in terms of previous statements as well as in terms of the 'basic statements'. The system is also recursive in the sense that the format patterns may include a repetitive element, for example, the user may specify that a phrase of a certain type may appear any number of times at some place in the format, or that in an algebraic expression involving brackets, the bracketing may extend to any depth. These are the brief particulars of the assembly routine described in the aforementioned papers.

It is the purpose of this report to describe how the Mercury autocode language (see *Computer Journal* **1,** Nos. 1 and 3, and *Annual Review* No. 1) which is largely a phrase structure language in the sense referred to here, can be described and defined in terms of such phrase definitions, etc. The reason for doing this is partly to illustrate the scope of the system, but also as a useful piece of work in its own right. It is hoped, in fact, to provide a translation program for MERCURY autocode tapes on

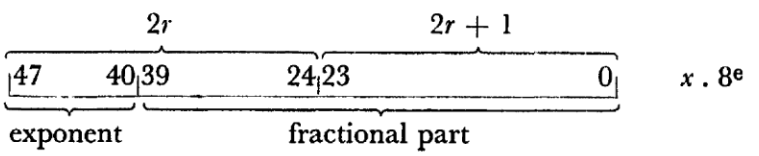
ATLAS. The only restrictions which arise are concerned with the facility in MERCURY autocode for 'breaking into' machine language. This is a computer oriented facility which to interpret correctly would mean constructing a digit for digit simulation program. Clearly this would be unrewarding both in terms of machine time and programming effort, and instead we propose to inform prospective users that if they wish to run their current Autocode programs on ATLAS, then they should avoid using machine instructions. Many forms of the machine instructions can be interpreted without ambiguity, and it is possible that we shall be able to relax these restrictions considerably in the final program. The 'statement definitions' of such machine instructions are of restricted interest, however, and we do not propose to present them here.

To understand the program below it is necessary to understand someting about MERCURY autocode, i.e. the source language, something about the target language, i.e. the order structure of ATLAS, and finally the meta-syntactical language of the assembly program itself. As already mentioned, MERCURY autocode has been described in several places, including the first volume of this Review. It is necessary to say something about ATLAS, however, since we are not yet able to refer the reader to an appropriate description. The assembly language will, we hope, be possible to follow after describing one or two statement definitions in detail; otherwise we must refer the reader to the papers mentioned earlier.

A SHORT DESCRIPTION OF 'ATLAS'

For the purpose of this report ATLAS can be regarded as a one-level store machine of 48-bit words addressed 0, 2, 4, . . ., or as a set of consecutive 24-bit words addressed 0, 1, 2, 3, 4, ... However, 48-bit words can only be selected automatically from locations $2r$, $2r + 1$, *not* $2r + 1$, $2r$.

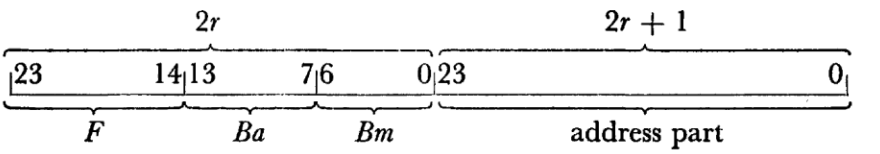
The instructions and floating-point numbers are each represented by 48-bit words, while 24-bit words are used to represent individual addresses, modifiers, etc. The structure of a floating point number is as follows:



where the most significant eight digits represent an octal exponent, and the remaining forty digits (the 'fractional part') give the precision of the

number in the form of one sign bit and thirteen octal digits. A number is in standard form when the fractional part lies in one of the ranges $-1 \leqslant x < -\frac{1}{8}$ or $\frac{1}{8} \leqslant x < 1$. The exponent lies in the range $127 \geqslant e \geqslant -128$. Automatic 'underflow' is provided, i.e. if as a result of an arithmetical operation the exponent becomes < -128 then the number is automatically replaced by $0{\cdot}8^{-128}$. If an exponent overflows, i.e. becomes >127, then a warning is given.

The structure of an instruction is as follows:



There are 10 function bits which provide, in theory, for 1024 operations. The two least significant digits of the address part are irrelevant and digit 2 is the 'units' digit. This allows for up to 2^{22} 24-bit words, but only half of this can be in the *main* store, which is available to the user. The other addresses relate to special forms of storage 'behind the scenes'. The bulk of this hidden store is referred to as the *fixed* store, and is used to store the library routines and *extracode* sequences (explained below).

The main store consists of a core store (with a cycle time of not more than 2 μsec) and a magnetic drum store, in amounts decided by the purchaser. The total number of 24-bit registers is theoretically limited to 2^{21}, but it is not expected that anyone will actually purchase this amount. The smallest amount of core store which is allowed for is 16,384 48-bit words. Arrangements are made so that whatever the proportion of core store to drum store, the whole can be made to look, with certain restrictions, like a large one-level core store. The means by which this is achieved is beyond the scope of this paper, and for our purpose we may assume a virtually infinite one-level main store.

The fixed store is constructed from ferrite rods in a woven wire mesh. The access time is approximately 0·2 μsec for reading only: it is not possible to alter the contents of the store except by manual means, which involves inserting and removing the ferrite rods by tweezers. There is a theoretical maximum of 2^{18} 48-bit registers, although it is anticipated that a store of 8192 words will be sufficient for most installations.

Associated with the fixed store is an erasible working store for the use of the routines in the fixed store. This is called the *subsidiary* store, and the standard size is 1024 words.

The *Ba* and *Bm* parts of the instruction refer to 128 *B*-registers which are separate from the main store. Most of the *B*-registers consist of 24-bit

core store registers with a cycle time of about 0·5 μsec, while the rest, which have special roles, are flip-flops.

The instructions fall into two classes: *A*-code instructions which relate to the *floating-point accumulator*, (the central arithmetic unit); and *B*-code instructions which are concerned with setting and adjusting the contents of the 128 *B*-registers.

In the *A*-code instructions the contents of *Ba* and *Bm* are first added to the *presumptive* address to give the *modified* address. The operand will usually be the contents of the 48-bit register thus referred to. In certain operations, however, the modified address may be interpreted in other ways.

Associated with the accumulator is an 8-bit octal exponent and a 79-bit fractional part. Operations are available for inserting and extracting information from the least significant 39 bits of *A*, but these do not concern us in the present application, and we shall regard *A* as referring simply to the most significant 40 bits and the exponent. There is also an accumulator test register of 2 bits only which record the sign of the accumulator and whether it is or is not zero.

In the *B*-code instructions *Ba* is used as a second operand, and the address is modified by *Bm* only. In some *B*-codes both *Ba* and *Bm* are used as operands, and in this case no modification takes place. Associated with the *B*-registers is a *B*-test register which, like the accumulator test register, can be set to record the sign and zero status of any *B*-register.

Certain *B*-registers are reserved for the use of fixed store routines and are therefore not generally available, i.e. information can only be left here at the owner's risk unless he is sure that any library routines which he is using do not use them, or alternatively that an 'interruption' will not arise. This refers to time-sharing features of the machine which are again beyond the scope of this report, but which it is appropriate to mention very briefly. The logic of the machine allows for peripheral equipment to be functioning while the main program is running, and only interrupts the latter when it is necessary to use the arithmetical units to assemble and dispatch information which has flowed into the peripheral buffers.

Certain *B*-registers also have a special purpose, most of which do not concern us here, except for *B*127 which is synonymous with the main control register, and *B*124 which is also the exponent of the accumulator. The latter in fact only contains 8 bits, the 16 most significant bits being zero. It is these *B*-registers which are constructed of flip-flops. B0 always contains zero.

As a result of the above provisions, only *B*-registers 1–80 are generally available.

The Extracode Instructions

In all instructions the function part is written in the form $B000$, where B denotes a binary digit 0,1, and the 0s are octal digits 0–7. They thus correspond to the internal representation of the function part. B is normally 0 unless the instruction is an *extracode* instruction. These are not built in instructions like the basic instructions, but instead call in an appropriate sequence of basic instructions in the fixed store. By this means it has been possible to introduce relatively complicated operations into the users' instruction code, while at the same time avoiding the corresponding complications in the 'hardware'. Except for the leading binary digit, extracode instructions have the same appearance and properties as basic instructions, and comprise both A-codes and B-codes. When the extracode sequence is entered, the main control ($B127$) is halted and the machine switches over to a special extracode control register ($B126$). On completion of the sequence main control is resumed in the usual way. The precise means by which the entry and exit is effected, and the way in which the information in Ba, Bm and S is made available to the constituent basic instructions will not concern us here.

The sort of operations for which extracode sequences are employed are as follows:

1. Instructions which are just too complicated to be included as basic instructions, e.g. discriminations involving 'greater than', 'modulus', etc.
2. The more substantial operations such as double length arithmetic, complex arithmetic, etc.
3. The elementary functions, i.e. sq. rt., sin, cos, etc.
4. Input and output operations. It will not be possible to enlarge on this aspect of the computer because it is very closely bound up with the peripheral supervisor system, which looks after the process of time-sharing. For this reason we cannot yet specify the forms which these extracodes will take.

SUMMARY OF ATLAS INSTRUCTIONS USED IN THE SEQUEL

(??? indicates that code digits have not yet been finalized).

Basic A-codes

0310	$A' = A + S$	Here A, A' indicate the contents of
0352	$A' = A \times S$	the accumulator before and after
0374	$A' = A/S$	the operation respectively.
0324	$A' = S$	Similarly for S, S' where S denotes
0311	$A' = A - S$	the floating-point number stand-
0325	$A' = -S$	ing in the register specified by the
0366	$S' = A$	(modified) address part.

The $A' = S$ and $S' = A$ instructions are straightforward copying operations and can be used for transferring 48-bit words from one part of the store to another. In the other operations the result is standardized and rounded. The precise nature of these operations, while relevant in some applications of the machine, will not concern us here.

A-extracodes

Two sets of extracodes will be provided, similar to the above, in which the operand is taken to be:

1. One half of the modified address itself, i.e. $\frac{1}{2}$ of (S + contents of Bm + contents of Ba). This allows the modifier indices (which are usually in 48-bit address units) themselves to be used as floating-point operands. The address is normally expressed in units of digit 2.
2. The floating-point number standing in the location following that of the instruction itself. The address part of the instruction is irrelevant. After the extracode sequence has been completed (main) control is returned to the instruction following this number. There is no equivalent of the '$S' = A$' code in this case.

A test codes

$0234 = 0$ $0235 \neq 0$ $1??? > 0$ $0236 \geqslant 0$	If the A test register satisfies the specified condition, then place n (the address part modified by Bm) in the B-register specified by Ba

Ba is usually taken to be $B127$ so that the effect is to transfer control if the condition is satisfied.

Basic B-codes

Here ba, bm, bt denote the initial contents of Ba, Bm, Bt and as before a prime indicates the final contents.

n denotes the modified address $S + bm$, and (n) the 24-bit content of this address.

0121	$ba' = n$	Some instructions for setting and altering B registers
0122	$ba' = ba - n$	
0123	$ba' = -n$	
0124	$ba' = ba + n$	
0127	$ba' = ba \,\&\, n$	
0101	$ba' = (n)$	

0170	$bt' = n - ba$	Instructions for setting the B test register
0172	$bt' = ba - n$	
0203	$ba' = n$ if $bm \neq 0$, $bm' = bm - 2$	These become conditional transfers of control if $Ba \equiv 127$. (The last is an extracode.)
0224	$ba' = n$ if $bt = 0$	
0225	$ba' = n$ if $bt \neq 0$	
0226	$ba' = n$ if $bt \geqslant 0$	
1???	$ba' = n$ if $bt > 0$	

Some extracode instructions

1. A set of ten extracodes for replacing A by $f(A)$ where f is one of the functions:

   sq. rt., sin, cos, tan, exp, log, mod, int. pt., fr. pt., sign

2. A pair of extracodes for the function:

   (1) $A' = \arctan(S/A)$
   (2) $A' = \sqrt{(A^2 + S^2)}$

   In (1) and (2) S will be double B modified in the usual way.
3. $A' = a_0 + a_1A + a_2A^2 + \ldots + a_nA^n$, where a_0 is specified by ba, and n is given by $S + bm$.
4. An instruction for the operation:

   $ba' = n + 2$ (*int pt A*)

5. A pair of instructions for transferring control to and returning from a subroutine. The subroutines may extend to any depth and for this purpose the links are nested in a set of consecutive 24-bit registers. The current position in this nest is kept in $B90$ which is reserved for this purpose. $B90$ is advanced and retarded by 1 (address unit) respectively for every entry and exit. The entry instruction ($E5a$) transfers control to n (i.e. $S + bm$). The exit instruction ($E5b$) causes control to revert to the original place in the program. The address part is irrelevant in this case.

THE NATURE OF THE TRANSLATED PROGRAM

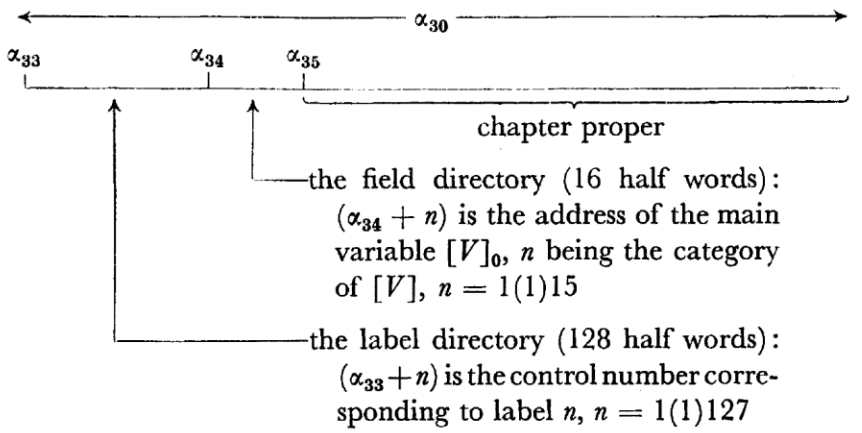
In this section we explain the general layout of the target program which is illustrated on the accompanying diagram. We have endeavoured to preserve a fairly close correspondence with the internal arrangement adopted for MERCURY itself. Thus each chapter of instructions is assumed to occupy a fixed allocation of consecutive registers. The chapters are stored in the space allocated to the auxiliary store and extend backwards from the end of this 'store'. What it is not possible to achieve is the

same ratio between the size of a chapter and the auxiliary store, which on the MERCURY is:

$$1 \text{ chapter} \equiv 512 \text{ auxiliary variables}$$

There, however, an instruction occupied only half the space of a floating-point number, and although the instruction code of ATLAS is considerably more powerful than that of MERCURY (especially with the aid of extracode) it is not powerful enough to render always in 440 ATLAS instructions the equivalent of 892 MERCURY instructions. Fortunately it is not necessary if an adequate span of storage is allowed for the combined material (i.e. auxiliary variables and translated program).

The α quantities given on the accompanying layout are used in the relevant statement definitions by the basic compiling instructions. During the execution of the translated program the index quantities $i, j, k, l, m, n, o, p, q, r, s, t$ are kept in B-registers 1–12 respectively. Each chapter is made up as follows:



α_{30} is the size of each chapter

α_{31} is the difference between absolute and relative (i.e. within a subprogram) chapter numbers of the current chapter

α_{32} is the absolute number of the current chapter

$\alpha_{33} = \alpha_7 - \alpha_{30}\alpha_{32}$ is the origin of the label directory of the current chapter

$\alpha_{34} = \alpha_{33} + 128$ is the origin of the field directory of the current chapter

$\alpha_{35} = \alpha_{34} + 16$ is the start of the chapter proper

α_{36} : at any stage in the assembly of a chapter α_{36}, $\alpha_{36} + 1$ give the (long) location of the next instruction

α_{37} : at any stage in the allocation of the main variables, i.e. execution of the directives $[V] \rightarrow [N]$, α_{37}, $\alpha_{37} + 1$ give the next available location.

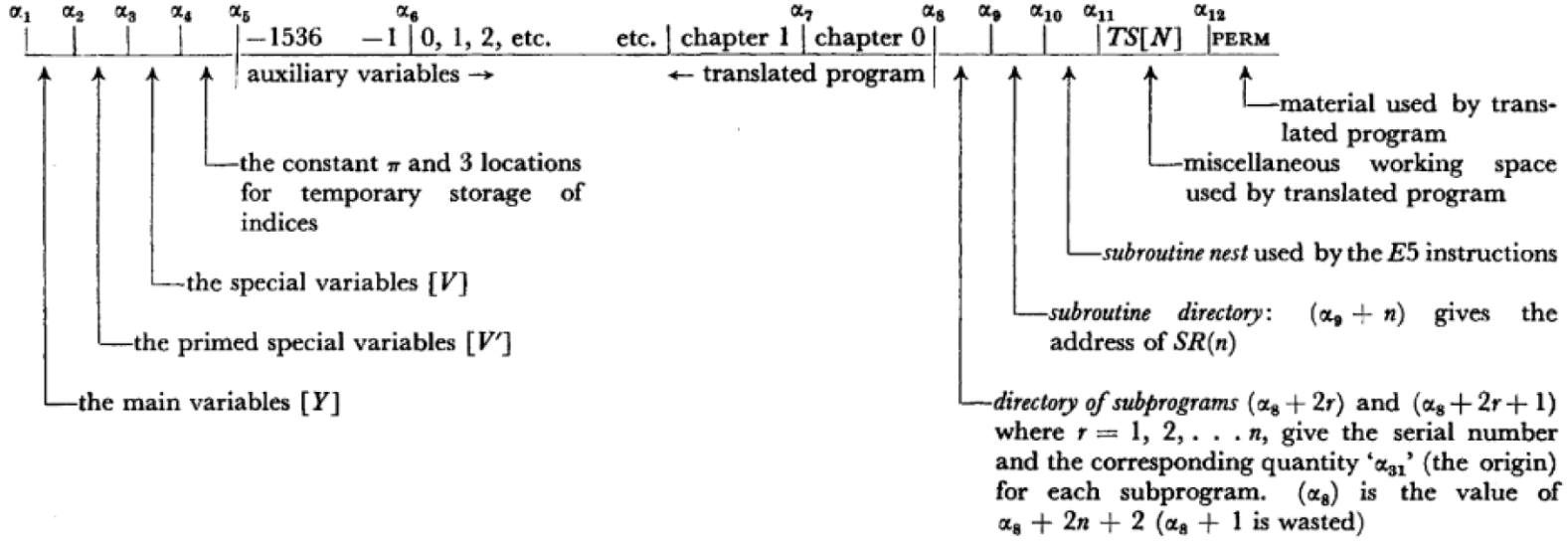


FIG. 1. Diagram showing layout of material involved in MERCURY autocode

In addition to the above the following α's refer to various *lists* and *nests* used in the process of translation. The terms list and nest are used here in the sense referred to in the earlier papers.

α_{21} refers to a double entry circular list giving the control number corresponding to local labels. Words are added to this list in pairs, the first being the numerical value of the label, while the second is the corresponding control number.

α_{22} refers to a nest giving the points in the target program at which reference is made to a local label. All references to local labels are filled in when the directive 'fill in local labels' appears. When each reference first appears the 24-bit location in which the corresponding control number will ultimately be placed is filled instead by the numerical value of the label. When the 'fill in' directive is encountered these locations are extracted one at a time from the nest, and then looked up in the list α_{21} to find the corresponding control number.

The labels dealt with in this way are 'local' labels, that is, labels introduced to facilitate the programming of the auxiliary material PERM and certain statement definitions.

α_{23} is similar to α_{22} but refers to the chapter labels, i.e. those which the autocode user himself uses. There is no equivalent to the list α_{21} because the functions of this are taken over by the label directory, which is part of each chapter and is needed during the execution of the program.

α_{24} is a nest relating to program cross references, i.e. jumps between one subprogram and another. Words are added to the nest in pairs, the first of each pair giving the location at which a presumptive reference must be corrected, while the correction itself is given in the second word of the pair.

α_{25} is the 'cycle' nest, and refers to the cycling instructions in MERCURY autocode which take the form, e.g.

$$i = 1(1)10$$

.

.

.

repeat

These translate into five instructions, three at the head of the cycle and two at the end. All five, however, are generated at the

beginning of the cycle, and the two which will ultimately be placed at the end of the cycle are recorded in the nest. Cycles within cycles up to eight deep are permitted on MERCURY itself, although this restriction could be relaxed on ATLAS.

THE METHOD OF DESCRIPTION OF A PHRASE STRUCTURE LANGUAGE

Before presenting the formal description of MERCURY autocode as a phrase structure language, it is desirable to say something about the method of description itself, although for details we must refer the reader to our earlier papers. The secondary or source language is described by means of the following primary statements: phrase definitions, statement formats, and statement definitions.

The phrase definitions

These are used to build up classes of logically similar phrases. To each class is assigned a name, the *class identifier*, which may then be used in further phrase definitions and statement formats to indicate that any phrase of the class in question may be substituted in its place. Class identifiers are represented by a string of characters enclosed in square brackets (e.g. $[IU]$ is the class identifier we subsequently assign to the alternatives: if, unless.) Thus, e.g.

phrase defn: $[V] = a, b, c, d, e, f, g, h, u, v, w, x, y, z, \pi$
defines the class $[V]$ as consisting of any of the lower case letters indicated. Similarly

phrase defn: $[V'] = a', b', c', d', e', f', g', h', u', v', w', x', y', z'$
defines $[V']$ as any 'primed' letter (except π').

Alternatively we could have defined both $[V]$ and $[V']$ in terms of an intermediate class $[a - h, u - z]$ say. Thus

phrase defn: $[a - h, u - z] = a, b, c, d, e, f, g, h, u, v, w, x, y, z$ followed by

phrase defn: $[V] = [a - h, u - z], \pi$
phrase defn: $[V'] = [a - h, u - z]'$

It is conceptually simpler, however, to write out both definitions in full and in fact is advantageous to keep the 'depth' of a definition as small as possible.

In MERCURY autocode language $[V]$ is the class of names of *special variables* and $[V']$ the *primed special variables*. The fact that there is no π'

is connected with the layout of the material in the high speed store of MERCURY. For the present purpose we just have to accept it as an instance of the many awkward features that are likely to occur in practical autocodes.

The phrase definitions following $[V]$ and $[V']$ in the formal description should be fairly clear to the reader who has used MERCURY autocode. Thus $[\pm]$ means simply 'plus or minus'. $[Y]$ is the class of names of floating-point variables. $[Q]$ is the class of operands which the user may write in a general arithmetical expression. $[F]$ and $[G]$ are similar to $[Q]$ and $[Y]$ respectively but include the expression $TS[N]$ referring to a set of temporary storage locations. These are used in certain auxiliary statement definitions leading up to the definition of the general arithmetic instruction $[Y] = [GE]$.

$[N]$ and $[K]$ are 'built in' classes and describe the form of an integer and a general constant respectively. Thus if

$$[D] = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

then $[N] =$ an arbitrary sequence of Ds

and $[K] = [N], [N]., .[N], [N]. [N]$

However, because they are built-in they do not have to be defined.

Two features which are often present in the form of a phrase or statement are *repetitive appearance* of an item and *optional appearance*. In order to describe these situations we introduce the qualifiers * and ?. Thus:

$[A][B^*][C]$ means $[A][B][C]$ or $[A][B][B][C]$ or $[A][B][B][B][C]$ etc.
$[A][B?][C]$ means $[A][C]$ or $[A][B][C]$
$[A][B^*?][C]$ means $[A][C]$ or $[A][B][C]$ or $[A][B][B][C]$ etc.

Although convenient these qualifiers are not essential and the above classes could be defined by means of *recursion* and the 'nil' class, thus

$[B^*] =$ (in order of preference) $[B][B^*]$, $[B]$
$[B?] =$ (in order of preference) $[B]$, nil
$[B^*?] =$ (in order of preference) $[B^*]$, nil

These are in fact how the * and ? qualifiers are handled behind the scenes.

The reader may have noticed in the foregoing examples that no commas occurred in the expressions listed as the alternative forms of a phrase definition. Since the comma is used as a primary separator we denote the appearance of a comma in the secondary phrases by [,]. Similarly for space $[SP]$ and end of line $[EOL]$. These special forms cannot therefore be used as class identifiers.

Statement formats

These are similar to phrase definitions except that they describe one phrase only, namely a form of statement. This may be a statement used in the source language or an auxiliary statement as mentioned earlier. In the latter case we allow the form to include primary commas (there is no need of a separator) since such statements will only appear in statement definitions where the distinction between the two is still preserved. The primary comma corresponds to an internal identifier which has no equivalent in the source language. Examples of statement formats are:

statement format (auxiliary): $A = [A.O.][F]$
statement format: $[Y] = [GE][EOL]$

The statement definitions

To each statement format corresponds a statement definition. The former describes the *form* of a statement, the latter describes the action which is to be taken when the form is recognized. The means by which a particular statement is recognized as representing some general form is described in the earlier papers, and does not concern us here. In most cases the action to be taken is to assemble the equivalent set of instructions in the target program but in the case of declarative statements such as $[V] \rightarrow [N]$ the equivalent action is to enter certain information in a list to be used later. Statements can of course be partly imperative and partly declarative, and both operations are in fact treated by means of *list compiling instructions*, the only difference being that in the first case the 'list' in question is the target program itself. Very often the 'meaning' of a statement, in the above sense, can be expressed in terms of the meaning of a sequence of other less complex statements, the subexpressions of the main statement being *parameters* of the *substatements*. It is necessary therefore to have some means of resolving a statement (and in general any expression) into subexpressions consistent with its known structure, and if necessary to build up new expressions from these subexpressions. It is also desirable to be able to compare different expressions in order to select distinct courses of action.

Thus in general a statement definition consists of three types of *primary instruction:*

1. Basic list compiling instructions.
2. Substatements.
3. Expression handling operations (resolving, testing, etc.)

A floating address system is employed for the purpose of control transfers and any primary instruction can be labelled; for example,

$$1] \rightarrow 2 \text{ unless } [Y] \equiv [V']$$

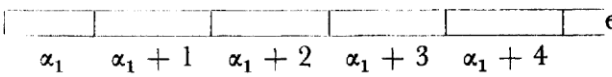
is an instruction (type 3) labelled 1.

The basic compiling instructions

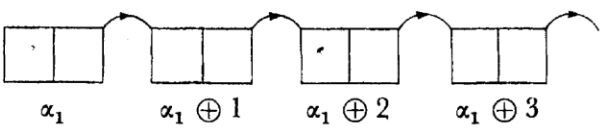
There is a central *global* group of 24-bit registers $\alpha_1\alpha_2\alpha_3 \ldots$ which do not form a field (i.e. cannot be referred to as α_r). In addition to this global set there is a set of local registers $\beta_1\beta_2\beta_3 \ldots$ associated with each statement definition. The compiling instructions are concerned with selecting processing and comparing the information in these registers, *and in the registers whose address is contained in these registers.* Thus, e.g.

$$\alpha_{10} = \beta_2 + (\alpha_1 + 3)$$

means $\alpha_{10} = \beta_2 +$ the contents of the register whose address is $\alpha_1 + 3$. We may speak of the conventional list α_1 as denoting the set of consecutive registers



Another kind of list is the *chain* in which each 24-bit word is accompanied (as the more significant half of a 48-bit word) by the address of the 'next' word. Thus:



$\alpha_1 \oplus 1$ denotes $(\alpha_1 + 1)$, etc. Further details can be found in the earlier papers.

The parameter operations

We discuss first the means by which the individual phrases are identified. Both the parameter phrases of the statement heading (i.e. the first 'line' of the definition) and those derived from them are normally referred to by writing the relevant class identifiers. If the same class of phrase appears more than once in any logical path through a definition, the different appearances of the class identifier are distinguished by means of a label (written inside the identifier brackets). Thus for example we may write ϕ divide ([*GE*/1], [*GE*/2]). The conventions of labelling are not rigid and we only require that subsequent references to each expression

bear the correct label, and that ambiguity is avoided. It is also necessary to be able to refer to a particular appearance of a 'repeated' phrase, and for this purpose we attach another label, in this case enclosed in round brackets. Thus, for example, given the 'repeated' phrase $[Q*/1]$ we refer to the second Q in the sequence as $[Q*/1(2)]$. In the use of floating references such as $[Q*/1(\alpha_1)]$ or $[Q*/1(\beta_1)]$ the current value of the α or β concerned determines the particular appearance of the phrase within a sequence. The above interpretation is not relevant in the case of $[N(\alpha_1)]$ and $[N(\beta_1)]$ etc., and we attach a quite different significance to these identifiers. Either one can be written in place of a conventional N in a substatement, and means insert the current value of the α or β in question at this point.

To return to the question of generating new phrases from those appearing in the statement heading, the instructions used for this purpose are:

$$\text{let } [\pi] \equiv \text{some } [\pi] \text{ expression}$$

and

$$\text{let } [\pi] = \text{some } [\pi] \text{ expression}$$

In the first of these the l.h.s. refers to a particular phrase appearing in the statement heading (or introduced by a previous 'let' instruction) while the r.h.s. is an explicit phrase of a form consistent with foregoing definitions of the l.h.s. and its subphrases.

For example, if the definitions

$$[\pm T] = [\pm][T]$$

and

$$[T] = [Q*][Q?]$$

have been previously given we may write

$$\text{let } [\pm T] \equiv [\pm][T]$$

or even

$$\text{let } [\pm T] \equiv +[Q]$$

if it is *known* that the $[\pm T]$ in question takes the form of a single $+[Q]$. When alternative forms are possible a single instruction can be used both to introduce the subphrases of a particular alternative, and to take some alternative course of action if (or unless) the specified alternative form appears. Thus for example

$$\rightarrow 1 \text{ unless } [T] \equiv [Q]$$

Throughout the logical path stemming from this instruction, the newly introduced phrase Q can be referred to; however, in the alternative path labelled 1 it remains 'unknown'.

In the second type of let instruction the new phrase introduced is the r.h.s. expression and this is subsequently referred to by the identifier appearing on the l.h.s. New phrases of this type are compounded from *previously* introduced phrases (and basic symbols), again in a manner consistent with the foregoing phrase definition of the l.h.s. This type of 'let' instruction does not give rise to conditional forms. Another type of conditional instruction is, however, required in order to compare different appearances of the same class of phrase. For example:

$$\rightarrow 1 \text{ if } [GE/1] = [GE/2]$$

Finally two further instructions should be mentioned, namely:

$[\alpha\beta] =$ category of $[\pi]$	e.g. $\beta_1 =$ category of $[Y]$
$[\alpha\beta] =$ number of $[\pi]$	e.g. $\beta_1 =$ number of $[\pm T^*]$

In both of these the π refers to a particular 'known' phrase. The former instruction determines the principal category to which the phrase belongs within its class, while the second instruction (which applies only to 'repeated' classes) determines how many times the phrase in question has actually been repeated.

Finally it should be noted that the statement heading itself is essentially a parameter operation of the form:

$$\text{let [instruction]} \equiv \text{[statement heading]}$$

It formally 'introduces' the constituent phrases of the statement.

Substatements

A substatement is a particular statement in which some of the constituent expressions (or subexpressions of them) may have been replaced by explicit phrases, and others by parameter phrases derived in one of the ways described above. Thus e.g. $A = A \times [Q^*(\beta_2)]$ is a particular form of $A = [A.0][F]$.

The effect of these is to cause the parameters in question to be substituted and then to call in the corresponding statement definition. In effect the basic compiling instructions, such as e.g.

$$\beta_2 = \alpha_{11} + [N] + [N] \quad \text{and} \quad [\alpha\beta] = (\alpha_{34} + \beta_2) + \beta_3 + \beta_3$$

which are particular forms of $[\alpha\beta] = [\text{word}][\theta][\text{word}]$ (see table below), are also treated in this manner. In this case, however, the corresponding statement definition is a built-in interpretive routine.

A summary of the built-in expressions and formats

In order to recognize the basic listing instructions and parameter operations in addition to the statements which the user defines, we supplement the *statement format dictionary* by a *dictionary of built-in operations*. This dictionary consists of the following formats:

$[\alpha\beta]$ = [word]
$[\alpha\beta]$ = [word]$[\theta]$[word]
([addr]) = [word]
([addr]) = [word]$[\theta]$[word]
→ $[\alpha\beta N][IU]$[word]$[\phi]$[word]
→ $[\alpha\beta N]$
let $[\pi]$ = [some π expression]
let $[\pi] \equiv$ [some π expression]
→ $[\alpha\beta N][IU][\pi] \equiv$ [some π expression]
→ $[\alpha\beta N][IU][\pi] = [\pi]$
$[\alpha\beta]$ = category of $[\pi]$
$[\alpha\beta]$ = number of $[\pi]$
END

The definitions of the following expressions are also 'built-in'.

[0–3] = 0, 1, 2, 3
$[D]$ = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
$[N] = [D^*]$
$[K] = [N]$, $[N]$., .$[N]$, $[N]$.$[N]$
$\alpha = \alpha 1, \alpha 2$, etc.
$\beta = \beta 1, \beta 2$, etc.
$[\alpha\beta] = \alpha, \beta$
$[\alpha\beta N] = \alpha, \beta, [N]$
[addr] = $[\alpha\beta]$, $[\alpha\beta] + [\alpha\beta N]$, $[\alpha\beta] - [\alpha\beta N]$, $[\alpha\beta] \oplus [\alpha\beta N]$
[word] = [addr], ([addr]), $[N]$.[0–3], .[0–3], $[N]$
$[\theta] = +, -, \times, /, \&, \mathrm{v}, \not\equiv$
$[\phi] = =, \neq, >, \geq, <, \leq$
$[IU]$ = if, unless
$[\pi]$ = 'any phrase identifier'
[some π expression] = 'any expression consistent with the definition of $[\pi]$'.

AN ILLUSTRATION OF A TYPICAL STATEMENT DEFINITION

As an illustration of a typical statement definition we shall describe the general structure of the definition of $A = [A.O.][F]$

The first step is to determine the category of the $[A.O.]$ phrase. Because

of the way [*A.O.*] has been defined this does not completely resolve the arithmetical operation involved and in the cases $A = A[\pm][F]$ and $A = [\pm ?][F]$ it is necessary to distinguish the + and − signs. This is done by means of the first half dozen instructions, as a result of which $\beta_1 = 1, 2, 3, 4, 5, 6$ corresponding respectively to the cases $A = A + [F]$, $A = A \times [F]$, $A = A/[F]$, $A = [F]$, $A = A - [F]$ and $A = -[F]$. We then examine the form of the $[F]$, distinguishing first whether it is a $[Q]$ or a $TS[N]$, and then if a $[Q]$ whether it is a $[Y][K]$ or $[I]$. The relevant instructions are $\rightarrow 3$ unless $[F] \equiv [Q]$, $\rightarrow 4$ unless $[Q] \equiv [Y]$, $\rightarrow 5$ unless $[Q] \equiv [K]$, let $[Q] \equiv [I]$ and let $[F] \equiv TS[N]$. The last two are necessary in order to 'introduce' $[I]$ and $TS[N]$ even though these cases follow as a result of exclusion.

In the case $[F] \equiv [Q] \equiv [Y]$ there are five alternative forms for $[Y]$ to be considered. In every case, however, the effective address of the operand is described by two quantities, the *presumptive address* and the category of the *index*. Thus e.g. if $[Y] = x_k$ then the presumptive address is the location corresponding to x_0, and the index category is 3 (corresponding to $i = 1$, $j = 2$, $k = 3$, etc.). These quantities are determined by separate statement definitions since they are also used elsewhere. They are called in by the instructions 'β_2 = presumptive address of $[Y]$' and 'β_3 = index of $[Y]$.' Given β_2 and β_3 the statement $A = [A.O.]\,[Y]$ translates into a single instruction with $ba = 0$, $bm = \beta_3$, $S = \beta_2$ and an appropriate function code $[FD]$. This latter is set by one of a 'table' of instructions 'let $[FD]$ = ????' selected through a multiway switch $\rightarrow \beta_1$. These all return to the substatement $[FD]$, 0, β_3, β_2 representing the general statement $[FD]$, [word], [word], [word]. This is followed by the END of the statement definition.

The case $[F] \equiv [Q] \equiv [K]$ is dealt with by a similar table but employs the set of extracodes which assume the operand is the number in the location following the instruction itself. Similarly the case $[F] \equiv [Q] \equiv [I]$ involves the extracodes which interpret the 24-bit integers $S + bm + ba$ as the operand. In this case $S = 0$, $ba = 0$, and bm is the category of $[I]$ so that the effect is to use the index quantity (in one of the *B*-registers *B*1 − *B*12) itself as the floating point operand. In the case $[K]$ we have introduced the substatement 'insert $[K]$' which (though not formally defined anywhere) means plant $[K]$ in locations α_{36}, $\alpha_{36} + 1$.

Finally the case $[F] \equiv TS[N]$ is treated by means of the first 'table' of (basic) instructions but with $\beta_2 = \alpha_{11} + 2[N]$ and $\beta_3 = 0$.

THE FORMAL DESCRIPTION OF MERCURY AUTOCODE

Note: The first set of primary statements describe the form in which

basic and extracode instructions are written in other statement definitions.

phrase defn: $[FD] = [BD][OD][OD][OD]$

phrase defn: $[BD] = 0, 1$

phrase defn: $[OD] = 0, 1, 2, 3, 4, 5, 6, 7$

statement format (auxiliary): $[FD]$, [word], [word], [word]

statement defn: $[FD]$, [word/1], [word/2], [word/3]

let $[FD] \equiv [BD][OD/1][OD/2][OD/3]$
$\beta 1 =$ category of $[OD/1]$
$\beta 2 =$ category of $[OD/2]$
$\beta 3 =$ category of $[OD/3]$
$\beta 4 =$ category of $[BD]$
$\beta 4 = 8 \times \beta 4\text{–}1$
$\beta 4 = \beta 4 + \beta 1\text{–}1$
$\beta 4 = 8 \times \beta 4$
$\beta 4 = \beta 4 + \beta 2\text{–}1$
$\beta 4 = 8 \times \beta 4$
$\beta 4 = \beta 4 + \beta 3\text{–}1$
$\beta 4 = 128 \times \beta 4$
$\beta 4 = \beta 4 +$ [word/1]
$\beta 4 = 32 \times \beta 4$
$\beta 5 =$ [word/2]/4
$\beta 4 = \beta 4 + \beta 5$
$(\alpha 36) = \beta 4$
$(\alpha 36 + 1) =$ [word/3]
$\alpha 36 = \alpha 36 + 2$
END

Note: The following relate to the simple arithmetical instructions used in MERCURY autocode.

phrase defn: $[V] = a, b, c, d, e, f, g, h, u, v, w, x, y, z, \pi$

phrase defn: $[V'] = a', b', c', d', e', f', g', h', u', v', w', x', y', z'$

phrase defn: $[I] = i, j, k, l, m, n, o, p, q, r, s, t$

phrase defn: $[\pm] = +, -$

phrase defn: $[Y] = [V]$, $[V']$, $[V][N]$, $[V][I]$, $[V]([I][\pm][N])$

phrase defn: $[Q] = [Y]$, $[K]$, $[I]$

phrase defn: $[F] = [Q]$, $TS[N]$

phrase defn: $[G] = [Y]$, $TS[N]$

phrase defn: $[A.O.]$ = (in order of preference) $A[\pm]$, $A\times$, $A/$, $[\pm?]$
statement format (auxiliary): $A = [A.O.][F]$
statement defn: $A = [A.O.][F]$

$\beta 1$ = category of $[A.O.]$
$\rightarrow 1$ unless $[A.O.] \equiv A -$
$\beta 1 = 5$
$\rightarrow 2$
1] $\rightarrow 2$ unless $[A.O.] \equiv -$
$\beta 1 = 6$
2] $\rightarrow 3$ unless $[F] \equiv [Q]$
$\rightarrow 4$ unless $[Q] \equiv [Y]$
$\beta 2$ = presumptive address of $[Y]$, $\beta 3$ = index of $[Y]$
6] $\beta 1 = \beta 1 + 10, \rightarrow \beta 1$
11] let $[FD] = 0310, \rightarrow 17$
12] let $[FD] = 0352, \rightarrow 17$
13] let $[FD] = 0374, \rightarrow 17$
14] let $[FD] = 0324, \rightarrow 17$
15] let $[FD] = 0311, \rightarrow 17$
16] let $[FD] = 0325$
17] $[FD]$, 0, $\beta 3$, $\beta 2$, END
4] $\rightarrow 5$ unless $[Q] \equiv [K]$
$\beta 1 = \beta 1 + 20, \rightarrow \beta 1$
21] let $[FD] = 1???, \rightarrow 27$
22] let $[FD] = 1???, \rightarrow 27$
23] let $[FD] = 1???, \rightarrow 27$
24] let $[FD] = 1???, \rightarrow 27$
25] let $[FD] = 1???, \rightarrow 27$
26] let $[FD] = 1???$
27] $[FD]$, 0, 0, 0, insert $[K]$, END
5] let $[Q] \equiv [I]$
$\beta 2$ = category of $[I]$
$\beta 1 = \beta 1 + 30, \rightarrow \beta 1$
31] let $[FD] = 1???, \rightarrow 37$
32] let $[FD] = 1???, \rightarrow 37$
33] let $[FD] = 1???, \rightarrow 37$
34] let $[FD] = 1???, \rightarrow 37$
35] let $[FD] = 1???, \rightarrow 37$
36] let $[FD] = 1???$
37] $[FD]$, 0, $\beta 2$, 0, END
3] let $[F] \equiv TS[N]$, $\beta 2 = \alpha 11 + [N] + [N]$, $\beta 3 = 0, \rightarrow 6$

statement format (auxiliary): $[G] = A$

statement defn: $[G] = A$

$\rightarrow 1$ unless $[G] \equiv [Y]$
$\beta 2 =$ presumptive address of $[Y]$, $\beta 3 =$ index of $[Y]$
2] 0366, 0, $\beta 3$, $\beta 2$, END
1] let $[G] \equiv TS[N]$, $\beta 2 = \alpha 11 + [N] + [N]$, $\beta 3 = 0$, $\rightarrow 2$

statement format (auxiliary): $[\alpha\beta] =$ presumptive address of $[Y]$

statement defn: $[\alpha\beta] =$ presumptive address of $[Y]$

$\beta 1 =$ category of $[Y]$, $\rightarrow \beta 1$
1] let $[Y] \equiv [V]$, $\beta 2 =$ category of $[V]$, $[\alpha\beta] = \alpha 3 + \beta 2 + \beta 2 - 2$, END
2] let $[Y] \equiv [V']$, $\beta 2 =$ category of $[V']$, $[\alpha\beta] = \alpha 2 + \beta 2 + \beta 2 - 2$, END
3] let $[Y] \equiv [V][N]$, $\beta 3 = [N]$, $\rightarrow 6$, END
4] let $[Y] \equiv [V][I]$, $\beta 3 = 0$, $\rightarrow 6$
5] let $[Y] \equiv [V]([I][\pm][N])$, $\beta 3 = [N]$, $\rightarrow 6$ unless $[\pm] \equiv -$, $\beta 3 = 0 - \beta 3$
6] $\beta 2 =$ category of $[V]$, $[\alpha\beta] = (\alpha 34 + \beta 2) + \beta 3 + \beta 3$
$\rightarrow 7$ if $[\alpha\beta] \neq 0$, print fault (7)
7] END

statement format (auxiliary): $[\alpha\beta] =$ index of $[Y]$

statement defn: $[\alpha\beta] =$ index of $[Y]$

$\beta 1 =$ category of $[Y]$, $\rightarrow \beta 1$
1] $[\alpha\beta] = 0$, END, 2] $\rightarrow 1$, 3] $\rightarrow 1$, 4] let $[Y] \equiv [V][I]$, 6]$[\alpha\beta] =$ category of $[I]$ END
5] let $[Y] \equiv [V]([I][\pm][N])$, $\rightarrow 6$

phrase defn: $[|Q] = /[Q]$

phrase defn: $[T] = [Q^*][|Q?]$

phrase defn: $[\pm T] = [\pm][T]$

phrase defn: $[GE] = [\pm?][\pm T^*?]$

statement format (auxiliary): $A = [GE]$

statement format (auxiliary): $A = [\pm?][T]$

statement format (auxiliary): $A = A[\pm][T]$

statement defn: $A = [GE]$

let $[GE] \equiv [\pm?][T][\pm\ T^*?]$
$A = [\pm?][T]$

```
    → 1 unless [±T*?] ≡ [±T*]
    β1 = number of [±T*]
    β2 = 1
 2] let [±T*(β2)] ≡ [±][T]
    A = A[±][T]
    β2 = β2 + 1
    → 2 if β1 ≥ β2
 1] END
```

statement defn: $A = [\pm?][T]$

```
    let [T] ≡ [Q*][|Q?]
    A = [±?][Q*(1)]
    β1 = number of [Q*]
    → 1 if β1 = 1
    β2 = 2
 2] A = A × [Q*(β2)]
    β2 = β2 + 1
    → 2 if β1 ≥ β2
 1] → 3 unless [|Q?] ≡ /[Q]
    A = A/[Q]
 3] END
```

statement defn: $A = A[\pm][T]$

```
    → 1 unless [T] ≡ [Q]
    A = A[±][Q]
    END

 1] TS1 = A
    let [T] ≡ [Q*][|Q?]
    A = [±][Q*(1)]
    β1 = number of [Q*]
    → 2 if β1 = 1
    β2 = 2
 3] A = A × [Q*(β2)]
    β2 = β2 + 1
    → 3 if β1 ≥ β2
 2] → 4 unless [|Q?] ≡ /[Q]
    A = A/[Q]
 4] A = A + TS1
    END
```

statement format: $[Y] = [GE][EOL]$

statement defn: $[Y] = [GE][EOL]$

$A = [GE]$
$[Y] = A$
END

phrase defn: $[N \text{ or } I] = [N], [I]$

statement format (auxiliary): $[FD]$, [word], 0, $[N \text{ or } I]$

phrase defn: $[\pm N \text{ or } I] = [\pm][N \text{ or } I]$

statement format: $[I] = [\pm ?][N \text{ or } I][\pm N \text{ or } I^* ?][EOL]$

Note: This represents only a subset of the index arithmetic instructions. In practice, however, the majority of them are of this form which lends itself to a more efficient treatment than the general case: $[I] = [GE][EOL]$. The special case must of course come before the general case in order of preference.

statement defn: $[I] = [\pm ?][N \text{ or } I][\pm N \text{ or } I^* ?][EOL]$

let $[FD] = 0121$
$\rightarrow 1$ unless $[\pm ?] \equiv -$
let $[FD] = 0123$
1] $[FD]$, 20, 0, $[N \text{ or } I]$
$\rightarrow 2$ unless $[\pm N \text{ or } I^* ?] \equiv [\pm N \text{ or } I^*]$
$\beta 1 =$ number of $[\pm N \text{ or } I^*]$
$\beta 2 = 1$
4] let $[\pm N \text{ or } I^*(\beta 2)] \equiv [\pm][N \text{ or } I]$
let $[FD] = 0124$
$\rightarrow 3$ unless $[\pm] \equiv -$
let $[FD] = 0122$
3] $[FD]$, 20, 0, $[N \text{ or } I]$
$\beta 2 = \beta 2 + 1$
$\rightarrow 4$ if $\beta 2 \leq \beta 1$
2] $\beta 1 =$ category of $[I]$
0121, $\beta 1$, 20, 0
END

statement defn: $[FD]$, [word], 0, $[N \text{ or } I]$

$\rightarrow 1$ unless $[N \text{ or } I] \equiv [N]$
$\beta 1 = 2 \times [N]$
$[FD]$, [word], 0, $\beta 1$
END

```
1] let [N or I] ≡ [I]
   β1 = category of [I]
   [FD], [word], β1, 0
   END
```

Note: The remainder of the index instructions are recognized under the following heading which in fact includes a wider class of instructions than is actually allowed in MERCURY autocode.

```
statement format: [I] = [GE][EOL]
statement defn: [I] = [GE][EOL]

   A = [GE]
   β1 = category of [I]
   1???, β1, 0, 0
   END
```

Note: The next section deals with the autocode instructions for evaluating elementary functions. The extracodes involved have been described in the introduction.

```
phrase defn: [Fx] = sq rt, sin, cos, tan, exp, log, mod, int pt, fr pt,
                    sign
statement format: [Y] = φ[Fx]([GE])[EOL]
statement defn: [Y] = φ[Fx]([GE])[EOL]

   A = [GE]
   β1 = category of [Fx]
   → β1
 1] let [FD] = 1???, → 11
 2] let [FD] = 1???, → 11
 3] let [FD] = 1???, → 11
 4] let [FD] = 1???, → 11
 5] let [FD] = 1???, → 11
 6] let [FD] = 1???, → 11
 7] let [FD] = 1???, → 11
 8] let [FD] = 1???, → 11
 9] let [FD] = 1???, → 11
10] let [FD] = 1???
11] [FD], 0, 0, 0
   [Y] = A
   END
```

Note: This example could be shortened by means of special table definition and look-up operations. Thus:

table defn: [*FD*/1] = 1???, 1???, 1???, 1???, 1???, 1???, 1???, 1???, 1???, 1???, and

statement defn: [*Y*] = ϕ[*Fx*]([*GE*])[*EOL*]

A = [*GE*]
β1 = category of [*Fx*]
[*FD*] = table [*FD*/1(β1)]
[*FD*], 0, 0, 0
[*Y*] = *A*
END

phrase defn: [*Fxy*] = divide, arctan, radius

statement format: [*Y*] = ϕ[*Fxy*]([*GE*][,][*GE*])[*EOL*]

statement defn: [*Y*] = ϕ[*Fxy*]([*GE*/1][,][*GE*/2])[*EOL*]

*TS*2 = [*GE*/2]
A = [*GE*/1]
β1 = category of [*Fxy*]
→ β1
1] let [*FD*] = 0374, → 4
2] let [*FD*] = 1???, → 4
3] let [*FD*] = 1???
4] [*FD*], 0, 0, α10 + 4
[*Y*] = *A*
END

statement format: [*I*] = ϕintpt([*GE*])[*EOL*]

statement defn: [*I*] = ϕintpt([*GE*])[*EOL*]

A = [*GE*]
β1 = category of [*I*]
1???, β1, 0, 0
END

Note: The extracode used here is E4.

statement format: [*Y*] = ϕpoly([*GE*])[*V*]0[,][*N* or *I*][*EOL*]

statement defn: [*Y*] = ϕpoly([*GE*])[*V*]0[,][*N* or *I*][*EOL*]

A = [*GE*]
0121, 24, 0, [*N* or *I*]
β1 = category of [*V*]

```
0121, 23, 0, (α34 + β1)
1???, 23, 24, 0
[Y] = A
END
```

Note: The extracode used here is $E3$.

statement format: $[Y] = \phi\text{parity}([GE])[EOL]$

statement defn: $[Y] = \phi\text{parity}([GE])[EOL]$

```
A = [GE]
1???, 22, 0, 0
0127, 22, 0, 2
0121, 22, 22, 0–1
1???, 0, 22, 0
END
```

Note: The extracodes used here are $E4$ and $A' = n$

statement format: intstep $([N])[EOL]$

statement defn: intstep $([N])[EOL]$

```
0324, 0, 0, α3 + 11
0121, 25, 6, 0
0121, 26, 0, (α34 + 13)
0121, 27, 0, (α34 + 6)
0121, 28, 0, (α34 + 8)
nest(α23) = α36 + 1
0121, 29, 0, [N]
1???, 25, 0, α3 + 7
END
```

Note: The extracode used here is $E5a$. The 'nest' instruction records the reference to a chapter label. The precise details of this instruction (which is really a substatement) can be found in Ref. (1), where it is described as 'add [word] to nest $[\alpha\beta]$'. We now prefer the form 'nest $([\alpha\beta])$ = [word]'.

statement format: 592, 0 $[EOL]$

statement defn: 592, 0 $[EOL]$

```
1???, 0, 0, 0
END
```

Note: The extracode used here is $E5b$.

Note: The next set of statement definitions describe the method of labelling autocode statements, the system of 'local' labels adopted in some of the auxiliary material, and all the statements involving reference to such labels. These include the conditional and unconditional *jump* instructions, the *across*, *down* and *up* instructions, the *cycling instructions*, and the auxiliary statement *call in SR*.

```
statement format: [N])
statement format (auxiliary): ([N])
statement format (auxiliary): SR([N])[EOL]
statement defn: [N])
    (α33 + [N]) = α36
    END
statement defn: ([N])
    list (α21) = α36
    list (α21) = [N]
    END
```

Note: 'list ([$\alpha\beta$]) = [word]' is described in Ref. (1) as 'add [word] to list [$\alpha\beta$]'.

```
statement defn: SR([N])[EOL]
    (α9 + [N]) = α36
    END
statement format (auxiliary): call in SR([N])
statement defn: call in SR([N])
    1???, 0, 0, α9 + [N]
    END
```

Note: The above is simply equivalent to the extracode instruction E5a.

```
statement format: jump [N][EOL]
statement format: jump ([I])[EOL]
statement format: [I]) = [N])[EOL]
statement format: [I]) = [I])[EOL]
statement defn: jump [N][EOL]
    nest(α23) = α36 + 1
    0121, 127, 0, [N]
    END
```

statement defn: jump ([*I*])[*EOL*]

$\beta 1$ = category of [*I*]
0121, 127, $\beta 1$, 0
END

statement defn: [*I*]) = [*N*])[*EOL*]

$\beta 1$ = category of [*I*]
nest($\alpha 22$) = $\alpha 36 + 1$
0121, $\beta 1$, 0, [*N*]
END

statement defn: [*I*]) = [*I*/1])[*EOL*]

$\beta 1$ = category of [*I*]
$\beta 2$ = category of [*I*/1]
0101, $\beta 1$, $\beta 2$, $\alpha 33$
END

statement format (auxiliary): fill in local labels

statement defn: fill in local labels

4] $\beta 1$ = nest ($\alpha 22$)
$\beta 2$ = $\alpha 21$
→ 5 if ($\beta 2+1$) = 0
3] $\beta 2$ = $\beta 2 \oplus 1$
→ 2 if ($\beta 1$) = ($\beta 2$)
$\beta 2$ = $\beta 2 \oplus 1$
→ 3 unless $\beta 2$ = $\alpha 21$
print fault (23)
2] ($\beta 1$) = ($\beta 2 \oplus 1$)
→ 4 unless ($\alpha 22 + 1$) = 0
1] delete $\alpha 21 \oplus 1$
5] → 1 unless ($\alpha 21+1$) = 0
END

phrase defn: [−] = −

phrase defn: [=≠>≥] = =, ≠, >, ≥

statement format: jump[*N*][,][*Y*][=≠>≥][−?][*K*][*EOL*]

statement format: jump[*N*][,][−?][*K*][=≠>≥][*Y*][*EOL*]

statement format: jump[*N*][,][*Y*][=≠>≥][*Y*][*EOL*]

```
statement format: (auxiliary): → [N] if A[=≠>≥]0
statement defn: jump[N][,][Y][=≠>≥][−?][K][EOL]
    A = [Y]
    → 1 unless [−?] ≡ −
    A = A + [K]
    → 2
 1] A = A − [K]
 2] → [N] if A[=≠>≥]0
    END
statement defn: jump[N][,][−?][K][=≠>≥][Y][EOL]
    → 1 unless [−?] ≡ −
    A = −[K]
    → 2
 1] A = [K]
 2] A = A − [Y]
    → [N] if A[=≠>≥]0
    END
statement defn: jump[N][,][Y/1][=≠>≥][Y/2][EOL]
    A = [Y/1] −[Y/2]
    → [N] if A[=≠>≥]0
    END
statement defn: → [N] if A[=≠>≥]0
    β1 = category of [=≠>≥]
    → β1
 1] let [FD] = 0234, → 5
 2] let [FD] = 0235, → 5
 3] let [FD] = 1???, → 5
 4] let [FD] = 0236
 5] nest(α23) = α36 + 1
    [FD], 127, 0, [N]
    END
```

Note: The machine codes involved here are the *A*-test instructions described in the introduction.

```
statement format: jump[N][,][I][=≠>≥][−?][N][EOL]
statement format: jump[N][,][−?][N][=≠>≥][I][EOL]
statement format: jump[N][,][I][=≠>≥][I][EOL]
```

statement format: (auxiliary): → [*N*] if *Bt*[=≠>≥]0
statement defn: jump[*N*][,][*I*/1][=≠>≥][−?][*N*/2][*EOL*]
$\beta 1$ = category of [*I*/1]
$\beta 2$ = [*N*/2]
→ 1 unless [−?] ≡ −
$\beta 2 = 0 - \beta 2$
1] 0172, $\beta 1$, 0, $\beta 2$
→ [*N*] if *Bt*[=≠>≥]0
END

statement defn: jump[*N*][,][−?][*N*/1][=≠>≥][*I*/2][*EOL*]
$\beta 1$ = [*N*/1]
→1 unless [−?] ≡ −
$\beta 1 = 0 - \beta 1$
1] $\beta 2$ = category of [*I*/2]
0170, $\beta 2$, 0, $\beta 1$
→ [*N*] if *Bt*[=≠>≥]0
END

statement defn: jump[*N*][,][*I*/1][=≠>≥][1/2][*EOL*]
$\beta 1$ = category of [*I*/1]
$\beta 2$ = category of [*I*/2]
0172, $\beta 1$, $\beta 2$, 0
→ [*N*] if *Bt*[=≠>≥]0
END

statement defn: → [*N*] if *Bt*[=≠>≥]0
$\beta 1$ = category of [=≠>≥], → $\beta 1$
1] let [*FD*] = 0224, → 5
2] let [*FD*] = 0225, → 5
3] let [*FD*] = 1???, → 5
4] let [*FD*] = 0226
5] nest($\alpha 23$) = $\alpha 36 + 1$
[*FD*], 127, 0, [*N*]
END

Note: These are the *B*-test instructions also mentioned in the introduction.

statement format: [*I*] = [*N* or *I*]([−?][*N* or *I*][*N* or *I*][*EOL*]
statement defn: [*I*] = [*N* or *I*/1]([−?][*N* or *I*/2])[*N* or *I*/3][*EOL*]
$\beta 1$ = category of [*I*]
$\beta 2 = \alpha 36 + 6$

```
    0121, β1, 0, [N or I/1]
    0121, 127, 0, β2
    → 1 if [—?] ≡ —
    0124, β1, 0, [N or I/2]
 2] 0172, β1, 0, [N or I/3]
    0225, 127, 0, β2
    nest(α25) = (β2)
    nest(α25) = (β2 + 1)
    nest(α25) = (β2 + 2)
    nest(α25) = (β2 + 3)
    α36 = β2
    END
 1] 0122, β1, 0 [N or I/2]
    → 2
```

statement format: repeat [*EOL*]

statement defn: repeat [*EOL*]

```
    (α36 + 3) = nest(α25)
    (α36 + 2) = nest(α25)
    (α36 + 1) = nest(α25)
    (α36) = nest(α25)
    α36 = α36 + 4
    END
```

Note: '([addr]) = nest([αβ])' is the reverse operation to 'nest([αβ]) = [word]'. It extracts from the nest the last word entered.

phrase defn: [—*N*] = —[*N*]

statement format: across [*N*]/[*N*][—*N*?][*EOL*]

statement defn: across [*N*/1]/[*N*/2][—*N*?][*EOL*]

```
    β1 = α31
    → 1 unless [—N?] ≡ —[N]
    β1 = 0
    nest(α24) = α36 + 1
    nest(α24) = [N]
 1] β2 = β1 + [N/2]
    β3 = α30 × β2
    β4 = α7 — β3 + [N/1]
    0101, 127, 0, β4
    END
```

statement format: down [*N*]/[*N*][−*N*?][*EOL*]
statement defn: down [*N*/1]/[*N*/2][−*N*?][*EOL*]

```
    β1 = α31
    → 1 unless [−N?] ≡ −[N]
    β1 = 0
    nest(α24) = α36 + 3
    nest(α24) = [N]
 1] β2 = β1 + [N/2]
    β3 = α30 × β2
    β4 = α7 − β3 + [N/1]
    0124, 21, 0, 512
    1???, 0, 0, β4
    END
```

Note: The extracode involved is *E*5a.

statement format: up [*EOL*]
statement defn: up [*EOL*]

```
    0122, 13, 0, 512
    1???, 0, 0, 0
    END
```

Note: The extracode involved is *E*4a.

Note: The following are the statement definitions for *preserve*, *restore* and the $\phi 6$ and $\phi 7$ instructions, all of which are concerned with the transfer of material between the working store and the auxiliary store.

statement format: preserve [*EOL*]
statement defn: preserve [*EOL*]

```
    call in SR(2)
    END
```

statement format: restore [*EOL*]
statement defn: restore [*EOL*]

```
    call in SR(3)
    END
```

phrase defn: [6 or 7] = 6, 7
statement format: ϕ[6 or 7]([*GE*])[*Y*][,][*N* or *I*][*EOL*]
statement defn: ϕ[6 or 7]([*GE*])[*Y*][,][*N* or *I*][*EOL*]

```
    A = [GE]
    β1 = presumptive address of [Y]
```

```
    β2 = index of [Y]
    → 1 unless [6 or 7] ≡ 6
    1???, 22, 0, 0
    0121, 23, β2, β1
 2] 0121, 24, 0, [N or I]
    call in SR(1)
    END
 1] 1???, 23, 0, 0
    0121, 22, β2, β1
    → 2
```

Note: The following are the declarative statements which come at the beginning and end of an autocode chapter.

```
statement format: chapter [N][EOL]
statement defn: chapter [N][EOL]
    → 1 unless [N] ≡ 0
    α32 = 0
    → 2
 1] α32 = α31 + [N]
 2] β1 = α30 × α32
    α33 = α7 − β1
    α34 = α33 + 128
    α35 = α34 + 16
    α36 = α35
    α37 = α1
    β2 = 0
 3] (α33 + β2) = 0
    β2 = β2 + 1
    → 3 if β2 ≤ 143
    END

statement format: [V] → [N][EOL]
statement defn: [V] → [N][EOL]
    β1 = category of [V]
    → 1 unless (α34 + β1) ≠ 0
    print fault (2)
 1] (α34 + β1) = α37
    α37 = α37 + [N] + 1
    → 2 unless α37 > 480
    print fault (6)
 2] END
```

```
statement format: variables [N][EOL]
statement defn: variables [N][EOL]
    β1 = [N] + α31
    β2 = α30 × β1
    β3 = α7 − β2 + 128
    β4 = 1
 1] (α34 + β4) = (β3 + β4)
    β4 = β4 + 1
    → 1 if β4 ≤ 15
    END

statement format: programme −[N][EOL]
statement defn: programme − [N][EOL]
    α31 = α32
    β1 = (α8)
    (β1) = [N]
    (β1 + 1) = α31
    (α8) = β1 + 2
    END

statement format: close [EOL]
statement defn: close [EOL]
 1] β1 = nest(α23)
    β2 = (β1)
    β3 = (α33 + β2)
    → 2 unless β3 = 0
    print fault (3)
 2] (β1) = β3
    → 1 unless (α23 + 1) = 0
    → 6 unless α32 = 0
 3] β1 = nest(α24)
    β2 = nest(α24)
    β3 = α8 + 2
 5] → 4 if (β3) = β2
    β3 = β3 + 2
    → 5 unless β3 = (α8)
    print fault (13)
 4] (β1) = (β1) + (β3 + 1)
    → 3 unless (α24 + 1) = 0
 6] END
```

Note: The 'faults' are those diagnosed by the standard MERCURY autocode program. Thus e.g. fault (3) means that a label has not been set. The interpretation of the statement format—print fault ([*N*])—like other output instructions, is tied up with the supervisor program which is beyond the scope of this report.

Normally translation ceases on encountering the *close* of a chapter 0 and the machine starts to execute the program at the first instruction of chapter 0. However, for reasons explained below it may be more convenient to postpone execution until all the material associated with a program has been read (i.e. its data and possibly further chapter 0's): and an 'end of message' directive is encountered. Chapter 0 will, in any case, be entered through the 'initial sequence' in PERM.

statement format (auxiliary): prepare to read a M.A. program [*EOL*]

statement defn: prepare to read a M.A. program [*EOL*]

set α1 — α12 inclusive
borrow α21 — α25 inclusive
set α30 = 512, α31 = 0
α36 = α12
0121, 118, 0, α10
0121, 21, 0, α6 — 1024
0121, 127, 0, α7 + 144
SR(1)
(1)0324, 22, 24, 0 — 1
0???, 23, 24, 0 — 1
nest(α22) = α36 + 1
0203, 127, 24, 1
1???, 0, 0, 0
fill in local labels
SR(2)
call in *SR*(4)
0121, 22, 0, α1
0121, 23, 21, 0
0121, 24, 0, 512
call in *SR*(1)
1???, 0, 0, 0
SR(3)
0121, 22, 21, 0
0121, 23, 0, α1
0121, 24, 0, 512
call in *SR*(1)

call in $SR(5)$
1???, 0, 0, 0
the sequence corresponding to $SR(4)$
the sequence corresponding to $SR(5)$
END

Note: The first three and last two lines of this definition are informal statements referring to sequences which we have omitted to describe in detail. $SR(4)$ is a sequence to pack the 12 B-registers containing the indices $i, j, \ldots, t$ into three spare locations at the end of the working store, namely $\alpha 4 + 1$, $\alpha 4 + 2$, $\alpha 4 + 3$. $SR(5)$ is the reverse operation.

The above statement must precede each MERCURY autocode program in order to prepare for its translation.

The following are the statement formats of the decimal input instructions and of numerical data in floating decimal form, also the *rmp* instruction.

statement format: read ([Y])

statement format: read ([I])

statement format: K[exponent ?] [EOL]

phrase defn: [exponent] = [,][N]

Note: This is not in fact adequate to describe the actual form used on MERCURY at present in which single spaces are ignored and a double space serves in place of an [EOL] as a terminal symbol. However, it will indicate the general idea.

statement format: *rmp*

In the general case a MERCURY autocode program consists of a *main program tape* which takes the form

chapter 1 chapter 2 . . . chapter N chapter 0

followed by a supplementary tape on which numerical data and/or further instructions (in the form of chapter 0's) can be punched. This supplementary tape is read under the control of the main program itself, the data by means of the *read* instructions, and the chapter 0's by means of the *rmp* ('read more program').

A possible arrangement on a machine like ATLAS is to translate the entire supplementary tape before starting to execute the main program. In the case of the numerical data this translation will amount to the conventional process of decimal to binary conversion, while the chapter 0's will be translated in the usual way. The supplementary tape can then

be simulated in the store of the machine by an ordered list whose items are floating-point numbers and/or chapter 0's. The effect of executing a read ([Y]) instruction therefore is to select the next item (a number) from this list and plant it in [Y]. Similarly the effect of an *rmp* instruction is to copy the next chapter 0 over the existing chapter 0 and enter this at the first instruction.

The following are the statement formats of the output instructions. We do not give the corresponding statement definition since the whole business of output (and input for that matter) is closely tied up with the *supervisor* system. The same applies to certain other autocode instructions, e.g. *end*, *halt*, *hoot*, etc.

statement format = print ([GE]) [N or I], [N or I]

statement format = newline

statement format = space

to which must be added

statement format (auxiliary) = print fault ([N])

Roughly speaking the effect of the output instructions is to feed the numbers and characters to be printed into a buffer store, where it takes its turn in the queue of material waiting to be processed and printed by the peripheral supervisor.

The foregoing description of MERCURY autocode is not complete. The principal omissions are the complex and double length arithmetic instructions, and the matrix operations. Given the appropriate extracodes however, (and of course this is 99% of the work!) the statement definitions would be comparatively trivial since all that is necessary in the target program is a call sequence to set the 'program parameters' for the extracode routine.

REFERENCES

1. An Assembly Programme for a Phrase Structure Language, *Computer Journal*, October 1960.
2. An Assembly Programme for a Phrase Structure Language (concluded), *Computer Journal*, January 1961.