The
Compiler Compiler

(Amended August 1963)

by   R.A. BROOKER

I.R. MacCALLUM

D MORRIS

J.S. ROHL

All of Manchester University

The Compiler Compiler

R. A. BROOKER, I. R. MacCALLUM, D. MORRIS and J. S. ROHL

University of Manchester

This paper is a sequel to the authors' previous papers in Volume 2 of this Review and elsewhere. It is a detailed specification of a system for describing the form and meaning of the statements in a phrase structure language (for example a scientific autocode). Given such a description the compiler compiler will generate (in machine code) a compiler for the language, i.e. a program which can read and translate (also into machine code) another program written in that language.

INTRODUCTION

This system may be considered to operate in two phases. In the *primary phase* it accepts and records the definition of a phrase structure language, and in the secondary phase it will translate a source program written in that language. The two phases are not completely separate and further definitions can be given in the middle of a source program. Their influence of course only extends forward and not back to the material already processed.

The primary material consists mainly of *format definitions* and *phrase definition*s which describe the form (or syntax) of statements and their constituent expressions, and *format routines* which describe their meaning (or semantics). The meaning of a new format is defined in terms of existing formats, which may be either built–in or previously defined ones. Both the form and meaning of expressions can be defined recursively, which is particularly useful in dealing with algebraic or other formulas which involve nested parentheses.

Five kinds of statements comprise the basis primary language (it can be extended) and they are recognized by the following headings or *master phrases*:

PHRASE

FORMAT CLASS

FORMAT

ROUTINE

The end of each statement is recognized by the start (i.e. the master phrase) of the next. There are two situations where there may not be a further master statement, namely when the material following is source language or when there is no further material. In these cases the master phrases:

END OF PRIMARY MATERIAL

END OF MESSAGE

should be used. The master phrase **'END OF PRIMARY MATERIAL'** will have the effect of removing all facilities for translating primary material (e.g. **PHRASE**s, **FORMAT**s, **ROUTINE**s) thus reducing the size of the generated compiler.

All subsequent references to composite symbols and the DCS are incorrect. The latter does not now exist. Composite symbols are now represented by 24 bit words which contain a 1 in digit 23 (most significant) and the basic symbols which form the composite symbol in digits $2 \rightarrow 22$. Seven digits are used for each symbol and the symbols are ordered starting with the one of smallest numerical value in digits $2 \rightarrow 8$. Composite symbols formed from more than three basic symbols are not allowed.

For example the internal representations of **A** and $\not\equiv$ in octal are:

4 0 1 2 6 2 0 4

6 5 3 4 0 7 4

Symbols and the line reconstruction process

In subsequent sections a meta–language is described, by means of which (in **PHRASE** and **FORMAT** statements) the syntax of a phrase structure programming language can be defined. Except for the master phrases and certain symbols which have meta–syntactical significance, the symbols used in this meta–language represent the same symbols in the language being defined. Although the language being defined, and the meta–language description, will eventually be prepared for input to the computer as a sequence of *characters* punched on paper tape (or cards), the emphasis is placed on the printed form of this input. In fact, the system reads the input stream a line at a time submitting each to a process called line reconstruction. This process results in a sequence of *symbols* one for each printing position across the page up to the last printed symbol on the line, which is then followed by an 'end line' symbol. The 'space' symbol will be used to represent spaces between printed symbols, but no space symbols will appear between the last printed symbol and the end of line symbol.

Each symbol in a reconstructed line represented by a code number (or serial number) contained in a 24–bit word. There are two kinds of symbols, namely basic symbols and composite symbols. However, the distinction is not usually important and composite symbols can only occur in the case of input from 7–hole tape (i.e. Flexowriter). Basic symbols are those which can be produced by depressing a single key on the 'editing equipment' (assuming that the required 'case' or 'shift' is already selected). Composite symbols are those which are synthesized by utilizing the back space key on the Flexowriter in order to overprint one symbol with another.

The code number representing a basic symbol is the 6–bit internal character code for characters on the 'inner shift' and 64 plus this quantity for characters on the 'outer shift' (see Atlas Manual). Thus the code numbers of basic symbols will be less than 128.[1] For the purpose. of converting composite symbols to the internal code numbers a dictionary of composite symbols (DCS) which relates to the groups of basic symbol comprising a composite symbol to the required code numbers. The symbols representing each composite symbol are ordered in ascending numerical order for uniqueness and looked up in the DCS. If an entry does not exist in the

---

[1] A table is given in the Appendix

DCS for a composite symbol, then it must be appearing for the first time. In this case it will be allocated the next available serial number (in the range 128–1000) and an entry will be made in the DCS. Some of the basic symbols available in the 5–hole tape code can only be reproduced as composite symbols on Flexowriters. The following permanent entries are therefore preloaded into the DCS to ensure that the composite symbols in question are converted to the correct internal code:

−	overprinted with	>	is given the code for	→
>	"	_	"	≥
0	"	/	"	φ
:	"	=	"	≈

If the erase symbol appears in any printing position across a line all other characters overprinted (or underprinted) by this symbol will be ignored. Also if a given symbol is overprinted by itself only the first appearance of the symbol will be noted. The erase symbol will not be put in the unedited list.

The meta–language has been designed so that only the following set of symbols is essential:

```
A B C D E F G H I J K L M O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 = , . ? ( ) [ ] $(or £) + − /
```

However, it will be an advantage if others (such as > ≥ ≡ ≠ ≢ etc.) are available in the set associated with the input media which is to be used. All the above symbols, except **[ ]** and **?**, are available in the codes of the three standard Atlas input media; namely; cards, 5–hole tape and 7–hole tape. In order to allow the meta–language description of a language to be input through any of the input media, the following punching conventions will be employed:

punch	**($** or **(£**	for	**[**	unless available
"	**)**	"	**]**	"	"
"	**$**	"	**?**	"	"

This will of course impose restrictions on the use of **$** (or **£**) in the metalanguage. In fact the combination (\$ or (£ must be avoided if either cards or 5–hole tape are to be used, and \$ must be identified in phrase identifiers (see later) if cards are to be used. If any other symbols have to be written in the meta–language which are not available in the symbol set for the input media being used, they can be represented by their internal code number enclosed in square brackets. Thus **[13]** is equivalent to **&** (see the Atlas Manual). Obviously this facility would only be used if the normal input media of the language being defined involves more symbols than that used to input the meta–language description.

The serial numbers of basic symbols, composite symbols, phrases and format routines (see later) will be distinct.

PHRASE

The **PHRASE** statement is used to associate a phrase or group of phrases with a single *phrase identifier* [2]. The identifiers thus defined may then be written in other phrases to indicate that any member of the associated group (or class) of phrases is a permissible substitution. For example writing:

> **PHRASE [IDENTIFIER 1]** = *ab, cd*
>
> **PHRASE [IDENTIFIER 2]** = *ef* **[IDENTIFIER 1]** *g*

means that *efabg* and *efcdg* are permissible substitutions for **[IDENTIFIER 2].**

Not all sequences of characters are permitted as identifiers and the symbols ( )/*? should not be used (except as described below) neither should] except to terminate the identifier. An identifier must also contain at least one non–numerical symbol, and must be distinct from all other identifiers including those used in the preloaded formats, etc. The symbols * and ? can be used to qualify any phrase identifier and are always interpreted as follows:

> **[IDENTIFIER*] = [IDENTIFIER] [IDENTIFIER *], [IDENTIFIER]**
>
> **[IDENTIFIER?] = [IDENTIFIER], NIL**

that is the * qualifier indicates that any arbitrary number of appearances of the qualified phrase is a permissible substitution and the ? qualifier indicates that the phrase may or may not appear. A query may he used to qualify a 'starred' identifier (but not vice–versa) and is interpreted thus:

> **[IDENTIFIER*?] = [IDENTIFIER*], NIL**

The symbol / is used only in format routines where it is necessary to distinguish (by means of integer labels after the /) different appearances of the same class of phrase, for example

> **[TERM/1] and [TERM/2]**

might denote two different appearances of the type of phrase associated with **[TERM]**. Another device only relevant in format routines is the phrase index. This is used to refer to a particular member of a sequence of phrases associated with a 'starred' identifier. Thus **[IDENTIFIER.*(3)]** means the third in the sequence. Also permitted are the forms **[IDENTIFIER*(α_1)]** and **[IDENTIFIER*(β_1)]** where α_1, and β_1, represent the working variables of a format routine (see later).

The order of the individual phrases within a definition is often significant, because they are always used in a left to right scanning process which attempts to match the various alternatives to a given source expression. Therefore if any phrase is a stem of any other phrase the stem must come last otherwise the longer alternative would never be recognized. It is for this reason that

> **[I*]**   is defined as   **[I] [I*], [I]**
>
>   and not   **[I], [I] [I*]**

Obvious violations of this rule like that above which can be detected without referring to other phrase definitions will be monitored (but the definition will still be

---

recorded) by the routine which assembles phrase definitions. However, the more subtle ones as in **[I2]** below will not.

> **PHRASE [I1] =** *ai, ad, ae*
>
> **PHRASE [I2] =** **[I1],** *adf*

Correct ordering is also required if one phrase is a special case of some other phrase in the same definition. The special case must be written first otherwise source expressions corresponding to the special case would always be recognized as the more general alternative.

It will have been noticed above that the phrase **NIL** has special significance in a phrase definition. One other phrase — **BUT NOT** — also has special significance, it is used to exclude specific members of earlier alternatives in a definition. Thus, writing:

> **PHRASE [I1] =** *abc, ade, ace*
>
> **PHRASE [I2] =** *g* **[I1], BUT NOT** *gade*

means that **[I2]** actually consists of *gabc* and *gace*. In general, more than one phrase may follow a **BUT NOT** and it qualifies them all.

The **BUT NOT** facility is implemented by 'looking for' the forbidden alternatives first in order of preference and if one is found signaling non–correspondence between the phrase definition and the sequence of symbols under examination. Thus if any of the permissible alternatives has a stem coincident with a prohibited phrase it too will be effectively excluded from the permissible set, for example, given:

> **PHRASE [A] =** *ab*
>
> **PHRASE [B] = [A]** *g*, **[A]** *de*, **[A]**
>
> **PHRASE [C] = [B]** *g*, **BUT NOT [A]** *g*

then **[C]** represents only *abdeg* and not *abgg* and *abdeg* as might be expected. Note also that a **NIL** alternative cannot be removed by use of the **BUT NOT** facility since **NIL** could be regarded as a stem of every other symbol string.

If an actual phrase starts with the stem **BUT NOT** which is not to be interpreted in the above way, then it must be disguised. For example, one could define the phrase:

> **PHRASE (CR) [BUT] = BUT**

and the stem **BUT NOT** would then be replaced by **[BUT] NOT**. The same remarks apply to NIL but only if it represents the entire last phrase in a definition. That is, no meta–syntactical significance would be attached to the **NIL** in:

> **PHRASE [NOTHING] = NIL, ZERO, NOTHING**

or **PHRASE [FIBRE] = HEMP, SISAL, MANILLA**

Finally, since the function of some phrase definitions is simply to allow for several ways of saying the same thing, a record of which alternative has occurred in any particular case may not be required. This may be indicated by using the symbols **(CR)** denoting 'contract record', e.g.

> **PHRASE (CR) [JUMP] =** $\longrightarrow$ **, JUMP**

FORMAT CLASS

A format class is similar to a phrase definition in that it consists of a set of alternative phrases (more precisely called formats in this context) represented by an identifier. However, it is not defined by a single statement like the **PHRASE** statement; but by a **FORMAT CLASS** statement which reserves an identifier for use in this way, followed by a sequence of **FORMA**T statements each of which adds (last in order of preference) a further alternative to the specified format class. Perhaps a more important difference between a format class and a phrase definition is that the individual members of the former will be associated (internally) with the serial numbers of the corresponding format routines whilst those of the latter are associated only with serially ascending category numbers (1, 2, 3 .. .). A format class is never regarded as complete and new formats may be added at any time providing it is before they are used elsewhere. Three format classes always exist, namely: **[SS]** the class of source language statements, and **[BS]** and **[AS]** two classes of statements for use only in **FORMAT ROUTINES** and whose significance will be explained under that heading. There is another format class associated with the master phrases, namely **[MP]**, but this does not generally interest the user except in so far as he must avoid the identifier **[MP]**. Further format classes may be introduced by the user if he prefers to define certain groups of statements (or formats) in this way rather than by phrase definitions. The **FORMAT CLASS** statement which reserves an identifier for use in this way has the form:

FORMAT CLASS [ABC]

where **[ABC]** is the identifier in question.

FORMAT

A format statement consists of the identifier of the format class to which the format is to be added followed by '=', then by the format itself. For example the statement

FORMAT [SS] = JUMP [LABEL]

would have the effect of adding the format **JUMP [LABEL]** to the class of source statement formats **[SS].**

PSEUDO–IDENTIFIERS

It is obvious from above that in primary material the symbols `,` and `[` play meta–syntactical roles. Some other means has therefore to be used to indicate the appearance of these symbols within phrases, formats and other primary statements. The same applies to the codes associated with: end of line space and erase which are ignored in all primary material, except routines where end of line is retained because it is used to separate instructions. The following pseudo–identifiers are therefore provided:

`[ , ]`	to indicate	`,`
`[ [ ]`	" "	`[`
`[EOL]`	" "	end of line
`[SP]`	" "	space
`[ERASE]`	" "	the erase symbol

Thus if a source language statement contains a comma, `[ , ]` will be used to indicate this fact in the format, and if this statement were used as an instruction in a format routine `[ , ]` would still be used, but in actual source language the symbol , is substituted. Since the comma is so frequently used as a separator a further pseudo–identifier is provided for use in formats which can be replaced by ',' when they are used as instructions within format routines. It is `[COMMA]` and can only be used in members of format classes `[AS]` and `[BS]`.

SOME EXAMPLES OF THE USES OF PHRASES AND FORMATS

As an example of the use of phrases definitions and formats in the syntactic definition of a language consider the following definition of general arithmetic instructions in MERCURY Autocode. That is, instructions such as:

```
Z(I-1) = 4.5 JK B3C/ π − A'CJ+C10 B(J+4)/3
PHRASE [±] = +, −
PHRASE [INDEX] =I, J, K, L, M, N, O, P, Q, R, S, T
PHRASE [V−LETTER] = A, B, C, D, E, F, G, H, U, V, W, X, Y,
                                                    Z, π
PHRASE [V−LETTER'] = [V−LETTER] ', BUT NOT π'
PHRASE [SUBSCRIPT] = [N], [INDEX], ([INDEX] [±] [N])
PHRASE [VARIABLE] = [V−LETTER] [SUBSCRIPT], [V−LETTER'],
                                                [V−LETTER]
PHRASE [FACTOR] = [VARIABLE), [K], [INDEX]
PHRASE [DIVISOR] = /[FACTOR]
PHRASE [TERM] = [FACTOR*] [DIVISOR?]
PHRASE [A−EXPR]= [ ±?] [TERM] [±TERM*?]
FORMAT [SS] = [VARIABLE] = [A−EXPR] [EOL]
```

The identifiers `[N]` and `[K]` represent the class of integer constants and the class of general constants respectively. They have not been defined above since they can be more conveniently defined as built–in phrases. This allows the decimal to

binary conversion to be carried out in the recognition phase and the actual number planted in the analysis record. If it were required to define them formally the following might be used:

```
PHRASE [D] = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
PHRASE [N]= [D]  [N],  [D]
PHRASE[K]=[N] . [N],  .[N],  [N].,  [N]
```

THE ROLE OF PHRASE DEFINITIONS AND [SS]

The formats in the format class **[SS]** and definitions associated with the phrase identifiers they involve are used by the expression recognition routine **(ERR)** in order to recognize to which source statement format a given sequence of characters from the input stream corresponds. At the top level this routine explores each alternative format in turn attempting to match it to the characters in the input stream. This process involves scanning the format and the input stream simultaneously and comparing their respective symbols. Whenever an identifier appears in the former, the **ERR** nests all the relevant counts and re–enters itself, this time to compare each of the alternatives in the specified phrase definition with the input stream. This recursion proceeds until at some level either the whole of one particular alternative is matched with the source string or until all the alternatives are exhausted without correspondence being achieved. In the former case control is returned to the level above and the next symbol or identifier in the current phrase is examined. If all the alternatives at a particular level are exhausted without success then control is again returned to the level above but this time the current alternative at this level is abandoned and the next one is explored. A record referred to as the analysis record is produced during this process which indicates which alternative phrase (i.e. its category number) was recognized at each level. When an instruction is eventually identified its analysis record is handed on to the associated format routine.

At each level it is the first alternative to match the source string which is accepted, no further alternatives at that level will be considered even though subsequent symbols or identifiers at the level above fail to match. It is interesting to note that because of this, for example, the standard method of writing the function digits of an Atlas instruction (namely, a binary digit followed by three octal digits or just three octal digits in which case zero is assumed for the binary digit) would have to be defined as

```
[FD] = [BD] [OD] [OD] [OD] , [OD] [OD] [OD]
```

(where **[BD] = 0 , 1**   and **[OD] = 0, 1, 2, 3, 4, 5, 6, 7)**

and not as

```
[FD] = [BD?] [OD] [OD] [OD]
```

The second definition will fail when the three octal digit form is used if the first is either an 0 or 1. In this case the 0 or 1 would be associated with **[BD?]** instead of the first **[OD]**. In the first alternative of the first definition the same thing would happen but when the third **[OD]** is not recognized the next alternative will be taken.

PRE–EDITING

A pre–editing routine (**R142**) exists in the compiler compiler. This routine is called each time a line of source program is reconstructed to edit the line before it is analysed. It removes all space and erases symbols from the line. The user may replace this routine by one to edit the source program in any other way he chooses. On entry to the routine **B62** is the address of a circular list (see later) containing the reconstructed line. The routine must generate a new list containing the edited line and set **B61** to its address. The original list must not be changed in any way.

FORMAT ROUTINE

To each member of a format class corresponds a format routine. This describes the meaning of the format or more precisely the action to be taken when an instruction of that form appears in a source program (or is to be interpreted in another format routine). For statements of an imperative nature the action will be to add the equivalent set of machine instructions to the target program; but in the case of declaratives, for example instructions defining store mapping strategy, the action will be to enter certain information in lists for reference by subsequent format routines.

The first line of each routine contains the routine heading. This indicates the format and format class with which the routine is associated. Its syntax may be described thus:

[format class] **[EQV]** [the format in question]

where **[EQV]** is defined in the appendix. The format routine associated with **[VARIABLE] = [A–EXPR] [EOL]** for example would start

    ROUTINE[SS] ≡ [VARIABLE] = [A–EXPR] [EOL]

The identifiers appearing in a routine heading, which in a format serve only to define syntax, are in a routine regarded as names referring to the principal expressions which comprise the format. That is, when an entry to a format routine is caused by the appearance of a particular instruction (i.e. one containing only basic symbols and not identifiers) of the form whose meaning the routine defines, the identifiers in the heading will be associated with the particular forms of phrase which have been substituted for them in the instruction. These identifiers may appear in the instructions of the routine, and the expressions associated with them will be substituted before each instruction is interpreted. The format written in the routine heading should generally be a copy of that written in the **FORMAT** statement. If however, the format contains identifiers defined as contract record **(CR)** phrases these identifiers must not appear in the routine heading; Instead particular forms of these phrases should be substituted.

Each instruction in a format routine may be from either the format class **[BS]** in which case it is called a basic statement (or built–in instruction), or from the format classes **[AS]** and **[SS]** and called a sub–statement. Basic statements are interpreted by conventional routines (i.e. routines containing only machine orders) built in to the system, whereas sub–statements are interpreted by their associate format routines. The built–in routines are constructed so as to use only a reserved set of *B*–lines and other machine registers, and the routine changing sequence takes advantage of this in bypassing some of the protective nesting of work space, etc.,

which is normally carried out. The difference between members of **[SS]** and **[AS]** is that those of the former may appear both in source language and in format routines but the latter (i.e.. auxiliary statements) can only appear in format routines. The instructions of a format routine break up the definition of the format into convenient logical steps. Auxiliary statements are introduced by the user to effect those operations for which no built–in or source statement exist. In MERCURY Autocode, for example the format

                [VARIABLE] = [A–EXPR] [EOL]

might be defined by the format routine

        ROUTINE [SS] [VARIABLE] = [A–EXPR] [EOL]
                ACC = [A–EXPR] [VARIABLE] = ACC
                END

The meanings of the auxiliary statements **ACC = [A–EXPR]** and **[VARIABLE] = ACC** (i.e. to compile orders to compute the **[A–EXPR]** in the accumulator, and to transfer the contents of the accumulator to the store register associated with the **[VARIABLE]**, respectively) will themselves be defined in terms of simpler auxiliary statements and/or basic statements and so on, until at the bottom level only basic statements will be used.

Any member of **[BS], [AS]** or **[SS]** used in a format routine may be a completely particular instruction (i.e. one consisting entirely of basic, or composite, symbols) or it may contain *parameters*, that is identifiers for which particular phrases are to be substituted (thus particularizing the instruction) each time it is obeyed. The instructions of a format routine should be separated by a comma or any number of newline codes, even in the case of formats which do not terminate with the phrase **[SEP]** (see Appendix). If an instruction overflows one line, continuation on the next line is indicated by starting that line with a solidus (/).

The syntax of the basic statements and some preloaded auxiliary statements is defined in the Appendix. Their meaning is described below.


BUILT–IN INSTRUCTIONS
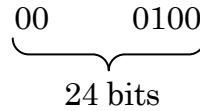### Basic listing instructions

These instructions are provided for manipulating 24–bit words and compiling lists of such words, for example the target program. Associated with them is a central group of 24–bit registers denoted by β_1, β_2, β_3 (or B_1, B_2, B_3). Also there is a further set of 24–bit registers α_1, α_2 α_3 (or A_1, A_2 A_3) local to each routine. Neither can be regarded as a field and referred to as α_i or β_i. The β–registers will be associated with '*B*–lines' and only β_1, $\rightarrow$ β_{40} are available to the user. In the case of the αs there is no practical limit but a block of 24–bit registers, whose size is determined by the highest α subscript appearing in a routine, is reserved in the main working area whenever that routine is in use.

In the words which the basic listing instructions manipulate, and these may be either the **[αβ]** registers themselves or store registers whose addresses are

contained in the **[αβ]** registers, a binary point is always assumed before the last two digits thus:

$$\underbrace{XX \qquad X}\,.\,\underbrace{XX}$$

$$\uparrow \quad 22 \text{ bits} \quad 2 \text{ bits}$$
$$\text{most significant end}$$
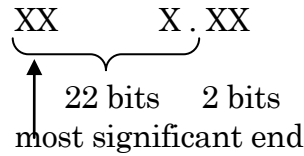
for example unity is represented by the word

$$\underbrace{00 \qquad 0100}$$

$$24 \text{ bits}$$

This is the most natural placing of the binary point in words which are used as 24–bit store line addresses, since adding one in the above scale will advance an address by one 24–bit line. When decimal integers are written in basic listing instructions they are converted to 24–bit numbers scaled as above and with their bottom two bits zero. The bottom two bits can be written into by using a substitution for **[WORD]** (see Appendix) which involves **[0−3]**. For example, 4.01 would be represented by the 24 bits 0 ... 010001. If a particular pattern of binary digits is required, say for use as a mask, the **[OW]** (octal word) form of **[WORD]** may be the most convenient. For example:

$$\textbf{111111111111000000000000}$$

may be written ***77770000** or just ***7777** since zeros at the least significant end may be dropped.

The instructions available for manipulating words are the following:

(1)   **[AB] = [WORD] [SEP]**

All arithmetic associated with **[WORD]**s is performed modulo 2^{21}.

The effect of this instruction is to compute the value of the **[WORD]** and place the result in the specified **[AB]**. An example is

$$\alpha_1 \ = \ \beta_3 \ + \ 4$$

whose effect would be to replace the number in register α_1, by the result of adding **4** to the number in β_3. The second alternative form of **[WORD]** namely **([ADDR])** requires further explanation. When the quantity **[ADDR]** is enclosed in parentheses it is interpreted as meaning the content of the store line whose address is given by **[ADDR]**. Thus:

$$\beta_3 \ = \ \alpha_3 \ + \ 4$$

means replace the content of β_3 by the result of adding **4** to the number in the store line whose address is given by the content of α_3. The significance of the operator '(**+**)' (see definition of **[ADDR]**) will be described later.

(2)   **[AB] = [WORD] [OPERATOR] [WORD] [SEP]**

This instruction is similar to (1) but permits a more complex R.H.S. For example:

$$\alpha_3 \ = \ \beta_4 \ - \ (\alpha_3 \ + \ 4)$$

which means replace the content of α_3 by the result of subtracting from the content of β_4 the number in store line α_3 **+** **4**. It should be noted with this type of

instruction that the two `[WORD]`s are computed before the `[OPERATOR]` is applied thus :

$$\beta_3 \; = \; 4 \; \times \; \beta_1 \; - \; 1$$

is interpreted as

$$\beta_3 \; = \; \{ \; 4 \; \} \quad \times \quad \{ \; \beta_1 \; - \; 1 \; \}$$

$$\underset{\text{1}^{\text{st}} \; \texttt{[WORD]}}{\uparrow} \qquad \underset{\text{2}^{\text{nd}} \; \texttt{[WORD]}}{\uparrow}$$

and not as might be expected from the usual precedence rules of algebra, and parentheses cannot be used to alter the meaning thus,

$$\beta_3 \; = \; (4 \; \times \; \beta_1) \; - \; 1.$$

(3)   `([ADDR]) = [WORD] [SEP]`

and (4) `([ADDR]) = [WORD] [OPERATOR] [WORD] [SEP]`

In these two instructions the R.H.S. is calculated as above. The result however is placed in the store register whose address is contained in a specified `[ADDR]`. For example:

$$(\alpha_1) \; = \; (\beta_4) \; \times \; (\alpha_2)$$

replaces the content of the store register whose address is contained in , by the product of the numbers in the store registers whose addresses are specified by β_4 and α_2.

(5)   `PLANT [FD] [COMMA] [ABN] [COMMA] [ABN] [COMMA] [WORD] IN`
`[B] [SEP]`

This instruction is provided for compiling Atlas machine instructions. The `[B]` in question is taken as the address of the first of a pair of 24–bit registers where this instruction is to be compiled, and it is advanced by 2 in the process. The instruction compiled is that produced by replacing the three words by their values. When this instruction is used to compile object program, β_1 should be substituted for `[B]` since β_1 is always preset to the address of the area where the object program is to be compiled.

(6)   `[FD] [COMMA] [WORD] [COMMA] [WORD] [COMMA] [WORD] [SEP]`

This represents a basic machine order to be executed (i.e. interpreted) at the point where it appears in the format routine. It is provided as an 'escape' for those situations in which the basic listing instructions are inadequate.

Control transfers

For the purpose of control transfers a floating address system is used and any instruction within a format routine can be preceded by an integer label, e.g.

3) $\beta_3 \quad = \; 4 \; \times \; \alpha_5$

There is no practical limit to the number of labels which can be used but a label directory will be recorded with each routine whose size will be determined by the highest label number which appears in the routine. Built–in instructions are provided which effect both conditional and unconditional transfers of control to labelled instructions.

(1)   `[JUMP] [LABEL] [SEP]`

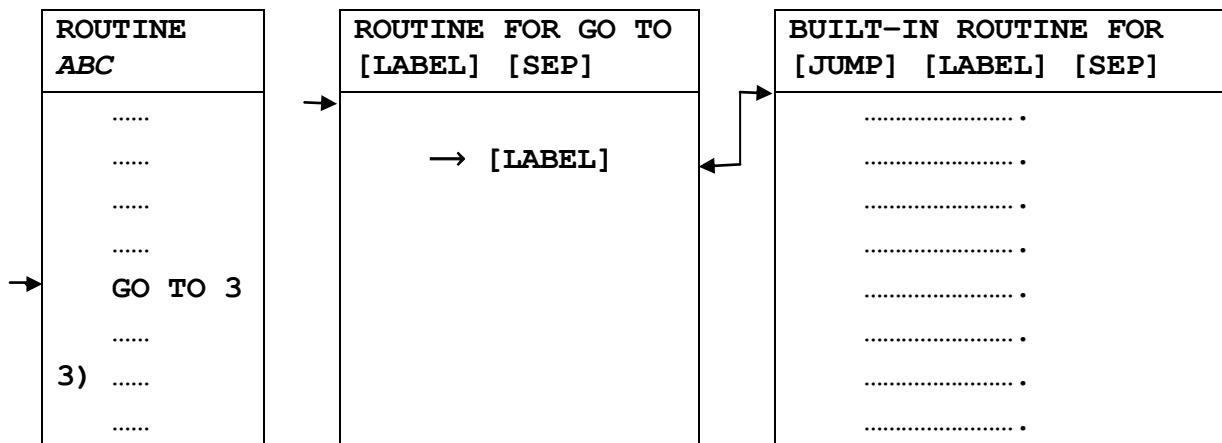An example of the unconditional jump instruction is

$$\longrightarrow \textbf{3}$$

This would have the effect of transferring control to the instruction labelled **3** within the same format routine. An **α** or **β** would also be a permissible substitution for the identifier **[LABEL]** but parameters as in **⟶ [N/1]** are not. The two instructions **α = [N/1]**, **⟶ α₁** should be used instead of **⟶ [N/1]**. The reason for this is that the phrase **[LABEL]** is built–in and parameters cannot be substituted into built–in phrases. It has been made a built–in phrase so that its analysis record can be made to contain a reference to the format routine in which it appears. This means that further jump orders, say **GO TO [LABEL]**, could be introduced as auxiliary statements, then in the associated format routine, namely:

    ROUTINE. [AS] ≡ GO TO [LABEL] [SEP]

            ⟶ [LABEL]

            END

when the parameter **[LABEL]** is substituted into the built–in instruction **[JUMP] [LABEL]** it will contain the necessary information to enable control to be transferred back to the specified label of the routine in which the expression represented by **[LABEL]** was first written.

    Consider for example a routine in which the instruction **GO TO 3** appears.

ROUTINE *ABC*	ROUTINE FOR GO TO [LABEL] [SEP]	BUILT–IN ROUTINE FOR [JUMP] [LABEL] [SEP]
.... **GO TO 3** ...... **3)**	**⟶ [LABEL]**	....

    When this instruction is reached control is transferred to the **GO TO [LABEL]** routine as shown, which at the **⟶ [LABEL]** instruction further transfers control down to the built–in routine for **[JUMP] [LABEL]**, which has finally to transfer control back to label **3** in the routine **ABC**. Thus the analysis record of **[LABEL]** which is carried forward in the process must contain the label number **3** and a reference to the routine **ABC**.

    (2) **[JUMP] [LABEL] [IU] [WORD] [COMPARATOR] [WORD] [SEP]**

    In this instruction the values of the two **[WORD]**s are computed and the predicate **[WORD] [COMPARATOR] [WORD]** is evaluated. If this is true, when the phrase **[IU]** takes the form **IF**, **[JUMP] [LABEL]** will be executed, otherwise control passes to the next instruction of the routine. The **UNLESS** form of **[IU]** reverses this procedure. Because the arithmetic is carried out modulo 2^{21} this test

may go wrong if: (1) the two **[WORD]**s differ in sign, and (2) the sum of their moduli is out of range (i.e. negative modulo 2^{21}).

(3) **CALL R [ABN] [SEP]**

This instruction transfers control to the first instruction of the routine whose 'serial number' is given by **[ABN]**. A link is nested and control will return to the next instruction when an **END** is encountered in the new routine. Almost any routine which has been built into the system can be used in this way and their specifications will be published elsewhere. Other routines of this type (referred to as system routines) can be introduced by the user through the format routine mechanism. The only difference is that the heading:

[format class**] [EQV] [**the format in question**]**

is replaced by

R [N]

where the **[N]** is the serial number to be associated with the routine which follows. Any of the instructions which are allowed in format routines can be used in system routines but it is unlikely that they will contain parameters (i.e. phrase identifiers). It is a convention that $\beta_{61} \longrightarrow \beta_{69}$ are used as the parameters (in the conventional sense) of system routines and basic listing instructions can be used to operate on this group of βs. Serial numbers $1000 \rightarrow 1023$ are reserved for additional system routines which a user may require.

(4) **CALL R [PI] [SEP]**

The identifier **[PI]** can in general be replaced by any phrase identifier but in the above instruction only a format class identifier is allowed. Its function is to transfer control to the format routine associated with the particular format which the format class identifier represents, and thus to interpret this format. (See also '**[ABN] = CATEGORY OF [PI]**' below.)

(5) **END [SEP]**

This instruction should be written at the end of every logical path through a routine. It causes control to return to the previous routine.

(6) **[FD] [COMMA] [WORD] [COMMA] 0 [COMMA] L [LABEL] [SEP]**

The **[LABEL]** in the address part of this instruction is interpreted as a label number in the usual way and when the instruction is obeyed its address part is replaced by the associated control number. Although any function code can be substituted for **[FD]** only those which represent control transfers will be sensible. It is provided mainly so that conditional accumulator testing instructions can be employed in routines which manipulate floating point numbers.

Parameter testing resolving, etc.

Sometimes the meaning of one source statement can be expressed as a sequence of less complex statements (or suitably chosen auxiliary statements) whose parameters are the principal expressions of the first statement. See for example the format routine for **[VARIABLE] = [A–EXPR] [SEP]** which was given earlier. In many cases however it is the sub–expressions associated with the parameters of a routine heading which are to be substituted into the sub–statements or basic statements of the routine. Some basic statements are therefore required to resolve expressions into the sub–expressions consistent with their known structure. Also if an expression can have several alternative forms it is necessary to have basic statements to discriminate between them and to switch control to different sequences of instructions. It is also necessary to be able to construct new expressions from existing ones.

The formats of the basic statements for carrying out these and other parametric operations contain the identifiers **[PI]**, **[RESOLVED–P]** and **[GENERATED–P]**. The identifier **[PI]** represents the class of phrase identifiers and can be replaced by an ordinary identifier (e.g. **[TERM]**, **[FACTOR*]**), or a labelled identifier (e.g. **[TERM/1]**, **[TERM/2]**, **[FACTOR*/1]**), or in the case of 'starred' identifiers, a phrase index may also be employed (e.g. **[FACTOR*/1 (α_1)]**, **[± TERM*(1)]**, **[±TERM*(β3)]**). It can also appear as a parameter in its own right (e.g. **[PI]**, **[PI/1]**) but we will consider the implications of this later. The phrase index is a device by means of which a particular sub–expression in the sequence of sub–expressions associated with a 'starred' identifier can be referred to. It is the only case in which a sub–expression of an expression on hand can be referred to without the expression being formally resolved. The required sub–expression is specified either directly by means of an integer or indirectly by means of an α or β, and for this purpose the sub–expressions are considered to be numbered consecutively from the left starting at 1. For example if a particular **[±TERM*]** is **+ABC – DEF**, then **[±TERM*(1)]** will represent **+ABC** and **[±TERM*(2)]** will represent **–[DEF]**. Also if α_1, = 2, then **[±TERM(α_1)]** will represent **–DEF**. In any format a 'starred' identifier with an index is a permissible substitution for the same identifier without the star, except within a **[RESOLVED–P]** and some appearances of **[PI]** in which it is specifically forbidden (see below).

The identifiers **[RESOLVED–P]** and **[GENERATED–P]** cannot exist on their own and are always related to the **[PI]** which precedes them in the same format Any substitution made for them must be a phrase of a form which ism associated with the identifier which replaces the **[PI]**. For example if the **[PI]** was replaced by **[±TERM]** then **[RESOLVED–P]** or **[GENERATED–P]** might be replaced by phrases such as **[±TERM]** or **[±] [TERM]** or **[±] [FACTOR*] [DIVISOR?]**. They do not have formal phrase definitions but they might be regarded as being defined thus

> **[RESOLVED–P]   = [P]**
>
> **[GENERATED–P] = [P]**

where the **[P]** is dynamically replaced by whatever identifier occurs in place of the preceding **[PI]**. This does not apply if the preceding **[PI]** is replaced by its own parametric form (say **[PI/1]**) and in this case the **[P]** would remain unset.

Therefore, only parametric **[RESOLVED-P]**s and **[GENERATED-P]**s (e.g. **[RESOLVED-P/1]**) can be used if the preceding **[PI]** is itself parametric. The implications of this will be considered later, and the other properties of **[RESOLVED-P]** and **[GENERATED-P]** will be apparent from the descriptions of the formats in which they appear.

(1) **LET [PI] [EQV] [RESOLVED-P] [SEP]**

The function of this instruction is to match a given expression on hand, whose identifier is substituted for **[PI]**, to the phrase substituted for **[RESOLVED-P]**. This last phrase need not be completely particular and any identifiers it contains will thereafter be associated with the corresponding sub–expressions of the **[PI]**. It must not however contain identifiers with phrase indices. An example is

LET [±TERM*/1] ≡ [±TERM] [±TERM*/2]

which would associate the first term of **[±TERM/1]** with **[±TERM]** and the rest with **[±TERM*/2]**. The form of an R.H.S. phrase is not restricted to that which is written in the phrase definitions for the L.H.S. identifier, since further substitutions can be made for the identifiers in the R.H.S. All the substitutions which are made must be consistent with foregoing phrase definitions, and whenever a choice of alternatives is made it must be consistent with the form of the **[PI]** expression which it is known will occur in practice. Thus the above instruction would only be sensible if it had been previously established that **[±TERM*/1]** was associated with more than one signed term. Without making further assumptions about the nature of **[±TERM*/1]** it could not be expanded beyond

LET [±TERM/1] ≡ [±] [FACTOR*] [DIVISOR?] [TERM*/2]

Since identifiers can be re–used dynamically as in conventional languages, an instruction such as

LET [± TERM* (α₁)] ≡ [±] [TERM]

could be used in a loop of instructions in which α_1 varied between 1 and the maximum number of **[±TERM]**s in **[±TERM*]**. At each pass through this instruction the two relevant sub–expressions of the α_1th **[±TERM]** would be associated with **[±]** and **[TERM]**.

(2) **[JUMP] [LABEL.] [IU] [PI] [EQV] [RESOLVED-P] [SEP]**

This instruction is used in those situations where an expression may have more than one form. If the expression has the form substituted for **[RESOLVED-P]** then the instruction will have the same effect as **LET [PI] [EQV] [RESOLVED-P]**, after which control will be transferred to the specified instruction if **[IU]** takes the form **IF**, or the next instruction in the case of **UNLESS**. If the expression corresponding to the **[PI]** does not match **[RESOLVED-P]** control will be switched in the reverse fashion to the above and no new sub–expressions will result. For example consider the instruction:

→ 2 IF [±TERM] ≡ [±TERM] [±TERM*]

Now if **[±TERM*]** initially represents more than one signed term this instruction will associate **[±TERM]** with the first and re–associate **[±TERM]** with the rest, then transfer control to the instruction labelled **2**. Otherwise **[±TERM*]**

must represent only one signed term and this could be referred to in succeeding instructions as `[±TERM*(1)]` or it could be formally resolved thus:

$$\text{LET } [±TERM*] \equiv [±TERM]$$

but it must not be referred to as `[±TERM]` without first being formally resolved.

    (3)   `LET [PI] = [GENERATED-P] [SEP]`

In this instruction the identifier substituted for `[PI]` is one with which a new expression is to be associated, and it must not involve a phrase index. The expression in question is that substituted for `[GENERATED-P]` and must be of a form which can be derived from the phrase definitions of the preceding identifier and any other identifiers which this involves. It may be a completely particular expression thus:

$$\text{LET } [± \text{ TERM*}] =+ abc/e - gh$$

or it may contain parameters, e.g.

$$\text{LET } [±TERM*] = [± \text{ TERM*(2)}] \ [± \text{ TERM*(1)}]$$

which would associate `[±TERM*]` with a new expression consisting of its previous first two signed terms in reverse order.

    (4)   `[JUMP] [LABEL] [IU] [PI] = [PI] [SEP]`

This instruction compares the analysis records for two expressions of like kind. These will only be 'equal' if the expressions look identical (except for sub–expressions replacing phrases with contracted out analysis records which can be ignored). In this sense *bac/e* is not equal to *abc/e*.

    (5)   `[AB] = NUMBER OF [PI] [SEP]`

Only 'starred' identifiers can be substituted into the R.H.S. of this instruction. Its function is to set `[AB]` equal to the number of expressions in the repeated sequence. Thus, if `[± TERM]` represents four signed terms,

$$\alpha_1 = \text{NUMBER OF } [± \text{ TERM}]$$

would be equivalent to $\alpha_1 = 4$.

    (6)   `[AB] = CATEGORY OF [PI] [SEP]`

This instruction examines the expression associated with the identifier which is substituted for `[PI]` and determines to which alternative form in the phrase definition of that identifier the expression corresponds. For example, if `[±TERM]` were associated with a single signed term:

    $\alpha_1 = \text{CATEGORY OF } [±TERM*]$

would set $\alpha_1 = 2$, since `[± TERM*]` is defined as `[± TERM] [± TERM*]`, `[± TERM]`.

Whereas in a format class the meaning of each alternative is defined by a routine and the category numbers of the alternatives are the serial numbers of the routines in question, meanings are very often assigned to the alternatives of a phrase definition by using the multi–way switch

$$\alpha_1 = \text{CATEGORY OF } [PI]$$
$$\longrightarrow \alpha_1$$

The meanings of the various alternatives would then be coded at the points labelled l), 2), 3), etc. It is when the number of alternatives is large that the format class becomes more convenient and in this case the multi-way switch is replaced by

$$\textbf{CALL R [PI]}$$

and the meaning of each alternative would then be defined in a separate format routine.

The identifiers **[PI]**, **[RESOLVED–P]** and **[GENERATED–P**] can be used as parameters in the usual way. Thus additional parameter manipulating instructions can be introduced as auxiliary statements. However if a **[RESOLVED–P]** or a **[GENERATED–P]** is used in a format it must be associated with a preceding **[PI**], for reasons given earlier. Because the identifier substituted for **[PI]** in any particular example of such an auxiliary statement will be local to the routine in which it appears, the analysis record for **[PI]** will contain, in addition to the identifier substituted for it, a reference to the routine in which it appears. Thus when a parameter such as **[PI]** is handed down to a format routine associated with an auxiliary statement for manipulating parameters, it can be substituted into a built–in instruction and the 'action' will take place in the routine in which the auxiliary statement was a sub–statement.

One use of this facility might be to represent an existing instruction by a different format thus:

```
FORMAT [AS] = RESOLVE [PI] INTO [RESOLVED–P] [SEP]
ROUTINE [AS] ≡ RESOLVE [PI] INTO [RESOLVED–P] [SEP]
        LET [PI] ≡ [RESOLVED–P]
        END
```

A further use would be to define extensions to the existing group of instructions such as the following instructions for testing if an expression has one of two alternative forms.

```
FORMAT [AS] = [JUMP] [LABEL] [IU] [PI] ≡ [RESOLVED–P]
                                [COMMA] [RESOLVED–P] [SEP]
ROUTINE [AS] ≡ [JUMP] [LABEL] [IU] [PI] ≡ [RESOLVED–P/1]
                            / [COMMA] [RESOLVED–P/2] [SEP]
        ⟶ 1 IF [PI]   ≡   [RESOLVED–P/1]
        ⟶ 1 IF [PI]   ≡   [RESOLVED–P/2]
        ⟶ [LABEL] IF [IU] ≡UNLESS
        END
   1)   ⟶ [LABEL] IF [IU] ≡ IF
        END
```

A particular example of the use of this instruction might be:

```
    ⟶ 3 UNLESS [VARIABLE] ≡ [V–LETTER] [N] , [V–LETTER] [INDEX]
```

If the **[VARIABLE]** in question had either of the two specified forms then its first sub–expression would henceforth be associated with the identifier **[V–LETTER]**, and its second would be associated with either **[N]** or **[INDEX].**

The remaining three instructions of this section are provided mainly for operating on expressions associated with parametric **[PI]**s in which case the associated 'action' takes place in the routines in which the expressions occur. They may also be used with particular identifiers substituted for the **[PI]**s and in this case the 'action' will take place in the same routine.

(7) **[AB] = CLASS OF [PI] [SEP]**

The function of this instruction is to determine the internal 'serial number' (see TREES and ROUTINES) of the class of phrase whose identifier replaces **[PI]**. If the parameter **[PI]** or **[PI/1]**, etc., is used the instruction will determine the serial number of the identifier associated with this parameter.

(8) **[AB] = ADDRESS OF [PI] [SEP]**

This instruction sets the **[AB]** in question to the address of the analysis record associated with the identifier which replaces **[PI]**. If a parametric **[PI]** is substituted the address of the analysis record associated with the identifier which the parametric **[PI]** represents will be obtained.

(9) **[PI] = [AB] [SEP]**

This instruction is for carrying out the reverse operation to (8). That is the identifier substituted for **[PI]** is henceforth associated with the analysis record whose address is given by **[AB]**. No attempt is made to check that this is a valid analysis record. The identifier substituted for **[PI]** in this instruction must not involve a phrase index.

SOME EXAMPLES OF FORMAT ROUTINES

The routines which define the meanings of the previously introduced auxiliary formats **ACC = [A–EXPR] [SEP]** and **[VARIABLE] = ACC [SEP]** and other auxiliary formats used in the process are given below. In practice the same meaning can usually be defined in several different ways and some variations in programming style will be evident in the routines given (for example two different techniques are illustrated for dealing with 'starred' sequences). It is not possible to formulate precise rules for determining the most efficient style, but one which results in the least number of instructions (either from **[BS], [AS]** or **[SS]**) being executed during the translation of any particular source statement should be near the optimum. In general basic statements without parameters will take the shortest time ; basic statements with parameters will take several (perhaps $10 \rightarrow 20$) times longer; and other parametric statement will take a comparable time to the latter plus the times for the individual 'instructions' in the associated format routines. More precise information can be derived from the description of the mechanics of the system given in 'TREES and ROUTINES'.

The phrase definitions given earlier will be assumed in what follows and only the required additional ones given.

Ambiguities

The order in which formats are introduced is generally significant, because they are scanned in this order and the instruction on hand is accepted as an example of the first format with which it matches: The problems which arise are similar to those involved in ordering the alternatives in a phrase definition. Thus **ACC = [± ?] [FACTOR] [SEP]** precedes **ACC = [± ?] [TERM] [SEP]**, for since the former is obviously a special case of the latter, it would never be recognized if they were introduced in the reverse order. For example, in one of the routines which follows, namely :

```
ROUTINE [AS] ≡ ACC = [± ?] [TERM] [SEP]
        LET [TERM] ≡ [FACTOR*] [DIVISOR?]
        …

        …

        ACC = [± ?] [FACTOR*(1)]
        …

        …
```

the sub–statement **ACC = [± ?] [FACTOR*(1)]** would be recognized as is parametric form of **ACC = [± ?] [TERM] [SEP]** and processed accordingly. When the routine came to be used in translating source material, this instruction would cause the routine **ACC = [± ?] [TERM] [SEP]** to re–enter itself. This is not, however, recursion; but merely a tight cycle where **[TERM]** is endlessly resolved into the same factor. Note also that the phrase **[SEP]** is essential in order to avoid another kind of ambiguity which is as follows.

If the formats were introduced in the correct order but without the phrase **[SEP]**, then a sub–statement such as **ACC = π[FACTOR] / 180**, appearing in a routine would not be recognized as a form as **ACC = [± ?] [TERM]**; instead **ACC = π.** would be recognized as a form of **ACC = [± ?] [FACTOR]**. An attempt would then be made to recognize **[FACTOR] / 180** (and what followed it) as some other format, and in general the machine would not recognize this and would stop. With the **[SEP]** added, however, recognition is not completed until either a comma or a new–line symbol is encountered. Thus in the above, the format **ACC = [±?] [FACTOR] [SEP]** would be rejected and others attempted until **ACC = [±?] [TERM] [SEP]** was encountered. The use of the phrase **[SEP]** eliminates all 'ambiguities of stems' as the above in reality are, and it is therefore recommended that all **[AS]** formats be terminated in this manner. In case there exist ambiguities between any of the basic listing instructions and any **[AS]** or **[SS]** formats which the user has introduced, then the latter classes of instruction may be distinguished by writing an asterisk in front of them. Formally, then, an instruction is defined as:

[BS], [ASTERISK ?] [AS], [ASTERISK ?] [SS]

and this is the order of preference used to identify instructions in a format routine.

There is no means of resolving ambiguities between members of **[AS]** and **[SS]** but since the formats of **[AS]** are chosen by the user, judicious choice should eliminate all these.

Apart from these considerations formats (and phrases) can be defined in any order providing they are always defined before they appear explicitly in format routines.

In the examples below the instructions of the form **PLANT [FD], [WORD], [WORD], [WORD] IN [AB]** are not written correctly. Instead of **[FD]** a symbolic description of the required operation is used.

```
FORMAT [AS] = ACC = ACC + DUMP [SEP]
FORMAT [AS] = ACC = [± ?] [FACTOR] [SEP]
FORMAT [AS] = ACC = [± ?] [TERM] [SEP]
FORMAT [AS] = ACC = ACC [OP] [FACTOR] [SEP]
FORMAT [AS] = ACC = ACC [ ±] [TERM] [SEP]
FORMAT [AS] = ACC = [A-EXPR] [SEP]
FORMAT [AS] = DUMP ACC [SEP]
FORMAT [AS] = [AB] = ADDRESS OF [K] IN NUMBER LIST [SEP]
FORMAT [AS] = [AB] [COMMA][AB] = ADDRESS AND MODIFIER OF
                                       [VARIABLE] [SEP]
PHRASE [OP] = ×, /, [±]


ROUTINE [AS] ≡ ACC = [A-EXPR] [SEP]
        LET [A-EXPR] ≡ [± ?] [TERM] [± TERM*?]
        ACC = [± ?] [TERM]
        ⟶ 1 UNLESS [± TERM*?] ≡ [± TERM*]
    3)  ⟶ 2 UNLESS [± TERM*] ≡ [±] [TERM] [± TERM*]
        ACC = ACC [±] [TERM]
        ⟶3
    2)  LET [± TERM*] ≡ [±] [TERM]
        ACC = ACC [±] [TERM]
    1)  END


ROUTINE [AS] ≡ ACC = [± ?] [TERM] [SEP]
        LET [TERM] ≡ [FACTOR*] [DIVISOR?]
        α₁ = NUMBER OF [FACTOR*]
        α₂ = 1
        ACC = [± ?] [FACTOR*(1)]
        ⟶ 1
    2)  α₂ = α₂ + 1
        ACC = ACC × [FACTOR*( α₂)]
    1)  ⟶ 2 UNLESS  α₂ = α₁
        ⟶ 3 UNLESS [DIVISOR?] ≡ / [FACTOR]
        ACC = ACC / [FACTOR]
    3)  END
```

```
ROUTINE [AS] ≡ ACC = ACC [ ±] [TERM] [SEP]
        →1 UNLESS [TERM] ≡ [FACTOR]
        ACC = ACC [±] [FACTOR]
        END
   1)   DUMP ACC
        ACC = [±] [TERM]
        ACC = ACC + DUMP
        END
```

The above routine illustrates the use of a parameter testing instruction in order to recognize a special case for which a more optimum translation can be provided than that which would otherwise result. A price which is paid for this, however, is an increase in compiler time and in the space occupied by the compiler. In general, some sort of compromise must be reached.

```
ROUTINE [AS] ≡ ACC = [±?] [FACTOR] [SEP]
     α₁ = CATEGORY OF [FACTOR]
     → α₁
1)   LET [FACTOR] ≡ [VARIABLE]
     α₂, α₃ = ADDRESS AND MODIFIER OF [VARIABLE]
5)   → 6 UNLESS [±?]  ≡   −
     PLANT (A = −S), 0, α₃, α₂ IN β₁
     END
6)   PLANT (A = S), 0, α₃, α₂ IN β₁
     END
2)   LET [FACTOR] ≡ [K]
     α₂ = ADDRESS OF [K] IN NUMBER LIST
     α₃ = 0
     → 5
3)   LET [FACTOR] ≡ [INDEX]
     α₂ = CATEGORY OF [INDEX]
     → 7 UNLESS [±?]  ≡  −
     PLANT (A = −n), 0, α₂, 0, IN β₁
     END
7)   PLANT (A = n), 0, α₂, 0, IN β₁
     END
```

In the above routine it is assumed that *B*–registers 1–12 are to contain the numbers associated with the index letters i to t. Decisions of this kind, relating to the mapping of the object program and its working space into the computer store, are generally made before the compiler is started.

```
ROUTINE [AS] ≡ ACC = ACC [OP] [FACTOR] [SEP]
```

```
         α₁ = 4
         ⟶ 1 IF [OP] ≡ −
         α₁ = CATEGORY OF [OP]
  1)     α₂ = CATEGORY OF [FACTOR]
         α₂ = α₂ + 1
         ⟶ α₂
  2)     LET [FACTOR] ≡ [VARIABLE]
         α₃, α₄ = ADDRESS AND MODIFIER OF [VARIABLE]
 14)     α₁ =α₁ + 5
         ⟶ α₁
  6)      PLANT (A = A × S), 0, α₄, α₃ IN β₁ , END
  7)      PLANT (A = A / S), 0, α₄, α₃ IN β₁, END
  8)      PLANT (A = A + S), 0, α₄, α₃ IN β₁, END
  9)      PLANT (A = A − S), 0, α₄, α₃ IN β₁, END
  3)      LET [FACTOR]  =  [K]
         α₃ = ADDRESS OF [K] IN NUMBER LIST
         α₄ = 0
         ⟶ 14
  4)     LET [FACTOR] = [INDEX]
         α₃ = CATEGORY OF [INDEX]
         α₁ = α₁ + 9
         ⟶ α₁
 10) PLANT (A = A × n), 0, α₃, 0 IN β₁ , END
 11) PLANT (A = A / n), 0, α₃, 0 IN β₁ , END
 12) PLANT (A = A + n), 0, α₃, 0 IN β₁ , END
 13) PLANT (A = A − n), 0, α₃, 0 IN β₁ , END
```

Before the routines for the remaining auxiliary formats can be written we must make some assumptions about the way the computer store is to be allocated to variables and constants. Therefore, let us assume that at the beginning of each MERCURY Autocode source program a routine is entered which sets the following βs

β_2 = the address of the first twenty–nine 48–bit registers to be used for the variables **A', B', ..., Z', A, B, ... Z, π** respectively.

β_3 = the address of the first fifteen 24–bit registers to be used as the variable directory of each chapter.

Let us also assume that the variable directory will be cleared at the beginning of each chapter (i.e. by the format routine associated with the format **CHAPTER [N])**. However, a copy of the variable directory associated with each chapter must be retained elsewhere in order to translate **VARIABLES [N]**. The entries are made in the variable directory when directives such as **A ⟶ 10** are encountered. The first register in the directory will be associated with '**A**', the second with '**B**' and so on, and

the entry made for each will be the base address of the vector in question (i.e. **A₀**, **B₀**, etc.).

β_4 = the address of the first register in an area reserved for constants and miscellaneous working. The that of these will be used as the accumulator dump and β_5 will contain twice the number of constants in the rest of the list (initially zero).

```
ROUTINE [AS] == [AB] = ADDRESS OF [K] IN NUMBER LIST [SEP]
        α₁ = ADRESS OF [K]
        β₅ = β₅ + 2
        α₂ = β₅ + β₄
        (α₂) = (α₁ + 1)
        (α₂ + 1) = (α₁ + 2)
        [AB] = α₂
        END
```

This routine presupposes that the analysis record of **[K]** consists of three words, **B₂ X X**, where the **X**s are the two halves of the floating–point number in question, an explanation of **B₂**, can be found in TREES AND ROUTINES. One way this routine might be improved is by writing a sequence of machine orders to test if the number in question is already in the list, and using this as its address instead of adding it.

```
ROUTINE [AS] ≡ DUMP ACC [SEP]
        PLANT (A ⟶ S), 0, 0, β₄ IN β₁
        END
ROUTINE [AS] ≡ ACC = ACC + DUMP [SEP]
        PLANT (A = A + S), 0, 0, β₄ IN β₁
        END
ROUTINE [AS] ≡ [AB/1] [COMMA [AB/2] =  ADDRESS AND MODIFIER
                              / OF [VARIABLE] [SEP]
        [AB/2] = 0
        α₁ = CATEGORY OF [VARIABLE]
        ⟶ α₁
    1)  LET [VARIABLE.] == [V–LETTER] [SUBSCRIPT]
        α₁ = CATEGORY OF [V–LETTER]
        α₁ = α₁ − 1
        [AB/1] = (α₁ + β₃)
        ⟶4 IF [AB/1] ≠ 0
        MONITOR (VARIABLE [SP] NOT [SP] SET)
    4)  α₂ = CATEGORY OF [SUBSCRIPT]
        α₂ = α₂ + 4
        ⟶ α₂
    5)  LET [SUBSCRIPT] ≡ [N]
```

```
            [AB/1] = [AB/1] + [N]
            END
   6)     LET [SUBSCRIPT] ≡ [INDEX]
   8)     [AB/2] = CATEGORY OF [INDEX]
            END
   7)     LET [SUBSCRIPT] ≡ ([INDEX] [±] [N])
            [AB/1] = [AB/1] [±] [N]
            ⟶ 8
   2)     LET [VARIABLE] ≡ [V-LETTER]
            α₁ = CATEGORY OF [V-LETTER]
            [AB/1] = α₁ + β₂ − 1
            END
   3)     LET [VARIABLE] ≡ [V-LETTER]
            α₁ = CATEGORY OF [V-LETTER]
            [AB/1] = α₁ + β₂ + 13
            END
```

Note: The instruction **MONITOR (…)** is an auxiliary statement which causes the symbols enclosed in the brackets to be output together with some other information which will indicate where the fault occurred in the source program. Although the object program compiled after such a fault occurs cannot be used, the compiler is allowed to continue in order to locate any further possible faults.

The examples given above do not illustrate how the syntax and semantics can sometimes be defined recursively. In order to do this let us now consider a hypothetical autocode, similar to MERCURY Autocode, but allowing parentheses in the general arithmetic expression **[A-EXPR]**. Thus, an example of an arithmetic instruction in this language might be:

$$\text{B}_{10} = \text{ABC } (\text{E}_1 \text{ -- } \text{G}_1(45.7 + \text{H}(_{\text{J}} - _1))/10 + 63.5 \text{ B}(_{\text{J} - 3}) \text{ (A + D)}$$

The only change that this would require in the previously defined syntax of **[A-EXPR]** is that the definition of **[FACTOR]** be replaced by

$$\textbf{PHRASE [FACTOR] = [VARIABLE], [K], [INDEX], ([A-EXPR])}$$

Note that the order of preference is such that **[V-LETTER]**s followed by **[SUBSCRIPT]**s of the form **([INDEX] [±] [N])** would be recognized as such rather than as the product of a **[V-LETTER]** and an **([A-EXPR])**. A result of this is that if a particular expression of the form **([A-EXPR])** has the same structure as the above alternative of **[SUBSCRIPT]** it must not be used in a position where a **[SUBSCRIPT]** is a legal substitution. For example, **+ A (I + 10)** would always be interpreted as **+ A($_{I + 10}$)** and if **A** multiplied by **(I + 10)** is intended it should be written **+(I + 10) A**.

In order to interpret the meaning of this kind of **[A-EXPR]** the following alterations must be made to the format routines already defined. Firstly provision must be made for a 'nest' of accumulator dumps. That is, assuming β_7 to be the address of a group of registers to be used as the accumulator dump nest and the β_6

the current position in this nest, initially zero, the format routine for **DUMP ACC [SEP]** becomes

  ROUTINE [AS] ≡ DUMP ACC [SEP]

    PLANT (A ⟶ S), 0, 0, β_6 + β_7 IN β_1

    β_6 = β_6 + 2

    END

The format **ACC = ACC + DUMP [SEP]** has now to be re-defined thus:

  ROUTINE [AS] ≡ ACC = ACC + DUMP [SEP]

    β_6 = β_6 − 2

    PLANT (A + S ⟶ A), 0,0, β_6 + β_7 IN β_1

    END

Also the format **ACC = DUMP [SEP]** which recovers the last value dumped will be required. However, the format statement defining this format cannot be written after the formats already defined since syntactically it is a special case of **ACC = [±?] [TERM] [SEP]** and **ACC = [A-EXPR] [SEP]**. It should therefore be inserted before **ACC = [±?] [FACTOR] [SEP]**. The associated routine is

  ROUTINE [AS] ≡ ACC = DUMP [SEP]

    β_6 = β_6 − 2

    PLANT (S ⟶ A), 0,0, β_6 + β_7 IN β_1

    END

Finally, the format routines for **ACC = [±?] [FACTOR] [SEP] and ACC = ACC [OP] [FACTOR] [SEP]** should be extended thus:

  addition to the routine for **ACC = [±?] [FACTOR] [SEP]**

   **4) LET [FACTOR] = ([A-EXPR])**

    ACC = [A-EXPR]

    ⟶ 8 IF [±?] ≡

    END

    8) PLANT (A = − A + n), 0, 0, 0 IN β_1

    END

addition to the routine for **ACC = ACC [OP] [FACTOR] [SEP]**

    5) LET [FACTOR] = ([A-EXPR])

    DUMP ACC

    ACC = [A-EXPR]

    DUMP ACC

    β_6 = β_6 − 2

    ACC = DUMP

    α_3 = β_6 + 2

    α_4 = 0

    ⟶ 14

PRELOADED AUXILIARY FORMATS

Most of these are concerned with manipulating information in lists and dictionaries. Two kinds of list are available. Firstly there is the conventional list in which consecutive items are recorded in consecutive store registers. With this kind of list it is necessary to estimate its size in order to allocate an appropriate area in the computer store. Since many lists required by compilers vary with different source programs, to allocate a safe maximum area to each would be wasteful. A second kind of list is therefore provided which does not assume any particular relative positioning of items within the computer store and can be extended as required. The items in these lists (called chain lists) are connected by means of a link. That is, for each item two consecutive storage registers are used where the item itself is recorded in the first and the second is reserved for the address of the pair of registers containing the next item. The address part of the last pair of registers in the list always contains the address of the first item in the list, and the address of the last pair of registers is referred to as the address or location of the list. Diagrammatically a *circular* list or chain (as it is called) of three items is

location of list
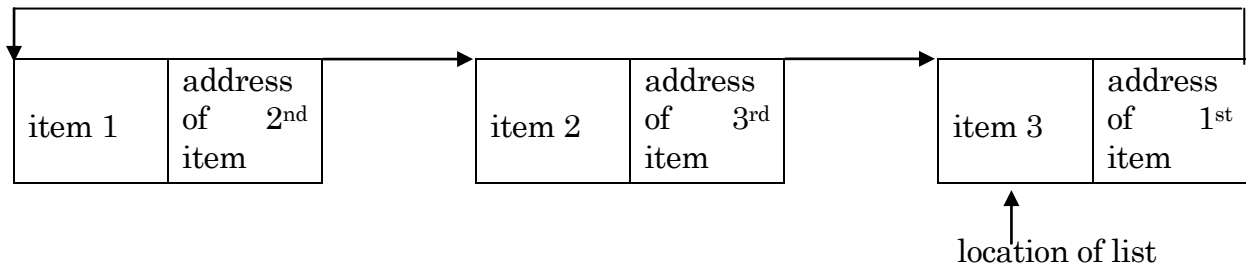
An empty list is represented by the address 0. All the preloaded auxiliary statements for manipulating circular lists assume these conventions. Initially all the register pairs in the area of store allocated for circular lists are linked together and the address of the first is in β_{89}. Words are removed from and returned to this main chain as required by the appropriate adjustment of links and the β_{89} register.

The operator '**(+)**' which appears in one alternative of **[ADDR]** namely **[AB] (+) [ABN]** is concerned with the chain type of list. Its interpretation is such that if β_4 is the address of one item in a chain that β_4 **(+) 1** is the address of the next, $\mathbf{B}_4$ **(+) 2** is the address of the one after that, and so on.

If β_2 is the location of a circular list then the instruction

$$\alpha_4 = (\beta_2 \ (+) \ 5)$$

copies the fifth item in the list into α_4. Behind the scenes the nth item is located by tracing through the links of the first $n - 1$ items. It is always more efficient therefore to scan the whole list systematically if this is possible. That is to set some **[AB]** to the address of the first item and to deal with this item and then move on the **[AB]** to the next item by the instruction **[AB] = ([AB] + 1)** and so on.

Conventional lists

Two areas of store are available for use as conventional lists. One of these is the area where the α list of a routine is placed. It is therefore local to a routine since the space occupied by these lists is recovered when control exits from the routine

(i.e. when an **END** is encountered). The address of the next available register in this area is contained in β_{90}. If a list of n registers is needed for local working in a routine then β_{90} should be copied to the α chosen to be the address of the list and β_{90}, must then be advanced by n if the area is to be protected from interference by further subroutines that might be called in.

A second area of working store is used for conventional lists required by more than one routine. Because this area is also the main working area of the system, and because information in this area may be moved about by the system, access is indirect. Associated with it is an index which contains the address of every item in the area. Although the position of an item may change it will always be associated with the same index position, and items are therefore referred to by their index positions or 'serial numbers'. The process of setting up a new item in this store for use as a conventional list is best done by means of the instruction:

$$\texttt{[AB] = CONVENTIONAL LIST OF [ABN] WORDS [SEP]}$$

and lists of this kind are deleted by the instruction:

$$\texttt{DELETE CONVENTIONAL LIST [AB] [SEP]}$$

No further auxiliary formats are provided for manipulating information in these lists since the basic listing instructions seem adequate. It must be remembered that while access to the first kind of list was direct that to the second kind of list is indirect. For example, if a list 10 words long is set up by the instructions:

$$\alpha_1 = \beta_{90}$$
$$\beta_{90} = \beta_{90} + 10$$

then the fourth word can be made unity by writing
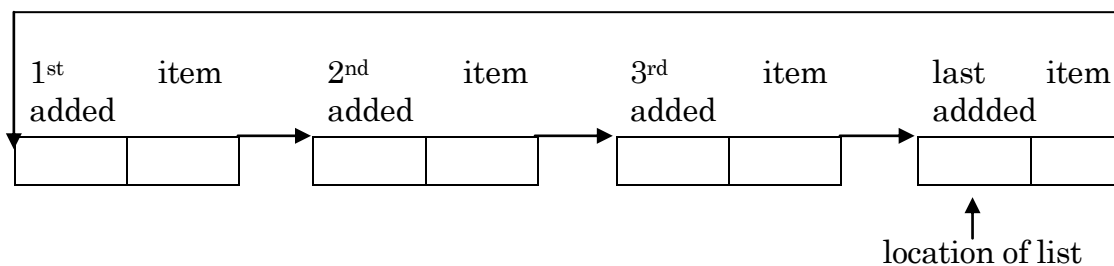
$$(\alpha_1 + 3) = 1$$

but if the list were set up thus:

$$\beta_3 = \texttt{CONVENTIONAL LIST OF 10 WORDS}$$

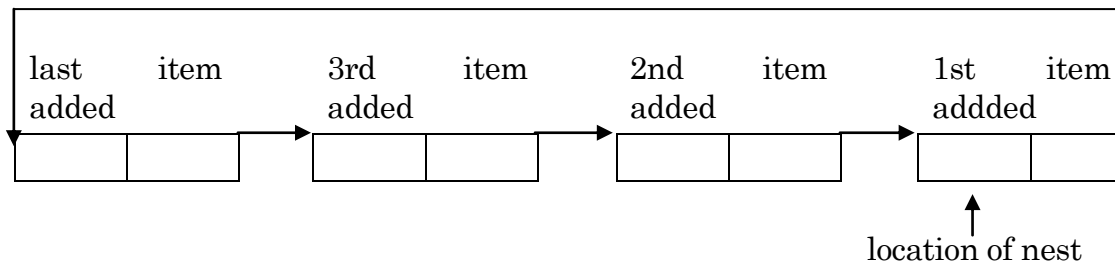then the same action would be represented by

$$\alpha_1 = (\beta_3)$$
$$(\alpha_1 + 3) = 1$$

Circular chain lists and nests

Both lists and nests have the structure described above and only differ in the way they are used. If a circular chain is referred to as a list when being constructed the result will be:


location of list

and if it is referred to as a nest it will be:

last item added		3rd item added		2nd item added		1st item addded	

location of nest

Thus items in a list are most easily processed or removed from the first through the last, whilst in the case of a nest this order is reversed. In other words, a nest is a last-in first-out device sometimes referred to as a pushdown list.

(1) [AB] = [LIST OR NEST] [WORD] [SEP]

This is the instruction which is used to set up a new list or nest of one item, namely the value of the specified **[WORD]**, and its address is recorded in **[AB]**. There is no difference in this case between using the words list and nest, both may be represented diagrammatically as
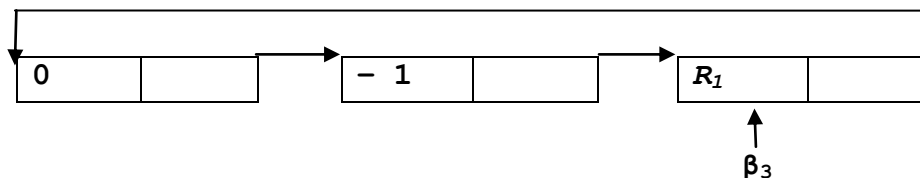
[WORD]	

[AB]
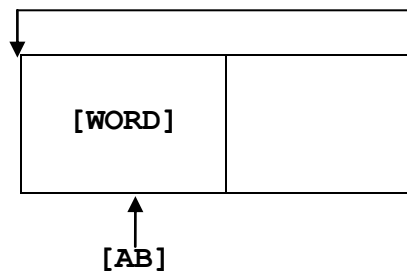
(2) [AB] = [LIST OR NEST] ([WORD] [,WORD*]) [SEP]

By this instruction a new list or nest can be set up which contains a sequence of **[WORD]**s. If a list is set up the first item will contain the value of the first **[WORD]** and the order of the rest will be preserved, but if a nest is set up this order will be exactly reversed so that the last **[WORD]** comes first. For example:

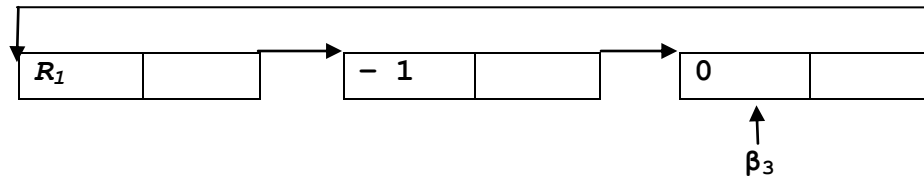$$\beta_3 = \text{LIST } (0, -1, \beta_1 + 5)$$

would produce the circular list:

0		− 1		R_1	

β_3

where R_1 is the result of adding **5** to β_1 and

$$\beta_3 = \text{NEST } (0, -1, \beta_1 + 5)$$

would produce the circular list:

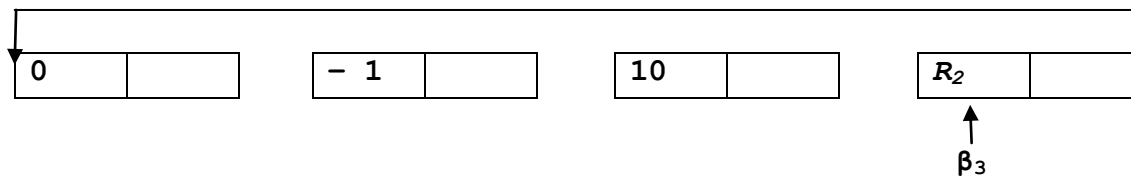| R_1		-1		0	

β_3

(3) `ADD [WORD] TO  = [LIST OR NEST] [AB] [SEP]`

Thus instruction is most easily explained diagrammatically as follows. Let the list β_3 have the structure:

0		-1		10	

β_3
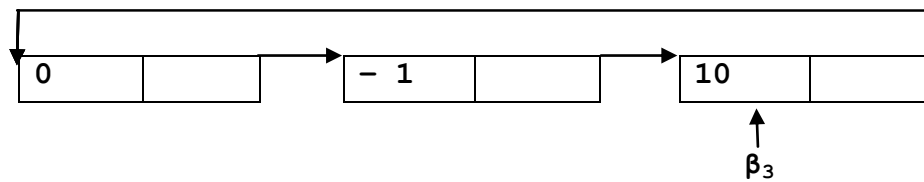
then the instruction `ADD α4 + 2 TO LIST β3` would transform it to:

0		-1		10		R_2	

β_3

where R_2 is the result of adding **2** to α_4. To operate on the nest

10		-1		0	

β_3
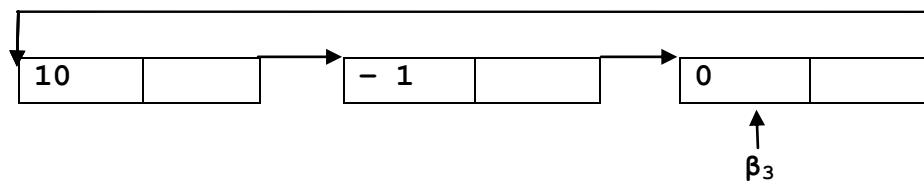
with an instruction such as `ADD α₄ + 2 TO NEST β₃` would result in:

R_2		10		-1		0	

β_3
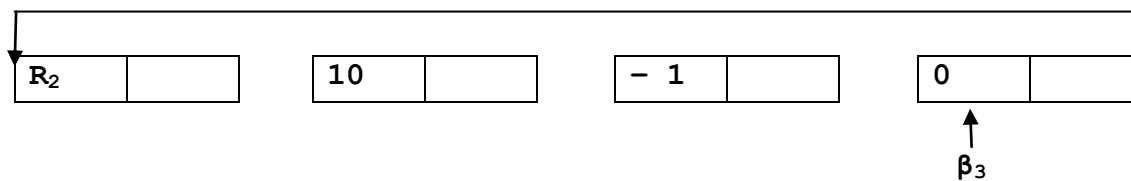
(4) `ADD ([WORD] [,WORD*]) TO  = [LIST OR NEST] [AB] [SEP]`

This instruction has the effect of adding the sequence of words in the specified order starting from the left. Its meaning is in fact, defined as follows:

```
ROUTINE[AS]≡ADD([WORD][WORD*])TO[LIST OR NEST][AB][SEP]
        1)   ADD [WORD] TO [LIST OR NEST] [AB]
             ⟶ 1 IF [.WORD*] ≡ ,[WORD] [WORD*]
             LET [,WORD*] = ,[WORD]
             ADD [WORD] TO [LIST OR NEST] [AB]
             END
```
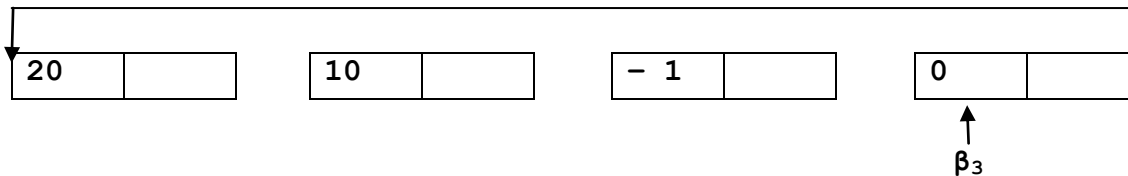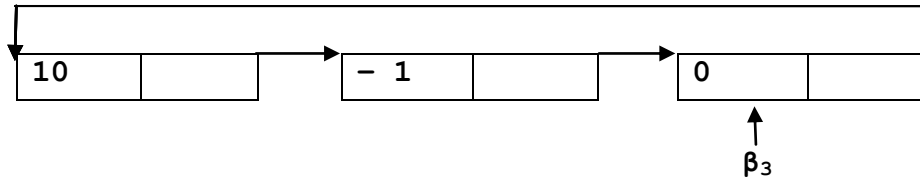
(54) `WITHDRAW [AB] FROM NEST [AB] [SEP]`

This instruction copies the last entered word in the nest `[AB]` into the other specified `[AB]`, and removes this entry from the nest. For example, given the nest

The effect of `WITHDRAW α7 FROM NEST β3` would be to set $α_7$ to **20** and to transform the nest to

Attempting to withdraw a word from an empty nest (e.g. $β_3$ `= 0`) will have no effect (i.e. the specified `[AB]` will not be altered).

(6) `DELETE [LIST OR NEST] [AB] [SEP]`

Whenever a list or nest is no longer required it should be linked back into the main chain so that the storage registers it includes can be used again in other lists. The above instruction carries out this action.

(7) `[AB] = LIST [AB] ([AB],?) [SEP]`

Very often it is required to keep a table of the argument-function value type, say one which relates labels to control numbers. If this table is recorded in a circular list by adding word pairs to the list with the instruction

`ADD ([WORD], [WORD]) TO LIST [AB]`

where the first `[WORD]` is the argument and the second `[WORD]` is the function value, the above instruction may be used to look up the function value associated with a given argument. The example $α_1$ `= LIST` $β_4$ `(3,?)` would record in $α_1$, the

value corresponding to the argument **3** in **LIST** β_4. If the required value does not appear in the list α_1, will be unaltered.

   (8) [AB] = LIST [AB] + LIST [AB] [SEP]

This instruction joins together the two specified lists. The result is a list in which the first item of the second list (on the R.H.S.) follows the last item of the first list and the relative ordering of all the other items in these two lists is unaltered. Since the last item of the second list thus becomes the last item of the new list, the location of the new list (i.e. the content of the **[AB]** on the L.H.S.) will be equal to the location of the second of the original lists. Although the two original lists will cease to exist as separate lists the two **[AB]**s which contain their locations will not be altered (unless the same **[AB]** appears on the L.H.S.). The two lists must be distinct. That is, a list cannot be added to itself.

Dictionaries

   Dictionaries are used to record 'symbol strings' and the single word (i.e. 24-bit) identifiers with which they are associated, in a manner which facilitates conversion from the former to the latter. The 'symbol strings' used in this connection are recorded in circular lists where each item in the list is regarded as a symbol. The two digits after the binary point must be zero (or 11) but the remaining 22 digits may be used to describe the symbol. The dictionaries themselves are recorded in circular lists, and an empty dictionary is represented by a list containing one item only, namely zero. An empty dictionary whose address is in β_{10} can therefore be set up by the instruction 'β_{10} = **LIST 0**'. In cases where the original sequence of symbols representing an expression is required for entering or looking up in a dictionary the instruction

   (1) [AB] = LIST [PI] [SEP]

is provided. This regenerates the symbol string representing **[PI]** in a circular list and sets **[AB]** to its address. Each character will be contained in a separate word in the 7 bits immediately before the binary point (composite symbols will be represented by their internal serial number). The most significant bit of the 7 will represent the internal shift (0 for inner shift) and the remaining 6 bits the internal code for the character (see Atlas Manual). In the case of those characters (such as space) which may appear on both shifts, the outer shift form will be used. If the **[PI]** in question represents a built-in phrase or if its definition involves a built-in phrase then it will not in general be possible to recover that part of the original symbol string which corresponds to this built-in phrase. Instead the analysis record for the built-in phrase will be recorded in the list in place of the original symbols which represented it. This analysis record will be preceded by two other words. The first of these will be an *I* word (see TREES and ROUTINES) containing the serial number of the built-in phrase in question, and the second will be a Bn word (also defined in TREES and ROUTINES) containing the number of words in the analysis record. These additional words will be required by the expression recognition routine if the reconstructed string is to be re-analysed (see later). The analysis record of the built-in phrase in question must not contain & words (see TREES and ROUTINES again).

(2) `ADD LIST [AB] [COMMA] [WORD] TO DICT [AB] [SEP]`

This instruction adds the new symbol string contained in `LIST [AB]`, together with the value of `[WORD]` as its associated value to the `DICT [AB]`.

(3) `[AB] = VALUE OF LIST [AB] IN DICT [AB] [SEP]`

If the DICT [AB] has an entry identical to the symbol string contained in use `[AB]`, then the `[AB]` on the L.H.S. will be set to its associated value. Otherwise the L.H.S. `[AB]` will not be altered.

(4) `DELETE LIST [AB] FROM DICT [AB] [SEP]`

This instruction has the obvious action of removing an entry from a dictionary. The `LIST [AB]` is not deleted and no action is taken if the entry is not in the dictionary.

(5) `CONVERT [PI] TO [AB] [SEP]`

Two preset parameters are assumed by this instruction β_2 and β_3. The first must contain the address of a dictionary and the second a provisional value to be associated with the new entry if one is made. In this case β_3, would also be advanced by 1. Its first action is to convert the `[PI]` to a list then this is looked up in the dictionary. If it appears in the dictionary `[AB]` is set to the associated value, if not it is entered together with the value β_3. `[AB]` is then set to β_3 and β_3 is advanced.

Looking up a dictionary in reverse

It will sometimes be necessary to recover the string associated with a given value. This can be done with the format

`LIST [AB] = ENTRY WITH VALUE [AB] IN DICT [AB] [SEP]`

It is assumed that no two entries will have same value, otherwise the 'first' entry will be taken (see TREES and ROUTINES).

Other preloaded auxiliary formats

(1) `ASSIGN VALUE [AB] TO [PI] [SEP]`

The only substitution for `[PI]` that will be accepted by the format routine associated with this format is `[N]` or `[N/I],` etc. Its purpose is to enable an integer in an `[AB]` register to be dynamically associated with the `[N]` type identifier. This then permits the value of the `[AB]` to be incorporated into a format where the expression `[N]` is the only permitted substitution.

(2) `MONITOR ( [ALL SYMBOLS EXCEPT RT BRACKET] ) [SEP]`

Some inconsistencies in a source program which a compiler can recognize (e.g. the same control label used twice) do not prevent it from continuing to translate the remainder of the source program in order to detect further possible errors. The above instruction is provided so that the occurrence of these faults can be monitored'. That is whenever this instruction is obeyed the symbols enclosed in parentheses will be output (in channel 0) together with some information indicating which source statement was being translated. Any basic or composite symbol can

be substituted into this instruction and the two pseudo-identifiers **[EOL]** and **[SP]** can also be used to influence the layout.

    (3) ANALYSE LIST [AB] W.R.T. [PI] [SEP]

This instruction uses the expression recognition routine in order to compare the sequence of source symbols in the **LIST [AB]** with the alternative phrases represented by the identifier substituted for **[PI]**. The analysis record which results will be recorded in the local working area (i.e. **(B90)** →) and **B90** will be advanced to the next available register. The identifier substituted for **[PI]** will be associated with this analysis record and it can be subsequently substituted into sub-statements or parameter resolving instructions in the usual way. That part of the **LIST [AB]** which is recognized as a phrase of the specified form will be deleted (if all the **LIST [AB]** is deleted **[AB]** will be set to zero). If no phrase of the specified form can be recognized the program will be halted. This instruction can be used to re-analyse the symbol string produced by the instruction **[AB] = LIST [PI]**. However if the regenerated sequence contains the analysis records of built-in phrases (see earlier), the same built-in phrases must appear in the same position in the new class of phrase to which it is to be matched.

    It is not anticipated that this facility will be generally used (or in fact generally required) and some further knowledge of the system will be necessary in order to use it safely. Particular care should be exercised if certain

preloaded built-in phrases are involved (e.g. **[PI], [α]** and **[LABEL]**)

    (4) LIST [AB] = NEXT LINE FROM INPUT [N] [SEP]

    This instruction reads all the characters in the specified input stream up to and including the next newline code, and records them in the **LIST [AB]**. The characters are recorded one per word in the 6 bits before the binary point. All characters including shift characters will be recorded. The main we of this instruction is to look ahead in the source program and to obtain the actual characters that appear before they are subject to the 'line reconstruction' process and the 'ignore '. One use for this facility in a MERCURY Autocode Compiler (say) is in the format routine associated with the instruction **'CAPTION'**. This instruction means that the next line of characters have to be recorded in the compiled program together with some instructions to output them each time the **'CAPTION'** instruction is obeyed. The source program is always input stream 0 and we shall not discuss here the reason for allowing the above instruction to specify other input streams.

OTHER MASTER STATEMENTS

The master phrases such as PHRASE, FORMAT, etc., are members of a format class **[MP]**, and it is possible to introduce new master phrases into the system, and to define the 'meaning' of these by means of format routines. For example if it was required to introduce a further master phrase 'format' to have exactly the same meaning as **'FORMAT'** this could be done thus:

FORMAT [MP] = format

FORMAT [MP] ≡ format

LET [MP] = FORMAT

CALL *R* [MP]

END

The meaning of new master phrases would not generally be described as simply as this and would probably involve operating on the input stream, and recording information in the central record store. This means that a knowledge of the inner working of the system is required which is not given here. In fact it is unlikely that ordinary users will make private additions to the system in this way but it is a convenient means by which the system can be generally extended from time to time. Some additions which have already been made to the system are described below.

Built-in phrase statement

The built-in phrase statement is a means of associating a sequence of instructions (a 'built-in phrase routine') with a phrase identifier. When an identifier associated with a built-in phrase is encountered by the expression recognition routine, control is transferred to the associated built-in phrase routine. These routines are designed to recognize in the input stream all members of the class or phrase with which they are associated and to plant an analysis record. The main advantage they have over the more usual phrase definitions is that they can generate unorthodox analysis records. They must however satisfy the main conventions relating to analysis records (see TREES and ROUTINES).

A built-in phrase statement takes the form:

BUILT-IN PHRASE [IDENTIFIER]

    [the routine proper**]**

That is to say, the identifier as written on the first line is followed on subsequent lines by the instructions which comprise the routine.

The routine proper must obey the following rules :

(1) The only instructions which it may contain are parameter-free forms of the built-in instructions. Jump instructions are further limited to the →**1** and →**1** if β_1 = β_2 forms ; →α_1 and →β_1 are not permitted. In fact α_i must not be used anywhere since these routines are not provided with local **(α)** working space. Instructions involving **(+)** are also prohibited and basic machine orders must not involve β_i (i.e. **121, 92, 0, −1** is allowed but **121, 92, 0,** β_{93} is not).

(2) The following are the parameters of built-in routines:

β_{61} =   the address of a circular list containing all the symbols in the line of source material currently being examined.

$\beta_{62} =$ the address of the last recognized symbol in the above list so that $(\beta_{62} + I)$ is the first character to be examined by the built-in routine. Accordingly, β_{62} must be advanced to the last symbol of any expression recognized by the routine.

β_{70} is the link set by the ERR and the built-in routine should be terminated by $B_{127} = B_{70}$.

$\beta_{63} =$ next available register in the conventional list containing the A.R. An analysis record satisfying the conventions of A.R.s (see TREES and ROUTINES) has to be recorded here and β_{63} advanced to the next available register if an expression is recognized, otherwise β_{63} must not be altered.

Bt must be set on exit either +ve for success or –ve for failure by means of the appropriate machine order.

The only other βs which may be used are $\beta_{91} \rightarrow \beta_{97}$. As already mentioned these routines have no local (α) working space.

Primary compiling routines

A primary compiling routine can be associated with any member of the format classes **[BS], [AS]** and **[SS]**. Its function is to compile machine orders to replace non-parametric forms of the format with which it is associated. The primary compiling routines that are provided will be used by the routine which assembles format routines inside the machine in order to produce more efficient routines, as follows. Each time the routine assembling routine recognizes an instruction it tests if its analysis record contains any parameters. If it does then the analysis record is copied into the routine to be interpreted whenever the routine is subsequently used. If however the analysis record is non-parametric then the routine assembling routine examines the list of primary compiling routines to see if one has been provided for the current instruction. The instruction is treated as before if there is not one available, but if there is then it is called in to translate the instruction into machine code.

The primary statement for defining primary compiling routines closely resembles that for format routines, and only the word **COMPILER** is used to distinguish the two thus:

    ROUTINE (COMPILER) [BS] ≡ [AB] = [WORD] [SEP]

    …………………………………………………………………..

    …………………………………………………………..

Any of the usual basic and auxiliary statements can be used in these routines. The instructions representing the format should be compiled in the store registers (β_{88}) $\rightarrow$ and on exit β_{88} should be advanced to the address of the next available register. If the compiled instructions require some *B*-lines *B82, B83, B84* may be used. In some primary compiling routines it is convenient to select a few special cases of the format and to compile orders for these, but to exit without compiling the remainder and to leave the routine assembling routine to record their analysis records instead. If this course is followed β_{54} must be set negative before exit. One primary compiling routine can call another as a subroutine by means of the instruction:

```
            CALL [PI] COMPILER [GENERATED-P]
```

For example, the routine for `[AB] = [WORD] [SEP]` might be called by this routine which compiles `[AB] = [WORD] [OPERATOR] [WORD] [SEP]` thus:

```
    ROUTINE (COMPILER) [BS] ≡ [AB] = [WORD/1] [OPERATOR]
                                            [WORD/2] [SEP]

    CALL [BS/1] COMPILER B83 = [WORD/1]

    CALL [BS/1] COMPILER B82 = [WORD/2]

    α₁ = CATEGORY OF [OPERATOR]

    ⟶ α₁

    1) PLANT 0124, 83, 82, 0 IN B88

    CALL [BS/1] COMPILER [AB] = B83

    END

        "

        "

        "
```

Primary compiling routines will be provided for all the basic formats and some of the preloaded auxiliary formats. The user considering it worthwhile to introduce additional ones for compiling non-parametric forms of his auxiliary statements is recommended to examine the provided ones.

Small routines

These routines are a special kind of system routine and they differ from the normal system routine only because of the way they are entered. The same instruction, namely **CALL R [ABN]** is used in both cases but the routine changing sequence distinguishes the two kinds of routine and in the case of small routines bypasses all the protective nesting of the α work space links, etc., which is normally carried out. Small routines are thus entered more quickly but as a result are subjected to the following restrictions. They must not involve αs or phrase identifiers and the only kind of jump instructions they may contain are those in which the label is specified explicitly (e.g. ⟶ **3**). All the logical paths through a small routine should end with the instruction **β127 = β70** and not **END**. The heading for the small routine statement is

```
            ROUTINE SMALL R [N]
```

As in the case of system routines **[N]** is the serial number to be assigned to the routine and one of the reserved set 1000-1023 should be used.

REFERENCES

1.   Brooker, R.A. and Morris, D., 'An Assembly Program for a Phrase Structure Language'. *Computer J.*,**3**, No. 3 (1960).

2.   Brooker, R. A. and Morris, D., 'Some Proposals for the Realization of a Certain Assembly Program'. *Computer J.*,**3**, No. 4. (1961).

3.   Brooker, R. A and Morris, D., 'A Description of Mercury Autocode as a Phrase Structure Language'. *Annual Review in Automatic Programming*, Vol 2, Pergamon Press, Oxford (1961).

4.   Brooker, R.A. and Morris, D., 'A General Translation Program for Phrase Structure Languages'. *J..A.C.M.*, **9**, No. 1 (1962).

5.   Brooker, R.A., Morris, D. and Rohl,J.S., 'Trees and Routines'. *Computer J.*, **5**, No. 1 (1962).

APPENDIX

LIST OF BUILT-IN AND PRELOADED PHRASES AND FORMATS

The built-in phrases. are denoted by b in the left hand margin, and in these cases the 'definitions' given below serve only to indicate the type of expression which can be substituted for them and are not necessarily consistent with the corresponding analysis records (only in fact in the case of **[B]** and **[N].** A c in the left hand margin indicates that the analysis record of the phrase gets contracted out by the ERR.

PHRASES

b	**[A]** =	= A1, A2, A3…….A0	(≡ [α] = α1…….)
b	**[B]** =	B1, B2, B3….….B0	(≡ [β] = β1…….)
b	**[N]** =	= 1, 2, 3……....0	
	[OPERATOR] =	-, ×, /, &, V, ≢, AND, NOT EQV	
	[COMPARATOR] =	=, ≠, >, ≤, <, ≥,)	
	[0-3] =	00, 01, 10, 11	
	[AB] =	[A], [B]	(≡ [αβ])
	[ABN] =	[A], [B], [N]	(≡ [αβN])
	[ADDR] =	[AB] + [ABN], [AB] - [ABN], [AB](+)[ABN], [AB	
	[WORD] =	[ADDR], ([ADDR]), [-?][N].[0-3],[-?].[0-3], [-?][N],[OW]	
b	**[-]** =	-	
b	**[FD]** =	[BD][OD][OD][OD], [OD][OD][OD] where [BD] denotes a binary digit **0,1** and [OD] an octal digit **0, 1, 2, 3, 4, 5, 6, 7**	
	[OW] =	* followed by up to 8 octal digits starting with the most significant	
b	**[IU]** =	IF, UNLESS	
b	**[LABEL]** =	[ABN]	
b	**[PI]** =	general phrase identifier | see text for	
b	**[RESOLVED-P]** =	some **[P]** expression following a **[PI]** phrase | further details	
c	**[GENERATED-P]** =	some **[P]** expression following a **[PI]** phrase	
	[JUMP] =	->, →, JUMP	
	note :	the second alternative → corresponds both to the genuine → on Mercury/Pegasus teleprinter keyboards and to the compound symbol - and > on Atlas Flexowriters.	
c	**[EQV]** =	≡, (≡)	
	[,WORD] =	[COMMA][WORD]	
	[LIST OR NEST]=	LIST, NEST	
b	**[ALL SYSMOLS EXCEPT RT BRACKET]** =	any symbol except)	

SPECIAL PHRASES

`[COMMA]`	denotes a , in a format routine
`[,]`	denotes a , in source language
`[[]`	denotes a [in source language
`[EOL]`	denotes a newline in source language
`[SP]`	denotes a space in source language
`[ERASE]`	denotes a \\\ in source language
`[SEP]`	`[COMMA], [EOL]`

note: this phrase which follows all the following instruction formats indicates that
they must be terminated either with a , or a newline.

BUILT-IN FORMATS

```
[AB] = [WORD][SEP]
[AB] = [WORD][OPERATOR][WORD][SEP]
([ADDR]) = [WORD][SEP]
([ADDR]) = [WORD][OPERATOR][WORD][SEP]
PLANT[FD][COMMA][ABN][COMMA][ABN][COMMA][WORD] IN [B][SEP]
[FD][COMMA][WORD][COMMA][WORD][COMMA][WORD][SEP]
[JUMP][LABEL][SEP]
[JUMP][LABEL][IU][WORD][COMPARATOR][WORD][SEP]
CALL R [ABN][SEP]
CALL R [PI][SEP]
END[SEP]
[FD][COMMA][WORD][COMMA]0[COMMA][LABEL][SEP]
LET[PI][EQV][RESOLVED-P][SEP]
[JUMP][LABEL][IU][PI][EQV][RESOLVED-P][SEP]
LET[PI] = [GENERATED-P][SEP]
[JUMP][LABEL][IU][PI] = [PI][SEP]
[AB] = NUMBER OF [PI][SEP]
[AB] = CATEGORY OF [PI][SEP]
[AB] = CLASS OF [PI][SEP]
[AB] = ADDRESS OF [PI][SEP]
[PI] = [AB][SEP]
[AB] = INDEX [ABN][SEP]
INDEX[ABN] = [AB][SEP]
SHIFT [AB] UP [ABN][SEP]
SHIFT [AB] DOWN [ABN][SEP]
PRINT [ABN][SEP]
PRINT SYMBOL [ABN][SEP]
[WORD]/[WORD][SEP]
PRINT [ABN] IN OCTAL [SEP]
```

AUXILARY-IN FORMATS

```
[AB] = CONVENTIONAL LIST OF [ABN] WORDS [SEP]

DELETE CONVENTIONAL LIST [AB][SEP]

[AB] = [LIST OR NEST][WORD][SEP]

[AB] = [LIST OR NEST]([WORD][,WORD*])[SEP]

ADD [WORD] TO [LIST OR NEST][AB][SEP]

ADD ([WORD][,WORD*]) TO [LIST OR NEST][AB][SEP]

WITHDRAW [AB] FROM NEST [AB][SEP]

DELETE [LIST OR NEST][AB][SEP]

[AB] = LIST [AB]([AB,?)[SEP]

LIST [AB] = LIST [AB] + LIST [AB][SEP]

[AB] = LIST [PI][SEP]

ADD LIST [AB][COMMA][WORD] TO DICT [AB][SEP]

[AB] = VALUE OF LIST [AB] IN DICT [AB][SEP]DELETE LIST [AB] FROM
DICT [AB][SEP]

CONVERT [PI] TO [AB][SEP]

LIST [AB] = ENTRY WITH VALUE [AB] IN DICT [AB][SEP]

CALL [PI] COMPILER [GENERATED-P]

ASSIGN VALUE [AB] TO [PI][SEP]

MONITOR ([ALL SYMBOLS EXCEPT RT BRACKET])[SEP]

ANALYSE LIST [AB] W.R.T. [PI][SEP]

LIST [AB] = NEXT LINE FROM INPUT [N][SEP]

LIST [AB] = NEXT RECONSTRUCTED LINE [SEP]

PRINT LIST [ABN][SEP]
```

APPENDIX 1
SERIAL NUMBERS OF BASIC SYMBOLS

Octal			Octal			Octal			Octal		
00	spare		40			100	spare		140	spare	
01	used		41	A		101	SPACE		141	a	
02	used		42	B		102	spare		142	b	
03	used		43	C		103	spare		143	c	
04	EOL		44	D		104	EOL		144	d	
05	[		45	E		105	COMA		145	e	
06	used		46	F		106	spare		146	f	
07	used		47	G		107	spare		147	g	
10	(		50	H		110	spare		150	h	
11	)		51	I		111	spare		151	i	
12	,		52	J		112	spare		152	j	
13	π £ $		53	K		113	spare		153	k	
14	?		54	L		114	spare		154	l	
15	&		55	M		115	spare		155	m	
16	*		56	N		116	spare		156	n	
17	/		57	O		117	:		157	o	
20	0		60	P		120	Φ ×		160	p	
21	1		61	Q		121	[		161	q	
22	2		62	R		122	]		162	r	
23	3		63	S		123	$\rightarrow$		163	s	
24	4		64	T		124	$\geq$		164	t	
25	5		65	U		125	$\neq$		165	u	
26	6		66	V		126	$\_$		166	v	
27	7		67	W		127				167	w	
30	8		70	X		130	% $^{-1}$		170	x	
31	9		71	Y		131	$\approx$ ~		171	y	
32	<		72	Z		132	α		172	z	
33	>		73	spare		133	β		173	spare	
34	=		74	spare		134	½		174	spare	
35	+		75	spare		135	spare		175	spare	
36	−		76	spare		136	spare		176	spare	
37	.		77	fault		137	spare		177		

ADDENDA

The following `[BS]` instructions are not described in the text:

(i)   `[AB] = INDEX[ABN] [SEP]`

(ii)   `INDEX[ABN] = [AB] [SEP]`

(iii)   `SHIFT [AB] UP [ABN] [SEP]`

(iv)   `SHIFT [AB] DOWN [ABN] [SEP]`

(v)   `PRINT [ABN] [SEP]`

(vi)   `PRINT SYMBOL [ABN] [SEP]`

(vii)   `SPACE [SEP]`

(viii)   `NEWLINE. [SEP]`

(ix)   `[WORD] / [WORD] [SEP]`

(x)   `PRINT [ABN] IN OCTAL [SEP]`

Notes

(i) and (ii) enable the user to access the index of addresses of items in the main working area of the compiler, (see e.g. *Conventional Lists*, p. 259).

(iii) and (iv) perform circular shifting operations on 24-bit words.

(v) prints the signed integer in digits 23-2 of the 24-bit word `[ABN]` followed by a single space.

(vi) prints the basic or composite symbol represented by `[ABN]`.

(ix) permits 24-bit words to be inserted in a format routine.

(x) prints the 8 octal digits corresponding to the word `[ABN]`.

The following [`AS`] instructions are not described in the text:

(i) `LIST [AB] = NEXT RECONSTRUCTED LINE [SEP]`

(ii) `PRINT LIST [ABN][SEP]`

Notes

(i) The next line of input on the currently selected channel is recorded in the circular list `[AB]`.

(ii) prints the list of basic or composite symbols `[ABN]`.

The following is a complete list of the formats in the Master Phrase `[MP]` dictionary.

(i)   `PHRASE`

(ii)   `ITEM`

(iii)   `END OF MESSAGE`

(iv)   `FORMAT CLASS`

(v)   `FORMAT`

(vi)   `DELETE ITEM`

(vii)   `REPLACE ITEM`

(viii)   `END OF PRIMARY MATERIAL`

(ix)   `ROUTINE`

> (x)   **BUILT-IN PHRASE**
>
> (xi)   **DEFINE COMPILER**
>
> (xii)   **AMEND COPILER**

Notes

(ii) and (vii) were used in connection with a primitive type of routine routine as part of the bootstrapping of the compiler compiler into Atlas. Their use should be avoided.

(iii) serves to denote the end of a section of primary material when a source statement is to follow.

(vi) recovers the space occupied by an item in the working area of the compiler when it is no longer required.

(viii), (xi) and (xii). The facilities afforded by these master phrases is due to the existence of the Atlas Supervisor System and in order to appreciate their use, the reader should be conversant with the Ferranti documents R55 and R58.

These master phrases are followed on the same line by the compiler name which is subsequently to be referred to in Job descriptions.

The symbols following the Master phrase will be treated in the same manner as, say, a Phrase definition, e.g. spaces and erases will be ignored, and significant spaces are indicated by **[SP]** etc. Up to 8 symbols or identifiers may specify a compiler name; the remainder of the line will be ignored.

The action of **DEFINE COMPILER** is to record the name which follows it in the list of available compilers held by the supervisor system and to write the compiler to the compiler magnetic tape.

If the master phrase **END OF PRIMARY MATERIAL** precedes **DEFINE COMPILER**, the compiler defined will consist mainly of phrases, formats and routines which the user has defined, together with those routines of the compiler compiler which make it a compiler complete, and independent of the compiler compiler.

If it is not preceded by the master phrase **END OF PRIMARY MATERIAL** then it will contain all the facilities of the compiler compiler and therefore corrections and additions can later be made to this compiler in compiler compiler language.

The master phrase **AMEND COMPILER** is to be used for recording corrected compilers. It will carry out the defining operation as above but also arrange to lose the copy of the compiler to which the corrections were made. Obviously, if a job ends with **AMEND COMPILER** the compiler name following must be the same as that referred to in the Job description, and if **DEFINE COMPILER** is used it must be followed by a new name which did not previously appear in the supervisor's list of compiler names.

Monitoring and diagnostics

When phrases and formats are being read into the computer serial numbers are allocated. In subsequent fault-finding it is useful to know the serial numbers in question, and therefore these are printed out, together with the phrase or format as

they are allocated. The serial number of a phrase identifier is not necessarily printed out when it is defined but rather when it first appears. When a format is printed, the serial number of any phrase identifiers it contains is printed instead of the identifier itself.

An example of the layout is

```
P — WORD , 280
P — ADDR , 281
F — ([281]) = [280] , 282
```

where `P —` indicates that a phrase identifier follows and `F —` indicates that a format follows.

In order to facilitate the interpretation of fault monitoring which occurs during the compilation of **ROUTINES**, the start of each routine is monitored in the form

<p align="center">START OF R 462</p>

(If the routine happens to be a primary compiling routine (see P.270) this monitoring is preceded by

<p align="center">CV for R 212</p>

where 212 would be the serial number of the corresponding format routine).

The monitoring of faults detected in primary material by the compiler compiler takes the following form:—

<p align="center">COMPILER COMPILER FAULT m R n L p</p>

where:

m is the fault number

n is the routine number

p is the line number of the master phrase which follows the primary statement in question.

The nature of the fault can be deduced from the following table:

R215   (analysis)

      FAULT 1   Analysis w.r.t. undefined phrase

R218   (phrase)

      FAULT 1   Phrase definition not recognized (∴ it is ignored)

      FAULT 2   Phrase defined twice (∴ first one lost)

      FAULT 5   **BUT NOT** in wrong place(i.e. as 1st alternative)

      FAULT 6   **NIL** after a **BUT NOT**

      FAULT 7   **NIL** as the only alternative

R 220   (format)

      FAULT 1   Format not recognized (∴ it is ignored)

      FAULT 2   Format class not defined.

R 221   (routine)

	FAULT 1	Routine heading not recognized.
	FAULT 2	Format class not defined
	FAULT 3	Analysis record of routine heading at lower level than format e.g.

```
FORMAT[AS] = [VAR]  = ACC [SEP]
ROUTINE[AS] = x   = ACC [SEP]
```

	FAULT 4	Routine defined twice (first version is deleted)
	FAULT 5	`PI` in small routine
	FAULT 6	`α` in small routine

R 230	(PI conversion)	
	FAULT 1	`PI` not recognized, e.g. `[A/I]` instead of `[A/1]`
	FAULT 2	`PI` not declared before used in a routine
	FAULT 3	Index occurs in a non-* class
	FAULT 4	Spurious index in a `PI` ⎤ Only occurs in phrases and
	FAULT 5	Spurious label in a `PI` ⎦ formats

R 238	(line reconstruct)	
	FAULT 1	Too many (> 3) characters overprinted
	FAULT 2	Too many (>160) characters on a line

R 253	(body of routine)	
	FAULT 2	Instruction not recognized (the faulty instruction will also be printed
	FAULT 3	Compiling version required, not available
	FAULT 4	Label not set
	FAULT 5	Label set twice.

R272	(format class	
	FAULT 1	Format class defined twice
	FAULT 2	Too many (> 16) format classes defined

R292	(built-in phrase)	
	FAULT 1	Defined twice (first copy is deleted)
	FAULT 3	`A` in a built-in phrase.

When a compiler is translating source material two further kinds of faults may be detected. One is an illegal source statement in which case the line number within

the source program, followed by **INSTRUCTION NOT RECOGNISED** are printed and the faulty instruction is printed on the next line.

The other kind of fault is concerned mainly with the illegal use of parameter resolving instructions, and the fault print-out takes the following form:

COMPILER COMPILER FAULT* *m* **R** *n* **L** *p*

> All non-zero B-lines from B1 — B124 in octal

> The stack

To users who do not have an intimate knowledge of the compiler compiler, the only useful information is the Routine number, *n*, which would be the serial number of the routine containing the instruction which has been wrongly used.

INITIAL ENTRY TO A COMPILER

Provision has been made for the user to pre-set various parameters for his compiler each time the compiler is entered from the supervisor. For this purpose he must supply a routine whose serial number is 162. In addition to B-lines and store-lines which he may wish to set on entry, he should also set the following index registers:

index 136 = maximum number of lines in a single source statement

index 140 = maximum number of illegal instructions to be allowed before terminating translation

Thus routine 162 might begin:

```
ROUTINE R 162
    B91 = 2
    INDEX 136 = B91
    B91 = 25
    INDEX 140 = B91
```