Trees and Routines

By R. A. Brooker, D. Morris and J. S. Rohl

The concepts of a phrase structure language and of a compiler to process it have been presented before. In this paper, the internal structure of phrases, formats and routines are described. The operation and linking together of format routines and the routines of the system are also discussed.

In two earlier papers (Brooker and Morris, 1960 and 1961a) we described some proposals for a programming system for writing compilers. We describe here the structure and operation of these compilers. The system now employed is not precisely that which was described earlier, and the relevant changes will be mentioned in the sections which they affect.

It will be recalled that the basic elements of the system were: phrases, statement formats (now abbreviated to formats) and statement definitions (henceforth called format routines). The first two describe the syntax of the statements in the source language, and the last define their “meaning” (or more precisely an algorithm for converting them to object coding). “Translation” of a source program starts by matching each source statement to the corresponding format, and generating a tree (or analysis record) indicating which alternative of each expression and subexpression had to be chosen in order to make the match. The associated format routine is then called in to process this tree and generate the equivalent object coding. The internal structures of the phrases, formats, and format routines, and the operation and linking together of the format routines and some of the system routines, are the main topics discussed below.

The Layout of the Store

This is illustrated by the following diagram.

![Diagram of the Layout of the Store](https://example.com/diagram.png)

It has already been explained how the upper part of the store is used for the main chain and the miscellaneous working chains, and we shall now describe the function of the conventionally addressed sections.

The index is a consecutive set of 1,024 words (one Atlas block) giving the addresses of up to 1,024 items in the record store. These items are of four kinds and are distinguished by attaching an appropriate tag to each entry in the index, thus:

- large routines 00
- small routines 01
- phrase definitions 10
- built-in phrase definitions 11

The position of each entry in the index is the serial number of the item to which it “points.” New items are placed at the right-hand end of the record store starting in the next available location given by \( B_{88} \) (which denotes the content of B-line (modifier register) 88). We have described the use of B-lines elsewhere (Brooker and Morris, 1961b). If an item becomes redundant it is removed, and the items to the right of it are moved along to recover the space vacated. This will be a comparatively rare event, however, and fortunately so, as it will involve the individual transfer of several thousand words.

The index will not in general be full, and the spare positions are chained together, each spare register containing the serial number of the next, thus:

![Chained Spare Positions](https://example.com/chains.png)

The head of this chain is given by \( B_{87} \). When a new item is entered, an index position for it is borrowed from the head of the chain and \( B_{87} \) is updated, by resetting \( B_{87} \) to \( (B_{87} + \) the absolute address of index position 0). Similarly, when an item becomes redundant, its index position is returned to the head of the chain. In this event it is also necessary to adjust the entries in the index of all those items which are moved down. Those are the entries whose value is greater than the address of the item removed. The entire index is scanned, the spare positions being automatically ignored, since the addresses in them are all less than 1,024.

In addition to the record store, other sections of conventional storage are needed for the main working stack, the object program (A), and possibly other purposes. It would normally be necessary to estimate roughly the amount of storage required for each section, but on Atlas we can avoid this because of the nature of the composite “one-level” store. This allows us to place our landmarks at very wide intervals indeed, provided they lie in the total range \((0, 2^{21} - 1)\), e.g. 0, 2^{15}, 2^{16}, 2^{17}, etc., and the drum supervisor program (see Kilburn, Payne and Howarth, 1961) will allocate storage space in blocks of 512 words covering those addresses which are actually used.
Representation of dictionaries and other branching structures (trees)

A variation of the form described in (Brooker and Morris, 1961a) is used in which the distinction of terminal words is dropped. It also allows us, in certain cases, to use the dictionary in reverse, i.e. to find the string corresponding to a given category number. We refer to the example of an ordered dictionary given in that paper, namely:

```
A B C 1
A B E 2
S H E C D 3
S B C 4
A Z O 5
A B D 6
```

which is represented by the list

```
A & B & C 1 2 3 4 5 6
```

Such a list can be kept either in the chain store or in consecutive storage locations. It is the convention when searching the tree (from left to right) to go straight on at each branch point (&) because this always represents the “first” alternative. The addresses of the alternative branches are recorded in a nest in case they should subsequently be needed. It is not really necessary to distinguish the terminal words since they are always followed by the start of the “next” branch to be explored, and hence are located by the “last” & word put in the nest. An extra & word at the very beginning is necessary, however, to locate the end of the last branch.

To find the string corresponding to a given category number we may use a simple “look ahead” process, thus. Scanning the list from left to right we look ahead at each branch point to see if the category number coming before the branch that it points to is either greater than or less than the given number. If less we take this new branch, otherwise we pass straight on. In this way we trace a path which terminates at the required category number. The “letters” on this path are the required string. This method only works, however, if the category numbers are in ascending order from left to right in the list, but fortunately, this is the case with the phrase dictionaries (since the various alternatives are presented in order of preference).

In order to use this method if the dictionary is in the chain store, it is necessary for the & words to point not to the start of the “next” branch, but to the category number lying before it. This is because we can only move in one direction along the chain (from left to right). Thus the representation of the above example becomes

```
A & B & C 1 2 3 4 5 6
```

where the letters and category numbers are untagged (or tagged 00), and the &’s bear the tag 01. This is the form of dictionary which we now use. In the chain store it may be a circular chain (of 25 words in the above example) or part of a longer chain, but the address of the dictionary is always that of the first & word. In an empty dictionary only the first word is present, and it is zero. Dictionaries are usually built up in the chain store and transferred to the record store when complete. Certain dictionaries, however (those representing the format classes), are continually being augmented, and so remain in the chain store for most of their working life.

Ordered dictionaries are usually assembled by merging each new entry, with the restriction that it must come “after” the last entry (or alternatively “before” the first), which usually results in incomplete merging as shown in the example above. Alternatively each new entry can be merged completely with the existing entries, so that the dictionary becomes

```
A & B & C 1 2 3 4
```

(Note: This destroys the left to right ordering of the category numbers, and we can no longer use the look-ahead process described above.) Both modes of assembly are used, and they are effected by two distinct routines. Incomplete merging is used to assemble the dictionaries in phrase definitions and format classes (see later), and complete merging is used in the case of the class identifier dictionary (CID), which records the correspondence between external multi-character identifiers and internal identifiers (which will be serial numbers in the range 0–1,023). The specifications of these routines are of interest.

R 224 (complete merging)

Function. To look up, enter, and delete entries in a chain store dictionary.

Input parameters.

- \(B_{61}\) = address of string to be looked up (a circular chain)
- \(B_{62}\) = address of dictionary
  - 1 for look-up
  - 2 for entry
  - 3 for deletion

Output parameters.

- \(B_{61}\) = unaltered
  - 0 for coincidence
  - 1 if string is a stem of existing entry
  - 2 if an entry is a stem of the string
    - three modes
    - 3 if there is a common stem
    - 4 if there is no common stem
  - in look-up mode it gives the location of the category word in case of coincidence, and if the string is a stem of an existing entry, it gives the category word of the first such entry.
- \(B_{64}\) = in entry mode it gives the location of the category word
Trees and Routines

Notes:

1. The address of the string to be looked up is that of the last word, i.e.
   \[ B_{61} \]
   \[ 11 \ldots 1 (1 \text{ denotes an untagged information word}). \]
   \[ B_{63} \]

2. It is very often required to know whether a given string is or is not present in the dictionary. If it is present we wish to know the category, and if it is not present then we wish to enter it and assign some category number to it. The routine enables us to do this using the entry mode. If the new entry coincides with one already present then \( B_{64} \) gives the category. If it is not present then it is entered together with a vacant location for the category number. This location is then filled in, outside the routine, by the user if he finds that \( B_{63} \neq 0 \).

3. The new entry is placed “after” the entries it is merged with, except where it includes an existing entry as a stem, in which case it is put “before” it.

4. Deletion only takes place in the case of coincidence, which can be checked by testing \( B_{63} \) on returning to the main routine.

R 225 (ordered merging)

Function. To merge a new entry either “after” the last entry or “before” the first entry. (Here we refer to the final order in the dictionary, not the order in which the entries were made.)

Input parameters.

\[ B_{61} = \text{address of string} \]
\[ B_{62} = \text{address of dictionary} \] as in R 224
\[ B_{63} = \text{new category word} \]
\[ B_{64} = \{ -1 \text{ if merging is to be before 1st entry} \}
\[ +1 \text{ if merging is to be after last entry} \]

Structure of Phrase Definitions

In the most general case these take the form

\[ [C] = (1), (2), \ldots \text{ BUT NOT } ( -1), ( -2), \ldots \]

that is as a sequence of alternative forms, separated by commas, which are allocated serial numbers as shown.

The BUT NOT is a new feature not previously described. It has been introduced in order that certain specific alternative forms of the foregoing phrases may be excluded. For example, in Mercury Autocode (Brooker, 1958a, b, and 1959) the set of primed variables should be defined as

\[ \text{PHRASE } [V'] = [V']. \text{ BUT NOT } \pi' \]

The BUT NOT qualifies all the phrases that follow it, and they are allocated negative serial numbers.

The positive alternatives are merged into a dictionary (using R 225, \( B_{64} = +1 \)) with the category numbers shown. The order is preserved because this is the order in which the ERR will look for them, i.e. it is the order of preference. Before entering each new alternative, however, we use R 224 (\( B_{63} = +1 \)) to ensure that it does not include any of the previous alternatives as a stem. The forbidden alternatives, i.e. those written after “BUT NOT” are then merged in front of these entries (i.e. with \( B_{64} = -1 \)) since the expression recognition routine (ERR) must examine them first; recognition of a case with a negative category number is treated as non-recognition for the phrase in question. When the dictionary has been assembled it is transferred to the record store, together with some further information (the M word and the K words), to form the phrase definition, thus

\[ M \& \ldots \ldots \ldots I \quad K \quad K \quad K \quad K \quad K \]

MISC. DICTIONARY KEYWORDS

In the transfer operation the &’s are made to refer to an address relative to the position of the M word as origin; thus the phrase definition is relocatable. M is the miscellaneous information word, and the K’s are the key words. (They are all untagged.)

The keywords have been described (in Brooker and Morris 1961a), but we shall say something below about their construction. The information (bits) in the miscellaneous word M are as follows

1. Whether the phrase is formal or built-in.
2. Whether it represents an elementary symbol.
3. Whether it represents an * class.
4. Whether the associated analysis record should be contracted out.
5. Whether it has an explicit nil alternative.
6. Whether it has an implicit nil alternative.
7. Whether the key words are present.
8. Whether they are complete.
9. Whether they happen to be all 1’s.

Most of this information is used by the ERR when looking for a phrase of the kind in question. Thus, depending on whether it is formal or built-in, the ERR either uses the phrase dictionary or calls in the associated sequence of instructions to do the job. For this purpose “formal” also includes elementary symbols, and a further digit is examined to see if the recursion has at last “touched bottom.” A built-in phrase definition takes the form

\[ M \& \text{[recognition sequence]} K \quad K \quad K \quad K \quad K \]

The analysis record of a phrase is contracted out if we are only interested in whether or not a phrase is present, and not in the particular form which it takes. This would be the case, for example, with the phrase

\[ \text{[JUMP]} = \text{JUMP}, ->, \rightarrow \] (see Appendix). The ele-
mentary symbols are represented by a pair of words MK, the M word containing in addition to the information bits 1, 2 a number in the range 1 to 5. This indicates which of the 5 key words is non-zero, since by definition only one of the 120 key digits can be a 1. Only that key word (K) is stored.

The key digits indicate the possible elementary symbols with which a phrase of the kind in question can start. This information is used by the ERR to cut short the search if the 1st symbol on hand is not one of them. The key digits of a phrase are constructed as follows.

Suppose, for example, that a phrase \([\text{A}]\) is defined by phrase \([\text{A}] = \text{BC} [\text{D}] \text{[E]} [\text{F}]\)

where phrase \([\text{D}] = [\text{G}], \text{H}, \text{NIL} \). The key digits of \([\text{A}]\) are those of \([\text{D}]\) and \([\text{F}]\) merged together (i.e. the logical “or”) along with that representing the elementary symbol B. However, \([\text{D}]\) contains an “explicit nil” alternative, which means that \([\text{A}]\) can also start with an \([\text{E}]\), so that the key digits of this phrase have also to be included. Now \([\text{F}]\) also has a “nil” alternative (since \([\text{F}] = \text{F}, \text{nil}\) and \([\text{A}]\) is therefore credited with an “implicit nil” digit in the information word, the key digits thus referring only to the non-nil branches. To the ERR a “nil” digit (either explicit or implicit) means that whatever the symbol on hand may be, it must always carry the recognition process to deeper levels, since even if there is only a nil branch it is necessary to elucidate its structure (i.e. the choice of alternatives which lead to the nil). Although the key digits of \([\text{A}]\) are not used by the ERR in this case they may be used in determining the key digits of other classes, just as the key digits of \([\text{D}]\) (which are those of \([\text{G}]\) and \([\text{H}]\) are used in determining those of \([\text{A}]\).

The key-digit construction routine makes use of the phrase dictionaries to search for the contributing classes of phrase, but since the phrase definitions are not necessarily presented in any logical sequence it may encounter a class where key words are not available. This information would be supplied by the key-word completion digit of each class. In this case it would abandon the attempt. Every time a new phrase definition is presented and assembled (with empty key words) in the record store, we attempt to construct the key words for it. If the attempt is not successful then the name of the class is placed in a special list—the list of classes whose key words are incomplete. If it is successful, however, we note the fact in the key-word completion digit, and then go on to consider the hitherto “incomplete” classes, since we may now be in a position to complete some of them. If any of them do become “complete” we remove them from the list and repeat the process on the remaining ones until it fails to complete any further classes.

Format Classes

The statement formats are not now merged into a single dictionary. Instead they are grouped together into format classes. For example, all the formats associated with source statements are kept in one class, \([\text{SS}]\), and those associated with auxiliary statements (these are statements defined exclusively for use as “instructions” in format routines and cannot appear elsewhere) are kept in another, \([\text{AS}]\). When defining a format the user indicates to which class it is to be added thus:

\[
\text{FORMAT} \ [\text{SS}] \ = \ [\text{Y}] \ = \ [\text{GE}] \ [\text{EOL}]
\]

\[
\text{FORMAT} \ [\text{AS}] \ = \ A \ = \ [\text{GE}]
\]

These innovations overcome some difficulties, as mentioned by us in reply to a letter from J. M. Watt of Liverpool University (this Journal, Vol. 4, p. 167).

Format classes are essentially similar to phrase definitions except that the formats of each class are presented individually in order of preference. These are merged into a dictionary with R 225 (B_{k4} = +1) after checking (with R 224) that each new entry does not include any of the existing entries as a stem. For each entry a place in the index is reserved for the associated routine, which is presented at a later stage. This index position is recorded as the category number of the entry. The resulting dictionary will not necessarily be ordered with respect to its category numbers, however, unless the “B_{k4}” chain is so ordered. We shall make no such assumption. The format class dictionaries are kept in the chain store until the source language is completely defined, or is sufficiently complete for it to be used for actual translation. In this case the dictionaries are transferred to the record store. Should more definitions be presented later on, the format class dictionaries are unpacked and returned to the chain store.

Because a format class is always open to further members we can never assume that key digits are “complete”; and since in any case, a large proportion of them would probably be 1 it is simpler to dispense with them entirely for these classes. A digit in the information word indicates this fact. This is, of course, distinct from the digit (set by the key-word construction routine) which indicates that they are present, but happen to be all 1’s. If a format class is encountered, in the right-hand side of a phrase definition, by the key-word updating routine, it is treated as if the key digits were all 1’s.

To summarize, a format class is represented by the chain

\[
\begin{array}{c}
\text{M} & \ldots & \ldots & \ldots & \text{I} \\
\end{array}
\]

to show its location being that of the M word.

The ERR and the Analysis Record

The general principles of the recognition routine ERR are as described (in Brooker and Morris, 1961a), but the form of the analysis records has undergone some
Trees and Routines

changes. They are now developed in two stages. The immediate result of the ERR is the compact form, similar to that described before, but embodying the changes made in the representation of a dictionary. This form is illustrated by the following analysis of \( \beta_8 - 4 \) with respect to the class [WORD] (see Appendix).*

\[
\begin{array}{c}
\text{[ADDR]} \\
\text{[WORD]} \\
\end{array}
\]

It is generated as follows. The ERR considers the first alternative form of [WORD] which is [ADDR]. It then proceeds to examine the key digits of [ADDR] and finds that such a phrase cannot start with a "(" which is the first symbol of the expression on hand. It therefore returns to the definition of [WORD] and examines the second alternative which is \([\text{ADDR}]\). It is able to match the "(" on hand and then tries to look for an [ADDR] stem in the remainder of the expression \( \beta_8 - 4 \). The key digits of [ADDR] confirm that \( \beta \) is a possible starting symbol for a phrase of this kind, and so it proceeds to consider the first alternative form of [ADDR] which is \([x\beta] + [x\betaN] \) (alternative names [A] for \( x \) and [B] for \( \beta \) are also used in the Appendix). It has therefore to look for an \([x\beta] \) stem. The key digits of \([x\beta] \) confirm that \( \beta \) is a possible first symbol and the ERR goes on to try the two alternatives of \([x\beta] \), namely \([x] \) and \([\beta] \). These phrase definitions are built-in, i.e. they are special sequences of instructions called in by the ERR in turn to try to identify these phrases. It gets a "yes" reply from the \([\beta] \) sequence which recognizes the phrase \( \beta_8 \). At this stage the provisional analysis record being generated, assumes the form

\[
\begin{array}{c}
\text{[ADDR]} \\
\text{[WORD]} \\
\end{array}
\]

where the category words 8, 2 indicate that \( \beta_8 \) is category No. 8 of \([\beta] \), and \([\beta] \) is category No. 2 of \([x\beta] \). The actual symbols of the expression do not appear in the analysis record, their presence being implied by the category numbers at each level. The first & remains to be filled in by the subsequent recognition process should it be successful at this level. If all the alternatives under examination are successful the record would take the form

\[
\begin{array}{c}
\text{[ADDR]} \\
\text{[WORD]} \\
\end{array}
\]

In fact, however, they are not. The next symbol on hand \(+\) does not match the \(-\) in \([x\beta] + [x\betaN] \) so that the ERR has next to consider the form \([x\beta] - [x\betaN] \). It is not necessary to retrace its steps, however, because the two forms are merged in the class dictionary for [ADDR], thus:

\[
[x\beta] \quad \quad \quad [x\betaN] \\
\downarrow \quad \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \quad \downarrow \\
[x\beta] \quad \quad \quad [x\betaN] \\
\]

and it proceeds at once to match the \(-\). Continuing in this way the ERR matches the 4 with \( [N] \) in \([x\betaN] \), and finally, on returning to the original level, it matches the final bracket in the expression on hand with the closing bracket of \([\text{ADDR}]\). The final form of the analysis record is that given earlier. Should the recognition process fail at any stage all the information necessary to retrace its steps to the previous levels (i.e. the relevant positions in the expression, the provisional analysis record, and the class dictionary) is nested (stacked) in the locations given by \( B_{88} \) onwards, i.e. at the right-hand end of the record store.

The above example uses only two kinds of words, the branch words & and the information words, the "ends" of the tree being located as in a dictionary by having their addresses quoted in the corresponding branch words. If the expression contained a parameter, e.g. \((\beta_8 - [x\betaN/2])\) then the record would be

\[
\begin{array}{c}
\text{[ADDR]} \\
\text{[WORD]} \\
\end{array}
\]

Here P is a third kind of word (with tag 11) containing the internal code corresponding to the name of the phrase to be substituted. This will be an integer in the range 1–N, N being the number of distinct combinations of identifier/label present in the format routine (see later). If the class of the * variety then there may be an index present as, for example, in \([* \quad T^*(1)]\). In this case the index is encoded in the most-significant half of the P word, the identifier number being that corresponding to \([* \quad T^*] \) which will have previously appeared.

* The roles of x and \( \beta \) are interchanged in the present paper. The \( \beta \)’s are now synonymous with the contents of the corresponding B-lines.
Trees and Routines

Thus the P word is of the form

```
10 bits         10 bits
k              P tag
```

The compact form of the analysis record (henceforth abbreviated to a.r.) is the most natural result of the recognition process (hence its name) used in the ERR, and for this reason we considered using it in the format routines. However, it does not lend itself very easily to the substitution of parameters, as the only way of doing this with the compact form is to keep it in the chain store and replace the P link in the chain by the chain representing the expression in question, which will itself have to be disconnected from its parent chain. Apart from the necessity of keeping a record of all the broken links to be stored, it is impossible to substitute an expression into more than one place at a time without making a physical copy of it, as would be necessary, for example, in the case of \([x^k/1] = [x^k/1] + 4\).

Instead we use the dual form where the role of the & word is reversed; it now points to the first word of a subanalysis record instead of the last. All the & words associated with a given level are in consecutive locations. The category number precedes them. The relationship between the two forms can be illustrated thus:

Compact form

```
\[\begin{array}{ccc}
\text{Sub A}\text{RP1} & \text{Sub A}\text{RP2} & \text{Sub A}\text{RP3}
\end{array}\]
```

Dual form

```
\[\begin{array}{ccc}
\text{Sub A}\text{RP1} & \text{Sub A}\text{RP2} & \text{Sub A}\text{RP3}
\end{array}\]
```

When packed in this way the end of each subanalysis record is implicitly indicated by the category word of the next. Provided, however, that the end of such subanalysis record can be identified by the presence of such an I word, they can be packed in arbitrary fashion. The dual form of the analysis record of \((\beta_n - 4)\) is thus represented in general by the tree

```
2 &
\downarrow
2 &
\downarrow
2 &
\downarrow
8
3
```

which, when packed in the canonical form, becomes

```
2 &
\downarrow
2 &
\downarrow
2 &
\downarrow
3
\text{&}
4
```

This occupies the same number of locations as the original compact form

```
2 &
\downarrow
2 &
\downarrow
2 &
\downarrow
8
```

The latter can be transformed into the canonical dual form by a recursive algorithm which starts on the top level at the right-hand end and works towards the left. This algorithm is in fact the final operation of the ERR.

The corresponding tree for the parametric expression \((\beta_8 - [x^kN/2])\) is

```
2 &
\downarrow
2 &
\downarrow
\beta_8
```

In this case parameter substitution is effected by replacing the P word with an & word pointing to the a.r. in question. The specification of the ERR, or recognition routine, is as follows.

**R 215**

Function. To analyse a string of characters with respect to some class of phrase giving a representation of the structure in the form of a tree (known as the analysis record, or a.r.). Parts of the record (or indeed, the whole) may be contracted out.

Input parameters.

- \(B_{61}\) = address of string to be analysed (a circular chain)
- \(B_{62}\) = class w.r.t. which the analysis is made
  - 0 for usual entry
  - 1 for a special entry
  - 2 for re-entry
- \(B_{63}\) = address of analysis record
- \(B_{64}\) = address of stem

Output parameters.

- \(B_{61}\) = address of ending
- \(B_{62}\) = address of recognized stem
- \(B_{63}\) = address of analysis record
- \(B_{64}\) = address of recognition
- \(B_{65}\) = 1 for recognition
- \(B_{66}\) = > 1 for “maybe”

† These terms are also used in the study of natural languages, where they refer to the parts of a word. In the present context they may stand for lengthy strings of material.
Trees and Routines

Notes:
1. The original string is split into two (circular) strings, the stem and the ending, thus:

![Diagram of a string being split into stem and ending]

If nothing is recognized then $B_{61} = 0$, and if everything is recognized $B_{61} = 0$.

2. The special entry ($B_{61} = 1$) is used if there is a possibility that the ERR will arrive at the end of the original string while following up some particular logical path. In this case, if the ERR does run out of material it records the intermediate state of recognition and returns to the main program with $B_{64} > 1$, otherwise (i.e. if $B_{63} = 0$) arrival at the end of the source string would be interpreted as failure to recognize the given stem. When an exit does occur with $B_{64} = 1$ further material is provided (in $B_{61}$) and the routine is re-entered with $B_{63} = 2$ in order to carry on where it left off.

There are then four possibilities:
(i) The ERR may recognize a stem which extends into the new string, thus:

![Diagram of a stem extending into a new string]

(ii) It may demand a further string.
(iii) The particular alternative which took it into the new string may fail to match, causing it to fall back and recognize something in the original string.

![Diagram of a new string failing to match]

(iv) It may fail to recognize anything at all.

It is appropriate at this point to say something about the use of the store when the ERR is in operation, although it cannot be fully appreciated until we describe the general system of working space for routines (see later).

$B_{90}$ denotes the extent of current working space. When the routine is called in 5 locations are used for the link (see later), and the analysis record is developed in the locations following this. $B_{90}$ is advanced to the next available location (allowing for an extra 1 word, following the a.r., which is sometimes necessary to locate the “end”). The ERR involves extensive internal nesting (being a recursive process) and this uses the locations given by $B_{63}$ onwards.

In the case of a “maybe” exit (as the case of exit with $B_{64} > 1$ is called), the ERR preserves the information (9 words) needed to take up the search again in case of a subsequent re-entry. This is done as follows:

![Diagram of information preservation at exit]

$B_{90}$ must be reset to the exit value before re-entry because it is used to locate the 9 words. The 9 words are recovered and further sections of the a.r. are written over the space that they occupied in the main stack.

Packing of Analysis Records

In (Brooker and Morris, 1961a) we described a scheme for compressing analysis records, which would result in a considerable saving of space in the case of deeply structured records. This has now been abandoned because (a) the unpacking operation was very time-consuming, and (b) most of the a.r.’s are not in fact very deep at all and would, if packed, yield only a comparatively small saving of space. The example given in that reference namely

$$[Y] = x[Y] + [V](i - 1)$$

which involves an a.r. of 33 words, is not in actual fact at all typical of instructions met with in a format routine. A more representative example is

$$A = \text{[GE/1]}$$

whose a.r. is merely IP. Here I is the serial number of the format $A = \text{[GE]}$ w.r.t. [AS], and P is the parameter word identifying the particular phrase [GE/1] of the form [GE] which is to be substituted in place of P to complete the instruction. (The symbols “A = ” do not appear in the record.)

The Analysis Records of Some Built-In Phrases

Some frequently occurring phrases, notably [N] (integer) and [K] (floating-point number) are recognized by built-in sequences of instructions. The resulting a.r.’s contain in one or two words the information which would otherwise occupy an extensive tree if these phrases were defined formally. This structure cannot therefore be resolved (by means of let [PI] [EQV] [RESOLVED-P]
and can only be interpreted by the routines associated with the formats in which they occur. (Another consequence is that it is not easily possible to recover the original string—but this matter is treated elsewhere.)

Three important a.r.'s are as follows:

[N] The a.r. consists of a single (1) word, the number itself.

[β] The a.r. consists of a single (1) word, the suffix \( n \) of \( \beta_n \).

[K] The a.r. consists of three consecutive words \( B_2 \)WW.

Hence \( B_2 \) is a fourth kind of word (tag 10) which indicates that the following \( k \) words are a self-contained piece of information, i.e., that all 24 bits are relevant. In this case the two words represent [K] in standardized floating-point form (though not necessarily in a complete 48-bit register).

The analysis records of [x2], [LABEL] and other built-in phrases will be presented at a more appropriate opportunity. These records are modified in the course of execution, and it is first necessary to explain how a routine is executed (see later).

Reconstructing the Original String from an Analysis Record

This operation is sometimes required. It uses the "look ahead" process described earlier for looking up in a dictionary (in this case a phrase dictionary) to find the string corresponding to a given category number. The process is a recursive one. Starting with the given class of phrase it first finds the alternative string which corresponds to the principal category number of the a.r. It then applies the process to each of the phrases encountered on this string, until eventually elementary symbols are reached. The process will fail in general if any "built-in" phrases are encountered; e.g. the original sequences 1, 01, 001, etc., would all have the same a.r., w.r.t. the class [N]. Some information in fact is usually lost in going from the original string to the a.r., and cannot be recovered.

The following is the specification of the relevant routine

**R 255**

Function. Given an a.r. (in dual form) and the class of phrase it represents, the routine will reconstruct the original string of elementary symbols—subject to the restrictions mentioned above.

Input parameters.

\( B_{61} \) = address of a.r. (i.e. of category word).

\( B_{62} \) = serial number of class (in record store).

Output parameter.

\( B_{63} \) = location of resulting string (a circular chain).

This routine can be used in conjunction with R 224 to convert external multicharacter identifiers to an internal integral equivalent. Thus, for example:

**ROUTINE [AS] \( \Rightarrow \) CONVERT [PI] TO [xβ]**

\( \beta_{61} \) = ADDRESS OF [PI] (see footnote)

\( \beta_{62} \) = CLASS OF [PI]

CALL IN R 255

\( \beta_{61} = \beta_{63} \)

\( \beta_{62} = \beta_2 \)

\( \beta_{63} = 2 \)

CALL IN R 224

IF \( \beta_{61} = 0 \)

(\( \beta_{64} = \beta_3 \))

(\( \beta_1 = \beta_3 - 1 \))

1) \( zβ = (\beta_{64} \)

END

In this routine \( β_1 \) is the location of the dictionary, and \( β_3 \) is the category number to be assigned to the next distinct identifier. The latter is advanced by 1 every time a new entry is made in the dictionary. The structure of the identifier is given by a phrase definition, e.g.

**PHRASE [IDENTIFIER] = [LETTER] [LETTER OR FIGURE]**

The instruction

**CONVERT [IDENTIFIER/2] TO \( x_1 \)**

is a particular case (i.e. parameter-free) of the format

**CONVERT [PI] TO [xβ]**

The Organization of Large Routines

Both the format routines defined by the user and those of the system itself, such as the ERR and the dictionary routines, operate on the same general principle. The administrative sequences DOWN and UP allow the routines to be used as subroutines to any depth, the links and local data being "stacked" in the same section of the store. The principle has been described elsewhere and allows the routines to use themselves recursively if necessary. The following diagram illustrates the store layout when a routine A is using B as a subroutine which in turn is using C.

The front of the stack is always given by the current value of \( B_{90} \) and it is up to the individual routines to see that

The \( β \)'s, which are used in format routines are, as mentioned earlier, synonymous with the corresponding B-lines. Consequently, when the user calls in one of the system routines, he may use \( β_{61}, \beta_{65}, \text{etc.}, \) for its input and output parameters \( B_{61}, B_{65}, \text{etc.} \)
this is advanced and retarded in the appropriate manner. If C calls in a further subroutine, D say, then the link (4 words) will be planted in $B_{90}$, $B_{90} - 1$, $B_{90} + 2$, $B_{90} - 3$, and $B_{90}$ will be advanced by 4. The locations following this will then be available for local use by D. However, the very first location of each local area is reserved for a forward address, so that we can trace the situation from behind if necessary (see later when the execution of [JUMP] [LABEL] is described).

Before we can describe the nature of the link it is necessary to explain the conventions regarding the use of B-lines when a routine is operating.

- $B_{89}$ is the head of the main chain (in the chain store).
- $B_{88}$ is the front of the record store.
- $B_{87}$ is the head of the index chain.

These quantities are used by all routines, whereas the following are of local significance.

- $B_{127}$ is the control register.
- $B_{90}$ is the front of the stack, i.e. the next available location.
- $B_{75}$ is the serial number of the current routine.
- $B_{73}$ is the location of the label directory.
- $B_{71}$ is the location of the current routine.

It will be seen further on that $B_{71}$ and $B_{73}$ can be recovered from a knowledge of $B_{85}$, and for this reason are not recorded in the link. During the operation of the routine, however, it is desirable for reasons of time economy to have immediate access to them.

Before calling in a subroutine its parameters are placed in $B_{81}$, $B_{82}$, etc. (up to $B_{99}$). The DOWN sequence is then entered with a single (extracode) instruction which puts the current value of control, $B_{127}$, in $B_{70}$, and the serial number of the subroutine (which is held in the address part of the instruction) in $B_{90}$. The subsequent operation of this sequence will be described later on. $B_{81}$-$B_{99}$ can also be used as miscellaneous working space provided this is not allowed to interfere with their role as parameter locations.

The B-lines $B_1$ to $B_9$ are common to all routines and must be allocated to them in a mutually consistent manner. The method we have adopted is to use $B_{39}$, $B_{35}$, $B_{37}$, etc., for the system routines, leaving $B_1$, $B_2$, $B_3$, etc., for the format routines (i.e. $\beta_1$, $\beta_2$, $\beta_3$, etc.). Within the system routines the B-lines are assigned as follows. A ground-level routine (i.e. one in which there are no subroutines) will use $B_{39}$, $B_{38}$ ... to $B_{54}$ (say), and the routines which employ this routine as a subroutine will use $B_{33}$, $B_{32}$, etc. It is not yet clear just how many B-lines will be left for the format routines.

**The DOWN and UP Sequences**

All (large) routines have the form

```
\text{ROUTINE PROPER} \rightarrow \text{LABEL DIRECTORY}
```

The first word N is the amount of immediate working space which is required for the routine on entry, and the stack is accordingly advanced by this amount plus one after first planting the link. The second word gives the (relative) location of the label directory.

The label directory is a list of control numbers corresponding to labels appearing in the routine, that corresponding to label n appearing in the mth location. The directory is only used to execute multiaxis switches (e.g., $\rightarrow \alpha_1$), otherwise (e.g. $\rightarrow \beta_3$) control numbers are incorporated into the jump instructions in the body of the routine. All control numbers and the & word giving the location of the directory are made relative to the first word (N) of the routine. The item is thus relocatable in the record store. This is more convenient than keeping a list of all those instructions whose address parts would have to be adjusted whenever the item was moved in the store.

![Diagram](https://academic.oup.com/comjnl/article-abstract/5/1/33/395478/5.png)

The first task of the DOWN sequence is to plant the link. This takes the form

- \((B_{72}) = B_{90}\) (forward link (see later)).
- \((B_{90}) = B_{90} - B_{71}\) (relative position in current routine).
- \((B_{90} + 1) = B_{75}\) (serial number of current routine).
- \((B_{90} + 2) = B_{90}\) (or $B_{77}$ in format routines (see below)).
- \((B_{90} + 3) = B_{72}\) (origin of local working space).

After this it sets $B_{72} = B_{90} - 4$.

$B_{71}$ and $B_{73}$ are not recorded: instead they are reconstructed from $B_{85}$ on coming UP, which is almost as quick as recovering them directly from the stack.

After the DOWN routine has planted the link it looks up the new routine in the index (its serial number being given in $B_{90}$) to find its absolute location and the quantities N and &. From this information it sets up the new values of $B_{71}$, $B_{73}$, $B_{72}$ and $B_{90}$.

The operation of the UP sequence is comparatively straightforward. It recovers the link from \((B_{72} - 4), (B_{72} - 3), (B_{72} - 2), (B_{72} - 1)\), and restores or reconstructs the original contents of the relevant B-lines.

**Small Routines**

These are routines which do not have any subroutines and do not disturb the administrative machinery of the main routine. Their parameters are placed in $B_{89}$, $B_{88}$, etc., in that order of preference, so as to avoid disturbing any parameters of the main routine which may still be standing in $B_{81}$, $B_{82}$, etc. The DOWN sequence finds out from the entry in the index whether a routine is large or small, and in the latter case enters it directly.
with the link in B₇₀. This is eventually used to effect a direct return to the main routine. There is no label directory, and hence it is not possible to use multiway switches.

The Structure of Format Routines

In the previous papers we classified the "instructions" of a format routine into substatements, basic list compiling instructions, and parameter operations. We shall now revise this definition. The "instructions" in a format routine may be any member of the format classes [BS], [AS], [SS]. [BS] is the class of built-in statements which include the list instructions and parameter operations; a complete list of these formats is given in the Appendix. An instruction is normally represented by its a.r. with respect to the relevant class, unless it is completely free of parameters in which case it may be represented by a sequence of machine instructions. Thus a format routine consists in the general case of a conventional sequence of machine instructions interspersed with a.r.’s.

An a.r. instruction is "obeyed" by calling in the corresponding format routine to analyse and interpret the record. The a.r. is not interpreted in situ, however; instead it is first copied out into the main stack by the sequence TRANSPLANT, which carries out the parameter substitutions and certain other adjustments (explained more fully later on), and then enters the DOWN sequence with the serial number of the routine in question (which is given by the principal category number of the a.r.). (The entry made via the TRANSPLANT sequence sets a switch to distinguish it from the usual entry.) The TRANSPLANT sequence is entered automatically by a special extracode instruction (E) which stands before each a.r. in the body of the routine. If the instruction is a [BS] format the corresponding routine will be a small "built-in" routine. The relevant storage layout is as follows

![Diagram showing storage layout]

The main reason for transplanting the record is that it may be part of a recursive cycle, in which case it is essential to work with copies of the a.r. entered in the main stack; also, when packed, the a.r.’s contain relative &’s which must be converted. In any case it is necessary to scan the record for the parameters which have to be substituted, and it is just as easy to transfer it at the same time, converting the &’s to absolute addresses.

After setting up the B-lines for the new routine the DOWN sequence examines the entry mode switch and if the entry has been via TRANSPLANT, then it calls in the HEADING sequence to enter the principal phrases in the new list of selected expressions (LSE).

The LSE/α List

The last N words reserved in the main stack on immediate entry to a format routine are used for the entries in the LSE and the α’s. It has already been mentioned that the external phrase names in a routine are replaced internally by an integer in the range 1 → m, m being the number of distinct combinations of identifier/label. These correspond to the first m positions of the N-word block. The following positions m + 1, m + 2, ..., m + n − 1 (= N) are allocated to α₁, α₂, ..., αₙ, n being the highest suffix employed. α₁, therefore, is associated with an internal identifier m − i − 1.

The phrase identifiers are ennumerated in the order in which they are encountered, starting with the heading, thus:

```
ROUTINE [AS] = ADD ([WORD] [,WORD]) TO
  [LIST OR NEST] [β]
```

Now the heading is simply a restatement of the format, so that the first entries in the LSE correspond to the & words on the top level of the a.r. on hand. These entries are actually made by the HEADING sequence which copies the contents of locations B₇₂ − r into locations B₇₂ + r + 1, (r = 1, 2, 3, etc.), until (B₇₂ − r) = an & word. Finally the value of B₇₇ is planted in B₇₂ − 1 which corresponds to [AS] in the above example. The rest of the LSE is cleared and subsequent entries are made by the parameter operations of resolution and generation.

The TRANSPLANTING Operation

When an a.r. is transplanted, that is copied into the main stack, the P words representing the parameters are replaced by & words pointing to the phrases in question. These & words will be found in the LSE. If zero entry is encountered then the routine must be inconsistent. A test for this is included. If the P word contains an index then the entry in the LSE must point to an a.r. with an "α" structure, thus:

```
α of first
appearance of
class
```

```
2nd
1 & α
```

```
3rd
```

```
last
```

42
In this case the P word substitution sequence must follow it through to find the & word pointing to the kth appearance of the phrase in question, k being the current value, i.e. at the time of transpaling of the index occurring in the P word.

In addition to parameter substitution all the & words of the copied a.r. are made absolute by adding the address of the first of the set of locations to which the record is transferred. If the last word of the a.r. being transferred is a P word then an extra 1 word (-10 in fact) is added, so that the end of the copied record can be recognized.

In addition, some modifications have to be made to those parts of the a.r. (if any) representing particular phrases of the form [x], [LABEL], [PI] or [RESOLVED-P]. These phrases describe information which is local to the routine in which they appear, and when they are handed down as parameters to other subroutines, the routines which ultimately interpret them must be able to locate this information in the original routines. For example, consider an instruction presented earlier, namely

\[
\text{CONVERT [IDENTIFIER/2] TO } x_1
\]

corresponding to the format

\[
\text{CONVERT [PI] TO } [x_\beta]
\]

The first step in "obeying" this instruction is to call in the corresponding [AS] routine (given earlier). The \( x_1 \) is the actual parameter corresponding to the formal parameter \([x_\beta]\), and similarly [IDENTIFIER/2] corresponds to [PI]. The routine which ultimately interprets \( x_1 \), however, is one level deeper. It is the small routine (corresponding to \([x_\beta] = [\text{WORD}]\)) which is called in to interpret the instruction \([x_\beta] = (\beta_{64})\), and this routine must have access to the \( x \) list of the routine in which "CONVERT [IDENTIFIER/2] TO \( x_1 \)" occurred. Similarly the routines which ultimately interpret [IDENTIFIER/2] are those called in to interpret the instructions

\[
\begin{align*}
\beta_{64} &= \text{ADDRESS OF [PI]} \\
\beta_{62} &= \text{CLASS OF [PI]}
\end{align*}
\]

and these must have access to the LSE in which [IDENTIFIER/2] appears.

The analysis record of \( x_k \) is a pair of words \( B_jW_k \), where \( W \) contains the integer \( m - k - 1 \). It is modified during the transpaling operation by adding to it the current value of \( B_{72} \) so that it gives the absolute address of the \( x_k \) register. To enable the TRANSPLANT sequence to recognize this special case the block word \( B_j \) is modified by making its most-significant digit a 1. The modification of the analysis record of [PI] will be described later when we discuss the parameter operations.

The (Small) Interpretive Routines

These are the routines which interpret the [BS] statements (see Appendix). They fall into three groups, the basic listing instructions, the jump instructions, and the parameter operations. As an example of the first group consider

\[
[x_\beta] = (\beta_{64}) \text{ representing } [x_\beta] = [\text{WORD}]
\]

This has the analysis record \( 1 \, P \, \& \, 2 \, \& \, 4 \, \& \, 2 \, \& \, 64 \). When this instruction is obeyed the a.r. is first transplanted, and the address of the a.r. of the phrase \([x_\beta]\) (which should be given in the LSE) is substituted in place of the P word. This a.r. will have been transplanted on some previous occasion (modified at the same time if it was an \( x \)). The storage layout is thus

\[
\begin{array}{c}
B_{72} \\
\hline
\text{LSE LIST MISCELLANEOUS} \quad 1 \, \& \, 2 \, \& \, 4 \, \& \, 7 \, \& \, 64 \\
\hline
\text{to a.r. of } [x_\beta]
\end{array}
\]

The (small) [BS] routine corresponding to \([x_\beta] = [\text{WORD}]\) is then called in, and a series of decisions based on the topology of the a.r. leads it to the appropriate action, which in this case is to put the contents of the address \( \beta_{64} \) into the relevant \( x \) or \( \beta \) register. It then returns control to the main routine (via the link in B_{70}) resetting B_{90} to the value B_{72}. The other list-compiling instructions are interpreted in a similar manner.

We next consider the jump instruction

\[
\text{[JUMP] [LABEL]}
\]

The analysis record of [LABEL] takes the form

\[
B_jW_iW_2W_3
\]

Here \( B_j \) is a block word in which the most-significant digit is a 1. This is to distinguish it from other block words in the transplanting operation, and causes the rest of the record to be modified as follows.

- \( W_i \) is spare (to make the form of the record consistent with those of [PI], etc.
- \( W_2 \) is initially vacant and filled in with the value of \( B_{72} \) at the time of transplanting.
- \( W_3 \) represents initially the actual label \( x_k \), \( \beta_k \), or \( k \).

It is of the form

\[
\begin{array}{c}
\text{tag} \\
0 \, 0 \text{ for no index} \\
0 \, 1 \text{ for } \alpha_k \\
1 \, 0 \text{ for } \beta_k \\
1 \, 1 \text{ for } k
\end{array}
\]

\footnotetext{If a primary assembly routine (see later) were available the instruction could be replaced by actual machine instructions because it is in fact parameter free, [IDENTIFIER/2] being a specific form of [PI].}
If it is an \( x_k \) or a \( \beta_k \), it is replaced by the value of this quantity (the \( x_k \) is local to the routine) at the time of transplanting. The eventual interpretation of this record by the (small) routine corresponding to [JUMP] [LABEL] is as follows. (The same technique is also used, of course, for the conditional jumps.) The quantity \( W_2 \) is compared with the "B72" of the current (large) routine. If they are identical the jump is effected, using the label directory given by B72. This gives a relative control number which is added to the address B71, and the result sent to B72. If the [LABEL] did not originate in the current routine, however, then \( W_2 \) is the "B72" of the routine in which it did originate, and the first entry in that section of the stack leads to the LINK which was recorded when control passed into the subroutine at the level below, thus:

\[
\text{original}
\]

From this LINK can be recovered all the necessary information to restore the operation of the original routine, i.e. to reset B71, B72, B75, B80, as well as B72. Finally, we refer to this local label directory using the value of \( W_3 \).

This latter situation would only arise if further jump orders had been introduced as auxiliary statements, thus:

\[
\text{FORMAT [AS] = GO TO [LABEL]}
\]

\[
\text{ROUTINE [AS] = GO TO [LABEL]}
\]

\[
\text{END}
\]

The Parameter Operations

Before describing how these are effected it is necessary to describe the form of the analysis records of the built-in phrase definitions [PI], [RESOLVED-P], and [GENERATED-P]. The first of these [PI] describes a general phrase identifier

\[ I \pm T, I \pm T/I, I \pm T^* (x_1), I \pm T^*/I (x_1) \]

and takes the form

\[ B_5 W_3 W_2 W_1 \]

Here \( W_1 \) contains the serial number \( (m_1) \) of the class of phrase described, together with its internal identifier \( (m_2) \) corresponding to the combination "identifier/label." These are packed into the 24-bit word as follows:

\[
\begin{array}{c|c|c|c}
\hline
m_1 & 0 & 0 & 0 \\
10 & 10 & 10 & 0 \\
\hline
\end{array}
\]

\[ B_3 W_3 W_2 W_1 \]

\( W_3 \) is initially vacant but is filled with the value of B72 at the time of transplanting (as in the case of [LABEL]). \( W_1 \), which is only relevant in certain cases, represents the index \( x_k, \beta_k \), or \( k \). Its form is the same as that of [LABEL]. The phrases [RESOLVED-P] and [GENERATED-P] mean "some expression of the class [P]" where [P] is the class appearing in whatever replaces a [PI] phrase in the same instruction format, more precisely the nearest one on the left. For example, if in the instruction

\[ [\text{JUMP}] [\text{LABEL}] [\text{IU}] [\text{PI}] [\text{EQV}] [\text{RESOLVED-P}] \]

the phrase [PI] were replaced by [WORD], then for [RESOLVED-P] we could write such expressions as [WORD/1], [ADDR/2], [(ADDR)], \([x_2] = [x_3N]\), etc. They are defined in effect by

\[ [\text{RESOLVED-P}] = [P] \]

\[ [\text{GENERATED-P}] = [P] \]

where [P] gets replaced (in the ERR) by whatever identifier occurs in place of the [PI]. In both cases the expression in question is analysed with respect to the class [P] (there is a slight modification in the case of [RESOLVED-P]), and the resulting analysis record, when suitably augmented, becomes the analysis record with respect to these special classes. An instruction of the form quoted above has an analysis record of the form

\[
\begin{array}{c}
I \& \& \& \\
B_3 W_3 W_2 W_1 \& \& \& \\
\hline
\end{array}
\]

(Note: the a. r.'s of [EQV] and [JUMP] are contracted out.)

If the expression in question represented a particular [P] as for example, in

\[ \text{LET [WORD/1] = [WORD/2]} \]

then we would have

\[
\begin{array}{c}
I \& \& \\
B_3 W_3 W_2 W_1 \& \& \\
\hline
\end{array}
\]

where the P is a parameter word describing [WORD/2] in the usual way.

If [PI] is replaced by its own parametric form (e.g. [PI/1]), then [P] cannot be set up in the first place, and
Trees and Routines

in this case only parametric [RESOLVED-P]'s and [GENERATED-P]'s can be used. For example

\[ \text{[JUMP][LABEL] IF [PI/1][EQV][RESOLVED-P/1]} \]

has the analysis record

\[
\begin{array}{c}
\text{I & & P P} \\
\text{B2WWW 1}
\end{array}
\]

where the two P words describe [PI/1] and [RESOLVED-P/1].

The forms of the analysis records for other parameter operations can be inferred from the foregoing examples.

We have already mentioned that there is a slight modification in the process of obtaining the analysis record for [RESOLVED-P]. This will now be explained.

The purpose of the operation

\[ \text{LET [PI][EQV][RESOLVED-P]} \]

is to resolve the particular phrase described by [PI] into the form described by [RESOLVED-P]. (If the two are inconsistent the operation will signal a fault.) To do this the interpretive routine compares the a.r. of the given phrase (the address is found in the LSE) with that of the [P] expression (the "template"). The "parameters" in the latter record do not, however, indicate substitution but rather the reverse. If the a.r.'s match then those parts of the former which correspond to the "parameters" in the latter are entered in the LSE. These parameters are described by P words (P words in which the most-significant digit is 1) containing the internal identifier of the phrase they represent. (There is no index part.) These words will be modified during the transplanting operation by the current value of B2, so that when they are ultimately interpreted they contain the absolute address of the relevant entry in the original LSE.

The parameters in [GENERATED-P] serve their usual purpose, namely to indicate parameter substitution when the a.r. is transplanted. The difference, therefore, between [RESOLVED-P] and [GENERATED-P] is that in the former case the ERR plants P words in place of parameters (which must be index-free) occurring in the expression in question, and in the latter case it plants P words.

The purpose of the operation

\[ \text{LET [PI] = [GENERATED-P]} \]

is, of course, to construct a new entry in the LSE ([PI] must be index-free), namely the given [P] expression which may include existing phrases in the LSE as substitution parameters. When the a.r. of this instruction is transplanted, the parameters are substituted in the usual way, and the a.r. of [PI] is modified as already described by the substitution of "B2". (If the [PI] and [GENERATED-P] are themselves parameters, then these operations will have been effected on some earlier occasion.) The operation of the associated interpretive routine is to locate the entry position described by the [PI] phrase and enter in it the address of the a.r. of the [P] expression. For example

\[ \text{LET [WORD/1] = (B8 - [xβN/2])} \]

substitutes the "known" phrase [xβN/2] in the right-hand side and enters the address of the resulting expression in the LSE so that it too becomes "known" as [WORD/1].

At the end of the operation B0 is not reset to B7 and the a.r. remains in the stack as a contribution to the miscellaneous working space following the x-list at each level.

The Remaining Parameter Operations

It should not be necessary to explain the other parameter operations in quite the same degree of detail.

The same comparison routine is used in the case of the instruction

\[ \text{[JUMP][LABEL][IU][PI] = [PI]} \]

Here the phrases referred to in the [PI]'s are looked up in the (relevant) LSE and their a.r.'s compared. If these match then their corresponding expressions are identical. Whether or not there is a transfer of control depends both on the result of the matching and the [IU] phrase.

To execute the instructions

\[
\begin{align*}
[xβ] &= \text{CATEGORY OF [PI]} \\
[xβ] &= \text{NUMBER OF [PI]} \text{(which must describe an * phrase)} \\
[xβ] &= \text{ADDRESS OF [PI]}
\end{align*}
\]

we first look up the phrase in question in the LSE and then examine its a.r. for the required information. The "CATEGORY OF" is the first I word of this record and the "NUMBER OF" is the number of recursions in its "*" structure. The "ADDRESS OF" is simply the entry in the LSE.

There is also the instruction

\[ \text{[xβ] = CLASS OF [PI]} \]

which uses the a.r. of the [PI] phrase for only one purpose, to select the serial number of the class of phrase referred to. This, it will be recalled, is packed into the most-significant half of the W1 word.

Finally, we have the instruction

\[ \text{[PI] = [xβ]} \]

which makes an entry directly into the (relevant) LSE ([PI] must be index-free.)

The Phrase Description Phrases as Parameters

To conclude this section on parameter operations we would point out that the phrases

\[ [PI], [RESOLVED -P], [GENERATED -P] \]
can themselves be parameters subject to certain restrictions. We have, in fact, already had an example illustrating this, namely

```
ROUTINE [AS] = CONVERT [PI] TO [zβ]
```

Another somewhat trivial illustration of this is the recasting of the "NUMBER OF" operations to use lower-case letters. Thus

```
FORMAT [AS] = [zβ] = number of [PI]
ROUTINE [AS] = [zβ] = number of [PI]
[zβ] = NUMBER OF [PI]
END
```

A more substantial example is the following definition of a compound testing instruction

```
FORMAT [AS] = [JUMP] [LABEL] IF [PI] [EQV]
            [RESOLVED -P] [COMMA] [RESOLVED -P]
ROUTINE [AS] = [JUMP] [LABEL] IF [PI] [EQV]
            [RESOLVED -P] [COMMA] [RESOLVED -P]
            [JUMP] [LABEL] IF [PI] [EQV] [RESOLVED -P]
END
```

An example of such an instruction is

```
  ← 10 IF [Y] = [V][I][N][±][N]
```

The double discrimination is replaced by a pair of simple tests. If both these fail the routine ends in the usual way, and is equivalent to the original test failing and control going straight on. If either test succeeds, however, it is equivalent to the original test succeeding, and hence we have to jump to the [LABEL] in question.

**Primary Assembly Routines**

It has been mentioned that the "instructions" in a format routine which are free of parameters can be represented directly by (Atlas) machine instructions, rather than by their a.r.'s. This will save time in transplanting and interpreting them. We have therefore made provision to associate with each instruction format a "primary assembly" routine, whose purpose is to assemble these machine instructions when a parameter-free specimen of that instruction format is encountered in the primary processing of a routine.

[The processing of the primary material, i.e. the way in which the phrase definitions, format routines, etc., are recognized was described in principle in an earlier paper (Brooker and Morris 1961a), but its details are beyond the scope of this paper.]

Thus, for example, when the [BS] instruction

```
  x₃ = β₁₀ + x₅
```

is encountered in a format routine, the primary assembly routine associated with [zβ] = [WORD] is called in and will plant the machine instructions

```
0121, B?, 10, 0
0104, B?, 72, 4 + m
0113, B?, 72, 4 + m
```

(where B? indicates some B-line used for intermediate storage and where m is the number of entries in the LSE in the appropriate part of the store, where the routine in question is being assembled. Since the function of these routines is to assemble machine instructions, they can be written on the same principles as the format routines, the only difference being that in the latter case the instructions are planted in the object program.

There are therefore two routines which may be associated with each format: the format routine itself, and the primary assembly version. To distinguish them we may write, e.g.,

```
ROUTINE (COMPILER) [BS] = [zβ] = [WORD]
```

(although such routines for most of the [BS] statements will be permanently available).

The correspondence between a format and its primary assembly routine is established by keeping a double entry list in which the index number of the routine is paired with that of the usual format routine.

**Acknowledgements**

The authors would like to acknowledge the discussions they have had with various people in the automatic programming field, in particular with Mr. A. G. Fraser of Ferranti Limited.

**References**


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**

$b$

$[A] = A1, A2, A3, \ldots, A0$

($=[a] = z1, \ldots$)

$[B] = B1, B2, B3, \ldots, B0$

($=[b] = B1, \ldots$)

$[N] = 1, 2, 3, \ldots, 0$

[OPERATOR] = $\cdot$, $\div$, $\times$, $\slash$, $\&$, $\mid$, $\not\equiv$, AND, NOT EQV

[COMPARATOR] = $=, \neq, >, <, \leq, \geq, \ldots$

[0-3] = 00, 01, 10, 11

[AB] = [A], [B] ($=[aβ]$)

[ABN] = [A], [B], [N] ($=[αβN]$)

[ADDR] = [AB] $\div$ [ABN], [AB] $-$ [ABN], [AB]+[ABN], [AB]

[WORD] = [ADDR], ([ADDR]), $[\ldots][N], [0-3], [-\ldots][N], [0-3], [-\ldots][N], [\ldots][N], [\ldots][N], [\ldots][N], [\ldots][N], [\ldots][N]

[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

$[\ldots]$ followed by up to 8 octal digits starting with the most significant

[IU] = IF, UNLESS

$b$

[LABEL] = [ABN]

[PI] = general phrase identifier

$[RESOLVED-P] = $some $[P]$ expression following a $[PI]$ phrase

$[GENERATED-P] = $some $[P]$ expression following a $[PI]$ phrase

$c$

$[JUMP] = \rightarrow, \rightarrow$, JUMP

**Note:** the second alternative $\rightarrow$ corresponds both to the genuine $\rightarrow$ on Pegasus/Mercury teleprinter keyboards, and to the compound symbol $\rightarrow$ on Atlas Flexowriters.

$c$

$[EQV] = \equiv, (\equiv)$

**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 an erase 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 new line.

**Built-In Formats**

[AB] = [WORD][SEP]

[AB] = [WORD][OPERATOR][WORD][SEP]

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

([ADDR] = [WORD][OPERATOR][WORD][SEP])

[PLANT] [FD][COMMA][WORD][COMMA][WORD]

[COMMA][WORD] IN [AB][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]127[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]

[PI] = [AB][SEP]