Some Proposals for the Realization of a Certain Assembly Program

By R. A. Brooker and D. Morris

Some proposals are described for a method of implementing an input routine, which would permit the user to define the meaning of the statements he uses. It is hoped that a program will be developed along these lines for the Atlas computer.

In a previous paper we discussed the principal user aspects of "An Assembly Program for a Phrase Structure Language" (Brooker and Morris, 1960). In the present paper we describe how the scheme is implemented, and for this purpose we assume a knowledge of the terminology and content of the earlier paper. It is the function of the mechanism described to read and store the definition of an autocode language (in the form of "phrase definitions" and "statement definitions"), and then to translate programs written in this language. We discuss first the method of using the store, and the storage of definitions.

The Method of Using the Computer Store

We have already described how the store is split into a conventionally addressed section and a "chain store," and have discussed the advantages which are offered by the latter for manipulating information in the form of lists. A conventional section is desirable because it offers two important advantages:

1. The storage requirements are approximately halved.
2. Lists of words can be more easily scanned in either direction.

The basis of our compromise is that most of the working operations (i.e. the assembly and manipulation of lists) are carried out in the chain store, and the assembled reference material is transferred to the conventional part of the store. It is chiefly as a *record store*, therefore, that the conventional section is used. We provide for the essential manipulation of items in this store by the following means.

At the beginning of the record store an index is kept which contains the address of every item in the record store, and items are always referred to by means of their index number. Whenever a new item is entered in the record store the index number allocated to it will correspond to the next available register in the index, unless some other item has become redundant, in which case its index number will be taken over by the new item. It is only the index register of a redundant item which is re-allocated. The space it occupies is recovered by other means, namely by "sliding" back all the items beyond the redundant one, so as to fill the vacant space and release an equal amount at the "available" end. This means that all the items should be "store invariant." They are made so by writing all internal references relative to the origin of the item (external references are simply index numbers). In order to up date the index after items have been "slid back" it is necessary to subtract a constant (equal to the number of words recovered) from every address in the index which is larger than that of the reclaimed space.

A similar operation to the above can be carried out in order to extend an item. This facility avoids the necessity of making extravagant (safe) estimates for variable-length items (e.g. the index).

Obviously it is not practicable to use the record store for items whose usefulness is of short duration, or for items which are subject to frequent alteration. In our application, however, a large volume of information, such as the phrase definitions and statement definitions, is relatively static and can be profitably "filed" away so as to release some storage space.

Storage of Phrase Definitions

A phrase definition is the list of alternative phrases which comprise the class of phrase in question. Each alternative phrase is a string of basic symbols and/or class identifiers. By means which are described later the external multicharacter form of the identifiers (e.g. [TERM]) is reduced to a single (address) word after input (both forms are referred to as class identifiers). Basic symbols are also represented in the same form, but the numerical values assigned to them are confined to a different range. Address words, as we have already mentioned, have two irrelevant digits at the least significant end which can be used to distinguish different types of word. An ordinary "untagged" word is referred to as W. At the end of each string (i.e. phrase) we add a word containing the category number of the phrase within its class. This word is given a tag which marks it as a *terminal word* (written $\underline{W}$).

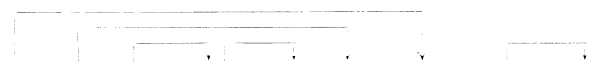
It is expected that in many definitions, some of the alternatives will have common stems, and dictionary grouping would therefore be advantageous. For example, the alternatives

A B C $\underline{1}$
A B E $\underline{2}$
B E C D $\underline{3}$
B E C $\underline{4}$
A E C $\underline{5}$
A B D $\underline{6}$

could be grouped as the dictionary

A { B { C $\underline{1}$; E $\underline{2}$; D $\underline{6}$ } ; E C $\underline{5}$ }
B E C { D $\underline{3}$; $\underline{4}$ }

In order to represent a dictionary linearly, we distinguish a further type of word called a *branch word* (&). This word contains the address of one branch whilst the other continues in the next location. For example, the above dictionary may be written



& A & B & C 1 & E 2 D 6 E C 5 B E C & D 3 4

This could be recorded either as a "chain" of 22 words in the chain store, or in a set of 22 consecutive registers in the conventional store.

We adopt the convention, when reading a "linear" dictionary, of taking the straight-on path first and recording the addresses of the alternative paths in a nest. If the straight-on path does not contain the information sought, then the last entry in the nest is explored, and so on.

The above grouping involves rearranging the alternatives within a class. If they are all mutually exclusive this is acceptable; if they are not, then we expect the order to reflect the order of preference of the non-mutually exclusive alternatives. In this case the user should write the phrase "in order of preference" before the first alternative. Limited dictionary grouping may still be possible whilst preserving the original order. For example, the above dictionary could be re-formed as

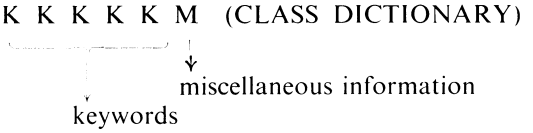
& A B & C 1 E 2 & B E C & D 3 4 A & E C 5 B D 6

A complete phrase definition is represented by a dictionary (called the *class dictionary*), supplemented by six other words. One of these additional words contains miscellaneous information, such as the number of alternatives in the class, and the other five are *key words*.

Each of the 120 binary digits of the keywords is associated with a particular basic (or composite) character; the digit is a 1 if the character it represents is a permissible starting symbol of the class of phrase in question, otherwise it is 0. The keywords are subsequently used by the expression recognition routine to answer questions of the type "can the word A be the starting symbol of a phrase of class C?" (say). If A represents a basic symbol, reference to the appropriate digit of the keywords will give a direct answer. However, if A is a class identifier, then C *may* begin with A only if all the 1's in the keywords for A are contained in those for C. That is an answer "may be" might be given and the alternatives in C are then tentatively compared with A.

If any member of a class *starts* with the identifier of a class which has not yet been defined then the keywords for the class cannot be completed. We therefore keep a list, in the chain store, of all the definitions which are incomplete, together with the outstanding definitions whose keywords they require. This list is updated whenever a complete set of keywords is obtained.

A complete phrase definition is assembled as a "chain" thus:



It is then transferred to the record store, and the index allocated to it becomes the associated internal class identifier. Conversion from an external multi-character identifier to the internal (index number) form is accomplished by means of the *class identifier dictionary*.

The Class Identifier Dictionary (CID)

As each phrase definition is recorded, a corresponding entry is made in the CID. This entry consists of a string of words containing the symbols of the identifier in question, terminated by a word containing the associated index number. If a class identifier occurs on the right-hand side of a definition before it has itself been defined, a provisional index number (to be used when the definition appears) is allocated to it and it is entered in the CID.

The structure of the CID is similar to that of a class dictionary, but it is retained in the chain store since it is frequently extended. All the entries are of course mutually exclusive, since each is a different string of basic symbols.

Qualified Class Identifier

In addition to writing simple class identifiers the user may also qualify them by means of "*" and "?" which indicate, "an arbitrary number of appearances of," and "an optional appearance of." Whenever an identifier, qualified in a particular way, is encountered for the first time, its formal definition is synthesized according to the rules:

[IDENTIFIER?] = [IDENTIFIER], nil
[IDENTIFIER*] = [IDENTIFIER] [IDENTIFIER*], [IDENTIFIER]
[IDENTIFIER*?] = [IDENTIFIER*], nil

The definition thus constructed is recorded along with the other phrase definitions, and the qualified identifier is added to the CID.

Statement Format Dictionary (SFD)

In the case of recursively defined statements it may not be possible to define the meaning of each in terms of previously defined statements only. For this reason we require that statement formats be defined independently of the definition of their meaning, and before they appear in any other context. Each format thus defined is added to the statement format dictionary. The terminal word of each entry (i.e. format) in this dictionary contains the index number to be allocated to its definition (when this is subsequently presented). In

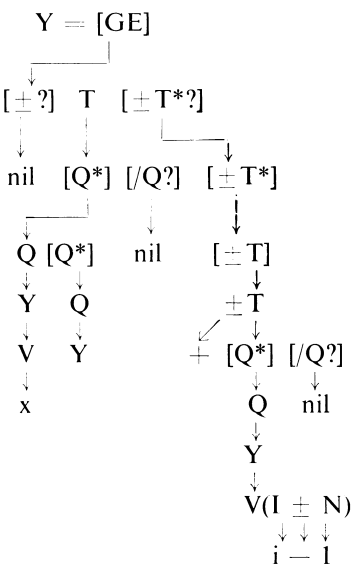
the meantime the corresponding register in the index will contain a dummy word which will not interfere with the working of the record store. Like the CID the statement format dictionary is also retained in the chain store.

Before we continue to describe the method of storing statement definitions, it is convenient to describe the routine which is used to recognize and analyse statements.

The Expression Recognition Routine (ERR)

This routine is concerned with recognizing expressions of basic symbols and/or class identifiers which can be generated from any of the phrase definitions or statement formats. An expression is generated by selecting a particular format (say) and replacing any or all of its class identifiers by particular phrases (or forms of phrases) consistent with their definitions. This process can be repeated until all the class identifiers have been reduced to basic symbols (as would be the case, for example, in statements occurring in the source program). Alternatively, the process can be stopped whilst some class identifiers remain, and these are then the *parameters* of the expression.

The generation of the parametric statement Y = xY + V(i − 1), for example, from the format Y = [GE] may be illustrated by the "substitution tree."



It is the function of the ERR to reconstruct the "substitution tree" and to determine the format from which it was derived.

For example, suppose the expression to be identified is $J_1, J_2 \ldots$, and that the stem $J_1, J_2 \ldots J_{j-1}$ has been recognized as being consistent with the stem $I_1, I_2 \ldots I_{i-1}$ of a statement format I (say). Now the ERR asks "is $J_j = I_i$?" If it is then i and j are advanced and the question is repeated. Otherwise the question "is J_j a possible first word of I_i?" is asked. A negative answer here means that $J_1, J_2 \ldots$ is not of the form I, and the next alternative statement format is investigated. However, if the answer is "yes," then the problem is to compare $J_j, J_{j-1} \ldots$ with the alternatives of class I_i. This is essentially similar to the original problem of comparing $J_1, J_2 \ldots$ to all the alternatives of the class of procedure formats, and can be dealt with by recursive use of the ERR.

For this purpose the ERR records all the current counts in a nest, and re-enters itself at a lower level with the address of the class dictionary for I_i replacing the address of the SFD. After this recursion has proceeded as far as is necessary, either to verify that a phrase $J_j, J_{j-1} \ldots J_{j-m}$ (say) is a particular member of I_i or that no such phrase exists, control will return to the primary level. If the phrase has not been recognized the alternative I is abandoned as above. Otherwise i will have been advanced by 1 and j by m + 1, and the process is repeated. Ultimate success is signalled if the end of the expression $J_1, J_2 \ldots$ and the end of I (i.e. the index number of its definition) are encountered simultaneously.

At all levels of the recursion the dictionary branch words are nested along with the current value of j, the dimension of the stem which has been recognized up to that point. When the search along a particular path proves fruitless, the path defined by the last entry in the nest is explored with j reset.

Nothing has so far been said about the form of the record produced by the ERR. It is, in fact, a one-dimensional representation of the substitution tree, and is called the *analysis record.*

The Analysis Record

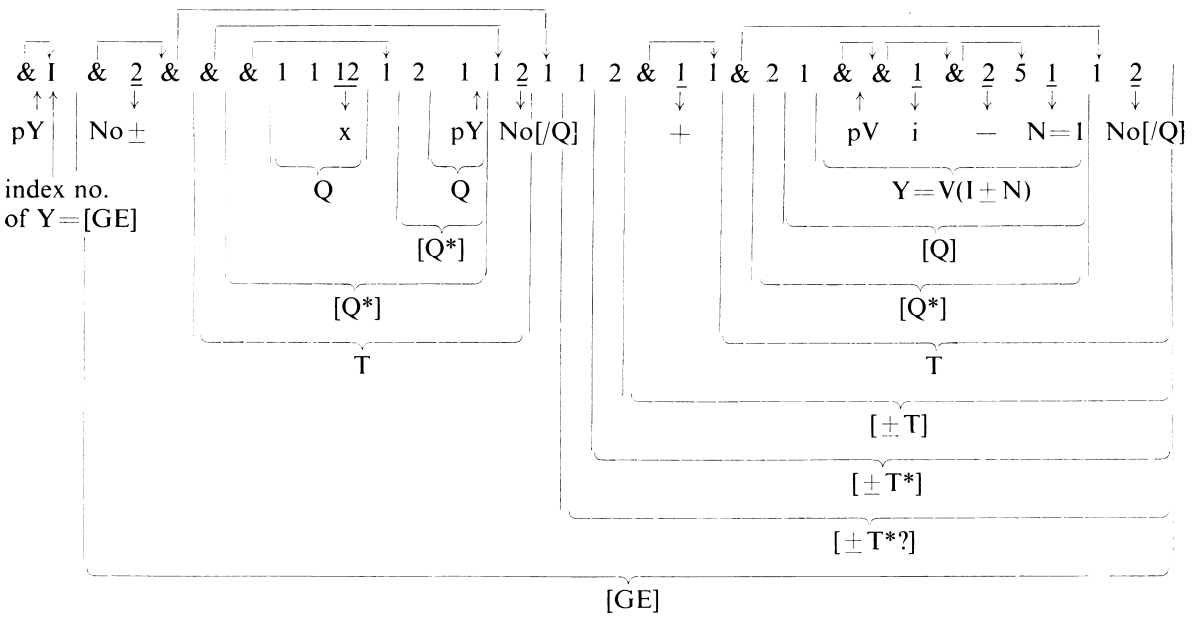
This takes the form of a list in which the items are sublists and hence are of variable length. In order to accommodate variable length items in a list we insert before each item an "& word," which contains the address of the & word preceding the next item. The word before the last item is replaced by an ordinary word and the list takes the form

& (item 1) & (item 2) . . . W (last item).

At the top level of the analysis record the items "correspond" (see below) to the identifiers of the recognized format, and the word W terminating the list contains its index number. No record is kept of the basic symbols which appear in the format because the success of the recognition process implies their presence. This "contracting out" of basic symbols is applied at all levels of the analysis record.

If a class identifier of the format also appears in the expression under analysis, then it represents a parameter whose value is to be supplied later. When this happens the analysis record will be contained in the chain store (see under Translation Routine), so that the actual values of parameters can be simply "linked" into the record. Hence there is no need to leave a provisional space, and instead the corresponding item in the record is completely omitted by linking the & word preceding it to

Table 1



the & word preceding the next item, and the parameter is recorded elsewhere (see below—the APL).

If a class identifier does not appear explicitly in the expression then it will have been replaced by one of the alternative strings of the class it represents. In this case the item in the record is itself a list whose items "correspond" to the class identifiers of the particular alternative, and the word terminating this list-structure will be the category number associated with that alternative. Eventually, where the regression does not end at parameters, it will end in a list having no items (i.e. only basic symbols which are omitted). The category number of these empty lists is recorded as an isolated *terminal* word.

The parameters of an expression analysed by the ERR are recorded in a separate list called the *associated parameters list* (APL). Each item in this list occupies five words which are used as follows:

(1) The first of these words is the class identifier.

(2) The second word contains the label (if any) which distinguishes it from other appearances of the same class of phrase. If the class identifier is not labelled, this word will be given a tag which indicates that its contents are irrelevant (note that we do not need to distinguish words for any other purpose in this list).

(3) The third type of word is associated with the bracketed type of label which indicates a particular appearance of a repeated item. If present this label may be either an α, a β, or an integer, or alternatively it may be omitted. We use all four permutations of the tag digits to distinguish these cases, and the index of any α or β concerned is recorded as an integer.

(4) The fourth word contains the address of the link word which has to be connected to the parameter in question.

(5) The last word is used to record the original value of the link so that the chain can be restored if a parameter is subsequently disconnected.

As an example of an analysis record we present in Table 1 the record for Y = xY + V(i − 1) analysed with regard to Y = [GE]. The APL is not given, but the points where the parameters have to be inserted are indicated by pY and pV.

In a later section we introduce a pseudo class identifier denoting "the class of phrase identifiers." When this is encountered it is treated separately by the ERR, and the record takes a special form. It utilizes a fourth kind of word, namely a *block word* denoted by B_n, and meaning "regard the next n words as a block of information and ignore their tag digits even though they may coincide with those of & and $\underline{W}$ words or even the block word itself."

Packing of Analysis Records

When the analysis records of the substatements in a statement definition are transferred to the record store, they are encoded (or "packed") according to the following scheme, which takes advantage of the fact that they are largely comprised of small integers. Since they

would not in any case be used *in situ*, the time taken to unpack them is worth expending for the resulting gain in storage space. If, however, the need for economy in space is not so great, then the more frequently used definitions could be transferred to the record section without being packed. In this case the APL would still be incorporated in the analysis record, as below.

In an analysis record the words &, W, $\underline{W}$, B_n are distinguished by means of different combinations of the two least significant digits. In what follows it is of interest to state the exact forms which these tags take.

& takes the form (address part) .11
W takes the form (address part) .10
$\underline{W}$ takes the form (address part) .00
B_n takes the form (address part) .01

The first step in packing an analysis record is to incorporate in it the APL. This means inserting each parameter (i.e. the first three words of each entry in the APL) at the appropriate point of the record. These parameter strings are prefixed by a special symbol p (say), which could be represented by a block word with a negative address part. Next we drop the addresses from the & words and regard them as left brackets. Consider for the moment a corresponding right bracket preceding each word whose address is contained in an & word. The analysis record given above would then appear as

(pWWW) I ($\underline{2}$)(((11$\underline{12}$)121pWWW)1$\underline{2}$)112($\underline{1}$)1(21(pWWW)($\underline{1}$)($\underline{2}$)5$\underline{1}$)1$\underline{2}$

↓ ↓ ↓

3 words describing Y 3 words describing Y 3 words describing V

Obviously the original form can easily be restored. Now we drop all the right brackets since it is evident that all terminal words, parameter strings (and blocks denoted by B_n), imply a following right bracket. We now have a string of words containing short integers (in which the least significant two digits are relevant), and the two separator words denoting (and p. This is packed as a sequence of consecutive characters (a 24-bit word comprises 4 characters) in the conventional store, according to the following rules.

(1) A word in which only the least significant 6 bits are relevant, and in which the two least significant bits are different from 11, is stored as one character. (Since the & word has been omitted only those (few) words following a B_n or p may terminate with 11.)

(2) A word in which not more than 9 bits are relevant, but excluding the above type, is stored as the most significant 9 bits of two characters whose remaining three bits are 111. (This includes words which terminate in 11.)

(3) Any word in which more than 9 bits are relevant is retained as 4 characters and is preceded by a character whose 4 least significant bits are 1011.

(4) A p symbol is represented by a single character $-\delta 10011$.

(5) A (symbol is represented by a single character $-\delta 00011$.

The procedure for unpacking a record is to examine it character by character. If the least significant two bits of a character differ from 11, it is restored as a 24-bit word and the next character is examined. Otherwise the least significant bits of the character must take one of the following forms: 111, 1011, 10011, 00011. A character different from all these could be used to mark the end of the record.

Recognition of the above character endings denote respectively:

the next 9 digits are to be assembled as a word,
the next 4 characters are to be assembled as a word,
a parameter is following,
insert a left bracket and subsequently convert it to an & word.

The Built-in Expressions and Formats

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

[$\alpha\beta$] = [word]
[$\alpha\beta$] = [word] θ [word]
([addr]) = [word]
([addr]) = [word] θ [word]
→ [$\alpha\beta$N] [IU] [word] ϕ [word]
→ [$\alpha\beta$N]
Let Π = [some Π expression]
Let Π = [some Π expression]
→ [$\alpha\beta$N] [IU] Π = [some Π expression]
→ [$\alpha\beta$N] [IU] Π = Π
[$\alpha\beta$] = category of Π
[$\alpha\beta$] = number of Π
END

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

N = [D*]
K = N, N., .N, N.N
α = α1, α2, etc.
β = β1, β2, etc.
[$\alpha\beta$] = α, β

[$\alpha\beta$N] = α, β, N
[addr] = [$\alpha\beta$], [$\alpha\beta$] + [$\alpha\beta$N], [$\alpha\beta$] − [$\alpha\beta$N], [$\alpha\beta$] ⊕ [$\alpha\beta$N]
[word] = [addr], ([addr]), N.D, .D, N
θ = +, −, ×, /, &, v, ⊽
ϕ = =, ≠, >, ≥, <, ≤
[IU] = if, unless
Π = "any phrase identifier"
[some Π expression] = "any expression consistent with the definition of Π."

With the exception of K, which is "built-in" because it is a frequently occurring expression, these definitions are required in order to interpret the dictionary of built-in operations. They are also available to the user as constituents of the phrases which he defines, although some have meanings so special that it is unlikely they would be used except in defining the system in terms of itself. The analysis record associated with most of the above expressions is consistent with previous descriptions, and we mention below the exceptions to this, which are treated as special cases by the ERR.

In the case of Π, the analysis record takes the form of a "block" of three words, containing the identifier and its labels (if any) encoded in the same manner as the parameters of a statement (see previous description of the APL). The record for the [some Π expression] will be consistent with the definition of the phrase identified in Π.

Four other built-in expressions, namely, N, K, α, β, are also treated in a special manner. Expressions of the type N are subjected to decimal-binary conversion and the resulting integers are recorded as if they were the category numbers of the class (i.e. N can be regarded as a class having $2^{21} - 1$ members). An expression of the type K will also be subjected to decimal-binary conversion, but this time the result will be recorded as a 48-bit floating-point number. This number will be added to a central list (unless it already appears there) and the "category number" in the analysis record will contain its address (or the address of the previous appearance of the same number). In the case of α the "category number" in the analysis record will contain its index. The record for a β is similar, but is more fully described under "Translation Routine."

The Storage of Statement Definitions

A statement definition is a list of substatements, basic listing instructions, and parameter operations. Each "line" in this list is submitted to the ERR on input, and an analysis record and APL are thus obtained.

The statement heading is also subjected to this treatment and analysed with respect to the corresponding format in the SFD (the purpose of this will become evident later). At the same time as the definition is read, a directory is assembled, noting the relative position within the definition of those "lines" which have primary labels. The assembled definition takes the form of a list employing & words, thus:

& (directory) & (statement heading) & (1st instruction) ...

which is then transferred to the next available space in the record store and the index is set accordingly. Here the & words and the directory occupy single registers, whilst each "line" of the definition is a packed analysis record. The first two registers of the directory are associated with the β's used in the definition and not with the primary labels. One gives the maximum index of the β's and the other (only relevant when a definition is in use) contains the address of the first of a "conventional" set of registers allocated to the β's. The addresses in the directory and in the & words of the list will be those of the & words preceding the lines to which they refer, given relative to the first & word of the definition.

We may now describe the sequence of events which occurs when a source program instruction is encountered.

The Translation Routine

Each source program instruction will be analysed by the ERR. This produces an analysis record, and the index number of the relevant statement definition. These are handed on to the master routine whose function is to "translate" the instruction with the aid of the definition in question, and any other definitions which this may call in.

Two working lists which this routine requires are set up on entry. The first is the list of registers to be used by the β_i of the definition. For this list we allocate the next available set of registers in a part of the conventional store reserved for this purpose. The address of the first of the allocated registers is entered in the first register of the directory associated with the current definition, and a count which contains the address of the next available register in the reserved store is advanced by the number of β's used in the current definition (i.e. the second word in its directory). We shall see later that a definition must be able to refer back to the β list of a previous definition. For this reason a β (say βn) is represented in an analysis record as (64 P + n), where P is the index number of the definition in which it occurs. Thus by extracting the P and referring to the associated definition (in particular the first word of its directory), the relevant β list is located. The above implies that no definition uses more than 64 β's. Definitions may also be used recursively and for this reason the contents of the first register in a directory are never destroyed. Instead they are nested when the β list is set up and restored when the present use of the definition terminates.

For the second working list we use a chain. This contains information relating to the expressions which are singled out in the analysis record of the instruction under analysis. It is called the *list of selected expressions* (LSE). The first entries in this list correspond to the

parameters of the statement heading, and are entered by using the analysis record of the statement heading as if it was an instruction of the type: "Let (the instruction under analysis) ≡ (the statement heading)." Subsequent entries are also made when further instructions of the above type (and other parameter operations) are encountered (see below). Each entry in the LSE consists of three words, namely, the class identifier, the label (if any) which distinguishes it from other expressions of the same class, and the address of the item corresponding to it in the analysis record.

After these two lists have been set up the definition is "obeyed" beginning at the "line" following the statement heading which, like any other "line," can be either a substatement, a basic listing instruction, a parameter operation, or an "END."

Before a substatement or basic listing instruction can be obeyed, its analysis record must be unpacked into the chain store. Next, the current values of any parameters it contains are substituted. If a parameter does not involve a particular appearance of a repeated item the substitution is straightforward (we discuss the alternative at the end of this section). The parameters to be substituted are described in the first two words of each entry in the APL. By comparing these with the first two words of each entry in the LSE the parts of the analysis record corresponding to the required sub-expressions are located. It then remains to take the address of the sub-expression from the LSE and enter it in the "link word" whose address is in the fourth position in the APL, and to record the previous value of this "link word" in the fifth position in the APL. Examination of the index number of the (now complete) analysis record decides whether it is a substatement or a basic listing instruction, which are dealt with as follows:

(1) *Substatements.* With the parameter substitutions effected, a substatement corresponds to a particular source program instruction, and the translation routine uses itself recursively in order to translate it. All the counts relating to the current level are recorded in a nest. These include

(*a*) the address of the current statement definition,
(*b*) the address within this definition of the current substatement,
(*c*) the address of the LSE,
(*d*) the next available location in the space reserved for β's,
(*e*) the address of the current analysis record.

The unpacked analysis record of the substatement now becomes the analysis record of the instruction under analysis, and the synthesis routine enters itself at a lower level to execute the associated definition.

(2) *Basic listing instructions.* These are executed by an interpretive routine which scans their analysis record to determine the required sequence of machine operations.

The Parameter operations are also unpacked and executed by interpretive routines, but the details are of more interest. In the type "Let $\Pi \equiv$ [some Π expression]; Π is the name of a particular sub-expression of the instruction under analysis, and [some Π expression] will be an analysis record for a similar expression involving parameters. These parameters will be in the APL for the instruction but will not appear in the LSE, since it is the function of the instruction to identify the parts of the analysis record corresponding to them, and to make the relevant entries in the LSE. For this reason the parameter linking operation is omitted when the instruction is unpacked. The left-hand side expression is located by means of the LSE, and the corresponding part of the analysis record is compared with that of the right-hand side [some Π expression], by means of the "tree comparison routine."

It is the function of this routine to verify that the two trees are of the same form and to locate the sub-trees of the former which correspond to the parameters of the latter. This means tracing through the structure of the former and checking that the latter contains the same integers in its ordinary and terminal words (W and $\underline{W}$) and that it branches at the same points. Complete correspondence is expected except when a branch of the [some Π expression] tree is prematurely terminated and the remainder is represented by a parameter. These points are recorded in the APL, and when the trace reaches them the address of the corresponding point in the other tree is noted and the next branch is explored. Any other discrepancy between the two trees violates the equality. If the equality is satisfied, the end of the two trees will be reached simultaneously. In this case the parameters contained in the APL are added to the LSE, together with the addresses of the corresponding parts of the analysis record.

In the case of the "let $\Pi \equiv \ldots$" instruction only success is acceptable, and failure leads to an "error print" routine. However, in the type "$\rightarrow$[$\alpha\beta$N] [IU] $\Pi \equiv$ [some Π expression]" which is otherwise identical, either success or failure is accepted and will decide which instruction is to be obeyed next.

The tree comparison routine is also used in the case of the instruction "$\rightarrow$[$\alpha\beta$N] [IU] $\Pi_1 = \Pi_2$." Here the expression on each side of the equality is looked up in the LSE and the trees are compared as before. They do not, of course, involve parameters.

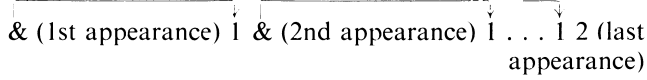
New expressions may also be introduced by means of the instruction "Let Π = [some Π expression]." In this case only previously introduced sub-expressions appear as the parameters of the right-hand side and the relevant substitutions are made when the instruction is unpacked (as in the case of a substatement). The name of the new expression is Π and this is added to the LSE together with the address of the analysis record of the right-hand side.

Finally, to execute the instructions:

$$[\alpha\beta] = \text{category of } \Pi$$
$$[\alpha\beta] = \text{number of } \Pi$$

we first look up the address of the expression Π in the LSE. It will be recalled that the category number of an

expression comes before the last principal item in the analysis record and is therefore located by tracing the & words to the end of the list. In the case of "number of" the expression will be a repeated one with a structure



so that again a simple count of the & words will provide the required information.

Similarly, when a parameter is encountered which specifies a particular appearance within a repeated sequence, the address of the list structure representing the repeated class is first found in the LSE, and then the particular appearance required is obtained by "skipping" over the appropriate number of & words.

It remains to discuss the operation "END." When this word is encountered at any level of the above recursion, control is returned to the level above, with all the counts reset. Eventually, when an "END" is encountered at the top level, the translation of a given source program instruction is complete.

Recovery of the Storage Space used by the Translation Routine

Another function of the instruction "END" is to initiate the process which recovers the space occupied by the redundant information left in the "wake" of a statement definition. The only conventional store used by a definition is its β list, and this is recovered by resetting the relevant count. At this point it is also appropriate to restore the original contents of the directory register which contains the address of the first β (i.e. the quantity nested when the new β list was set up). The redundant information in the chain store comprises

(1) all the unpacked analysis records and APL's,
(2) the LSE,
(3) a list containing the addresses of (1).

The analysis records are first restored to the chain form they had prior to the parameter substitution. For this purpose the broken links were noted in the APL (as the fifth word of each entry). The lists (2) and (3) and the APL are also in chain form, so that all the redundant space can now be recovered by linking each chain back on to the main chain. Obviously this operation will be simplified if we work throughout with *circular* chains (or lists).

The Input Process

In this section we describe how the relevant material is prepared for and read into the computer. The input medium is 7-hole paper tape and is prepared on a "Flexowriter." This is a machine similar to a typewriter but, in addition to giving a printed record, as each key is depressed a characteristic pattern of holes is punched on a new *row* of the paper tape. In each row there are seven positions in which a hole can be punched (i.e. seven *tracks*), but as there are less than 64 patterns involved only six of these are necessary. The seventh position is used as a parity check and is only punched in those rows which would otherwise have an even number of holes. (Input from 5-hole tape will also be provided, but the techniques are similar to those described below.)

The principal features of the Flexowriters to be used in conjunction with Atlas are as follows. There are 43 keys each of which is associated with a pair of visually distinct characters. Depression of any key results in one or other of the two characters being printed according as to whether the machine is the *upper case* (UC) mode or the *lower case* (LC) mode. The tape code (i.e. the pattern of holes punched on the tape) is the same in both cases, however. The two cases are selected, as on a normal typewriter, by means of shift keys, the operation of which is also recorded on the tape so that the following symbols can be correctly interpreted. The tape codes associated with the case shift keys are themselves independent of the mode of the machine so that, e.g., depressing the UC key produces the same pattern of holes regardless of whether the machine is already in the UC mode or not. The same applies to the keys for effecting the operations of *space* (SP), *backspace* (BS), *newline* (NL), and *erase* (EX); and certain other keys whose function is not relevant here. The *erase* code consists of a hole punched in all 7 positions. There is a mechanical tape reader attached to the machine by means of which a tape can be "played back" to produce a further printed record (and if necessary another tape record). During a playback an UC tape code has no effect if the machine is already in the UC mode, and similarly for LC.

When presented to the computer, the Flexowriter tape is scanned by a photo-electric tape reader under the control of the *initial input program* which discards the parity bit of each row (after checking the parity), and prepares an ordered list of 6-bit codes inside the store of the computer. Initially these may be packed 4 to a word (of 24 bits), but for the next stage they must be represented by individual words as integers in the range 0–63 (in address units).

The first step is to eliminate the UC and LC codes by introducing an extra bit in each code, in order to represent the 86 distinct printed characters. This is most simply done by adding 64 to the UC codes and the operational codes, and leaving the LC codes unaltered.

The next stage amounts to a line-by-line reconstruction process in which the information between successive *newline* codes is standardized. For this purpose each line is regarded as being made up of a fixed number of cells, one for each position across the line of a paper record. By scanning a line in the "forward" sense (i.e. in the order in which it was punched) and noting the *space* and the *backspace* operations, we can determine the characters which fall into each cell. Neither space nor backspace are regarded as characters in this sense. If the same character appears more than once in any cell, then appearances after the first are ignored.

The information in each cell is called a *symbol*. If it

consists of a single character only it is called an *elementary symbol*, otherwise a *composite symbol*. Thus, e.g., 0 is a composite symbol consisting of the superimposed characters "0" (zero), "—" (minus), and "_" (underline). The list of characters in each cell is now rearranged in ascending numerical order. Thus if "0" ≡ 80, "—" ≡ 94, and "_" ≡ 22, then the resulting string is "_0—," irrespective of the order in which the symbols were punched. This ordered string of characters is then looked up in a *dictionary of recognized composite symbols* (DRCS), which gives the corresponding single word identifier (or serial number) to be used in place of the string. Henceforth it is treated in the same way as an elementary symbol, and the dictionary can be regarded as a means of extending the elementary symbol codes. More than one composite string may lead to the same identifier. Thus 0 can also be formed with the UC *letter* "O" in place of *figure* "0." Since *letter* "O" ≡ 111 the equivalent string becomes "_ — O." The DRCS may also be extended so as to recognize the dominant character in certain cases of overpunching (e.g. to replace "— (BS) +" by "+").

The empty cells in a reconstructed line are regarded as being filled by *spaces*, and the line is terminated with an *end of line* symbol (EOL).

Not all possible composite symbols are included in the dictionary, and if a given string is not found it is left in the standard form

S (BS) S (BS) S (BS) . . . (BS) S

in which *backspaces* are inserted between the constituent elementary symbols. *Backspace* itself therefore becomes an elementary symbol. The underlined symbols will probably be treated in this way, since otherwise it would mean almost doubling the number of "elementary" symbols and hence the number of digits in the key words of a class definition. As proposed at present there are 5 key words which provide up for to 120 "elementary" symbols. These must include the elementary printed symbols, namely

(i) the 26 upper case letters A–Z,
(ii) the 26 lower case letters a–z,
(iii) the 10 decimal digits 0–9,

and (iv) the 24 miscellaneous separator symbols

[ ] ( ) < > + − × β π . , * | _ ½ ⁻¹ ? & = : ′ /

which together with the three special symbols *space backspace* and *end of line* make a total of 89. This leaves room for 31 recognized composite symbols, such as

⩽ ⩾ ← → ± ≠ θ ϕ ≡ ≠

However, for a reason explained later, 5 key digits are reserved for certain special symbols, so that only 26 such composite symbols are recognized.

The reason for the somewhat elaborate reconstruction process just described is to enable the machine to interpret correctly any tape which reads correctly when played back on the Flexowriter (ignoring erases). Thus, for example, it is impossible to say from looking at the play-back just how many "spaces" there are on the tape, following the last significant (i.e. printed) symbol on the line. The eye merely sees the gap between this symbol and the right-hand margin, and this correspondence is preserved in the machine by making each line consist of a standard number of character positions and defining the "empty" position as *spaces*. This fact should be borne in mind when specifying formats. Thus, if each instruction is intended to start on a new line, for example, then the last significant symbol of the instruction would be followed by [SP*?] EOL. Similarly, if the spacing of the individual symbols of the instruction is irrelevant, then, strictly speaking, we should put [SP*?] between each member of the relevant phrase definitions. This would be rather tedious, however, and so we have introduced a primary statement of the form "ignore [ ]'s" where [ ] refers to some previously defined phrase (which may include the "nil" form). It applies to all subsequent phrase definitions and statement formats, until superseded by a further statement of the same kind. It means ignore the phrase or symbol in question where it occurs between the principal members in any subsequent appearance of the phrase or format being defined. With the aid of this device, only the EOL need be specified in the above example.

Recognition of the Primary Material

In the remaining sections we discuss how the compiler recognizes the material which is presented to it in the form just described. For this purpose we shall need to define the syntax of our *primary* language precisely. Previously a rigid syntax has not been used, and the reader may therefore notice discrepancies between the definitions which follow and some of the foregoing examples. It is intended that the syntax described below should be observed in practice. The primary material consists of statements describing the form and treatment of the *secondary* or source language material. We have introduced three main types of primary statement, namely *phrase definitions*, *statement formats*, and *statement definitions*. The relative ordering of these statements is only restricted by the following:

(1) A phrase must be defined before it appears in a statement definition, but not necessarily before it appears in other phrases, or in statement formats.
(2) A statement format must be defined before any substatement derived from it appears in a statement definition, and before the statement definition associated with it is given.

The individual primary statements and blocks of secondary material each begin on a new line with one of the master phrases: *phrase defn*, *statement format*, *statement defn*, *ignore*, and *secondary statements*. There may also be other master phrases of a supervisory

nature, e.g. *title*, *end of message*, *estimated running time*, etc.

For various reasons we have avoided introducing a specific symbol to indicate the end of a primary statement or sequence of secondary statements. Instead the end is recognized by the appearance of the next master phrase. Thus as each line is reconstructed it is submitted to the ERR to see if it starts with one of the foregoing master phrases. It is necessary, of course, that the master phrases be distinct from all other possible starting phrases. The "program" is thus decoded and reconstructed section by section, where each section represents a primary statement or sequence of secondary statements. In either case the sections are terminated by an internal *end of section* symbol, which is distinct from the 115 "elementary" symbols. Each section is then processed in accordance with its category, starting at the first symbol after the master phrase.

The Primary Statement

Certain elementary symbols may not appear directly in descriptions of source language statements and expressions, because they are used for meta-syntactical purposes in the primary statement itself. These are "SP" "EOL" "," "[", which instead are written as [SP] [EOL] [,] [[]. The meta-syntactical use of these characters is as follows: "SP" and "EOL" are usually ignored; "," is used as a separator (e.g. between source language expressions in phrase definitions), and "[" is used to introduce phrase identifiers† and, of course, the special forms of these characters themselves. The internal identifiers corresponding to these four symbols are replaced by values 117–120 (the *end of section* symbol corresponds to 116), while the special sequences will eventually be replaced by the corresponding "elementary" identifiers normally associated with these symbols.

Note: The *backspace* symbol is not available to the user in any form, so that a phrase definition such as

$$[\underline{A} - \underline{Z}] = [A - Z]\ [BS]\ \_$$

where $[A - Z] = A, B, C, \ldots, Z$

must instead be written as

$$[\underline{A} - \underline{Z}] = \underline{A}, \underline{B}, \underline{C}, \ldots \underline{Z}.$$

The primary statements are scanned for the identifiers enclosed in square brackets, which take the general form [[ ] [identifier] [asterisk?] [query?] [label?] [index?]] where

[identifier] = any sequence of symbols other than / ( ] * ?
[asterisk?] = *, nil
[query?] = ?, nil
[label?] = /N, nil
[index?] = ([α β N]), nil

† For the purpose of the present discussion we assume that all phrase identifiers are enclosed in square brackets, contrary to the description given earlier where single letter identifiers were also used. (The special forms [SP] [EOL] [,] [[ ] are recognized as particular cases of this general form.)

The general phrase identifiers are now replaced by the internal phrase identifier, namely a group of three words corresponding to (1) the class identifier, (2) the label, and (3) the index (as explained in the description of the APL).

The class identifier is obtained by looking up the identifier sequence in the CID, which also includes the special character sequences, as well as the "built-in" identifiers α, β, N, etc. The serial numbers of the different categories lie in different ranges and enable the type of entry to be distinguished in the ERR. Those of the four special characters are, of course, the internal form of the corresponding elementary symbols, and lie in the range 1–115. Those corresponding to the "built-in" expressions will occupy a higher range of values, and the general class identifier will occupy a higher range still. When a special symbol sequence is identified it is replaced by one word only, namely the basic symbol it represents. If the identifier string is not present in the CID, then it is entered (i.e. merged into it) and allocated the first available serial number.

The treatment of qualified classes has already been explained in principle. The qualifying symbols * and ? are regarded as part of the identifying sequence which is looked up and, if necessary, entered in the CID in the usual way. If it is present then the class has already been defined and no special action is necessary; but if it is not present then it is necessary to set up the appropriate class definition, e.g.

[identifier * ?] = [identifier *], nil.

This is subsequently treated in the same way (see below) as any other class definition: if the [identifier *] involved is not present in the CID then it is entered as before and a further class definition is set up, thus:

[identifier *] = [identifier] [identifier *], [identifier].

If the reduced [identifier] is not present in the CID then it also is entered, but even if it is present the class may not have been defined (since the phrase definitions can be introduced in any order), and in this case further action (that is, attention to the key digits) must await a formal definition (i.e. by the user) of this reduced class.

In the course of obtaining the internal identifier we can also carry out certain checks: for example, that an index occurs only in conjunction with an [identifier *], except in the special case of [N([$\alpha\beta$])]. We can also ensure that the label and index parts are absent in phrase definitions and statement formats.

As a result of the foregoing operations, the internal form of the primary statements now consists of a string of "elementary" symbol identifiers, phrase identifier triplets, and the four primary separators SP EOL COMMA and EOS (end of section). This string of identifier words is then processed according to the nature of the primary statement.

The Phrase Definitions

These have the form:

phrase defn: [Π] = [preference clause?] [phrase * ?] [last phrase] EOS (ignoring SP's and EOL'S), where:
[preference clause] = (in order of preference)
[phrase] = [Π, ES*] COMMA
[last phrase] = nil, [Π, ES*]
[Π, ES] = [Π], [ES]
[Π] = phrase identifier
[ES] = elementary symbol identifier

The description could be strengthened by insisting that the [Π] employed on the left-hand side must not be qualified in any way.

The Statement Formats

These have the form (again ignoring SP's and EOL'S):

Statement format = [Π, ES*] EOS or
Statement format (auxiliary): [Π, ES, COMMA *] EOS where
[Π, ES, COMMA] = [Π], [ES], COMMA

The second form allows the user to introduce statements which are not intended for use in the source language itself, but only as intermediate steps in the construction of certain statement definitions. In this case the format may include primary COMMA's which otherwise cannot be used. The two kinds of statements are kept in distinct dictionaries so that the auxiliary statement formats can be omitted during the translation of source language material.

An internal description in terms of such "phrase definitions" and "statement formats" will, in fact, be used to provide, with the aid of the ERR an "analysis record" of the structure of the primary statement string. It is the task of the corresponding "statement definition" to merge the alternative phrases of a phrase definition into a class dictionary, preserving the order of preference if necessary; or, in the case of a statement format, to add it to the end of the appropriate SFD (since the statement formats are given in order of preference).

In the case of a class dictionary the key words are also set up, and subsequently updated as described in the account of the CID.

These operations can be described in terms of the list-compiling instructions which we have already introduced.

The "Ignore" Statement

This takes the form: ignore [Π]'s. It is treated as follows. A facility is built into the ERR whereby at each level it will ignore (i.e. pass over) all phrases of a specified class, when trying to identify a particular class of phrase in the string under comparison. (No mention of whether such phrases are present is made in the analysis record, however.) In the internal form of the class definition, each alternative phrase is prefixed by the identifier of the particular phrase (or symbol) which it is required to ignore in that particular case. If this is *nil* then this word will indicate the empty class. In a phrase definition, the "ignore" applies to all the alternative phrases, and so, as a result of the merging process, its identifier appears only once at the head of the dictionary "tree." This is not the case, however, with the statement format dictionary, where the "ignore" may apply to the individual formats.

Statement Definitions

These take the form:

statement defn: [substatement] [line *] [separator * ?] EOS

where:

[substatement] = [auxiliary substatement], [secondary substatement]
[line] = [separator * ?] [primary label ?] [line proper]
[separator] = COMMA, EOL
[primary label] = [N]]
[line proper] = [built in operation], [substatement]

The combinations SP and EOL [SP * ?] / (i.e. EOL followed by solidus) are ignored. This directive is in addition to any ignore statements applying to the substatements involved.

The complexity of the foregoing "phrase defns" is necessary in order to allow the freedom of layout which is usually taken for granted in an informal program.

The formats of the built-in operations do not, as they stand, include any characteristic terminal symbol, and we therefore arrange for the corresponding internal format descriptions to terminate with the [separator] phrase. The same applies to auxiliary statement formats: it is automatically included at the end of the format. These operations must therefore be followed in a statement definition by a COMMA (i.e. a proper comma) or an EOL (i.e. a new line). A secondary statement format will normally include its own terminal phrase, e.g. [EOL] or [,] (corresponding to actual EOL or comma in a source language program), and it must terminate in this way when it appears in a statement defn, although it may be followed by further primary [separator]'s. With these precautions, a sequence of substatements and built-in operations can be recognized unambiguously provided they do not start with a primary COMMA or EOL. The latter is impossible and the former unlikely. If any material "overflows" the end of a line, then it can continue on the next line *provided that it is prefixed by a /*, since this is ignored after an EOL.

The formats of the built-in operations constitute a dictionary distinct from both the secondary SFD and the auxiliary SFD. They include the parameter operations and the basic compiling instructions. These may not appear directly in the source language material, and those of them which could be used there, namely the compiling instructions:

$$[\alpha\ \beta] = [\text{word}] \quad [\alpha\ \beta] = [\text{word}]\ \theta\ [\text{word}]$$
$$([\text{addr}]) = [\text{word}] \quad ([\text{addr}]) = [\text{word}]\ \theta\ [\text{word}]$$

must be formally redefined, thus, e.g.:

statement format: [α β] = [word] [EOL]
statement defn: [$\alpha\beta$] = [word] [EOL]
[α β] = [word],
end

Such a statement occurring in the source language program, e.g. $\alpha_1 = (\alpha_3)$, would have a purely declarative significance, corresponding to those statements in classical input routines which manipulate "preset parameters." The expressions α and β may also occur in any permissible source language statement built up from these compiling instructions. In this case, however, it is necessary to restrict the number of β's (say to 50) for a reason which is explained in the next paragraph.

The other built-in operations involve either parameter operations, or transfers of primary control which have no significance in the source language program because, unlike a statement definition, such a program is read, analysed, *and translated* statement by statement. A further consequence is that the range of β's involved cannot be determined. Normally this is done with the aid of the routine (part of the ERR) for recognizing phrases of the form βn, which also notes the maximum value of n. This value is then recorded when all the lines of the statement definition have been read and analysed. In the case of a source language program this never happens because each statement is translated as soon as it is analysed.

In the case of the basic compiling instructions, which are redefined for use in the source language program we have the choice of either form for use in a statement definition, e.g.

$$\alpha_1 = (\alpha_3), \text{ or } \alpha_1 = (\alpha_3) \text{ [EOL]}$$

Naturally one would employ the original form.

It should be remembered that the expression which replaces [some Π expression] in certain of the parameter operations is normally a source language expression, so that any commas appearing in it must be written in the form [,].

The format of the parameter operations (see *Storage of Statement Definitions*) can be strengthened by differentiating between the [Π]'s used in the different operations, as was suggested in the case of the phrase definition. Thus we have the following rules.

1. In the operations

$$\text{Let } [\Pi] \equiv \text{[some } \Pi \text{ expression]}$$
$$\rightarrow[\alpha\beta\text{N}]\ [\text{IU}]\ [\Pi] \equiv \text{[some } \Pi \text{ expression]}$$

the phrase identifiers occurring in [some Π expression] must not be indexed. This can be checked by examining the resulting APL. Also, [Π] itself cannot take the form N([$\alpha\beta$]).

2. The [Π] employed in the operation

$$\text{Let } [\Pi] = \text{[some } \Pi \text{ expression]}$$

must not involve an index. The phrase identifiers appearing on the right-hand side, however, can take the most general form, as in a substatement.

3. The [Π]'s appearing in the operations

$$\rightarrow[\alpha\beta\text{N}]\ [\text{IU}]\ [\Pi] = [\Pi]$$
$$[\alpha\beta] = \text{category of } [\Pi]$$

must not take the [N([$\alpha\beta$])].

4. In the operation

$$[\alpha\beta] = \text{number of } [\Pi]$$

the [Π] must represent an "*" class and, of course, must not be indexed.

The above checks would be included in the subroutines involved by the ERR when attempting to identify the different types of [Π].

The "Statement Defn" of a Statement Definition

The string of identifier words representing the statement definition is compared, by means of the ERR, with an internal description in the above terms. This results in an analysis record which provides almost directly the information needed to assemble the final form of the statement definition. (See *Storage of Statement Definitions.*)

The Secondary Material

It remains to recall what happens to the source language material. This, of course, is submitted for comparison with the secondary statement format dictionary by means of the ERR.

When the appropriate form is recognized, the corresponding statement definition is entered and executed (see the description of *The Translation Routine*). This results in the statement being translated unless it has an operational significance, in which case it may cause the material translated thus far to be obeyed as a program. Otherwise the compiler continues to read and translate secondary material until another section (i.e. a master phrase) is encountered.

Conclusion

As already mentioned, the system could be formulated entirely in its own terms, using phrase definitions, statement formats, and statement definitions. This would mean including such operations as dictionary search, packing and unpacking, etc., among those used in the statement definitions. These could either be built into or built up from the basic listing instructions.

Reference

BROOKER, R. A., and MORRIS, D. (1960). "An Assembly Program for a Phrase Structure Language," *The Computer Journal*, Vol. 3, p. 168.