A General Translation Program for Phrase Structure Languages*

R. A. Brooker and D. Morris

The University of Manchester, England

Introduction

A program input system for the Atlas [1] is described which operates in two phases. In the *primary phase* it accepts the definition of a phrase structure language, and in the *secondary phase* it will translate a source program written in that language. The primary material consists of *format definitions* and *phrase definitions* describing the form (syntax) of the sentences, clauses and constituent expressions, and *assembly routines* which describe their "meaning" (i.e. semantics). The system is extendable and allows the user to define the meaning of new formats in terms of existing formats as well as in terms of *basic assembly instructions* (whose meaning is built in). Both the form of expressions and their meaning can be defined recursively, a feature which is particularly useful, for example, in dealing with algebraic formulas involving parentheses.

It is unlikely that every machine user will want to write his own autocode: what is more likely is that he may wish to extend one of the standard languages to include statements suited to his own problem area. To build up a reasonably useful language from "scratch" would take about one to three months. The mechanism is explained below with reference to Mercury Autocode [2].

Phrase Definitions

Phrase definitions are used to build up classes of logically similar phrases. To each class is assigned a name, the class identifier, which may then be used in further phrase definitions and format definitions to indicate that any phrase of the class in question may be substituted in its place. Class identifiers are represented by a string of characters enclosed in square brackets (e.g. INDEX might be assigned to the class of index letters). The following, therefore, could be the start of a formal definition of Mercury Autocode:

phrase [V] = a, b, c, d, e, f, g, h, u, v, w, x, y, z, π
phrase [V′] = a′, b′, c′, d′, e′, f′, g′, h′, u′, v′, w′, x′, y′, z′
phrase [INDEX] = i, j, k, l, m, n, o, p, q, r, s, t
phrase [VARIABLE] = [V′], [V] [N], [V] [INDEX], [V] ([INDEX] [±] [N]), [V]

Note. The four symbols [, *space* and *end of line* have a metasyntactical significance in all the primary material. Appearances of these symbols in the source material being described are denoted by [[] [,] [*SP*] and [*EOL*] respectively. (The principal input medium of Atlas is 7-hole punched paper tape prepared on Flexowriters.)

* Received July, 1960; revised July, 1961.

The class [N] denotes an integer and it is a built-in class which does not require further definition. The same applies to the class [K], used below, which describes a general constant.

Two features which are very often present in a phrase or source statement are "repetitive appearance" and "optional appearance" of some item. In order to describe these two situations the qualifiers * and ? may be used and the relevant formal definitions will be constructed behind the scenes. Thus:

[A] [B*] [C] means [A] [B] [C] or [A] [B] [B] [C] or [A] [B] [B] [B] [C] etc.
[A] [B?] [C] means [A] [B] [C] or [A] [C]
[A] [B*?] [C] means [A] [C] or [A] [B] [C] or [A] [B] [B] [C] etc.,

where only [A], [B], and [C] require formal definitions.

The general arithmetic expression in Mercury Autocode could now be defined as follows:

phrase [Q] = [VARIABLE], [K], [INDEX]	x_2, 3.6, i
phrase [TERM] = [Q*] [DIVISOR?]	$2x_1u/3$
phrase [DIVISOR] = / [Q]	/n
phrase [±] = +, −	−
phrase [± TERM] = [±] [TERM]	$-2x_1u/3$
phrase [GENERAL ARITHMETIC] = [±?] [TERM] [± TERM*?]	$2u_1u_2 - 3v_1v_2/4$

When there is more than one alternative in a phrase definition the ordering may be important as in the definition of [VARIABLE] given earlier. Here the last category [V] logically includes the 2nd, 3rd, and 4th categories, in the sense that [V] by itself means [V] followed by "anything." By placing it last, therefore, in order of preference the expression recognition routine (ERR) will only fall back on this possibility when having recognised a [V] it fails to identify an [N], [INDEX], or ([INDEX] [±] [N]). The ordering of the categories of a phrase definition is then the order in which the ERR attempts to identify them when looking for a phrase of the class in question. To take a more abstract example we must also exclude such definitions as C = A,B where A = xyy,x and B = xy. Here B includes some of the phrases represented by A, and since the reverse also applies we cannot have C = B,A. Instead, we must write *C* = xyy, xy, x —if this is what we mean.

Format Definition

Certain expressions are more conveniently defined in a progressive fashion. This is the case where the number of alternative formats is large and likely to grow. Two such classes are of permanent significance, the class of *source statements* (identified by [SS]) and the class of *auxiliary statements* (identified by [AS]). The source statements are the "sentences" which can appear in the source language, while the auxiliary statements are, as their name suggests, used in building up the meaning of the source statements from the *basic assembly instructions*.

The individual members of a format class are defined in order of preference by *format definitions*, for example:

format [SS] = [VARIABLE] = [GENERAL ARITHMETIC] [EOL]
format [AS] = ACC = [GENERAL ARITHMETIC]
format [AS] = [VARIABLE] = ACC
format [AS] = ACC = [±?] [TERM]
format [AS] = ACC = ACC [±] [TERM]

(Both these and the phrase definitions given in the previous section will be referred to in the subsequent examples of assembly routines.)

Format classes other than [SS] and [AS] need to be introduced by a special statement, for example:

format class [IDENTIFIER]

Individual members are then presented as described above. In Mercury Autocode, however, no format classes other than [SS] and [AS] are needed.

Assembly Routines

To each member of a format class corresponds an assembly routine. This describes the "meaning" of the format, i.e. the action to be taken when it is encountered in the analysis of a source program. For source statements of an imperative nature the action is to plant the equivalent set of machine instructions in the target program, but in the case of declarative statements such as $x \longrightarrow 10$ representative of the general source statement defined by

format [SS] = [V] → [N] [EOL]

(and analogous to the dimension statement of FORTRAN) the action is to enter certain information in a list for reference by subsequent imperative source statements. Statements can of course be partly imperative and partly declarative and both are ultimately expressed in terms of *list compiling operations*, the only difference being that in the former case the "list" in question is the target program itself. These list compiling operations represent a group of the basic assembly instructions.

Very often the "meaning" of a source statement in the above sense can be expressed in terms of a sequence of other less complex source statements (or suitably chosen auxiliary statements) where the subexpressions of the main statement are the *parameters* of the assembly routines corresponding to the "sub" statements. It is necessary therefore to have some means of resolving a source statement (or any other expression) into subexpressions consistent with its known structure. If a phrase has alternative forms it is necessary to be able to discriminate between them, selecting appropriate courses of action. It is necessary to be able to construct new expressions from the phrases of existing expressions, and finally it is necessary to be able to compare existing phrases for identity. These operations are performed by another group of the basic assembly instructions, the *tree operations* (so called because a source statement is repre-

sented in the machine by a branching structure or "tree"). For the purpose of "control" transfers, a floating address system is employed and any substatement or basic assembly instruction can be labelled, for example

10] → 3 unless [VARIABLE] ≡ [V] [INDEX]

is a basic tree operation labelled 10. The effect is to jump to an "instruction" labelled 3 unless a specific phrase [VARIABLE] takes the form [V] [INDEX], the 3rd principal category. If it does take this form then the phrases [V] and [INDEX] automatically become "known".

To summarize, an assembly routine will consist of the following types of "instruction":

1. Substatements (i.e. calls for the corresponding assembly routines)
2. List compiling operations (basic assembly instructions built into the system)
3. Tree operations

List Compiling Instructions

Associated with these instructions is a *central* group of 24-bit registers denoted by α_1 , α_2 , α_3 , $\cdots$ which do not form a field (i.e. cannot be referred to as α_r). In addition to these, there is a further set of *local* 24-bit registers β_1 , β_2 , β_3 , $\cdots$ associated with each assembly routine. The list compiling instructions are concerned with selecting, processing, and comparing the information in these registers and in the registers whose addresses are contained in them. Thus, e.g. $\alpha_{10} = \beta_2 + (\alpha_1 + 3)$ means set register α_{10} equal to the contents of register β_2 plus the contents of the register whose address is given by $\alpha_1 + 3$; and $\rightarrow 1$ if $(\beta_1) > \alpha_3 + 2$ is typical of the testing variety and means 'jump' to the instruction labelled 1 if the number in the register whose address is in β_1 is greater than $\alpha_3 + 2$.

Now in general if α_1 (say) is the address of the first item of a conventional list (i.e. one in which consecutive items lie in consecutively addressed registers) then $(\alpha_1 + n)$ is the $(n+1)$-th item in that list. In addition to the conventional list, however, the *chain* list is also of frequent use. In this list each 24-bit word containing an item is accompanied (in the next register) by a further word which contains the address of the next item (i.e. word pair). Generally the address of the first word of such a list is recorded in the link word of the last item, thus making the list circular. The advantage of the chain list is that it is easy to manipulate, e.g. to insert and delete items simply means inserting or detaching a link. However, given α_1 (say) as the address of the first item of such a list the $(n+1)$-th item is not now $(\alpha_1 + n)$. Instead, the address of the second item in fact is given by β_1 (say) $= (\alpha_1 + 1)$, and the third by $\beta_1 = (\beta_1 + 1)$, and so on. As a shorthand way of referring to various items in a chain, the symbol $\oplus$ is used. The $(n+1)$-th item is thus denoted by $(\alpha_1 \oplus n)$, and this item can be transferred to β_5 (say) by the instruction $\beta_5 = (\alpha_1 \oplus n)$. Thus $(\alpha_1 \oplus 1)$ is equivalent to $(\alpha_1 + 1)$ if α_1 is an address in a chain.

Some Examples of Assembly Routines

The use of substatements and the basic assembly instructions is illustrated by the following examples.

```
routine [SS] ≡ [VARIABLE] = [GENERAL ARITHMETIC] [EOL]
               ACC = [GENERAL ARITHMETIC]
               [VARIABLE] = ACC
               END
```

The first item in every assembly routine is the *heading*. This serves the dual purpose of relating the routine to a particular format (of the class [SS] in this case) and also resolving the actual format into its principal subexpressions, [VARIABLE] and [GENERAL ARITHMETIC]. These then become "known", i.e. are available for use as parameters in subsequent instructions. In the above case the two instructions which follow are routines, using as actual parameters the phrases [GENERAL ARITHMETIC] and [VARIABLE] which appeared in the heading, i.e. in the particular source statement being processed.[1] In the above example the principal phrases of the heading pass directly into the substatements as parameters.

Ultimately however they will have to be broken down into further subexpressions before the information can be used, and in general a new level of stratification is revealed in each subroutine. The next example illustrates this point.

```
routine  [AS] ≡ ACC = [GENERAL ARITHMETIC]
         let [GENERAL ARITHMETIC] ≡ [±?] [TERM] [± TERM*?]
         ACC = [±?] [TERM]
         → 1 unless [± TERM*?] ≡ [± TERM*]
      3] → 2 unless [± TERM*] ≡ [±] [TERM] [± TERM*]
         ACC = ACC [±] [TERM]
         → 3
      2] let [± TERM*] ≡ [±] [TERM]
         ACC = ACC [±] [TERM]
      1] END
```

In this case only one parameter appears in the routine heading. So obviously if the auxiliary statement is to be defined in terms of simpler statements this parametric expression must be resolved into its subexpressions. The first "instruction" of the definition does this, and the expressions appearing on the right-hand side of the equality can then be referred to. The next "instruction" is an auxiliary statement which is not defined here but which is intended to plant instructions in the target program to set the accumulator equal to the first (possibly signed) term of the [GENERAL ARITHMETIC]. Next the nature of the [± TERM*?] has to be determined as this may be either [± TERM*] or "nil". In this latter case, control is passed to the end by the conditional param-

[1] If there is more than one phrase of the form [VARIABLE] (say) in a routine heading, then we distinguish them by writing [VARIABLE/1], [VARIABLE/2], etc.

eter-resolving instruction. If control passes sequentially to the next "instruction", the [GENERAL ARITHMETIC] must involve a subexpression [± TERM*]. Now in order to resolve this into a more elementary form it is necessary to know how the "*" classes are defined inside the machine. In general [IDENTIFIER*] is defined recursively as [IDENTIFIER] [IDENTIFIER*], [IDENTIFIER]. The recursive structure of [± TERM*] is thus expanded by a cycle of instructions which deal with each [± TERM] in the sequence until the last, which is dealt with at the point labelled 2. As in conventional programming, the same name can be dynamically assigned to a sequence of different expressions. In this example the names [±], [TERM], [± TERM*] are all used in this way. The auxiliary statement

ACC = ACC [±] [TERM]

which appears in every cycle including the last makes use of the [TERM]'s resolved in each cycle: the corresponding assembly routine plants instructions in the target program to add (or subtract) each [TERM] to (or from) the accumulator.

Another method of access to the individual [TERM]'s in the heading is illustrated in the next example, which is an alternative version of the same assembly routine.

```
routine  [AS] ≡ ACC = [GENERAL ARITHMETIC]
         let [GENERAL ARITHMETIC] ≡ [±?] [TERM] [± TERM*?]
         ACC = [±?] [TERM]
         → 1 unless [± TERM*?] ≡ [± TERM*]
         β1 = number of [± TERM*]
         β2 = 1
       2] let [± TERM*(β2)] ≡ [±] [TERM]
         ACC = ACC [±] [TERM]
         β2 = β2 + 1
         → 2 if β1 ≧ β2
       1] END
```

The tree instruction β_1 = number of [± TERM*] determines the number of [± TERM]'s actually present in the particular phrase [± TERM*] on hand. This is used to count the number of cycles in the loop which follows. In this loop the instruction

let [± TERM*(β_2)] ≡ [±] [TERM]

automatically selects the β_2'th appearance of [± TERM] and resolves this into the expressions [±] and [TERM]. Strictly speaking, this is not as efficient as the process of recursive resolution used in the first example, since the counting is done from the beginning each time by counting through the branches of the tree representing [±TERM*].

Yet another version of the same routine makes use of a rather different definition of [GENERAL ARITHMETIC], thus:

```
phrase [GENERAL ARITHMETIC] = [±?] [UNSIGNED ARITHMETIC]
phrase [UNSIGNED ARITHMETIC] = [TERM] [±] [UNSIGNED ARITHMETIC],
                                                        [TERM]
```

With these definitions we can associate the following assembly routines.

```
routine  [AS] ≡ ACC = [GENERAL ARITHMETIC]
         let [GENERAL ARITHMETIC] ≡ [±?] [UNSIGNED ARITHMETIC]
         ACC = [UNSIGNED ARITHMETIC]
         → 1 unless [±?] ≡ −
         change sign of ACC
      1] END
routine  [AS] ≡ ACC = [UNSIGNED ARITHMETIC]
         → 1 if [UNSIGNED ARITHMETIC] ≡ [TERM]
         let [UNSIGNED ARITHMETIC] ≡ [TERM] [±] [UNSIGNED ARITHMETIC]
         ACC = [UNSIGNED ARITHMETIC]
         ACC = [TERM] [±] ACC
         END
      1] ACC = [TERM]
         END
```

In these routines the instructions

```
change sign of ACC
ACC = [TERM] [±] ACC
ACC = [TERM]
```

are auxiliary statements whose assembly routines will plant instructions to perform the tasks indicated.

Although the above examples do not contain actual listing instructions, the definitions of some of the substatements used will eventually lead to sequences of basic instructions which will compile the target program. For example:

```
routine  [AS] ≡ change sign of ACC
         (α1) = N1
         (α1 + 1) = N2
         α1 = α1 + 2
         END
```

Here α_1 is always the current address in the target program, that is, the next instruction will occupy the pair of 24-bit locations α_1 , α_1+1. The function and modifier digits (N_1) stand in α_1 , and the address part (N_2) in α_1+1. In this case a single instruction of the form "ACC = S − ACC", where S = N_2 is the address of floating-point zero, will suffice to change the sign of the accumulator. Clearly a format for any machine instruction—or a general format for them all—could be defined and interpreted in the same way.

Some of the lower level auxiliary statements of Mercury Autocode will require a knowledge of the variable directives (i.e. source statements of the form [V] → [N]) which have gone before. One way this information might be made available is for the assembly routine associated with [V] → [N] to record the [N] associated with each of the [V] letters in a particular position in a conventional list of 15 registers separate from the target program. The obvious way of associating the letters of [V] with positions in the list is to use the same ordering as in the phrase definition of [V], i.e. a in position 1, b in position 2, etc. This, however, requires that a mechanism be provided to determine which

alternative within a phrase definition a particular expression represents. The built-in instruction,

$$[\alpha\beta] = \text{category of ["any phrase identifier"]}$$

is provided for this purpose. Thus, in the example below, whenever the [V] in question is a "d" (say) then β_1 will be set to 4. It is assumed below that α_3 has been reserved for the address of the directive list and that it will not be altered by other statement definitions, but, of course, they may refer to it.

```
routine  [SS] ≡ [V] → [N] [EOL]
         β1 = category of [V]
         β1 = β1 − 1
         (α3 + β1) = [N]
         END
```

Other Format Classes

We have associated assembly routines with the individual members of a format class. The two classes [SS] and [AS] enjoy a privileged position insofar as they also represent two forms of instruction found in an assembly routine. This is expressed more formally as:

$$\text{[instruction]} = \text{[built-in instruction]}, \text{[AS]}, \text{[SS]}$$

The formats of all three classes must be mutually distinct. An assembly routine associated with a format class other than [SS] or [AS] might be headed (to borrow an example from a commercial autocode):

```
routine [logical description statement] ≡ units = [LITERAL]
```

corresponding to a format definition:

```
format [logical description statement] ≡ units = [LITERAL]
```

To call in the routine associated with the category or format of a particular [logical description statement/1] on hand (i.e. already resolved), we use the instruction

```
call in R [logical description statement/1]
```

(representative of the more general form "call in *R*[format class identifier]").

We have not associated assembly routines with the principal categories of a phrase definition. The meanings of phrases are embodied in the routines associated with the formats which employ them. If a phrase has several alternative forms this is reflected in the relevant routines by the appearance of a multiway switch, e.g.,

```
β1 = category of [Y]
→ β1
```

or other means of discrimination. If the number of alternatives is very large and likely to grow the routine will become unwieldy and in this case the phrase definition is better treated as a format class.

Conclusion

The foregoing examples are intended to give a general idea of how the system is used, but nothing has been said about how it is realized behind the scenes. Some of our earlier ideas on this matter are presented in [10], but the techniques currently employed will have to be the subject of another paper. There is space here for only a very brief outline of how assembly routines are used in the process of translation (the secondary phase).

Each source statement is first analyzed with respect to the dictionary of source statement formats by the *expression recognition routine.* This produces a record of the phrase structure in the form of a branching structure or tree. The assembly routine associated with the particular source statement is then entered with this tree as working material. The routine heading picks out the subtrees corresponding to the principal subexpressions and enters their locations in a *list of selected expressions* (LSE). There is a local LSE for each assembly routine. Any subsequent tree instructions in the routine will operate with these subtrees, e.g. resolving them into further subtrees, identifying categories, constructing new trees, etc. The subtrees may also be employed as parameters in "substatement" instructions and it is appropriate to explain how this is effected. Instructions involving parametric phrases (which may include basic instructions) are recorded in the assembly routine as incomplete tree structures, the "loose ends" indicating the points at which the subtrees representing the parameters should be connected. When such an instruction is executed these loose ends are first connected to the "tops" (i.e. the main branches) of the trees in question, which will be found in the LSE, since these phrases will have already been resolved. With the tree complete the corresponding assembly routine is called in, and in the case of a basic instruction a built-in interpretive sequence is entered. (Basic instructions which are free of parameters are replaced by the equivalent machine instructions when the assembly routine is first read into the machine.) Control eventually returns to the original routine after passing down and up a heirarchy of similar routines in the usual way.

REFERENCES

Some of the ideas embodied in the scheme we have described have been in circulation for some time. For example the use of chaining to represent lists of unpredictable length can be found in [3–7]. (In fact, however, we have found comparatively little need to resort to this technique, most of the relevant data being stored in a perfectly conventional fashion.) Phrase definitions were used by Backus [8] in describing an early version of ALGOL.[2] References [9] and [10] outline some earlier ideas on the form and realization of the kind of language described here. Eventually it is hoped to publish a more readable "compiler writers' guide" illustrating the use of the various types of assembly instructions.

[2] The referee has pointed out that there is a very close correspondence between our "phrase definitions" and Kleene's "regular expressions" [see *Representation of Events in Nerve Nets and Finite Automata*" (p. 24) by S. C. Kleene, Automata Studies, Annals of Math Studies 34, Princeton University Press, 1956]. Although introduced for a rather different purpose, they employ the same ideas of recursive definition and include an operator equivalent to our combined "*?".

1. KILBURN, T.; EDWARDS, D.; WARBURTON, E. T.; AND PAYNE, R. B. The Atlas computing system. (In preparation).
2. BROOKER, R. A. (1958) The Autocode programs developed for the Manchester University computers. *Comput. J. 1*, 15.
3. NEWELL, A.; AND SHAW, J. C. (1957) Programming the logic theory machine. Proc. Western Joint Comput. Conf. (1957), 230.
4. NEWELL, A.; AND TONGE, F. (1960) An introduction to information processing language V. *Comm. ACM 3*, 205.
5. WINDLEY, P. F. (1960) Trees, forests, and rearranging. *Comput. J. 3*, 84.
6. PERLIS, A. J.; AND THORNTON, C. (1960) Symbol manipulation by threaded lists. *Comm. ACM 3*, 195.
7. MCCARTHY, J. (1960) Recursive functions of symbolic expressions and their computation by machine, Part 1. *Comm. ACM 3*, 184.
8. BACKUS, J. W. (1959) The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. Proc. Internat. Conf. Informat. Proc., UNESCO, Paris, June, 1959.
9. BROOKER, R. A.; AND MORRIS, D. (1960) An assembly program for a phrase structure language. *Comput. J. 3*, 168.
10. BROOKER, R. A.; AND MORRIS, D. (1960) Some proposals for the realisation of a certain assembly program. *Comput. J. 3*, 220.