Experience with the Compiler Compiler

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

This paper records some experience with the Compiler Compiler, a compiler designed to facilitate the writing of compilers. Since the project has been the subject of several articles,† we shall assume the reader is conversant with the general ideas involved, and describe mainly how they have worked out in practice, in the dozen or so general and special purpose compilers which have been written with it. Finally we discuss the general utility of the project.

Just as ALGOL and FORTRAN are primarily designed to handle arrays of floating point numbers, so the Compiler Compiler (henceforth CC) is designed to handle data structures of tree form which arise out of syntactical analysis. Apart from this the two kinds of languages have much in common. Thus a CC program consists of a master routine and a set of subroutines, one to process each type of source statement in the language, and one for any auxiliary statements that it may be convenient to introduce. These subroutines are all at the same textual level, as in FORTRAN. The master routine is always the same—it is supplied by the CC—and operates as follows. It reads the source program a line at a time and parses it with respect to the grammatical rules of the language which are the phrase and format definitions supplied by the user. The parsing algorithm (actually a subroutine of the master routine) identifies the format to which the line of source program corresponds and produces a record (in the form of a tree) of its internal syntactical structure. The master routine then calls the routine associated with the format and translates the instruction recognized. After this control returns to the master routine which reads the next line of the source program.

The following extracts from a FORTRAN compiler should remind readers of the main features of the language. We consider the ARITHMETIC STATEMENT

```
FORMAT [SS] = [VARIABLE] = [EXPR]
```

where the meta-variables are defined as

```
PHRASE [EXPR] = [PRECEDING ±][TERM]
  [TERMS]
PHRASE [PRECEDING ±] = [±], NIL
PHRASE [±] = +, −
PHRASE [TERMS] = [±][TERM][TERMS], NIL
PHRASE [TERM] = [FACTOR][FACTORS]
PHRASE [FACTORS] = *[FACTOR][FACTORS],
  /[FACTOR][FACTORS], NIL
```

† See References.

None of the published descriptions of the system is entirely satisfactory, that of Rosen (1964) being perhaps the most objective. R. B. E. Napper (Computer Science Dept., Manchester Univ.) has also written a very readable account of the system from the users' point of view, which is available on request.

We introduce the auxiliary statements

```
FORMAT [AS] = LOAD [PRECEDING ±]
  [TERM]
FORMAT [AS] = ACC = ACC[±][TERM]
FORMAT [AS] = DIV BY [FACTOR]
FORMAT [AS] = MULT BY [FACTOR]
FORMAT [AS] = LOAD [PRECEDING ±]
  FACTOR
FORMAT [AS] = DUMP ACC IN [VARIABLE]
```

Examples of two of these routines are

```
ROUTINE [SS] ≡ [VARIABLE] = [EXPR]
    LET [EXPR] ≡ [PRECEDING ±][TERM]
      [TERMS]
    LOAD [PRECEDING ±][TERM]
  2) → 1 UNLESS [TERMS] ≡ [±][TERM]
      [TERMS]
    ACC = ACC [±][TERM]
    → 2
  1) DUMP ACC IN [VARIABLE]
    END

ROUTINE [AS] ≡ LOAD [PRECEDING ±]
      [TERM]
    LET [TERM] ≡ [FACTOR][FACTORS]
    LOAD [PRECEDING ±][FACTOR]
  2) → 1 UNLESS [FACTORS] ≡/[FACTOR]
      [FACTORS]
    DIV BY [FACTOR]
    → 2
  1) → 3 UNLESS [FACTORS] ≡ *[FACTOR]
      [FACTORS]
    MULT BY [FACTOR]
    → 2
  3) END
```

The other [AS] routines follow similar lines calling in progressively simpler routines until at the lowest level the actual translated instructions are planted.

* *Department of Computer Science, The University, Manchester* 13.

Table 1

Storage space used by compilers

	EMA	ALGOL	FORTRAN 2	AA	AB
Phrases	998	871	494	369	
Phrase routines	470	1,389	528	531	
Formats	642	77	261	293	6,656
Format routines	9,670	9,695	9,194	6,359	
CC routines	4,528	3,326	4,244	3,452	
PERM lists	0	250 (approx.)	200 (approx.)	920	764
Size of compiler proper	14,218	15,358	15,798	11,924	7,420
PERM	2,908	602	1,299	4,895	4,895
Deleted material	7,168	10,240	8,704	12,213	—

Notes 1. The CC routines include the MASTER routine, and the routines for parsing, line reconstruction and pre-editing, and some of the [BS] interpretive sequences.
2. PERM refers to the preloaded library of input/output routines and special functions used by the object programs of the compilers.
3. PERM lists refer to the name and property lists of the preloaded PERM. These lists are needed (and expanded) in the course of compiling.
4. The deleted material has been referred to in the text: it includes phrases, formats, and routines.

The phrases and formats

Phrase definitions are represented in the computer by the tree structures described in Brooker, *et al.* (1962), henceforth referred to as *Trees and Routines.*

The format definitions are recorded as though they were the alternatives in a single phrase definition. Four classes of format are basic to the system (others can be introduced if desired).

[SS] the class of source statement formats.

[MP] the class of master phrases (e.g. PHRASE, FORMAT).

[BS] the class of built-in statements for use in routines.

[AS] the class of auxiliary statements which the user may introduce for use in routines.

The last three are only used in the PRIMARY phase of the CC in which it reads the phrases, formats, and format routines, i.e. the definitions, syntactic and semantic, of the language. These formats and the phrase definitions specific to them can be deleted once the compiler is "defined" and becomes operational (the SECONDARY phase). In its undeleted form the compiler remains an incremental compiler, i.e. further statement definitions can be added.

The actual storage space occupied (after deletion) by the various language definitions is given in **Table 1.**

The parsing algorithm

This has been fully described in *Trees and Routines.* It is a top to bottom recognizer taking advantage of common stems and precedence of alternatives in the phrase definitions.

The parsing record for the Atlas Autocode statement

$$l = 1 + sqrt\,(x(i)^2 + y(i)^2 + z(i)^2)$$

amounts to 127 half words. It will be clear from this that the analysis record resulting from parsing an entire program as a single source statement may occupy a great deal of space. In the case of ALGOL this amounted to about four times that of the source program (packed 4 symbols to a word).

This was overcome in the Atlas ALGOL compiler (written by R. H. Kerr and J. Clegg of I.C.T. Ltd.) by using a phrase routine (see below) to define a [BASIC STATEMENT] as text terminated by a semicolon, etc., the parsing record consisting simply of the characters suitably packed. It would be the function of the routine for processing the [BASIC STATEMENT] to further parse the text, that is to analyze it and identify assignment statements, "go to" statements, etc. The detailed parsing of the lower levels which consumes a great deal of space is thus suspended until it is actually needed, and then carried out (basic) statement by statement. In this way the convenience of using the original syntax is

preserved. Alternatively after each basic statement in the syntax a pseudo phrase routine could be inserted which would convert (and hence condense) the parsing record of the basic statement to semi-compiled object code (thereby partly usurping the function of the corresponding format routine).

The parsing algorithm has sometimes been criticized because it operates top-down instead of bottom-up. However, the grammatical definitions are usually such that it would make little difference which method is used *if the complete parsing record is to be determined.* More to the point is the fact that complete parsing of a fairly simple statement such as $X = 1$ with respect to [VARIABLE] = [EXPR] involves the construction of trees with many empty branches. It is the time spent in this activity, and on the subsequent inspection of the empty branches in the processing routines, that makes our approach inherently less efficient than (say) the technique of operator precedence for arithmetical expressions. If necessary, however, we can always fall back on such methods through the use of phrase routines.

Phrase routines

The CC allows the user to substitute in place of a formal phrase definition an informal phrase routine which may employ more efficient recognition techniques, e.g. table look-up (although still using [BS] statements as far as possible to avoid the use of machine code). In such cases the parsing algorithm will abandon the formal procedure and enter the routine provided, which must of necessity construct its own analysis record. If this too is informal (which is usually the case) it will also require informal treatment in the processing (format) routines. This is in fact the procedure adopted with most of the permanent or built-in phrase routines, e.g. [N] ≡ integer. There is no limit to the amount of analysis which can be put into a phrase routine and if necessary entire statements can be recognized in this way. Table 1 indicates the extent to which such routines have been used.

The format routines

These also are implemented as described in *Trees and Routines*. The main point which we would emphasize here is the desirability of providing compiling versions (mentioned briefly at the end of that paper) for as many formats as possible. These enable statements to be translated into machine orders. In the case of the [BS] formats the compiling versions are already provided, and consequently most [BS] statements are fairly efficient. For example, $\beta 1 = \beta 2 + 3$, which refers to two index registers of Atlas, is replaced by a single machine order. Control transfers and index testing instructions are also close to machine code.

Most of the parameter resolving and testing instructions are compiled by the same criterion. For example

→ 3 UNLESS [FACTORS] ≡ * [FACTOR] [FACTORS]

is replaced by machine orders which

1. take the address of the sub-tree for [FACTORS] from the stack;
2. compare the category number at the top of this tree with the category number of *[FACTOR] [FACTORS] in the definition of [FACTORS] (i.e. 1) and transfer control to the instruction labelled 3 if they differ;
3. otherwise (if they are equal) further instructions take the next two words in the tree for [FACTORS], which will be the pointers to the trees for [FACTOR] and the second [FACTORS], and copy them into the stack positions allocated to the meta-variables [FACTOR] and [FACTORS]. Thus subsequent instructions may access those newly located sub-trees.

The operation of the auxiliary statements introduced by the user may also be of interest. For example

LOAD [PRECEDING ±][FACTOR]

would be compiled into orders to

1. pick up the address of the sub-trees for [PRECEDING ±] and [FACTOR] from the stack and plant them into that part of the stack which will become the working space for LOAD [PRECEDING ±][FACTOR];
2. call a shortened version of the routine-changing sequence.

Those instructions which cannot be compiled into machine orders and which have to be packed up as trees within a routine, are interpreted at run time. The parameter resolving and testing instructions are interpreted by a hand-coded subroutine which compares the trees involved. Auxiliary statements are transplanted into the stack (as described in *Trees and Routines*), so that the relevant subroutine may be entered in the same way as the top-level routine is entered with the analysis record of the original statement. Interpretation is time-consuming (say 50 orders for resolution instructions and auxiliary statements) and is avoided as far as possible.

A format routine therefore may be partly compiled and partly interpretive, though generally it is completely compiled. To give some idea of the space occupied the LOAD [PRECEDING ±][TERM] routine occupies 36 orders. The total space occupied by the format routines of the various languages is given in Table 1. Again this refers to the deleted compilers, since all compiling versions and many of the [AS] and [BS] interpretive routines are no longer needed once a compiler is "defined".

An overall assessment

Compilers for the following languages have been written with the CC:

EMA	ALGOL	FORTRAN IV	D. Morris and I.C.T. Compiler team (R. H. Kerr, J. Clegg and others).

Table 2

Breakdown of compiling times

Per cent of total compiling time

	EMA	ALGOL	FORTRAN 2	AA	AB
Input	17·3	27·1	11	13	19·8
Line reconstruction and pre-editing				23	44·8
Analysis	57·9	39·2	29·3	46	35·4
Processing	24·8	33·9	59·7	18	
Speed 1	1,452	1,100	2,275	537	298
Speed 2	303	169	355	197	112

Notes 1. Speed 1 is the number of machine instructions obeyed per instruction compiled. Speed 2 is the number of machine instructions obeyed per character of program tape. These are average figures for a number of programs. In each case Speed 1 showed the most individual variation (the speeds for some programs being as much as ±50% about the mean). The other measures, Speed 2 and the distribution ratios, showed much less variation.

2. One or two miscellaneous figures are also available: Speed 1 for an early version of AA was approx. 1,500. The overall compiling times for the same program run with EMA and CHLF3 was in the ratio of 3·9 (CHLF3 is a hand coded compiler for a language amounting to a subset of EMA).
3. In none of these compilers were any special measures taken to optimize object code. (FORTRAN 2 was essentially a first exercise in using the CC.)

ACL	D. Hendry, University of London Atlas Computer Service.
CPL	G. Colouris, University of London Institute for Computer Science.
Atlas Autocode (AA)	J. S. Rohl.
Elliott Autocode Mk 111	R. de Morgan, Department of Social Medicine, University of Manchester.

There also exist some augmented versions of the above, e.g. EMA and ALP, AA and survey analysis package, as well as purely special-purpose compilers. In addition the CC has been used as the basis of a scheme for programming in "natural" language (Napper, 1964).

In only two cases has it been possible to make any comparison with a hand coded compiler on the same machine, these being Atlas Autocode and Mercury Autocode, and comparative figures are given in **Table 2.**

It will be seen from Table 1 that the (deleted) compilers produced with the CC are comparatively large, occupying nearly twice as much space as a hand coded version. This is no disadvantage if the storage space is available, although it may restrict the possibilities of time sharing.*

The size of an undeleted compiler, however, could be embarrassing, and it may be desirable to temporarily delete it for testing purposes, although, in practice it is often left in undeleted form until it is debugged and commissioned.

From Table 2 it will be seen that the total compiling times of a CC compiler can be up to four or five times as slow as a conventional compiler, unless special provision is made to optimize those paths most frequently taken through the compiler, in which case it is possible to reduce this factor to 2 or less, as has been done in the case of AA. This could be done by recasting (and if necessary hand coding) certain phrases as phrase routines, and the relevant sections of the format routines (see Table 1). Certain [AS] routines may need compiling versions. One may also rearrange the phrase definitions to give frequently occurring phrases order of precedence (where this does not conflict with the logic of the definitions) and possibly recast them to reduce the logical depth of the set.†

Coding with [BS] statements and hand coding, though a chore, are less repugnant when using a modular subroutine system such as the CC provides, in which formal and informal routines can be interchanged. Optimization by recasting the logic of the definitions may or may not be convenient, and if it results in lack of transparency may defeat the purpose of the CC.

* It was for this reason that the hand coded version of Atlas Autocode, known as AB, was written by one of us (R.A.B.). This includes full syntax checking.

† A recent investigation has shown that reorganization of the internal representation of phrase definitions together with the parsing routine could improve the time for formal syntax analysis by a factor of 2.

Whatever lengths one may go to, however, formal use of the CC will inevitably result in loss of compiling speed, partly because of the formal division between syntax analysis and processing, but mainly perhaps because of the way in which arithmetical expressions are handled (see the remarks made earlier under The parsing algorithm). Some speed is also lost in the processing routines because we chose to make them dynamically relocatable, e.g. by use of relative jumps. (See the account of the "sliding store" in *Trees and Routines*.)

With the CC as with any other programming language one may write good programs and bad programs, and before attempting any detailed optimization one should ensure that the program is conceived on a sound logical basis. One may also write more, or less, ambitious programs depending on how efficiently the object program of the compiler is to run. In most of the compilers listed above the object code is fairly good, but no serious optimization has been attempted. The CC compares favourably with conventional languages such as FORTRAN used for the same purpose since the compilers for these languages do not usually compile particularly efficient code for integer manipulation and logical operations.

We feel it is quicker to produce compilers by using the CC than by hand coding them, although it is not easy to substantiate this. The compilers listed above took between 0·5 and 2 man-years to produce, depending upon the complexity of the language and the degree of optimization built-in. The hand coded compiler for AB took less than one man-year, but it used many ideas developed in AA and also the AA PERM package. (A significant fraction of the time taken in producing a compiler is absorbed by deciding the form of the compiled code and in coding the PERM package.) Coding using the CC is certainly less demanding, and it is more feasible to produce compilers as a part-time activity, and to use lower grade staff to do this. In this case considerable debugging assistance can be given during short and infrequent periods of supervision. Finally we would stress that the CC is particularly valuable when the syntax is complicated.

Other applications

The CC has also been used in some data processing applications, one of the first being the interpretation of the Atlas logging data.* The Atlas produces on a special punch at the completion of every job a great deal of logging information about the job. For example, time on and time off, a job title, a code to indicate from which department, university or firm the job originated, how much time was spent on input, compiling, computing and output, how much store was used during compiling and running. Further information is punched every time the machine detects a fault and restarts itself, and during the engineers' test programs.

The tape was processed each day by a CC program which, using phrase structure definitions of the logging information, read it and converted it to a canonical form on magnetic tape, for subsequent processing by an Atlas Autocode program which produced daily and monthly statements for each department, and other miscellaneous statistics.

Acknowledgements

We are indebted to the writers of the compilers concerned for many of the figures given in Tables 1 and 2.

* A special compiler LOG was written for this purpose by S. Moore (now in the Economics Department University of Manchester).

References

BROOKER, R. A., MACCALLUM, I. R., MORRIS, D., and ROHL, J. S. (1963). "The Compiler Compiler", *Annual Review in Automatic Programming*, Vol. 3, London: Pergamon.

BROOKER, R. A., MORRIS, D., and ROHL, J. S. (1962). "Trees and Routines", *The Computer Journal*, Vol. 5, p. 33.

BROOKER, R. A., ROHL, J. S., and CLARK, S. R. (1966). "The Main Features of Atlas Autocode", *The Computer Journal*, Vol. 8, p. 303.

NAPPER, R. B. E. (1964). "A System for defining language and writing programs in 'Natural English' ", *Formal Language Description Languages for Computer Programming* (Ed. Steel), Amsterdam: North Holland.

NAPPER, R. B. E. (1966). *An Introduction to the Compiler Compiler*, Manchester University (unpublished).

ROSEN, S. (1964). "A Compiler-Building System Developed by Brooker and Morris", *Comm. A.C.M.*, Vol. 7, No. 7, p. 403.