Compiler Compiler facilities in Atlas Autocode

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

This paper describes how the essential phrase structure facilities of the Compiler Compiler have been added to Atlas Autocode. Although the description is in terms of Atlas Autocode they could just as easily be inserted in other languages.

As we have demonstrated in another paper in this issue (see page 345), the Compiler Compiler has proven itself to be a very useful tool in writing compilers. As we came to use it for more general data-processing tasks, the lack of any comprehensive arithmetic facilities became a limitation. Rather than add these to the Compiler Compiler, we chose to add the phrase structure features of the Compiler Compiler to an algebraic compiler.

This is Atlas Autocode (AA), which has been described elsewhere (Brooker, Rohl and Clark, 1966), but which can be thought of as a dialect of ALGOL. So far we have implemented a minimum of facilities, and we may be forced to add others in the light of experience. It is the broad outlines, however, which are of interest. In this description we assume a knowledge of the concepts discussed in "Experience with the Compiler Compiler" (see page 345).

Phrases, format classes and formats

PHRASES, FORMAT CLASSES and FORMATS can be regarded as declarations which instead of reserving space as the type declarations do, cause dictionaries to be created or added to, and packed up within the object program. For PHRASE names we use ordinary AA identifiers (instead of enclosing them in square brackets), and basic symbols are enclosed in quotes. Concatenation is indicated by a period. Consider the phrases for "digit" and "integer" in Compiler Compiler format

PHRASE [INTEGER] = [DIGIT][INTEGER],
[DIGIT]

PHRASE [DIGIT] = 1, 2, 3, 4, 5, 6, 7, 8, 9

In Atlas Autocode these appear

phrase *integer* = *digit* . *integer*, *digit*

phrase *digit* = '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'

The phrase identifiers in FORMAT CLASSES and FORMATS are also replaced by AA identifiers so that we might have, for example:

format class *ss*

ss **format** *variable* . '=' . *expr*

Thus the syntax can be easily defined in terms of **phrase**, **format class** and **format** statements. There is, of course, no built-in master routine to control the reading, recognition and processing of data. A compiler written using these features will contain **phrase** and **format** definitions (with the usual type declarations, routine specifications and routines themselves) followed by a very short piece of program which will read in a line (or, if translating ALGOL, a whole program), pre-edit it if necessary, call the analysis routine and, if analysis is successful, call the processing routine to enter the correct format routine (see below). **Fig. 1** gives an outline of such a compiler.

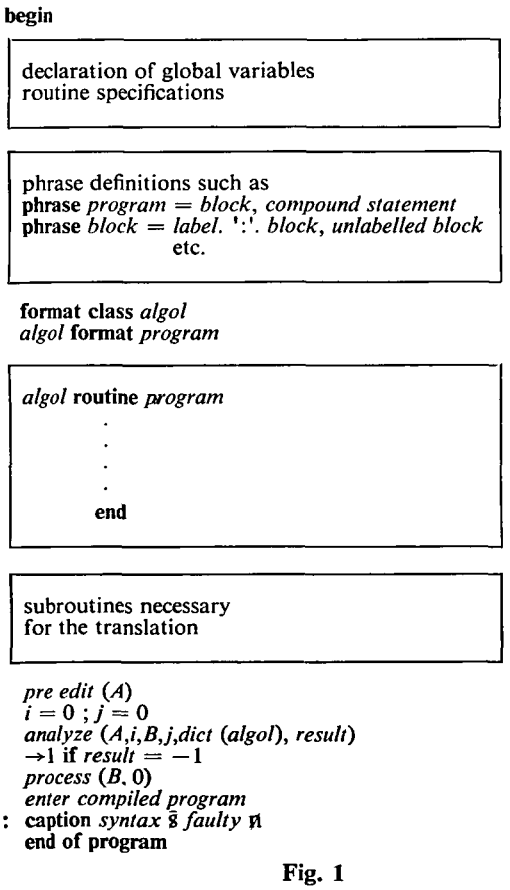

Fig. 1

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

Permanent routines

Two routines are provided within the AA PERM material.

(i) **routine** *analyze* (**integer arrayname** *A*, **c**
integername *i* **integer arrayname** *B*, **integername** *j*, **c**
integer *dictionary*, **integername** *result*)

This routine analyzes (i.e. parses) the pre-edited source string in *A* starting at *A*(*i*) with respect to the dictionary whose address is given by *dictionary* to produce an analysis record in *B* starting at *B*(*j*). Both *i* and *j* are advanced over the recognized string and analysis record, respectively, and *result* is set to 1 if recognition has taken place, to −1 otherwise. Since addresses are not explicitly known in AA, the function *dict* is provided to find the address of any phrase or format class.

(ii) **routine** *process* (**integer arrayname** *B*, **integer** *j*)

This routine inspects the analysis record in *B* starting at *B*(*j*) and calls the relevant format routine.

Format routines

A FORMAT ROUTINE has a similar heading to its corresponding format. For example:

ss **routine** *variable* . '=' . *expr*

Within the routine **integer**s are used to store information about **phrase**s. In the above example *variable* and *expr* are automatically declared integers but all others must be declared explicitly. The statements of the routines are normal AA statements to which have been added two extra resolution statements corresponding to, for example:

LET [EXPR] ≡ [PRECEDING ±][TERM] [TERMS]
→ 1 UNLESS [TERMS] ≡ [±][TERM] [TERMS]

and a built-in function *category*.

Let us consider how the routine

ROUTINE [SS] ≡ [VARIABLE] = [EXPR]

presented in "Experience with the Compiler Compiler" would appear in Atlas Autocode.

ss **routine** *variable* . '=' . *expr*
integer *plus or minus*', *term*, *terms*, *plus or minus*
resolve *expr* **into** *plus or minus*' . *term* . *terms*
load term (*plus or minus*', *term*)
2: → 1 **unless** *terms* **resolves into** *plus or minus* . *term* **c**
. *terms*
add term (*plus or minus*, *term*)
→ 2
1: *dump acc in variable* (*variable*)
end

Whereas the Compiler Compiler version used [AS] routines as subroutines, the Atlas Autocode version uses ordinary AA routines such as *load term*. These may, of course, also use the resolution instructions described above.

routine *load term* (**integer** *plus or minus*', *term*)
integer *factor*, *factors*
resolve *term* **into** *factor* . *factors*
load factor (*plus or minus*', *factor*)
2: → 1 **unless** *factors* **resolves into** '/' . *factor* . *factors*
div by factor (*factor*)
→ 2
1: → 3 **unless** *factors* **resolves into** '*' . *factor* . *factors*
mult by factor (*factor*)
→ 2
3: **end**

Implementation

The action of compilers written this way should be fairly clear, but some idea of the implementation may be useful.

PHRASE definitions give rise to dictionaries of exactly the same form as those of the Compiler Compiler, except that the BUT NOT and NIL features do not exist. The dictionaries are packed up within the compiled program and a jump planted around them.

FORMAT CLASS definitions cause an empty dictionary to be created using a chain store.

FORMAT definitions cause the formats to be added to the relevant FORMAT CLASS dictionary. Before the end of the program a statement such as **pack** *ss* **dictionary** is required.

ROUTINE headings are first looked up in the appropriate dictionary to check that a format has already appeared. (At present, a format is inserted if one has not already appeared.) The phrase names in the heading are then treated formally as **integer** declarations.

At run time, the analysis routine produces an analysis record which is exactly the same as that produced by the Compiler Compiler. The *process* routine inspects this analysis record and calls in the relevant format routine, handing on the addresses of the principal subtrees as the parameters. Thus in our example on entry, *program* contains the address of that part (almost the whole) of the analysis record which refers to *program*.

RESOLUTION instructions are much simpler than the corresponding Compiler Compiler formats since resolution can only proceed one level at a time. At compile time, the dictionary corresponding to the left-hand name is scanned to ensure that the right-hand side is an acceptable alternative. For example, the dictionary

corresponding to *terms* is scanned to ensure that *plus or minus . term . terms* is an alternative when compiling the instruction

→ 1 **unless** *terms* **resolves into** *plus or minus . term* **c** *. terms*

At run time, the tree whose address is in *terms* is inspected to see whether it actually conforms to the right-hand-side. If it does the addresses of its principal sub-trees are planted in *plus or minus*, *term* and *terms*; otherwise control passes to the instruction labelled 1.

Conclusion

The system just described uses **integers** to refer to trees and sub-trees. A theoretically more satisfying scheme using more formal concepts of type and reference for this purpose has been described by Brooker and Rohl (1965). The present scheme, however, has the advantage that it can be fairly easily grafted on to any compiler in which **integers** can serve as store addresses.

The facilities have been part of the Atlas Autocode (AA) compiler since April 1966, and have found a wide variety of uses. A compiler for a language called BB, which is a subset of AA, has been written and debugged, and a compiler for a logical design language is at present under development.

They have found further use in more general data processing jobs. The logging program mentioned earlier has been re-written, and a program to process all the applications for places in this department's honours course has been constructed. This program accepts as data (i) the original application of the student to the U.C.C.A., (ii) any updating material such as offers from other universities and interview assessments, and (iii) interrogatory statements about the state of applications, such as

LIST ALL NAMES INTERVIEWED IN ALPHABETICAL ORDER

Two translation programs have also been written. One converts an AA program from its normal mode into upper case delimiter mode (see Brooker, *et al.*, 1967) and at the same time allows the user to change whatever identifiers he chooses throughout the program. The second converts ALGOL procedures into AA routines. This is in an undeveloped state yet but it is hoped shortly to be able to translate completely some 90% of the published algorithms.

References

BROOKER, R. A., MORRIS, D., and ROHL, J. S. (1967). "Experience with the Compiler Compiler", *The Computer Journal*, Vol. 9, p. 345.

BROOKER, R. A., and ROHL, J. S. (1965). "Simply Partitioned Data Structures", *Proceedings of a Machine Intelligence Workshop 1965* (Ed. Michie, to be published by Oliver and Boyd).

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

Book Review

A Syntax-Oriented Translator, by P. Z. Ingerman, 1966; 131 pages. (New York: *Academic Press*, 48s.)

It is difficult to review a book the chief deficiencies of which are announced as such by the author by the end of the third paragraph of his preface. Indeed, the author further disarms us by overstating the case against himself and his book; he has not committed hubris and the wrath of the gods will be withheld so that he may live to assay Olympus yet again.

The title well describes the book, which was originally intended "for the home compiler-writer". When the author concluded that people do not write compilers at home he forgot that they may be required to do so in a provincial computer installation, very often a more difficult task. It is thus sad that the book needs a "patient reader". Would it have been possible to produce a simple cookbook, three times the size and five times the value to the trade? In the present overburdened state of education everywhere we need more instruction manuals.

A reader with enough persistence can certainly use this book to help implement a syntax-oriented translator, and to devise the language and machine descriptions for some cases of interest. The major, all too common, defect of the book is the use of the assignment statement of ALGOL and FORTRAN as the worked example. We hope for an author who will choose the COBOL option "Move Corresponding", preferably from a group of computational fields to a display record; or perhaps the "Preserve", "Restore", or "Task" options of PL/1. These are examples where this type of translator might show advantages over the "operator/operand stacks" method of Dijkstra, and Randell and Russell, which is perfectly adequate for arithmetic expressions and perfectly general if designed properly to be table-driven.

More precisely, given a machine with basic software and an arbitrary language not foreseen by the designers of the software to implement on that machine, then the design of the structure of the object program to model the semantics of the language is far more difficult than the syntax analysis of the language. But a generalized syntax analyzer is a worthwhile convenience.

H. D. BAECKER