The Atlas ALGOL preprocessor for non-standard dialects

By F. R. A. Hopgood and A. G. Bell*

This paper describes the preprocessor written to replace the original input routine of the Atlas ALGOL compiler. The main purpose of this preprocessor is to allow programs punched in a wide range of conventions and codes to be run, without change, on Atlas.

1. Introduction

In the summer of 1964 the Atlas Computer Laboratory accepted delivery of the S.R.C. Atlas and started to provide a service to the Universities, Colleges of Technology and some Government Departments. The service was intended to cater for programs which were either too large or, in some way, too awkward to be run on the customer's own computer. The bulk of the work originally programmed for other computers was written in the three languages Mercury Autocode, FORTRAN and ALGOL.

Mercury Autocode was acceptable on Atlas with virtually no change as a compatible Atlas EMA (Extended Mercury Autocode) compiler had been provided by I.C.T. Ltd. FORTRAN programs tended to require slight modification for, although incorporating many dialects as subsets in the HARTRAN system, the compiler did not accept every dialect. However, almost all FORTRAN programs are punched on cards with standard punching conventions so that changes, when required, tended to be systematic and easily made by card replacement.

The situation for ALGOL was the worst. The two machines for which ALGOL programs were most frequently written were the English Electric KDF9 and the Elliott 803. In each case both the input/output facilities provided and the punched form of the program were completely different from the Atlas ALGOL conventions. The Atlas compiler had a small set of input/output procedures reminiscent of Mercury Autocode. The KDF9 set was more comprehensive including extremely flexible formats for output and also magnetic tape procedures. The Elliott 803 dialect included two non-ALGOL statements **read** and **print**. These statements could have a variable argument list consisting of expressions, strings and procedure calls. Furthermore, whereas the Atlas compiler only accepted I.C.T. 7-hole tape the KDF9 programs were punched on 8-hole tape and the 803 programs on 5-hole tape.

The preprocessor was therefore initially designed to translate these two dialects into a form acceptable to the Atlas compiler. Also, the possibility of readily incorporating additional dialects of ALGOL, if required, was borne in mind.

2. Atlas ALGOL compiler

The Atlas ALGOL compiler provided by I.C.T. Ltd. was written using the Compiler Compiler of Brooker and Morris (1963). The compiler is basically a two-pass one producing reasonably efficient code without spending too much time in compilation. It loads a set of pre-compiled "permanent procedures" as though they had been defined in the block surrounding the actual program.

The compiler accepts programs punched on 7-hole paper tape one character of which is a moveable back-space (BS). Reserved words such as **for** are punched thus:

f o r BS BS BS UL UL UL

or any other order of punching which would produce the same printed layout (where UL produces underlining).

Due to the form of the Compiler Compiler, the input routine of the compiler itself does not interface with the rest of the compiler in terms of ALGOL Basic Symbols but by character positions on the printed sheet. The input routine reconstructs the incoming program a line at a time and then passes on "composite characters" from this buffered line. A more reasonable interface for ALGOL compilers is in terms of the 116 ALGOL Basic Symbols and, before the preprocessor was written, the compiler was modified so that the output of the input routine was ALGOL Basic Symbols which were then converted by an "Interface Table" into the form required by the compiler. This is slightly inefficient, of course, but it is hoped eventually to absorb this Interface Table into the compiler by suitable modification.

A diagram of the compiler is given in **Fig. 1**.

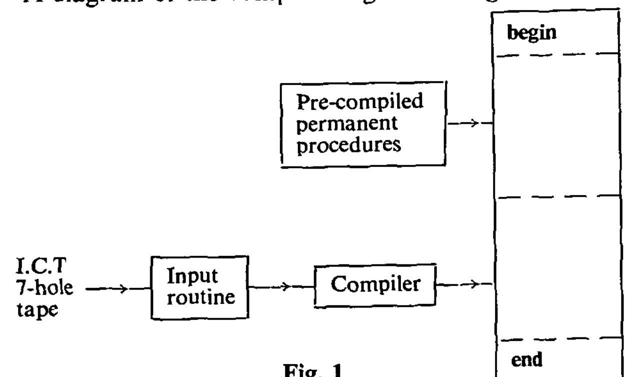

Fig. 1

* *Atlas Computer Laboratory, Science Research Council, Chilton, Didcot, Berks.*

Downloaded from https://academic.oup.com/comjnl/article/9/4/360/390224 by guest on 19 April 2024

3. Strategy

The initial aim was to replace the input routine by a table-driven routine (henceforth called the preprocessor). For each possible dialect a set of tables is provided which defines the type of input and the characteristics of the dialect, e.g. 5-hole paper tape with space dependent symbols as in Elliott 803. To load the appropriate tables a command (Processor Command) is added to the head of the program and this defines which of the four available Permanent Procedures should be inserted around the program. The Processor Command can be described thus:

INPUT ⟨INPUT TYPE⟩ WITH ⟨PERM TYPE⟩ I/O PROCEDURES;

The form of the compiler with preprocessor added can be represented by **Fig. 2.**

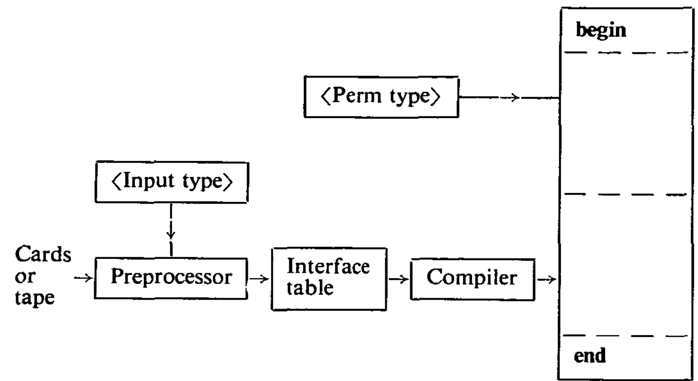

Fig. 2

4. Preprocessor tables

Each dialect of ALGOL translated via the preprocessor has three tables which define the action to be taken on the input stream. These tables are:

(*a*) Initial Input Table.
(*b*) Single Character Conversion Table.
(*c*) Composite Symbol Conversion Table.

The Initial Input Table is used to build up a line of original program in a buffer, marking certain characters to aid in the later stages of conversion to ALGOL symbols. Although in general it is not necessary to build up a complete line before translation can start, it is necessary for the standard 7-hole tape input which requires line reconstruction. At the beginning of the project it was not known whether any other dialects requiring line reconstruction would need to be accepted and so, to generalize the approach, all dialects are buffered a line or card at a time.

The binary value of each character arriving from the input medium is used to look up the relevant position in the Initial Input Table corresponding to this character, and to cause a jump to the address contained therein. The addresses are to small routines which execute the appropriate action on the incoming character and the input buffer.

There are some 30 of these small routines which are designed to perform a variety of functions. Some examples are:

1. NORMAL. Store character in buffer, increment buffer point.
2. IGNORABLE. No action required. Used for redundant spaces, stop codes and erases.
3. CASE SHIFT. The Initial Input Table is divided into several sections corresponding to the possible states of the input stream; a look-up is done relevant to the position of the section pointer. For case-shift characters the action is to change the position of the section pointer to either the upper or lower case section.
4. UNDERLINE. In the preprocessor system this is an extremely important symbol. Underlining of characters is represented by setting a marker bit denoting underline on the corresponding buffer entry.

The Normal Routine will add the current value of this bit to the character, before storage in the buffer. For example, in the KDF9 dialect with non-escape underline, **begin** is punched uniquely as b_e_g_i_n; the underline character enters a small routine which sets the underline bit "on". The normal routine, having stored the next character complete with the underline bit, automatically resets the bit to "off". In a dialect which represents **begin** as 'begin' the initial prime sets the underline bit "on" and the Normal Routine will store all characters, including the bit, until the closing prime resets the underline bit to "off". The action of the Normal Routine is slightly different in these two cases, and this is achieved by setting switches according to the dialect under consideration.

With the set of about 30 routines the mechanism of the initial conversion is very flexible. New dialects can be implemented by defining a new set of Dialect Tables; there is no further coding required.

The ability to have a number of states of the input stream can be used for purposes other than case shifts. Different action on input symbols may be required, for example, inside string quotes or comments. Specifically, the case of the I.C.T. 1900 dialect, which allows **begin** to be punched as 'bEgIn', 'BeGiN' or any other combination of upper and lower case within the primes, was implemented by making the initial prime cause the section pointer to access a part of the Initial Input Table where upper and lower case letters were not differentiated.

Once a line has been reconstructed in the buffer, the composite ALGOL symbols, which occupy two or more locations in the buffer, must be converted to a form the compiler will recognize. In most dialects almost all the composite symbols will have been underlined by the processing described above; hence a test for the next

Downloaded from https://academic.oup.com/comjnl/article/9/4/360/390224 by guest on 19 April 2024

character in the buffer being underlined would be a good criterion for judging whether or not the character could start a composite symbol. This would, of course, be quite efficient as non-underlined characters could be assumed to be single characters and converted immediately. However, all dialects have exceptions to this rule. For example some dialects denote the exponentiation sign by ** and the majority represent := (becomes) by the two symbols : and =. Therefore the Initial Input Table was also used to force an underline marker on every symbol which could possibly start a composite symbol. In the cases given these would appear as ** and := in the buffer. Similarly, in a space dependent dialect, such as Elliott 803, where ALGOL symbols such as **begin** are reserved and delimited by spaces or other non-alphameric characters, then the initial letter of an alphabetic string would have an underline forced on it, **begin real** appearing in the buffer as BEGINREAL.

5. Line translation

The recognition process, when analyzing the buffered line, can be described thus:

1. If character is not underlined then convert via Single Character Conversion Table; continue.
2. If next character is underlined then compare characters in the buffer against entries in the Composite Symbol Table to see if a match can be found. If a match is found convert and proceed, otherwise convert the first symbol using Single Character Conversion Table; continue.

Both the Composite Symbol and Single Character Conversion Tables used were, in fact, Computed Entry Tables (Hopgood, 1966) to ensure that look-up times were kept to a minimum in the space available. By careful choice of the method used for computing the table entry point and the ordering of the symbols in the table it was possible to get the average number of attempts required to identify a symbol down to about 1·2. Entries in the single character conversion table are, of course, unique.

The converted forms of the ALGOL symbols are then passed to the computer itself via the interface table. The Composite Symbol and Single Character Tables were produced initially by a process which amounted to running the preprocessor backwards. The physical punched form of the ALGOL basic symbols, in the new dialect, was read in a certain order, the entries calculated, and the binary form of the characters entered in the table. If no equipment was available to punch the information then the tables were calculated by hand.

Space dependent dialects require some modification of the Line Translation Algorithm described above. Consider in such a dialect the identifier BEGINA. In this case after recognition of the Composite Symbol **begin** is achieved, it is then necessary to examine the following character to see if it is alphameric. If it is then the recognition is not allowed. The most convenient solution is to have an alphameric marker, in addition to the underline marker, added to all alphameric characters, apart from the first, of an alphameric string. Again, this is readily accomplished by the Initial Input Table.

6. Illegal ALGOL statements in the dialect

Most dialects tend to have a certain amount of illegal ALGOL statements or systems information included with a program. For example, titles or comments are required before the actual start of the program. In general these can be removed by fairly simple procedures—in the case of titles by setting a pseudo comment state and treating the title terminator as a semicolon. However, the Elliott dialects have **read** and **print** statements which can only be translated by source to source transformations. The items in a **print** statement can be either expressions to be output, text to be output or calls of setting procedures which locally alter the output parameter settings. An example of the Elliott print statement is:

print *A*, *B*, *punch* (2), '*C*=', *C*;

which would print the contents of *A* and *B* on the currently selected output and then, switching to punch 2, the text *C*= followed by the contents of *C*. The statement is translated by the preprocessor which switches to a special state upon encountering **print** and produces as output:

```
begin
  start print elliott;
  print elliott (A);
  print elliott (B);
  punch (2);
  print text elliott ('C=');
  print elliott (C);
  terminate print elliott
end;
```

The names of the procedures are verbose to avoid clashes with identifiers used in the program. The set of statements has to be in the form of a compound statement to retain the correct ALGOL form of the program. The "start" and "terminate" procedures are inserted for storing and re-storing the global values of the setting parameters. The Elliott **read** statement is dealt with in a similar way.

7. Dialects available

The system came into use in a pilot form in the spring of 1965 and has been gradually extended until at present (September 1966) programs written in the following dialects can be accepted and executed in a single run without alteration to the program tape or card deck:

1. I.C.T. 7-hole tape.
2. KDF9 8-hole tape.

Downloaded from https://academic.oup.com/comjnl/article/9/4/360/390224 by guest on 19 April 2024

3. Elliott 5-hole tape (803).
4. Elliott 8-hole tape (503).
5. Atlas cards (a space dependent dialect).
6. I.C.T. 7-hole tape (with upper case delimiters).
7. Elliott cards (basically the Alcor dialect with Elliott **read** and **print** statements).
8. Elliott 8-hole tape (4100).
9. I.C.T. 8-hole tape (1900).

The ease in which additional dialects can be added is exemplified by the fact that the Elliott 410 form was available for use within two days of receiving its definition and upper case delimiters was a record of only 30 minutes.

The dialects allowed above have different input/output procedures and it is necessary, at present, to provide in the system four different sets of permanent procedures. These are:

1. I.C.T.
2. KDF9
3. Elliott
4. Graph

ALGOL programs written for other computers tend to use the input/output package defined for the particular machine so that a strong link exists between the dialect and the set of permanent procedures used. This could have been built into the system. However, it was thought that the flexibility of allowing any of the four sets of permanent procedures with any of the dialects was desirable, and this was done. Any user, therefore, has a choice of input/output procedures to use.

The first three sets of permanent procedures are described in the manufacturers' manuals. The fourth set, Graph, is an extension of the I.C.T. set so that, as well as normal printed output, graphical output on the Laboratory's Benson Lehner plotter is available automatically.

8. Additional facilities

Once the preprocessor was working in the form described, it was necessary to add editing and library facilities to help users to manipulate and alter their programs with the equipment available at the Laboratory.

8.1. Program listing

The facility of listing the program is necessary to editing. It was decided that the listing of a program should be in a standard form independent of the original layout submitted by the programmer. This has two advantages. One, the layout of most programs is not good, especially during the debugging phase, and programmers welcomed a listing with all the indenting set correctly. Secondly, as the listing algorithm produced a unique layout (inserting new lines and spaces at well-defined positions) there was no need to store layout information with library and editing text.

A listing of the program is obtained on the line-printers. Lines can be numbered for editing purposes. Alternatively it can be punched out on either I.C.T. 7-hole tape or Elliott cards as a conversion mechanism. Both these forms are re-entrant. Upper and lower case letters are differentiated on the cards.

8.2. Editing

Commands such as:

delete 10
replace 5 **by** $a := D$;
after 7 **insert** $b := 0$;

are allowed for correcting individual lines.

8.3. Library facilities

Ideally, programmers should be allowed to define and call library items from magnetic tapes as well as to edit, compile and execute their programs, all in a single run. A library item consists of a piece of ALGOL text and is stored in a packed form of ALGOL symbols on a magnetic tape. Definition of a library item is produced by enclosing the piece of ALGOL text in the directives **commence** ⟨library name⟩; and **define**. Insertion of a library item into a program is by the directive **library** ⟨library name⟩; at the required point.

A standard library tape is available containing the set of algorithms defined in the *Communications of the A.C.M.* together with code procedures for system tapes. The user may define his own library tape, and this provides a convenient way of storing uncompiled programs. Having done this then the user may, in later runs, call this library item, insert corrections using any of the accepted dialects, redefine the result as either the same or a different library item, and execute the program. The original physical program is not required again. At each debugging run only the corrections need be input. When the program is fully debugged a copy can be obtained by the listing facilities or, alternatively, the program can be compiled and stored on magnetic tape in binary form. Production runs can then be made without requiring recompilation.

9. Conclusion

It has been shown that in a period of $1\frac{1}{2}$ man years it is possible to produce a system capable of accepting nearly all ALGOL programs currently in use in the British Isles. This has been achieved with virtually no alteration to the compiler itself. With a little extra effort (about 8 man months) it has further been possible to provide a good system surrounding the compiler.

The authors would like to thank J. Clegg and other members of the staff of I.C.T. Ltd. at Manchester who provided the assistance necessary to ease the interface problems. More details of the system are given in the manuals of reference to the system (Hopgood and Bell, 1966).

(*see overleaf for references*)

Downloaded from https://academic.oup.com/comjnl/article/9/4/360/390224 by guest on 19 April 2024

References

BROOKER, R. A., MACCALLUM, I. R., MORRIS, D., and ROHL, J. S. (1963). "The Compiler Compiler", *Annual Review in Automatic Programming*, Vol. III, p. 76.

HOPGOOD, F. R. A. (1966). "Hash Tables" from *A Series of Lectures on Systems Programming*, H.M.S.O. (to be published).

HOPGOOD, F. R. A., and BELL, A. G. (1965). "Atlas Algol Processor for Non-Standard Algol Programs, Atlas Computer Laboratory, Algol Paper No. 7.

HOPGOOD, F. R. A., and BELL, A. G. (1966). "The Listing, Editing and Library Facilities of the Atlas Algol Compiler", Atlas Computer Laboratory, Algol Paper No. 11.

Book Review

Readings in Automatic Language Processing, edited by David G. Hays, 1966, 202 pages. (Barking: *Elsevier Publishing Co. Ltd.*, 80s.)

"The literature in computational linguistics is not yet sufficiently stable to meet the needs of teachers and students. Some of the best work has been presented only in semi-published research reports. No text-book is in print and the collections of papers available as books are mostly conference proceedings and too advanced for the student who needs an introduction to the field.

"The selections in this book . . . are generally comprehensible to the newcomer. . . . They cover the field as it exists today, taking a fairly broad view of computational linguistics.

"I have included these papers because they epitomize, in their various ways, methods, solutions to central problems, or approaches to the use of the computer as a processor of natural language. Other papers will undoubtedly refine, or perhaps supersede, the concepts that the reader can learn from these pages. But I have tried to choose papers in which sound concepts were developed with enough richness of detail to let the reader see how it all works."

These quotations from the dust jacket and from the editor's preface clearly indicate the purpose of this book. It is a very worthy purpose and it is amply fulfilled. In his introductory essay, Dr. Hays places the work of "Computational Linguistics" in the setting of the progress of information processing as a whole. He surveys the various branches of the subject, many of them potentially of great practical value, and in doing so identifies the basic ideas which are developed in the collected papers. It is as well to list these papers here:

(1) Introduction: David G. Hays
(2) Specification Languages for Mechanical Languages and their Processors—A Baker's Dozen: Saul Gorn (1961)
(3) Natural Language in Computer Form: Martin Kay and Theodore W. Ziehe (1965)
(4) A High-Speed Large-Capacity Dictionary System: Sydney M. Lamb and William H. Jacobsen, Jr. (1961)
(5) Parsing: David G. Hays (1966)
(6) The Predictive Analyser: Susumo Kuno (1965)
(7) Connectability Calculations, Syntactic Functions, and Russian Syntax: David G. Hays (1964)
(8) The Grammar of Specifiers: David A. Dinneen (1962)
(9) Research Methodology for Machine Translation: H. P. Edmundson and David G. Hays (1958)
(10) On the Mechanization of Syntactic Analysis: Sydney M. Lamb (1961)
(11) Keyword-in-Context Index for Technical Literature: H. P. Luhn (1959)
(12) Automatic Phrase Matching: Gerard Salton (1965)
(13) A framework for Syntactic Translation: Victor H. Yngve (1957)

Gorn's contribution (2) serves to introduce the problem of the description of languages. It sketches a number of techniques for language specification, from natural language to Turing machines, by way of Backus Normal Form and incidence matrices. The bibliography leads into the world of Chomsky and formal grammars and to the theory of automata. Kay and Ziehe (3) give a very readable account of a system for encoding a natural language text in a computer prior to processing. The difficulties of creating a standard internal code are discussed in terms which will be readily appreciated, particularly by those concerned with making processors for programming languages.

A full and easily accessible dictionary is an essential part of a language processing system. In existing computers, however, there is a danger of a dictionary becoming too unwieldy in terms of space requirements and access times. Lamb and Jacobsen (4) give an account of the extremely ingenious techniques they have developed to achieve both high economy of storage and reasonable operating speeds. The papers 5 to 8, which deal with the central problems of translations require concentrated attention, though they should not be beyond the reach of anyone with a feel for languages and experience of programming. On first reading, the non-specialist will probably benefit by skipping parsing, predictive analysers, connectability tests, and the grammar of specifiers, and passing direct to Edmundson and Hays (9). This paper, originally published in 1958, was the first of the *RAND* series on machine translation, and it is remarkable that eight years later it is still valid. Another paper, even older (1957) but still fresh, is that by Yngve (13). This marks the step forward from satisfaction with word-for-word translation to the concept of full translation "based on a thorough understanding of linguistic structures, their equivalences, and meanings".

Specialist reviewers have written about these papers separately in the specialist journals. Here we are concerned with the collection as a whole. Clearly the editor is an authority in his field, with long and creative experience behind him. He has displayed his subject in a very simple and effective way by choosing a dozen papers which span the field and which stimulate the lay reader's interest. Most of us find it increasingly difficult to escape from ever narrowing specialization. A book like this enables us to appreciate, if only to a limited degree, what some of our fellow-specialists have been doing. Is it too much to hope that this volume will start a fashion and we may see equally digestible anthologies on, for example, simulation, computer design, . . . even, perhaps, programming languages?

F. G. DUNCAN

Downloaded from https://academic.oup.com/comjnl/article/9/4/360/390224 by guest on 19 April 2024