The Atlas compiler system

By D. Morris and J. S. Rohl*

This paper describes those features of the Atlas system which comprise the supervisor/compiler interface, with particular reference to how an Atlas Autocode user sees them.

A design aim of the Atlas system was to produce a wide choice of programming languages which implies, of course, a number of compilers. Further it was intended that the majority of these would be developed by jobs run within the normal system. Therefore, the supervisor contains facilities for:

(1) specifying the compiler to be used at the start of a job
(2) calling compilers into store within a job
(3) replacing the system copy of a compiler by a modified version.

The system runs with a permanently mounted *supervisor tape* which contains the supervisor and the compilers. Whenever the supervisor is restarted, it (or more strictly the main part of it) is transferred to main store in which it operates. The compilers are transferred to main store as required and are generally lost when compilation ends. A facility known as *batch processing* exists, by means of which selected compilers could be retained for subsequent jobs, but the large storage demands of user jobs has precluded its use.

A job specifies the required compiler by the statement

COMPILER (name)

in its job description. Its position on the supervisor tape is then found by looking up the entry for that name in the *compiler directory*, within the supervisor. This directory contains information such as

(1) compiler name
(2) internal number
(3) address on tape
(4) maximum size.

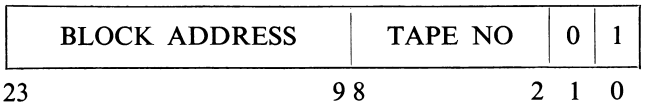
All compilers on tape are preceded by a *title block* which gives its size in blocks and the main store address to be associated with each block transferred to store. Normally these will be consecutive but a certain amount of scatter can be accommodated. The address assigned to the first block of a compiler is taken as its *standard entry point*.

It quickly became apparent that frequent modification of the supervisor tape created maintenance problems. Therefore the instructions described below for calling and redefining compilers were extended to function on non-listed compilers which are stored on private (user) magnetic tapes. Only the *standard compilers* are now stored on the supervisor tape.

Atlas instructions in which the most significant function digit is a 1 cause entry to a subroutine specified by the rest of the function digits. They are called *extracodes* and both the *call compiler* and *define compiler* instructions are of this type. The input parameters to the subroutines are the first modifier (Ba) and the result of modifying the address part of the instruction by the second modifier (N).

Call compiler extracode

This routine operates in two modes. If the bottom digit (0) of N is zero then digits 2 → 23 define the internal number of the compiler to be called. Its address on the supervisor tape is found by looking up N in the compiler directory, and the compiler is transferred into store. If digit 0 of N is a 1 then N is decoded thus:



and the compiler is read from the specified block on the specified tape. In both cases if Ba is zero, control is transferred to the standard entry point of the compiler, but if Ba is non zero control is transferred to Ba.

Define compiler extracode

Ba for this extracode is the logical number of the magnetic tape on which the compiler is to be recorded. The supervisor tape is denoted by 127. N is the address of the first of five 24-bit words describing the compiler. The first of these contains the first four characters of the compiler name, which may extend into the second. Within each half word if there are less than four characters they should be right-justified and filled out with zeros. The address of the compiler on tape will be found by looking up this name in the compiler directory. Providing tape 127 is not specified, the address can be given instead of the name. In this case the first half word must be zero and the second should contain the compiler address. The remaining three 24-bit words are

(1) address of the compiler in store
(2) address of the end of the compiler in store
(3) address where the compiler is to operate.

This last address will also be regarded as the compiler

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

standard entry point. If the compiler address on tape is specified by name its size will be checked against the maximum size listed in the supervisor's compiler directory.

Compiler Compiler facilities

Since most of the compilers on Atlas were written using the Compiler Compiler, then it was natural that these extracodes be formally embedded in it. The master phrase DEFINE COMPILER was introduced for replacing standard compilers. The action of the associated routine is best explained by means of an example:

DEFINE COMPILER AA
ATLAS AUTOCODE

The define compiler routine first remembers the name of the compiler being defined, here AA. It then reads in the following line of text, adds the current date to it and stores the resulting record within the compiler for use as described below. It then sets up the appropriate parameters and obeys the define compiler extracode.

Whenever the compiler is subsequently used, the initial entry routine will always print out the record described above. Thus from the output we are able to tell which compiler was used to run any program, a useful piece of knowledge in the period immediately following the commissioning of a new compiler.

Because of the large amount of input involved in writing compilers they are defined in a number of steps. This was encouraged by two of the characteristics of the Compiler Compiler system. Firstly, the Compiler Compiler accepts the definitions of a language (both syntax and semantics) and the total becomes a compiler for this language. At the end this compiler can accept further definitions to expand itself. Secondly, the Compiler Compiler uses a sliding store principle. Each item (phrase definition, routine, etc) is intrinsically relocatable, i.e. all addresses are relative. When a second copy of an item is presented to the Compiler Compiler, it is compiled on to the end, then the relevant part of the compiler is moved down over the space occupied by the first copy. Thus an obvious way of developing a compiler is to define it initially in a number of steps, then as tests indicate faults in the compiler, to produce correction tapes from which updated compilers can be created. This process can be continued until the compiler is "bug free". The same principle can also be used when modifications are being made to increase efficiency.

Since the supervisor tape contains only the standard compilers, all development is done on private magnetic tapes. As mentioned earlier this requires the ADDRESS on a magnetic tape to which the compiler is to be defined. Since numbers are more susceptible to punching and tape reader errors, with consequent overwriting of valuable information on other parts of the tape, a simple scheme was devised to enable users to give compilers defined in this way (called SPECIAL COMPILERS) a name rather than a number. The tape to be used must be defined in the job description, and given the logical number 1.

DEFINE SPECIAL COMPILER AA1
COMPILER AA1

has the following effect. Firstly, the date is added to the string "COMPILER AA1" and the resulting record stored in the compiler for printing out on all subsequent entries. Secondly, block 1 of magnetic tape number 1 is read into the store. The user will have previously written a directory of names and addresses of all his compilers on this block. This directory consists of pairs of 24-bit words, the first being the first four characters of the compiler name (left-justified and filled out with zeros if necessary), the second the block address of the compiler on tape 1 coded as for the call compiler extracode. These pairs are preceded by the number of compilers on the tape. This table is searched for the name AA1 and if found, the compiler is defined on this tape. As a further aid to the elimination of confusion due to accidental overwriting the name of the compiler is printed out thus:

COMPILER AA1 DEFINED

If the compiler name does not appear in the directory, then a fault is monitored:

COMPILER AA1 NOT AVAILABLE

Most of the Atlas compilers were developed this way using the "father and son" principle. The first tape used the CC to define a special compiler say AA1. Then AA1 was used to define say AA2, AA2 to define AA3 and so on.

Compiler Special

As mentioned earlier these special compilers are not on the Supervisor Tape but on private tapes, and a new compiler, COMPILER SPECIAL, was written to access them indirectly. We describe its action with respect to the following statement in the job description:

COMPILER SPECIAL
AA1

Compiler Special, reads in the symbols on the next line, ignoring set and shift characters and all characters after the first 4, reads down block 1 of magnetic tape number 1, which contains the same directory as was used to define the compiler, and looks for the name in this directory. From the address in the directory, it can call the special compiler AA1 by means of the call compiler extracode.

Atlas Autocode facilities

Atlas Autocode (AA) is an ALGOL-like language which has been described elsewhere (Brooker, Rohl and Clark, 1966) but those characteristics which are impor-

tant for this discussion can be described with reference to **Fig. 1.**

(1) The appearance of a statement on a new line terminates the previous one.

(2) There is a simplified block structure in which the *statement* **begin** is used to introduce a new level of identifiers and labels, and **end** to revert to the old level. In this context we speak of "textual levels". The main program is at level 1, its subroutines at level 2 and so on. The main program is itself a block in which the **end** is replaced by **end of program.**

(3) A *routine* has associated with it a *routine spec*, the action of which will be described with reference to Fig. 1.

Just as the statement **integer** *n* declares the name *n* at textual level 1 to be an integer, so **routine spec** *sort* (**arrayname** *A*, **integer** *n*) declares the name *sort* (at level 1) to be the name of a routine with 2 parameters of type **arrayname** and **integer** respectively. (The names *A* and *n* are ignored at this point.)

The routine heading initiates a new level and declares *A* and *n* to be an **array** and an **integer** (at level 2) respectively.

```
begin
integer n
routine spec sort (arrayname A, integer n)
read (n)
array x(1 : n)
read array (x)
sort (x,n)
print array (x,2,3)
    routine sort (arrayname A, integer n)
    integer i, j, switch
    real x
    cycle i = 1,1,n
    switch = 0
    cycle j = 1,1,n−1
    if A(j) ⩾ A(j+1) then → 1
    x = A(j)
    A(j) = A(j+1)
    A(j+1) = x
    switch = 1
1:  repeat
    → 2 if switch = 0
    repeat
2:  end
end of program
```

Fig. 1. An AA program to sort a sequence of numbers into descending order

To return, one of the design criteria for Atlas Autocode was that the routines permanently available to all users such as input/output routines, matrix routines and differential equation routines (called the PERM) should be written in AA. (They use of course the AA machine code formats for efficiency.) The main program and all the PERM routines are considered as being embedded in a textual level 0 in which the *routine specs* for the PERM routines have also been declared.

When a tape containing the PERM routines and specs is presented to the compiler it is translated in the same way as a program. This produces as well as translated program a series of lists which are required to accomplish the translation not only of the PERM but also of all source programs. These lists include a constant list, a name list, a property list and a routine directory and are spaced at wide intervals along the store. There are also a number of pointers associated with these lists which have to be preserved. The DEFINE COMPILER mechanism as it stands cannot handle this situation since it requires the compiler to occupy consecutive blocks in the store.

It was convenient to introduce the AA statements

define compiler

define special compiler

rather than modify the Compiler Compiler master phrases to which they closely correspond. These store pointers within the compiler, add the translated program and all the lists to the compiler, and then define the whole as a compiler using precisely the same mechanism as the Compiler Compiler master phrases.

Originally the PERM and its lists were *copied* on to the end of the compiler and any previous copy deleted. The Initial Entry Routine was modified to copy out this material, reset the pointers and then to print out the compiler title as before. This had the advantage that the compiler itself was left undisturbed and so could be used for batch processing, but the store requirements of every program were enlarged by the two copies of the PERM.

Subsequently the **define compiler** and **define special compiler** statements and the Initial Entry Routine were modified so that instead of copying information from block *A* to block *B*, the block *A* itself was *renamed* to be block *B*, taking advantage of the Atlas paging system. This meant that with only one copy of PERM in existence at any time, the store requirements were minimized. On the other hand it prevented batch processing.

Using these facilities PERM was added to the basic AA compiler (again using the "father and son" technique), in 3 stages.

User features

After AA had been in use for some time, there were numerous requests for the PERM to be expanded. The extent of the current PERM is an indication of these requests, though many routines suggested were not of sufficiently wide application to be included. Users may, however, starting with the AA compiler on the Supervisor, expand the PERM in any way they wish, to form their own special compiler (on their own magnetic tape). An obvious example is a special AA compiler which contains a large general purpose magnetic tape routine to enable a number of users to store data on the rest of the tape with a minimum chance that one user will overwrite that of another.

The usefulness of this facility was considerably increased by allowing **define special compiler** to appear anywhere at level 1. The effect is to add to the compiler all the translated program preceding the **define special compiler** together with an updated copy of the lists.

Ideally, it should also cause label lists and so on to be packed on the end of the other lists at the end of the compiler. However, by imposing two conditions on the position of this statement:

(1) no explicit jumps across this point

(2) no **cycle** and **repeats** split across the point

then only a minor alteration to the **define special compiler** routine was required. This modification caused the labels to be filled in and **cycle-repeat** pairings to be checked before performing the defining of the compiler described above.*

A program to call this compiler consists only of that part of the original program which came after the point at which **define special compiler** was inserted. In general this consists of a series of routine calls (followed by **end of program** and data, if any). That is, it is rather like a steering tape. This is a useful facility for debugging programs, especially as any incorrect routine in the compiler can be replaced by a corrected version on this steering tape. This has proved a valuable feature since the tapes for each run can be kept very short, a couple of feet or so. Many programs spend all their working life in this state.

When a program has been completely debugged and is in production, a special compiler containing everything up to **end of program** can be defined; programs using this compiler contain only **end of program** and the data. This is unsatisfactory for a number of reasons. First, since this special compiler contains the AA compiler and all the lists required for compilation it is unnecessarily long. Secondly, it is aesthetically unsatisfactory to have to start each program with **end of program.** Consequently a statement **end of compiler** was introduced. This does all the tasks, such as filling in labels, that are associated with **end of program** (which it replaces) and then defines the program as a compiler. The compiler consists of the program plus PERM, and the two lists necessary at run time—the constant list and the routine directory. These are packed on the end of the program as is a sequence (the entry point of the compiler) to unpack these into their correct position during running. Although some of these compilers are compilers in the conventional sense, they are more often just a convenient way of storing a translated program on magnetic tape and accessing it as if it were a compiler.

A program to use this compiler consists merely of the data of the original program.

* These restrictions could easily have been lifted but there has been no call for it as control can easily be transferred across the point by routine calls.

Conclusions

We have been concerned with making provision within our system for keeping the length of large programs as small as possible, consistent with ease of debugging and the efficiency of the whole system. In this light, the facilities outlined above have proved quite valuable. Many uses suggest themselves; for example, an AA compiler containing a list processing package, or an AA compiler containing grading programs, and so on. However, we have regretted the unnecessary restrictions which were programmed into the define compiler extracode requiring the compiler to occupy a contiguous set of blocks and not to use block 0. Also the restriction that the standard entry point of a compiler be the first word of the first block has occasionally been inconvenient.

Reference

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