R E Thomas

UNIVERSITY MATHEMATICAL LABORATORY
CAMBRIDGE

Technical Memorandum No. 65/5

THE PLANNING OF AN AUTOCODE COMPILER

by

H.P.F. Swinnerton-Dyer

September 1965

THE PLANNING OF AN AUTOCODE COMPILER

by

H.P.F. Swinnerton-Dyer.

Introduction

This memorandum consists of the planning documents which I wrote before starting the detailed programming of the new Titan Autocode compiler; they are given in their original form, even though this is not always a correct description of the compiler as it now is. They have been published here for two reasons - first as a description of one possible way to write a compiler, and second to show the extent to which the structure of a complicated program should be worked out before any actual orders are written. The sections of the memorandum were written in a random order, and there is considerable cross-referencing between them: the reader is advised to read them all superficially before trying to follow the details of any one of them. He is warned that the terminology used here is not the same as that in the Autocode manual. The version of the compiler described here compiles from Autocode into IIT; but in the planning I bore in mind the need to be able, with as little alteration of the compiler as possible, to compile directly into binary in the store.

This memorandum is not of course the complete documentation of the compiler: there are also the detailed specifications of the open and closed subroutines in the compiler, the allocation of B-lines and working space in core store to particular duties and the details of the way in which information is packed into them, and the actual annotated machine-code program. All this, however, describes the detailed implementation of particular facets of the compiler. It is not meant to be read as a whole, but it enables anyone who needs to understand a particular section of a program to do so. The principal beneficiary of this is the original programmer himself - either when he is writing other sections of the program, or when he is debugging - and even if one never expects to have to change a program it is a false economy to skimp on the documentation. (Apart from slips of the pen, the most common type of error in writing a large program arises from misremembering the specifications of other sections of it. This is also the most difficult type of error to find, since no individual section of the program is wrong in itself.)

The first Titan Autocode compiler was written by M.V. Wilkes and others as an example of list-processing technique. As is inevitable with any

list-processing program, it used a great deal of store. Since the main supervisor has been planned in such a way as to run several programs simultaneously, which avoids the computer lying idle during magnetic tape and disc transfers, it was necessary to provide a smaller compiler; in writing the present compiler small size and high compiling speed have been the primary objectives, even at the expense of some lack of optimization in the object program compiled. It is independent of the previous compiler except that the 'Autocode package' has been taken over almost unchanged. (This is the set of closed subroutines in the object program which are used to implement LIBRARY and similar instructions. They are the same, and occupy the same fixed area, for all object programs; they are held in binary form on magnetic tape, and are called down into the store by the first few obeyed orders of the object program.) Because the Autocode package was in existence when I started planning this compiler, its specifications in this memorandum are much more specific than those of the rest of the compiler.

Because of the conditions that it had to meet, the compiler described here was designed and written in as low-brow a manner as possible.** The advantages of such an approach are that the resulting compiler will be both smaller and faster than one written by more sophisticated methods. The disadvantages are

(i) that the compiler is more liable to contain undetected errors, since each branch of it has been written individually and so there is no necessary uniformity of treatment;
(ii) that because the compiler takes advantage of incidental features of the language from which it is compiling, it is very much more tiresome to alter it when the language is modified.

It is generally believed also that a compiler of this sort is more laborious to write than one which uses more sophisticated techniques. However, the

** For some alternative compiling techniques, see

Dijkstra, E.W. ALGOL 60 Translation, Algol Bulletin Supplement No.10 (1961).

Randell, B. and Russell, L.J. Algol 60 Implementation. A.P.I.C. Studies in Data Processing No.5 (1964).

Irons, E.T. The Structure and Use of the Syntax Directed Compiler, Annual Review in Automatic Programming, Vol.3 (1963).

Brooker, R.A., MacCallum, I.R., Morris, D., and Rohl, J.S. The Compiler Compiler, Annual Review in Automatic Programming, Vol.3 (1963).

efficiency of the compiled object program does not depend on the type of compiler, but only on the amount of effort that has gone into writing it; the price of better object program is just a larger and slower compiler.

The character input routine

An Autocode instruction consists of a string of characters, and the routines which read and edit each Autocode instruction must obtain the characters and operate on them one by one. In principle we could simply read each character by a 1054 order, but it would then be necessary to find the type of the character and in some cases (such as shifts) to take special action. Instead, all reading is done through a subroutine: this supplies the next effective character, both in internal-code form and as classified by type and value. It also does certain necessary book-keeping operations, and copes with error messages caused by a wholly illegal character - that is, a character illegal regardless of its context.

The information which the subroutine must hold from one entry to the next consists of three marker digits and the current line number. The markers are as follows:

(i) Current internal-code shift. This is reset whenever a shift character or a newline is read.
(ii) Within text, title or optional section not being compiled. This marker is set and unset outside the subroutine, but is used inside it. It gives a convenient way of dealing with those characters which are legal in text or title but illegal otherwise.
(iii) Current line contains effective characters. This is set by any character which corresponds to a printed symbol in the input document, and is unset by a newline. It is used when updating the current line number.

The line number is held in the form 'line α after label β of block γ ', or 'line α of block γ ' if no label has yet been read in this block. The compiler in fact holds two line numbers, one for the next character to be read and the other for the head of the current instruction. The first of these is advanced whenever a newline is read and the last line contained effective characters; it is reset at the start of a new block or on dealing with a legitimate label. The second is set from the first whenever the compiler starts to read a new instruction.

To each internal-code character correspond two entries in a look-up table. For a normal character these give its type, which is a positive half-integer, and its value, which is an integer or half-integer; for a character which has to be treated specially, the type is replaced by the

address of the first order of the associated open routine, with the sign digit set as a marker. When the subroutine is entered, it will usually use a 1054 order to read the next internal-code character and will then increase this by 8 if it is in outer set; sometimes, however, the character will already have been read, increased and stored, and then it need only be recovered from the store. In either case, the apparent type and value are found from the look-up table. If the type is positive, which happens when the character is normal, set the 'character-in-line' marker and exit. If the type is negative, enter the associated open routine; note that for a newline this routine is entered directly from the 1054 order, without any table look-up. Some characters are abnormal only because the marker digits or line count need special treatment; in such cases the routine does this, sets the type number, and exits. Others are abnormal because they may form part of composite characters; in this case further characters must be read and if necessary (that is, if a composite character cannot be formed) stored for later use. For other abnormal characters the routine does any necessary book-keeping, including possibly putting out an error message, and then returns to the start of the subroutine in order to read a further character from the input stream.

Those abnormal characters which may not produce a normal exit, and the actions taken in the corresponding routines, are listed below. These characters do not have a type number in the normal sense, and the values ascribed to them are never actually used by the compiler.

(i) Characters illegal regardless of context. This includes any character which in theory cannot appear in an input stream, together with 0.3, 3.2, 4.0, 7.6 and 7.7 of inner set and 2.6, 2.7, 3.0, 3.1, 3.2, 3.4 and 7.6 of outer set. The routine puts out an error message; it may or may not set the 'character in line' marker.

(ii) Character illegal except in text or title. These are 0.0 and 1.2 of inner set and 3.6 of outer set. The routine examines the 'in text' marker, and proceeds as in (i) if this is not set. If it is set, ignore 0.0; for the other two characters, set 0.1 as the character read, set the 'in line' marker and exit normally from the subroutine.

(iii) Character wholly ignored. These are 0.6 and 0.7 of inner set and 0.6, 0.7, 1.4 and 7.7 of outer set; note that on the last of these the 'character in line' marker must be set.

(iv) shift characters. These are 0.4 and 0.5 of either set; they cause the 'internal-code shift' marker to be updated.

There are two characters either of which may be the first part of a composite character. In each case we read further characters one by one,

testing each to see if it is the next one needed to form the composite character. If the composite character is formed then it is used; otherwise the initial character is used and the other characters are stored, to be used on later entries:

(v) 'Greater than'. This may be followed by 'equals' to form a composite character which is given a normal type and value.

(vi) 'Arrow'. This may be followed by another arrow and a newline; if so, the line count is increased by one and the composite character treated as a space.

The other characters which have their own special routines are as follows:

(vii) Space and Tab. Reset the type number and exit from the subroutine, without changing the 'character in line' marker.

(viii) Newline. If the 'character in line' marker is set, advance the current line number by one and clear the marker to zero. Set the character read to a negative value (for use in text or title), set the type number and value, and exit from the subroutine.

It remains to classify those characters which produce a normal exit from the subroutine, and to ascribe values to them. Any scheme of classification is a compromise between the demands of speed and of size, and is to some extent chosen on non-logical grounds. There are in this compiler six types, with type numbers 0.0, 0.4, ..., 2.4; the corresponding characters and values are as follows:

(i) Letter, no distinction being made between upper and lower case. The value of any letter is given by the following table:

A to D	0 to 3
E to H	33 to 36
I to T	-31 to -20
U to Z	37 to 42

Thus from the value of a letter one can immediately deduce whether the corresponding variable is index or floating point, and what is the corresponding B-line or register of store.

(ii) Digits. The value of any digit is the digit itself.

(iii) Newline and semicolon. These have values 0 and 1 respectively.

(iv) Space and tab. These have values 0 and 1 respectively; but in fact both here and in the previous case the value is never used.

(v) Characters which can appear inside an arithmetic expression. There are ten such characters, and their values are given by the following table:

Suffix ten	-1	Square ket	1.4
Point	-0.4	Minus	2
Round bra	0	Plus	2.4
Round ket	0.4	Stroke	3
Square bra	1	Asterisk	3.4

(vi) Other characters. There are six of these, with the following values:

Colon	-0.4	Greater than	1
Equals	0	Greater than or equals	1.4
≠ or ~	0.4	Arrow	2

These values have been chosen so as to simplify both table look-up and the discrimination of 'colon' or 'equals' within the type.

As has already appeared from the description above, the character input routine needs a small amount of working space in which it can store those characters which have been read from the input stream but do not yet have to be handed over to the main input routine; and it must also keep a count of these characters. They are stored one to a halfword, with the internal shift already included in them. Because of the presence of this buffer of characters, it is possible to implement a further subroutine, which hands back an internal-code character to the input stream; this is done by writing the character to the buffer, each member of which has to be pushed down a place, and increasing the count. It is useful for making uniform the entry conditions to some of the other input routines - for example, when reading an arithmetic expression one or two of whose characters have already been read before the nature of the expression has been recognized.

Reading and editing Autocode instructions

The routine which reads an Autocode instruction and checks the correctness of its syntax should be thought of as operating in two modes - one for reading arithmetic expressions and the other for reading the rest of the instruction. In practice it is convenient to read arithmetic expressions by means of a closed subroutine with unusually elaborate entry and exit conditions. An arithmetic expression is stored in compiler working space, where it can later be dealt with by the translation routines. Other parts of an Autocode instruction contain only small amounts of information; this is either held in B-lines or used at once to control the operation of the compiler. (But there are exceptions to this, such as the table in a switch instruction.)

There are three important ways in which several characters have to be

combined to form a single symbol; in each case the treatment is different according as the characters are inside an arithmetic expression or not:

(i) A preset word. If the first two characters of an instruction are letters, then the instruction must start with a word; this is read by an open routine which also notes the terminating character, and is identified by comparison with a table. Once the word is found, it identifies the type of instruction. A word in any other position outside an arithmetic expression is predictable; its presence is checked by a closed subroutine, which may or may not be satisfied by a newline instead, according to context. Within an arithmetic expression, a sequence of letters is both treated as a product of variables and also built up as a potential word; on reaching a non-letter, the compiler tests whether a function name has been built up, and if so replaces the product of letters by the function.

(ii) A number. Within an arithmetic expression this is built up by one of three open routines, according as the number is fixed point, floating point, or in an undetermined mode. Elsewhere, the number must be an integer and well within the range of a B-line, so that there is no difficulty in testing for overflow; it is read by a closed subroutine, which has two entries according as the first digit has already been read or not. The subroutine notes the number formed, and also the terminating character; there is an error exit on near-overflow.

(iii) A multicharacter identifier. This is read by one of two essentially similar subroutines, one of which is used for floating point and the other for integer identifiers. Each of these has two entry points, one for when the asterisk or point alone has been read, and the other for when the letter following it has also been read; the first is normally used outside and the second inside an arithmetic expression. The identifier is built up in a pair of halfwords, characters after the eighth being ignored; the terminating character is preserved for use outside the subroutine. When the identifier is complete, it is compared with those already in the corresponding table; if it is not already present it is inserted in the next vacant position. On exit, its position in the table is given. There is an error exit if the identifier is not in the table and there is no room to insert it.

There are other ways in which several internal-code characters may be combined, for example to form '$\geq$', but these are dealt with ad hoc in the character input subroutine.

The following types of error will be detected by the input routine:

(i) Codeword not recognized by the compiler; this can only happen at the beginning of an instruction, since elsewhere the analogous fault will be treated as a character out of context.
(ii) Appearance of ROW, CHAR or INPUT twice in the right-hand side of a calculation instruction, or at all in any other arithmetic expression.
(iii) Character out of context; in this case the character is named in the error message, even though it may not be the one really at fault.
(iv) Mismatched brackets; this may be a round or square ket when the other was expected, or an unpaired bra or ket in an arithmetic expression.
(v) Expression too large for the compiler to handle.
(vi) Too many multicharacter identifiers.

When the routine detects any one of these, it puts out a message which gives the type of error and the line on which it occurs. If the error precedes the first PROGRAM instruction, or is of type (vi), then compiling is abandoned; otherwise the current instruction, including subsequent characters up to the next newline or semicolon, is ignored and the input routine is re-entered to deal with the next instruction.

The part of the input routine which reads an arithmetic expression into the store must also update certain lists and markers which are used elsewhere in the compiler. The following markers are set inside the routine, and apply only to a single expression:

(i) The expression is a single variable, possibly subscripted. This is examined to see that the left-hand side of a calculation instruction is acceptable.
(ii) The function INPUT has appeared. This is examined on reading '→ n IF /' after a calculation instruction.
(iii) INPUT, ROW or CHAR has appeared. This is examined on reading any of these three, and also at the end of any arithmetic expression other than the right-hand side of a calculation instruction; there is an error exit if it is set then.
(iv) First character is a minus. This is used only in dealing with the step in a FOR instruction, and is most conveniently set outside this routine.

There is also a three-valued marker which determines whether the part of the arithmetic expression currently being read is index, floating point or of type as yet undecided. It is initially set before starting to read the arithmetic expression, and its value may be carried over from one expression to the next within a single instruction; it has the value 'type undecided'

only for the argument of a PRINT instruction or for the arithmetic expression involved in a conditional jump or switch. Its value is changed within the routine by the machinery which checks the pairing of brackets: see below. At the end of any arithmetic expression, the routine also updates the list of working-space intervals in which arithmetic expressions are held; this list has bounded length since the left-hand sides of calculation instructions are treated in a different way.

There are also two lists concerned with the bracket structure, the first being meaningful for an entire instruction and the second only within a single expression:

(i) List of arguments of the function INTEGER. These are identified by the positions of the opening and closing brackets. Because of the way the list is formed, if one argument contains another the inner one is the earlier in the list.

(ii) List of bras not yet paired with a ket, and associated information. For this purpose there are three sorts of bracket: square, normal round, and those which enclose the argument of the function INTEGER. Each entry in this list gives the type of the bra and the current 'head of product' information described below; for an INTEGER bra it also gives the store address of the bra and the current value of the 'index/floating point' marker. This list is a pushdown list.

While an arithmetic expression is being read, enough information has to be held to enable the sign of the current product to be changed from minus to plus if possible, by changing the sign of a constant or by using negative multiplication. This information consists of a marker showing whether the sign of the product is plus or minus, and if it is minus what is its store address and whether it is unary or binary; note that a unary plus is never recorded in the store. The two lists described above are altered only when a bra or ket is read. On reading a bra, set the relevant information and the type of bra in list (ii); note that the type of bra is the last thing written, so that it will be the first thing read. For a square bra, change the expression type to 'index'; for an INTEGER bra change it to 'floating point'. In any case reset the 'head of product' information. On reading a ket, read the type of the corresponding bra from list (ii) and check that it agrees; it is only at this point that an INTEGER ket and a normal round ket can be distinguished. If list (ii) is empty, there is an error exit; but note that this is the correct exit when reading the argument of say a PRINT instruction, since in this case the initial bra is not recorded and the final ket acts as a terminator. Now suppose that the ket is correctly paired. In each case the 'head of product' information is recovered; for an INTEGER ket the 'index/floating point' marker is

recovered from the list, and a new entry is made in list (i); for a square ket the marker is reset to 'floating point'. At the end of an arithmetic expression list (ii) should be empty, and there is an error exit if it is not.

The other sorts of error that the input routine must test for are the illegal juxtaposition of two symbols, and the presence of a floating point symbol in an index expression. A symbol is represented by a pair of numbers packed into a halfword, one giving the type and the other the particular symbol within the type. The possible types are as follows:

(i) Index identifier, whether single or multicharacter. The identifying number is that of the corresponding B-line.
(ii) Simple floating point variable identifier. The identifying number is that of the corresponding word of store.
(iii) Subscripted variable identifier. The identifying number is as in (ii); indeed this type is obtained by conversion from (ii) after reading a square bra.
(iv) Floating point constant, identified by its position in the list of these.
(v) Fixed point constant, identified by its position in the list of these.
(vi) Functions, including INPUT, ROW, CHAR and INTEGER.
(vii) Arithmetic signs.
(viii) Brackets.

During compilations, three more types will appear: these are null halfwords and index or floating point working variables. Types (ii), (iii), (iv), most functions and stroke are of floating point type. If any of these occur in an index expression this causes an error exit; if it occurs in an expression of unknown type then the type is forced to 'floating point'. While a constant is being read in an expression of unknown type, it is formed in both modes; if it has neither fractional part nor decimal exponent, it is stored as a fixed point constant. In this way we may have fixed point constants in what subsequently turns out to be a floating point expression; such constants are converted during compiling. In testing for illegal juxtaposition we keep a note of the last symbol, classified on a rather different basis. Here the possibilities are as follows:

(i) Operand, that is, an identifier, a constant, a round or square ket or one of the functions INPUT, ROW or CHAR. If this is followed by an identifier, constant, function or round bra, then the implied multiplication sign must be inserted between the two. It can be followed by a square bra only if it is a floating point identifier; and in this case its type must be changed from simple to subscripted.

(ii) Function having an argument. This must be followed by a round bra.

(iii) Arithmetic sign. This must be followed by an identifier, constant, function or round bra.

(iv) Start of expression or round or square bra. This must be followed by a plus or minus sign, identifier, constant, function or round bra.

Whenever the last character read either is a complete symbol in itself or terminates one, the symbol is checked by these rules; if it is legitimate, it is written to the store and also recorded as the new value for the last symbol read. There is a special difficulty in dealing with a sequence of letters which might compose a word. These are initially treated as simple variables multiplied together; if when the word is terminated it can be identified as a function, then it overwrites them. (The function will certainly be legitimate in its context; but it may be necessary to restore the sign of the product.)

The type of an instruction is determined by its first few characters, more than one being needed only if the first is a letter. The possible non-letters that can begin an instruction, and the corresponding types of instruction, are as follows:

(i) A digit. If no PROGRAM instruction has yet been read, this must be the start of 144/#2; thus the entire line can be ignored. Otherwise read the number which begins with this digit, and store it in a B-line. Ignoring spaces, the following symbol must either be TRACK, which concludes the instruction, or a colon which marks the number as a label and terminates it.

(ii) An arrow. This must introduce a jump or switch, either conditional or unconditional, or a subroutine jump. For the rest of the routine see below.

(iii) A square bra, round bra or round ket. These introduce respectively a comment and the beginning and end of optional compiling, and cause in each case an immediate jump to the associated routine.

(iv) A point or asterisk. This must introduce a calculation instruction.

If the first character is a letter but the second one is not, we are again dealing with a calculation instruction; in fact the second character is bound to be a space or tab, 'equals' or a square bra, but we need not check this now. If the first two characters are both letters, then the instruction starts with a word; the input routine reads this word and identifies the type of instruction from it by comparison with a table.

If the instruction starts with an arrow, read the following symbol; this must be either a number, for an ordinary or a subroutine jump, or the name of an index variable, for a switch. In the former case store the number in a B-line and examine the rest of the line. There are now three possibilities:

(i) The rest of the line is empty; the instruction is an unconditional jump which is now complete and can be handed over for compilation.

(ii) The next symbol is 'IF', introducing a conditional jump. Set the expression type to 'undetermined', and read an arithmetical expression and the symbol which terminates it, which must be a relation; then read a further arithmetic expression, and check that it is terminated by a newline. (Note that for convenience in editing, each expression should be preceded in the store by a spare halfword.) Now edit so that

$\alpha = \beta$ becomes $\alpha - \beta = 0$ $\alpha \neq \beta$ becomes $\alpha - \beta \neq 0$
$\alpha \geq \beta$ becomes $\alpha - \beta \geq 0$ $\alpha > \beta$ becomes $-\alpha + \beta < 0$

if α or β is just the constant zero it should be deleted at this point. The editing process consists of changing all signs outside brackets in that expression whose sign is to be reversed, and re-absorbing minus signs into constants or products as far as possible; note that before doing this we must restore any suppressed unary plus at the beginning. The instruction is now complete and may be handed over for compiling.

(iii) The rest of the line is an arrow; the instruction was a subroutine jump and it is now complete and can be handed over for compiling.

If the instruction is a switch, note the index variable and check that it is followed by a square bra. Then read a sequence of numbers and store them in consecutive halfwords of working space, continuing this as long as they are terminated by colons; the last number will be terminated by a square ket. This concludes the switch part of the instruction; the information it contains consists of the name of the index variable, the count of numbers read and the numbers themselves. Now proceed as for a jump, except that of the alternatives above only the first two are allowed.

To read a calculation instruction, first hand back to the input stream the one or two characters that were read before recognizing it. Now read an arithmetical expression; this is bound to be a single variable, possibly subscripted, and to be terminated with an 'equals' sign. If it is an index variable, change the type to 'index'; and in any case delete the entry in the list of arithmetic expressions. Read another arithmetic expression, and if it is terminated by 'equals' repeat the tests and actions above.

Eventually we shall read an arithmetic expression with some other terminator, and this concludes the calculation part of the instruction. There are two possible additional complications:

(i) An expression of the form '→ n IF /'. This is allowed only if the function INPUT has appeared in the calculation instruction; in this case the number n is recorded.

(ii) One or more expressions of the form *m where $0 \leq m \leq 5$. These are in fact not read until the rest of the instruction has been translated; they are then read one by one and compared with the keys currently set, and if necessary an optional printing section is compiled.

If both these occur, they must be in the order given above.

All other instructions begin with a word, which identifies the type of the instruction. The most complicated to deal with, in that it can contain arbitrary floating point expressions, is the 'FOR' instruction. After the initial word, read a simple or multi-character identifier and check that it is followed by 'equals' after perhaps some spaces. If this is an index identifier, set the arithmetic expression type to 'index'. Read an arithmetical expression and check that it is terminated by a colon. Then read the next character other than a space and note, for use in compiling, whether it is a 'minus'; give it back to the input stream and read a second arithmetic expression, checking that it also is terminated by a colon. Finally read a third arithmetic expression and check that it is terminated by a newline; this completes the instruction.

We regard a LIBRARY instruction as broken up into subtypes by the number that follows; and we also treat in the same way TRAP, UNTRAP, PRINTCHAR and the five magnetic tape instructions. To each of these corresponds a codeword which shows for each argument in turn whether it is a floating point variable, an index expression or a label, and whether it is terminated by a colon or by an unpaired round ket. Check that the initial word of the instruction is terminated by a round bra; then read the arguments one by one, using the appropriate subroutine and checking that the terminator is the correct one. The last argument is that terminated by an unpaired round ket. A similar technique works for PUNCH and PRINT, except that we use an explicit sequence of subroutine calls, and the last argument is terminated by a newline; note that for PRINT we do not know in advance the number of arguments. LAYOUT is also dealt with in this way, except that the key word is terminated by a space.

The other types of instruction are simpler, though they give rise to a considerable variety of possibilities. TEXT and TITLE have been dealt with elsewhere; we verify that the key word is followed by a colon, and then

enter a routine which deals with both input and output simultaneously. The other possibilities are as follows:

(i) EQUATE. Check that this word is terminated by a colon; then read a sequence of multicharacter identifiers, checking that they are all of the same type. The instruction is terminated by reading a newline. In practice it may be convenient to treat the first one specially and process each of the subsequent ones as soon as it is read.

(ii) PROGRAM. This is followed by a number, which may indeed terminate the keyword. After this there is a possibly empty sequence of items, each of which consists of a floating point identifier terminated by a square bra, followed by a number terminated by a square ket. Each of these items can be processed separately. The whole instruction is terminated by a newline.

(iii) RETURN may be a complete instruction, or it may be followed by an unconditional jump or switch. To test which, find the next character, which must be a newline or an arrow; in the latter case hand back the arrow to the input stream and treat the word as a complete instruction.

(iv) KEY or KEYS is followed by a series of digits, each of which is most easily dealt with as soon as it is read.

(v) START is followed by a pair of numbers, the first being terminated by a stroke and the second by a newline.

(vi) READER and OUTPUT are each followed by a single number.

(vii) The other six words are each complete instructions. The only unusual feature about them is that the content of the rest of the line must be ignored, since it can have the status of a comment.

Object program space allocation

The space available to the object program consists of the B-lines, up to B90 inclusive, and the core store. The B-lines are allocated as follows:

B1 - B12 Each of these holds the current value of one of the twelve single character index variables I to T.

B13- B63 These are allocated one by one to the fiftyone permitted multicharacter index variables, in the order of their first appearance. Note that if one of these B-lines is released by an EQUATE instruction, it does not become available again to the compiler.

B64 This holds, in coded form, the line number of the Autocode instruction whose translation is currently being obeyed; its content can be unpacked into the form 'line ℓ after label m of block n'.

B65 - B67	Reserved for special purposes in the Autocode package routine dealing with INPUT. Of these, B67 holds the address to which to jump on reading a stroke; and on an error exit B66 holds the illegal character and B65 the number of previous successful entries.
B68 - B79	General working space for index calculations; and data for entry to library-type routines in the Autocode package.
B80 - B89	Working space for library and other routines in the Autocode package. Note that the functions ROW, CHAR and index INPUT leave their results in B88.
B90	Return jump address for exit from routines in the Autocode package.

Except for B64, which is needed for the monitor routines, these are identical with the allocations in the previous Autocode compiler.

The principle of compatibility has also had some effect on the allocation of space in core store. The bottom end of the store is assigned in an absolutely fixed manner:

0 - 3	Floating point variables A to D.
4 - 32	General working space for floating point calculations; library routines are allowed to use 6 to 19 inclusive.
33 - 42	Floating point variables E to H and U to Z.
43 - 142	These are allocated one by one to the hundred permitted multicharacter unsubscripted floating point variables, in the order of their first appearance.

The space immediately beyond this is reserved for the Autocode package read down from magnetic tape, and for the magnetic tape buffer which occupies 512-1023 inclusive. The routines in the package themselves start at 150, but the entire block 0 - 511 has to be read from the tape; this enables the initial values of the floating point variables to be set automatically.

The upper end of the region reserved for the Autocode package is defined by two parameters; (1) is the address of the first register not used for the routines, and (60), which is the first subsequent multiple of 512, is the first register whose contents are not overwritten when the package is brought down from magnetic tape. The subscripted variables occupy a region starting at (1), whose size is determined by the declarations of the first program block. The object program follows immediately after this region, with the proviso that in no circumstances may it begin before (60).

The fixed starting routine which is obeyed at the beginning of any

object program ensures that the index variables I to T are set to 0.1 and the floating point variables A to G and U to Z to non-standardized zero; by convention H is set to π. This makes it possible to arrange that in the post mortem only those variables which have been set are printed out.

Preset object program routines

The object program as actually obeyed consists of two parts: those orders and constants which have been explicitly compiled and assembled in the store as the translation of the original Autocode program, and a preset package which is brought down from magnetic tape in binary form. This package is brought down to a fixed area in the store, so that it does not have to be relocatable, and entries to it are easy to compile; but this implies that for each object program we have to bring down the entire package and not just those parts of it which may be needed. The package is in fact brought down by the first few obeyed orders of the object program, which are explicitly compiled but have a preset form; they are followed by a jump to the initializing routine in the package, with a link set to the address in the Autocode start directive. It is only when this link has been obeyed that the object program proper can be said to have started.

The package is read down onto the first few blocks of object program space, and therefore formally contains some registers allocated to floating point variables. Apart from this, its contents are as follows:

(i) Closed subroutines, and the long-term working space used by them. These correspond to Autocode words, such as LIBRARY, for which we do not wish to have to compile explicitly the corresponding sequence of orders.
(ii) The initializing routine. This sets all the initial conditions assumed in the description of the Autocode: input and output streams, traps, etc.
(iii) The routines for Postmortems and Rescue.
(iv) Possibly some useful constants.

For short-term working space these routines can, with certain precautions, use the registers allocated to temporary floating point variables; any additional space they need is within the package. Like orthodox Library routines, they use B80 to B89 and hold a link in B90.

The closed subroutines are entered by 1362 orders in the compiled program. This involves some loss of time in obeying the object program, by comparison with the usual method of loading B90 and then jumping explicitly; indeed some of the subroutines are so short that it would be better to compile them in open form into the program. Part of the excuse for the present

treatment is the compiler-writer's convenience; but there is also a compensating saving of time from reducing the number of orders that have to be compiled and assembled. When a version of the compiler is produced which generates binary program in the store, instead of IIT in an output stream, this should be reconsidered.

The various routines are described in detail below. The parameters which are used to label the entry points in this description are those which appear in the IIT form of the package. They also appear in the Autocode compiler, where they are explicitly set by directive; in the object program, therefore, jumps to these entry points can be compiled with absolute addresses and the corresponding parameters never occur. The parameters which address long-term working space only appear in the IIT form of the package; they are used here solely to simplify the description.

In general, the arguments which a routine needs are given on entry to it by the contents of B68, B69 and so on; an index expression or a label is called by value, and a floating point variable by reference. In some circumstances the value of a floating point expression may be in the accumulator.

In order to organize the nesting of subroutines, we maintain a pushdown list of subroutine links and a count of the number of links in the list; the former of these is held in 32 halfwords starting at (100), the latter in the address part of a variable order in the package. There are three entries:

(i) At (3), corresponding to → label →. Set (b90+1) in pushdown list and increase the count by 0.4; and then jump to b90. Error exit if the list was already full.

(ii) At (4), corresponding to RETURN. Decrease count by 0.4, remove last link from the list and jump to the address given by it. Error exit if the list was already empty.

(iii) At (5), corresponding to RETURN → label. Decrease count by 0.4 and remove the last item from the list; then jump to b90. Error exit if the list was already empty.

There are essentially two print routines, the first for PRINT i:j:k and the second for PRINT i ; we regard PRINT i:j as the special case of the former in which k = 1. Some constants are common to both routines, and the first one calls in the second in case of overflow. Entry conditions are as follows:

(i) For PRINT i:j:k set b68=i, b69=j and b70=k. Enter at (9) to print from the accumulator, or at (10) to print from B71.

(ii) For PRINT i set b68=i; enter at (11) to print from the accumulator, or at (12) to print from B69.

In each case there is an error exit if the print parameters are outside the permitted range. Note that the print routines do not themselves put out the spaces at the end of a number; they only set a count in B88 which is used by the layout routine. The only case in which PRINT is not followed immediately by an entry to the layout routine is in optional printing; and in that case a newline is explicitly output.

Each of the seven output streams for which provision is made has its own layout counts. These occupy four halfwords for each stream; two of these hold the number of items in each block and the number of columns, both decreased by one, and the other two hold the current values of the corresponding counts. Working space for the counts of items starts at (164), and for the counts of columns at (165). There are two entry points, the first to set up the layout for one output stream initially, and the second to advance the layout counts for the current stream:

(i) For LAYOUT i:j:k set b68=i, b69=j and b70=k, and enter at (13). This sets up the counts for a new block for output stream i ; if j>0 - that is, if we are setting rather than unsetting the layout - it also puts out a newline on stream i. Error exit unless $1 \leq i \leq 7$.

(ii) To advance the layout counts after printing a number, enter at (14) with the value of b88 that was set by the print routine; this entry should not be used except immediately after PRINT. It advances the block and row counts, and prints two newlines for the end of a block, or one for the end of a row; failing either of these, it puts out the preset number of spaces.

There is also a routine for binary output. To implement PUNCH (i) j set b68=j and b69=i, and enter at (26). There is an error exit unless j = 5,7 or 8.

There is a routine for each of the three input functions. For CHAR, enter at (15); on exit the value of the function will be in B88. For ROW, enter at (16); after exit, mask the content of B88 with a suitable constant, whose value has been set by a previous TRACK instruction, and the result will be the value of the function.

For INPUT, enter at (7) to form the result in the accumulator, or at (8) to form it in B88. In either case B67 should hold the address to which to jump on reading a stroke; on a normal exit this is reset to lead to an error routine, so that the compiled part of the object program need only set it in response to '→ n IF /'. There is an error exit caused by reading any unsuitable character; in this case the character is given, in its original internal code form, in B66, and the number of previous successful entries to

this routine is in B65. (This is the only use of these three B-lines in the object program.) This routine uses internal working space only.

To LIBRARY n corresponds a routine entered at (30+n) with information about its parameters held in B68, B69 and so on. All library routines treat their parameters in the same way. A parameter which is a floating point variable or constant is called by reference, and an index or label is called by value; thus any index parameter can be an expression rather than a single variable. If a library routine has error exits, these will be of the form 1156 0 0 n+10. As well as the library B-lines, these routines may use registers 6-19 as working space.

The magnetic routines are called in essentially the same way as library routines; in particular the treatment of their parameters is identical. They use 512 to 1023 as a buffer for magnetic tape transfers, but the rest of their working space is internal. Enter at (17) for FILE, at (18) for READ, at (19) for WRITE, at (20) for POSITION and at (29) for NEWFILE; in each case there is an error exit if the values of the parameters are inadmissible. It is also necessary to enter this package at the end of the object program, in order to tidy up the buffer; for this purpose, enter at (21) instead of a 1117 order. This routine terminates with an 1117 order.

The special functions RADIANS and DEGREES are obtained by multiplying the content of the accumulator by constants whose address are (24) and (25) respectively.

All errors in an Autocode program are in fact automatically trapped; the normal trapping address is at the start of the post-mortem routine. The instruction TRAP (i:j) therefore alters the trapping address for the corresponding fault, and the instruction UNTRAP (j) restores it. To implement TRAP (i:j) enter at (210) with the value of label i in B68 and with j in B69; to implement UNTRAP (j) enter at (211) with j in B68. In either case there is an error exit unless $0 \leq j \leq 7$.

The initializing routine is a closed routine which is entered as soon as the autocode package has been brought down from magnetic tape; enter at (2) with B90 containing the value of the label in the Autocode start directive. It sets initial conditions as follows:

(i) Set stream 1 for both input and output, provided in each case that it has been specified in the job description. (In each case, the error exit from the stream-setting extracode is trapped; hence this will not at the moment work correctly for output.)

(ii) Set (30) as the address for private monitor and 2(30) for all traps.

(iii) Set the impossible value 0.1 in B1 to B63 inclusive (the B-lines which hold the index variables) so that the post-mortem routine can detect which of them have been set in the Autocode program and need not print the others.

(iv) Set 0 in B65 and (200) in B67, for use in the routine for INPUT.

(v) Set the file boundary by means of the 1000 extracode.

When all this has been done, control is transferred to the compiled program, at the label specified in the Autocode start directive.

The post-mortem routine is entered after any fault, other than one for which a trap has been set in the Autocode program. The entry for faults 97 to 99, which cannot be trapped, is at (30) and is a private monitor routine; all other faults enter the trap routine at 2(30). The two extra orders are used to reconcile the entry conditions. The post-mortem routine falls into two parts, printing first the values of those single-character variables which have been set and then information about the type of error and, if possible, where in the program it has occurred. Since the information obtained by the trap is held in B78 and B79, the preset print routines described above can safely be used. There is also a closed subroutine which is used to print messages; it is entered with a link in B90 and the address of the first of the sequence of halfwords containing the message in B68, and it puts out characters until it encounters 0.0, which marks the end of the message. Note that newline is stored as 7.6 for this purpose.

The post-mortem routine first puts out a suitable heading. It then examines B1 to B12 in turn; and for any of these whose content contains no fractional part it puts out an equation stating the value of the corresponding index variable. It then examines similarly those registers of store which correspond to a single-character floating point variable; and for each of these whose content is not /0/0 it puts out an equation stating the value of the corresponding variable.

The next step is to print out an indication of the cause of the error. To each fault number correspond two halfwords, one giving the address of a suitable error message and the other giving the address of the corresponding open routine. Except for error type 14, the action taken is straightforward:

(i) For faults 1,5,8-10,12,13,19 and 97-99 a standard message is put out; in each case the description of the fault in the Autocode program is essentially the same as in machine code terms.

(ii) Faults 0,2 and 4 imply that the program has been overwritten because of a previous misuse of subscripted variables. For fault 0 the post-mortem routine also examines the value of B127 when the trap occurred; this shows whether it was the compiled program or the preset package that was overwritten, and hence whether the subscript was too large or was negative.

(iii) Faults 3 and 6 are the immediate consequence of out-of-range subscripts, the subscript being negative in the former case and too large in the latter.

(iv) For fault 11 the current input stream number is put out as part of the obvious error message.

(v) Faults 20-25 are magnetic tape faults not caused by the Autocode program; they have a common message which includes the fault number.

Error type 14 represents the error exits from the preset routines and those deliberately compiled into the object program. The type of error is indicated by the address of the 1156 order; the post-mortem routine therefore examines this to find a suitable message.

(i) If the address is negative, then the object program has reached the end of a block and not found a jump instruction. The block number is n, where n-100 is the address of the 1156 order.

(ii) Addresses 1,2,5,6 and 8 correspond to errors in the preset routines as follows:

1 Subroutines nested more than 32 deep.
2 No address in list when exiting from subroutine.
5 Print parameters outside permitted range.
6 Layout parameters outside permitted range.
8 PUNCH instruction with impossible track number.

Address 7 corresponds to an illegal character in the data read by an INPUT instruction; the character is in B66 and the count of successful entries in B65.

(iii) Address n+10, where $1 \leq n \leq 20$, corresponds to misuse of Library n.

(iv) Addresses 31 to 33.3 indicate misuse of the magnetic tape package, the integral part indicating the type of misuse and the fractional part the type of magnetic instruction in the Autocode program. The corresponding meanings are given in the following two tables:

31 File number outside the permitted range.
32 The file has not been initialized.
33 The block at which the file is to start is outside the permitted range.

0.0 In FILE 0.1 In POSITION
0.2 In READ 0.3 In WRITE

When the post-mortem routine has finished outputting the type of error and any information associated with this, it puts out the position of the error in the original Autocode program. This is held in coded form in B64,

in which the digits are assigned as follows:

0 - 8	Last label number; this is forced to 511 if no label has yet been read in this block.
9 - 19	Number of lines since the last label.
20 - 23	Block number.

Compilation of arithmetic expressions

The subroutines described below compile those parts of the object program which calculate the values - whether in floating point or index mode - of algebraic expressions. In each case the data for the subroutine is a formula, or section of a formula, after it has been compressed and edited by the input routines; as the formula is compiled, so corresponding alterations are made in the data. The data will have been examined for syntactic errors before these subroutines are entered. A formula is packed into a sequence of halfwords, the content of a halfword representing a symbol. The following types of symbol are generated by the input routine:

(i) A variable, whether single or multi-character. The halfword contains an indication of whether the variable is simple, subscripted or index; and it also contains an indication of the associated address. For a simple variable this is the store address, for an index it is the associated B-line, and for a subscripted variable it is a pointer to the place in a list in which the base address for that row of variables is held.

(ii) A constant. The halfword contains an indication of whether the constant is floating point or index, and also its position in the associated list of constants. Duplicates are not eliminated from these lists, because there may be changes of sign during editing. The list of floating point constants is a genuine one which can be output at any opportunity as part of the object program; the list of index constants is merely a necessary piece of indirect addressing.

(iii) Functions. These include the special functions INPUT, ROW and CHAR, which have no arguments.

(iv) Arithmetic signs. By this stage all implied multiplication signs have been made explicit. In addition to the usual four signs, +, -, × and /, the sign for negative multiplication may have been inserted during editing, as part of the attempt to replace minus by plus signs.

(v) Brackets. Distinctions are drawn between bra and ket, and between round and square brackets; but it is not necessary to decide whether round brackets are functional or algebraic.

(vi) Beginning and end of the expression to be compiled.

The following symbols are generated during compiling:

(vii) Null halfwords. These are produced because the formula is not necessarily closed up to the left as it is simplified. They are wholly ignored in the descriptions that follow.

(viii) Temporary variables, both floating point and index. The halfword indicates the type of the variable and the associated store address, in much the same way as in (i) above.

There is also the following notional symbol, which will not appear explicitly in the data but which it is convenient to use in the description:

(ix) The current content of the accumulator.

The subroutines have been designed to compile reasonable, though not optimal, object program; clearly they could be changed so as to embody more elaborate rules. The emphasis has been on minimizing the number of orders generated, rather than on minimizing object program running time; thus there are no qualms about extracodes such as the 1302 order. The primary reason for this is that it leads to simpler rules for compiling; but it is hoped that the saving in compiler and assembly time will balance the increase in object program running time. Similarly, index variables in a floating point expression are converted to floating point mode before any arithmetic is done on them; the main reason for this is simplicity, but it does also decrease the risk of undetected overflow. No special attempt has been made to use the minimum amount of object program working space, either for index or for floating point variables. If the space allowed is exhausted, this is taken to indicate that the expression is too complicated, and an error message is printed.

The subroutines have a strict hierarchical order; thus each subroutine may call in others lower than itself, but none of them can be used recursively. This condition has to be imposed because the total working space of the compiler is limited; in fact it saves both machine time and programming effort. It can be achieved by a suitable choice of the order in which the subformulae are to be compiled.

The rules of the compilation process are stated in such a way that they enable any syntactically correct formula to be compiled; but there are two contexts in which they have not been made optimal because they will already have been changed by the input editing process. One of these is the unary plus, which should always have been deleted. The other is a minus sign preceding a product; here the minus should have been changed to a plus (or deleted if unary) and one multiplication changed from positive to negative.

The highest level subroutine merely removes all occurrences of the special function INTEGER. This has to be dealt with specially because it occurs in an index expression but involves the use of the accumulator. This process is described as happening inside an arithmetic expression; but in practice it will usually be done over a whole line at once. It is in order to save time in this subroutine that the input routine forms a list of the positions of the arguments of the function INTEGER in an instruction. This subroutine is supplied with an auxiliary, depending on the nature of the line. The auxiliary determines the index variable to which the value of each function INTEGER will eventually be assigned, and is not described here.

Subroutine 1. Reduce arithmetic expressions by removing INTEGER. For each item in the list of arguments of the function INTEGER, treated on a first-in, first-out basis, use subroutine 2 to compile the argument into the accumulator, and delete it from the expression; then enter the auxiliary to obtain a suitable index variable, and compile a 1301 order to it; then replace INTEGER by a reference to this variable.

The second level subroutine compiles floating point expressions which do not involve the function INTEGER. (The corresponding subroutine for fixed point expressions is notionally at a lower level, because it does not involve subscripts.) The structure of this subroutine is governed by the fact that all arithmetic must be done in the accumulator; thus once we have the value of a subexpression in the accumulator (in the object program), we should do as much work on it as possible before storing the result. The sequence of orders compiled by this subroutine necessarily leaves the value of the expression in the accumulator, which is where it will be needed.

In order to simplify the description of the subroutine, we shall use 'operand' to include variables (whether simple, subscripted or index), constants and the two special functions ROW and CHAR. This involves certain conventions in interpreting the description. An operand is regarded as a single symbol, even though it will spread over several halfwords if it is a subscripted variable; in other words, pointers ignore square brackets and their contents. By 'compiling' an accumulator order involving a subscripted variable, we understand the appropriate entry to Subroutine 3.3; this will compile the accumulator order preceded by any necessary modifier orders. A constant will be floating point and is treated exactly like a simple variable; any advantage that can be obtained by changing its sign will already have been gained in the editing routine. If ROW or CHAR appears, it is replaced by a temporary index variable and the corresponding sequence of orders is compiled; this happens before the accumulator order is compiled, perhaps even as soon as the pointer reaches it.

Subroutine 2. Compile reduced floating point expression. This subroutine has two phases; in the first it finds a suitable operand to replace by

'Acc', and in the second it absorbs symbols of the expression into 'Acc' step by step until this is no longer possible. If at the end of the second phase the expression has been reduced to the single symbol 'Acc' then the subroutine is finished; otherwise we replace 'Acc' by a temporary variable, compile a suitable 356 order and return to the first phase.

The search for a suitable operand is controlled by a pointer. On the original entry to the subroutine this pointer starts at the right-hand end of the expression; on a re-entry to phase one from phase two, it starts at the temporary variable which has just replaced 'Acc' in the expression. During phase one the pointer moves from right to left symbol by symbol until it finds one of the following situations:

(i) The function INPUT, which is the only function other than ROW or CHAR which can be found in this phase. Replace this by 'Acc' and compile the appropriate library call.

(ii) Round bra, or the left-hand end of the expression. This must be followed by operand, 'plus' operand or 'minus' operand. In the first two cases compile a suitable 324 or 1524 order; in the third case compile a 325 or 1525 order. Then delete the sign, if any, and replace the operand by 'Acc'. However, there is a possible refinement. If the bra is preceded by a function other than RADIANS or DEGREES, and the argument of the function is an unsigned floating point operand, then we can form the function by a single order.

(iii) Stroke, provided that the next non-operand on the right is a stroke or a positive or negative multiplication. Replace the symbol after the stroke, which must be an operand, by 'Acc' and compile a 324 or 1524 order.

(iv) Plus or minus, provided that the next non-operand on the right is a stroke or a positive or negative multiplication. Replace the symbol after the plus or minus, which must be an operand, by 'Acc'; if the sign was plus, compile a 324 or 1524 order; if it was minus, change it to plus and compile a 325 or 1525 order.

These situations are mutually exclusive, and can therefore be tested for in any order; moreover one of them must happen eventually. As soon as we have found one and taken the proper action, we enter the second phase.

In phase two we examine the neighbourhood of 'Acc' to see if it fits any one of the following descriptions. Note that these are not mutually exclusive, and have therefore to be examined in the order given; also that we only examine a description if we have failed to fit any of the previous descriptions.

(i) Next symbol to the right is a stroke. Delete the stroke and the symbol following it, which must be an operand, and compile a 374 or 1574 order.

(ii) Next symbol to the right is a positive or negative multiplication sign. Delete this sign and the symbol following it, which must be an operand, and compile a 362 or 1562 order for a positive multiplication, or a 363 or 1563 order for a negative multiplication.

If neither of these cases has occurred then the next symbol to the right must be plus, minus, a round ket or the end of the expression.

(iii) Previous symbol to the left is a positive or negative multiplication sign. If the symbol before that is an operand, delete the operand and multiplication sign, and compile a 362 or 1562 order for a positive multiplication, or a 363 or 1563 order for a negative one.

(iv) Previous symbol to the left is plus, minus, round bra or start of expression, and next symbol to the right is plus or minus. In this case the next but one symbol to the right must be an operand; delete this operand and the plus or minus that precedes it, and compile an order according to the following table:

Operand	Sign of operand	Character before 'Acc' Minus	Other
Floating	+	322	320
"	-	320	321
Index	+	1521	1520
Index	-	1520	1521

Moreover, change the symbol before 'Acc' to plus if a 322 order was compiled.

(v) Previous symbol to the left is plus or minus, which is preceded by an operand. If the symbol before the operand is plus, minus, round bra or the start of the expression, proceed as follows. Compile an order according to the following table:

Operand	Preceded by	Symbol before 'Acc' +	-
Floating	Minus	321	320
"	Other	320	322
Index	Minus	1521	1520
"	Other	1520	1521

Delete the operand, and the character before it if that is a sign;

and if a 322 order was compiled then change the symbol before 'Acc' to a plus.

(vi) Previous symbol to the left is plus or minus, preceded by a round bra or the start of the expression. Delete the sign; and if it was a minus compile a 322 order which refers to the constant +0, which is available in the package.

(vii) Previous symbol to the left is round bra. In this case the next symbol to the right must be round ket. If the symbol before the round bra is a function, compile the order that calculates it and delete function, bra and ket. Otherwise simply delete the bra and ket.

If the description that was found led to any action, then repeat phase two with the new situation. If it led to no action, or if no suitable description was found, then either exit from the subroutine or return to phase one as described above.

It is also necessary to consider in detail the relation between the symbols used in the description above and the actual string of halfwords which represents the expression. The symbol 'Acc' is never held in the store; it corresponds to a sequence of null halfwords, and it is represented by pointers to the first and last of these halfwords. Deletion in phase two involves increasing the scope of 'Acc'. Pointers are moved by a pair of internal subroutines to find the symbol before or after a given symbol; these move one halfword at a time, testing for the ends of the expression and ignoring null halfwords and square brackets and their contents. On returning from phase two to phase one, the expression is contracted to the left by removing those null halfwords which constituted 'Acc', except for the one into which the temporary variable is put; this is not logically necessary, but it should save time.

This is the only subroutine which uses temporary floating point variables; and the description above shows that it uses them on a strict 'last in, first out' basis. We can therefore control the allocation of storage registers to these variables as follows. (We assume that the registers available are consecutive, and that those with the lowest addresses are to be used first; but only minor changes in the description would be needed to change these conventions.) We hold throughout the subroutine a number, which is the address of the next register available for use. On entry to the subroutine, this is set to the first address allocated to temporary variables. It is increased by one every time 'Acc' is replaced by a temporary variable at the end of phase two; and it is decreased by one whenever any other order involving a temporary floating point variable is compiled - the new value of the number being in this case also the address of the order compiled. Thus in the representation of the expression in the store, we do not need to hold the address of the temporary

variable; we can simply use the same symbol for all temporary floating point variables.

The remaining subroutines deal with index expressions which do not involve the function INTEGER. Because of the method used for dealing with this function in Subroutine 1, we cannot use the simple technique above for controlling temporary index variables. Instead, a temporary index variable in the expression to be compiled is represented by a half-word which contains the number of the variable; and to each potential temporary variable we assign a marker which denotes whether it is in use at the moment or not. This marker is set to 'in use' whenever a new temporary variable is supplied - the variable being the one with smallest address among those at present not in use; the marker is reset to 'not in use' whenever the variable is used to change the value of another variable.

Apart from the special treatment of the functions INPUT, CHAR and ROW in Subroutine 4, the only orders assembled in these routines are 120 to 124, 1302 and 1303. If these combine the values of two temporary variables, then the result is the new value of the temporary variable with the lower address and the other is released; this convention is imposed so that Subroutines 3.2 and 3.4 should compile assignments to the temporary variable with the lowest possible address, as the specifications of Library routines assume. If the order assembled combines the values of a temporary and a named variable, then normally the result is the new value of the temporary variable; the only exception is described explicitly in Subroutine 3.1, where the result is the new value of the named variable and the temporary variable is released. Subroutine 3.1 is also the only place where the values of two named variables are combined. The function part of the order compiled is determined by the arithmetic involved, except in the choice between a 120 and 122 order; here we choose so that the result has positive sign in the expression being assembled.

The other operands that can occur are constants and functions - the latter only in Subroutine 4. To each of the three possible functions corresponds an Autocode-package subroutine which loads B83; as soon as one of them is encountered, we compile an entry to the corresponding subroutine, followed in the case of ROW by a masking order whose argument has been set by a previous TRACK instruction. We then replace the function by a pseudo-variable associated with B83, which behaves exactly like a temporary variable except in two respects. It does not have a 'used - unused' marker, to be reset when it is released; and it does not count as a temporary variable during the search for one in Subroutine 3.2

So far as multiplication is concerned, a constant is treated exactly

like a named variable; this copes with Subroutine 4. The active part of any other subroutine is concerned with manipulating an elementary sum: that is, an expression containing no brackets, products or functions. The first step in this is to scan through the expression (from left to right so as to simplify dealing with signs) looking for particularly desirable variables; the definition of these depends on the subroutine, but they always include the temporary variable with the smallest address. The same scan deals with constants: we set a number initially to zero, and when we find a constant we delete it (and the plus or minus that precedes it) and add it into the number. The number is then used as the address part of the next order compiled, with sign changed for a 122 or 123 order; and it is then reset to zero as a marker.

Subroutines 3.1 and 3.3 wholly delete the original expression, replacing it by null halfwords; the other subroutines replace the expression by a reduced form of it, not necessarily closed up to the left. For this purpose, Subroutines 3.3 and 3.4 regard the expression as including the subscripted variable and the square brackets.

Subroutine 3.1. Compile reduced index expression as value for named variable. We call this named variable the 'target variable'. First use Subroutine 4 to reduce the expression to an elementary sum. Scan through this sum, dealing with constants and also looking for the target variable (on its first appearance only) and the temporary variable with smallest address; if either or both of these are found, note them and delete them and their signs from the expression. If no temporary variable has been found but there are at least two operands left in the expression (or if there is just one operand, but both it and the target variable are negative) then delete one operand, obtain and note a new temporary variable taken positively, and compile a suitable 121 or 123 order.

If we now have a temporary variable, scan through the expression again, merging operands one by one into the temporary variable; each merge involves deleting the operand from the expression, compiling a 120, 122 or 124 order and (for a 120 order only) changing the sign of the temporary variable from minus to plus. If there is no temporary variable but the expression still contains an operand, then note this as if it were a temporary variable and delete it from the expression. (Thus at this stage the expression is totally deleted.) Finally, find which of the four alternatives below corresponds to the operands noted and act accordingly:

(i) No variables. Compile a 121 order to set the target variable.
(ii) Target variable only. Compile a 124 or 120 order according to the sign of the target variable, to incorporate the constants.
(iii) Non-target variable only. Compile a 121 or 123 order to set the target from the non-target variable, according to the sign of the latter.

(iv) Target and non-target variables. First, if both these have negative signs compile a 120 order to change the sign of the temporary variable; and note this change of sign. Then compile a 120, 122 or 124 order to absorb the non-target into the target variable.

Subroutine 3.2. Compile reduced index expression as value for temporary variable. Use Subroutine 4 to reduce the expression to an elementary sum; and then use Subroutine 5 to deal with this. The temporary variable assigned to this will be the one with lowest address among those which are in the expression or available for use in it.

Subroutine 3.3. Implement compilation of an order referring to a subscripted variable. The data for this subroutine consists of the function part of the order and a pointer to the halfword in an expression which contains the name of the subscripted variable. First use Subroutine 4 on the subscript to reduce it to an elementary sum. Scan through this sum, dealing with constants and also looking for the temporary variable with smallest address, and for one other positive operand; if either or both of these are found, note them and delete them from the expression. If no temporary variable has been found but there are at least two operands left in the expression (or one negative one), then delete one operand, obtain and note a new temporary variable taken positively, and compile a suitable 121 or 123 order. If there is no temporary variable but the expression contains just one operand, which is positive, note this as if it were a temporary variable and delete it from the subscript.

If there are still operands in the subscript, and hence certainly a temporary variable, scan through the subscript again merging operands one by one into the temporary variable; as above, each merge involves deleting the operand from the subscript, compiling a 120, 122 or 124 order and (for a 120 order only) changing the sign of the temporary variable from plus to minus. If at the end of this scan the temporary variable is still negative, compile a 120 order and note its change of sign.

At this stage the subscript is totally deleted. Add to the number representing the contribution from the constants the base address corresponding to the name of the subscripted variable; and delete this name and the square brackets. Finally compile the order given, using as modifiers the two variables noted.

Subroutine 3.4. Set reference address for subscripted variable as temporary variable. The data for this is a pointer to the halfword in an expression which contains the name of the subscripted variable. Replace the name by the base address associated with it, delete the square ket, and delete the square bra or replace it with plus according as the subscript does or does

not start with a sign. Use Subroutine 4 to reduce the resulting subexpression to an elementary sum; and then use Subroutine 5 to deal with this.

In practice this and Subroutine 3.2 will be done by the same group of orders.

Subroutine 4: Put reduced index expression into the form of an elementary sum. The subroutine is controlled by a pointer which starts at the right-hand end of the expression and moves back step by step, ignoring null halfwords. After each step the subroutine tests whether the pointer points to any of the four situations below; note that these are not mutually exclusive and hence must be tested for in the order stated:

(i) Temporary variable, followed on the right by a multiplication sign. Merge this variable and the next operand, compile a 1302 or 1303 order, and delete one operand and the multiplication sign.

(ii) Temporary variable preceded on the left by a multiplication sign. If the previous character is an operand, merge it with this variable, compile a 1302 or 1303 order, and delete one operand and the multiplication sign.

(iii) Operand followed on the right by a multiplication sign but not preceded by one. Obtain a new temporary variable and set it in the expression in place of this operand, and compile the corresponding 121 order.

(iv) Round bra. Apply Subroutine 5 to the subexpression strictly between this round bra and the next round ket to the right; then delete the round brackets.

If the description that was found led to any action, then reset the pointer to the result of the action and test again for any of the four situations listed; note that this may mean moving the pointer to the right. If no action was taken, or if the actual situation fitted none of the descriptions, then move the pointer back another step. When the pointer reaches the left-hand end of the expression, the subroutine is finished.

Subroutine 5. Compile elementary sum as value for temporary variable. Scan through the sum, dealing with constants and also looking for the temporary variable with smallest address; if this is found, note it and delete it from the sum. If no temporary variable has been found, obtain and note a new one taken positively; if there are operands still in the sum, delete one of them and compile a 121 or 123 order according to its sign; if not, compile a 121 order to deal with the constants. Now scan through the expression again, merging operands one by one into the temporary variable; each merge involves deleting the operand from the expression, compiling a 120, 122 or 124 order and (for a 120 order only) changing the sign of the

temporary variable from minus to plus. If at the end of this the temporary variable is still negative, compile a 120 order to change its sign. Then if the constants marker is non-zero (so that no order has yet been compiled), compile a 124 order to deal with the constants.

Finally, write to some halfword of the expression (which is now null) the symbol for this temporary variable; and exit from the subroutine.

Translation of Autocode instructions

An Autocode instruction of any particular type may be regarded as a fixed syntactic formula whose gaps, if any, contain numbers, variables, labels, or index or floating point expressions. The contents of the gaps provide all the variability in the instruction; in effect they are the arguments of a rather clumsily written function. To each type of instruction corresponds a translation routine; this may compile organizational orders for the object program, or update the long-term information held by the compiler, or both. If necessary, the translation routine will call in a subroutine for dealing with arithmetical expressions.

With this method of translation, there is a correspondence between Autocode instructions in the source program and sections of machine code in the object program. To simplify Rescue, we wish to make this correspondence explicit; so we arrange that while the object program is being obeyed a particular B-line shall always contain the line number of the current line in the source program. The content of this B-line must be reset at the beginning of each section of object program; because of the possibility of exceeding time or output limits, we must do this even if the section of object program appears incapable of monitoring. Most translation routines will therefore compile an order to reset this B-line, although this will not be stated explicitly; the current line number is necessarily held by the input routine, since it is needed by the error exits from the compiler.

It is convenient to think of the compiler as working in two phases. In the first phase it reads an Autocode instruction into the store, edits and compresses it, and checks it for syntactic correctness. In the second phase it recognizes the type of the instruction, and enters the appropriate routine to translate it; these are the routines described below. In practice, the interface is somewhat different, because the type of an Autocode instruction is uniquely determined by its first few characters; thus the first phase contains one routine for each instruction type, and this routine leads straight on to the corresponding routine of phase two. For certain types of instruction, input and translation go on at the same time; these are the cases in which the translation process is unusually simple and in which the instruction might reasonably be so long that it could not all be held in the store at once:

(i) Program block and subscripted variables declaration. In fact, each declaration is read and translated by itself, in order to save space; but we shall ignore this when we come to describe it.

(ii) Square brackets enclosing comment. On reading the square bra, set up a count of bracket depth, whose original value is one. This count is to be increased by one on reading a square bra and decreased by one on reading a square ket. The comment ends with the square ket at which the count becomes zero; we must then verify that the rest of the line in the source program is empty.

(iii) TITLE. Verify that the next character is a colon. Then read characters from the input stream and write them to the system output stream until another colon is read; this terminates the title and is not itself output. The compiler must keep a record of the current internal-code shift in the output stream and put out shifts when necessary; this record is initially set to 'inner' and must be left so at the end of the title, if necessary by outputting an extra 'inner set'. Two characters, other than colon, need special treatment: a suffix 2 must be replaced by a space, and a newline must be put out by a 1065 0 0 2,1 order and must reset the output stream shift marker. At the end of the title, verify that the rest of the line is empty.

(iv) TEXT. This is treated in essentially the same way as TITLE, except that instead of outputting a character we compile a 1064 order in the object program; note that for a newline this is 1065 0 0 2,1 again.

There are two other types of Autocode instruction which are anomalous in that neither occupies a line to itself:

(i) An Autocode label causes the corresponding machine code label to be compiled; this compilation includes a check that the label has not been used before, and any necessary updating of lists. After this has been done, the various line counts in the compiler are changed so as to refer to this label rather than the previous one; but there is no change if the previous label was on the same line.

(ii) Round brackets denoting key 0 optional compiling. These may be treated as one Autocode instruction or as two. If a round bra is read when key 0 is not set, then we read and ignore characters from the input stream until we encounter a newline followed by a round ket; this ends the optional section, and we then verify that the rest of the current line is empty. (This treatment leads to certain anomalies for TEXT or TITLE; it is dictated by compatibility.) If a round bra is read when key 0 is set, this merely sets a flip-flop; and when a round ket is read at the end of the section the flip-flop is unset again. In either case there is an error

exit if the flip-flop is in the wrong position. (A KEYS instruction in the middle of an optional section will therefore not cause the rest of the section to be ignored.)

The instructions EQUATE and PROGRAM control the allocation of space to subscripted and multicharacter variables; their immediate effect is to update the tables which the compiler uses for this purpose. There are three such tables, the first for multicharacter index variables, the second for multicharacter floating point variables, and the third for subscripted variables whether single or multicharacter. The first table assigns to each multicharacter index variable a B-line; when such a variable is first encountered by the input routine it is assigned the next unused B-line, but this may later be altered by an EQUATE instruction. The second table assigns to each multicharacter floating point variable a storage register; remarks similar to those above apply in this case also. The third table gives for each storage register assigned to a floating point variable the base address for the corresponding subscripted variable name; if no such subscripted variable has been declared this base address is negative, as a marker. (Note that the argument for entering this table is not the name of the simple variable, because we have to allow for the effects of EQUATE.) The third table is altered by a PROGRAM instruction.

An EQUATE instruction has the form EQUATE : $\alpha_0 \alpha_1 \ldots \alpha_n$ where $n \geq 1$ and the α_i are multicharacter identifiers of the same type. Without loss of generality assume this type to be index. We then read the entry in table 1 corresponding to α_0 and write this into the positions in table 1 corresponding to $\alpha_1, \ldots, \alpha_n$.

A PROGRAM instruction has the form PROGRAM m $\beta_1 \ldots \beta_n$ where n can be zero and each β has the form $\alpha [\nu]$ with α the name of a floating point variable and ν a non-negative integer. On reading m, compile a parameter row directive for the object program, set -1 in every position in table 3 and reset the count giving the first register still available for subscripted variables to its base value. On reading any β, find the address assigned to α as a simple variable, store the current value of the count in the corresponding position in table 3 (verifying that that position previously contained -1), and increase the count by $\nu + 1$. At the end of the instruction, if this is not the first program block to be dealt with, verify that the value of the count is not greater than it was for the first program block.

There are two other instructions which only alter the information held in the compiler. For n TRACK verify that n = 5,7 or 8 and then set 31, 127 or 255 respectively as the mask to be used when compiling the function ROW. For KEY or KEYS $\nu_1 \ldots \nu_n$ clear all key marker digits and then reset those

for $\nu_1, \ldots, \nu_n$, verifying that each $\nu_i \leq 5$.

The main body of a calculation instruction has the form $\alpha_1 = \ldots \alpha_n = \beta$ where the α_i are variables, with the floating point preceding the index ones, and β is an arithmetical expression which is index or floating point according as α_n is. If this is followed by '→ m IF /', which can only happen when the instruction contains the function INPUT, compile an order to set the value of label m in B67. If the instruction is followed by one or more optional print messages of the form *m, test whether key m has been set, and if so set the optional print marker; this will be used when the calculation instruction has been compiled. Next, apply Subroutine 1 to the entire expression (not just to β) to remove all appearances of the function INTEGER.

There is now a division, according as α_n is an index or a floating point variable. If α_n is an index, use Subroutine 3.1 to compile the orders which implement $\alpha_n = \beta$. Now if $n > 1$ delete '$= \beta$', regard α_n as a new β and return to the head of this paragraph. If $n = 1$, test if the optional print marker is set; if so, compile the orders to set the value of α_1 in B69, to output a newline followed by *m and a space, to print from B69 and to output another newline. If, however, α_n is a floating point variable, use Subroutine 2 to compile the orders which load the value of β into the accumulator. Then for each α in turn, compile a 356 order to set α from the accumulator, using Subroutine 3.3 for this if α is a subscripted variable. Finally test if the optional print marker is set; if so, compile the orders to output newline followed by *m and a space, to print from the accumulator and to output another newline.

Autocode cycles are organized by the instructions FOR $\alpha = \beta_1 : \beta_2 : \beta_3$ and REPEAT, which are paired. In principle, a FOR instruction causes two chunks of program to be compiled; one of them is output immediately as part of the object program, the other is put into a pushdown list. A REPEAT instruction causes the top chunk to be removed from the pushdown list and output as part of the object program. There is the further complication that the first chunk involves forward references to the second one, and must therefore be supplied with object program labels; this has to be done by a special subroutine, since it will have to be changed for direct compiling into binary.

Suppose first that α is a floating point variable (necessarily a simple one) and that the count is an increasing one; this last fact will have been noted in the input routine. Enter a subroutine to obtain two object program labels λ and μ. Compile object program orders to implement the calculation instruction $\alpha = \beta_1$ and follow these with the order 121 * 0 (λ). Find ν, the current value of (0), and set μ and ν in

the pushdown list. Compile object program orders to implement the calculation instruction $\alpha = \alpha + \beta_2$, where the plus sign is omitted if the first character of β_2 is plus or minus. Output the object program label λ; then use Subroutines 1 and 2 to compile the orders which load $\beta_3 - \alpha$ into the accumulator, and follow these by 237 * 0 (μ). This completes the translation of a FOR instruction. If the count was a decreasing one, the only change is that we compile $\alpha - \beta_3$ instead of $\beta_3 - \alpha$.

If α is an index variable the treatment is similar; suppose again that the count is an increasing one. Proceed as above up to outputting program label λ; then use Subroutines 1 and 3.2 to compile the orders which load $\beta_3 - \alpha$ as a temporary variable (necessarily B68) and follow them with 217 * 68 (μ). If the count is decreasing, compile $\alpha - \beta_3$ instead. Certain refinements to this are available. For a decreasing count with $\beta_3 = 0$, we need not compile $\alpha - 0$ but can use a 217 order to test α directly. If β_3 is at worst the sum of an index variable and a constant, we can compile $\alpha - \beta_3$ or $\beta_3 - \alpha$ into Bt by a single 170 order, and test it with 227 * 0 (μ).

To translate REPEAT, remove the top pair μ, ν from the pushdown list, compile the order 121 * 0 ν and then output the object program label μ. There will be an error exit if the pushdown list was empty; there will also be one on a FOR instruction if the pushdown list is already full.

There is scope for considerably greater saving by putting the test into the part of the program compiled by REPEAT; this is worth doing if the count is not too complicated. For example, suppose that α is an index variable, that $\beta_2 = 1$ and that β_3 is at worst the sum of an index variable and a constant; then a better translation procedure is as follows. Enter a subroutine to obtain one object program label λ. Compile object program orders to implement the calculation instruction $\alpha = \beta_1$ and follow these with an order to set $\beta_3 - \alpha$ in Bt and then the order 227 * 0 (λ). Find ν, the current value of (0), and set α, β_3, λ and ν in the pushdown list. To translate REPEAT, remove them from the pushdown list, compile an order to set $\beta_3 - \alpha$ in Bt, and then the order 221 * α ν, and finally output the object program label λ. There are obvious minor changes for $\beta_2 = -1$, and even further simplification when $\beta_3 = 0$.

An Autocode jump instruction may be unconditional or conditional, and it may be either a simple jump or a switch; this gives four possibilities. In addition there are the entries to and exits from subroutines.

A simple unconditional jump has the form '→ m' and is translated into the single order 121 * 0 (m). The input routine edits a conditional jump, so that it is handed over to the translator in the form '→ m IF α = 0'

or '$\alpha \neq 0$' or '$\alpha \geq 0$' or '$\alpha < 0$' where α is an index or floating point expression. In all the descriptions we assume that the condition is $\alpha = 0$; the other cases only involve changing the function part of the test order. If α is floating point, use Subroutines 1 and 2 to compile orders which load α into the accumulator, and follow these with 234 * 0 (m). If α is an index expression, compile orders which load it as a temporary variable, necessarily B68, and follow these with 214 * 68 (m); but note that if α is the difference of two variables, plus perhaps a constant, it would be better to compile the one order which sets it into Bt and follow that by 224 * 0 (m).

A switch has the form $\rightarrow \beta[\lambda_0, \ldots, \lambda_n]$ where β is an index variable and the λ_i are labels. To translate it, compile the four orders 217 * β n+5(0), 170 β 0 n, 227 * 0 n+3(0), 101 * β 1(0) followed by the n+1 pairs $/(\lambda_0)/0 \ldots /(\lambda_n)/0$. If the switch is conditional then the condition is edited as above; suppose it is $\alpha = 0$. If α is a floating point expression, compile the orders to load α into the accumulator, followed by 235 * 0 n+6(0) and then the orders to translate the corresponding unconditional switch; similarly for α an index expression.

Subroutine entries and exits involve a jump to a preset routine, and the parameters explicitly numbered are the corresponding compiler ones. For $\rightarrow$ m $\rightarrow$ compile the two orders 1362 0 0 (3), 121 * 0 (m); for RETURN compile 1362 0 0 (4). For the instruction obtained by following RETURN by an unconditional jump, compile 1362 0 0 (5) followed by the orders corresponding to the jump.

There are a variety of instructions of library type, for all of which we compile the orders which set the parameters in their proper places, followed by a 1362 order to enter a preset subroutine whose address is a parameter of the compiler. The print routines are anomalous, in that their parameters are taken in an unnatural order, and in that the 1362 order compiled must be followed by a further order 1362 0 0 (14) to deal with layout:

(i) For PRINT (α) i:j:k use Subroutines 1 and 3.2 to compile orders to set the value of i in B68, then that of j in B69 and then that of k in B70. If α is a floating point expression, use Subroutines 1 and 2 to compile orders to set its value in the accumulator, and follow these with 1362 0 0 (9); if α is an index expression use Subroutines 1 and 3.2 to compile orders to set its value in B71 and then compile an order 1362 0 0 (10).

(ii) Translate PRINT (α) i:j as if it were PRINT (α) i:j:1.

(iii) For PRINT (α) i use Subroutines 1 and 3.2 to compile orders to set the value of i in B68. If α is floating point, compile orders to set it in the accumulator, followed by 1362 0 0 (11); if it is an index expression, compile orders to set it in B69 and then compile 1362 0 0 (12).

The instruction PUNCH (i) j is anomalous only in the order of its parameters. Compile orders to set j in B68, and then to set i in B69, and then compile 1362 0 0 (26).

The other instructions of this sort are LAYOUT, LIBRARY n, TRAP, UNTRAP and the five magnetic-tape actions; for this purpose we regard each LIBRARY n instruction as a distinct type, rather than treating n as a parameter. To each of these types corresponds an address (m) in the preset routines, and a codeword which shows how many parameters should be supplied and whether they are floating point, index or label. For each parameter in turn, compile the orders to set the corresponding B-line. For floating point we either compile a 121 order or use Subroutines 1 and 3.4; for an index expression we use Subroutines 1 and 3.2; for a label we compile a 121 order. When all the parameters have been dealt with, compile the order 1362 0 0 (m).

The instruction PRINTCHAR (α) would be dealt with in exactly the same way, but there happens to be no preset routine associated with it. Use subroutines 1 and 3.2 to compile orders to set the value of α in B68, and then compile 163 68 0 0 three times, followed by 1064 0 68 0.

Any other type of instruction leads to the compilation of a fixed sequence of orders, whose addresses may depend on the numbers in the instruction:

(i) For START m, where m is a label, first note ν, the current value of (0). Compile the sequence of orders 1260, 1267, 1274 which will bring the preset package of routines down from magnetic tape; the rest of these orders depend on the compiler-supervisor interface and are not yet fixed. Then compile 121 90 0 (m), 121 * 0 (2) to jump to the preset starting sequence and thence to the Autocode starting address. Finally compile the start directive $\rightarrow \nu$. This ends the compilation process; the compiler then discovers whether the process has been satisfactory or not, and exits accordingly.

(ii) The following instructions contain a number, and this must be inserted into the machine code formula which is compiled as a translation of them:

READER n	1050	0	0	n
OUTPUT n	1060	0	0	n

(iii) The following instructions lead to the compilation of the fixed order or sequence of orders shown opposite them:

RUNOUT	121	68	0	40
	1064	0	0	0
	203	*	68	-1(0)
PAGE	1061	68	0	0
	1071	0	68	0
CRLF	1065	0	0	2.1
STOP	121	*	0	(21)

(iv) The instructions WAIT and 144/$\neq$2 are totally ignored.

Output of object program

The first version of the compiler will produce object program in IIT form as a character stream; but there will be a later version which assembles it in binary directly into the store. For this later version it will be necessary to rewrite the routines which put out the individual items of the object program, and those which are in any way concerned with object program parameters. These routines are described below; some of the subdivisions of cases are more elaborate than we need for the first version of the compiler, but this will prove advantageous for the later one.

To each Autocode program block corresponds one object program parameter row, the number of the row being one greater than that of the block. Parameters 1 to 256 are assigned to the labels of the Autocode program, parameter $N+1$ corresponding to label N. Parameters 257 to 511 are available to the compiler for references to lists of floating point constants and for the forward references implicit in 'for - repeat' cycles. To ensure that any misuse of labels in the Autocode program is detected during compilation, rather than in the IIT assembly routine, two marker digits are assigned to each possible label: one of them records whether the label has been used, the other whether there has been any reference to it. When a label is used, the routine checks that the first marker was unset and then sets it; when a reference is made to a label, the routine sets the second marker whether it was previously set or not. Before implementing the start directive, the compiler checks that there is no label whose second marker is set but whose first is not.

The compiler also keeps a list of floating point constants appearing in the Autocode program; this list is built up by the input routine, and is cleared by being put out as part of the object program at any reasonable opportunity - that is, after any unconditional jump or at the end of a block. In addition the compiler has five counts associated with the output routines to preserve:

(i) Current value of (0). This is initially set during compilation of the first PROGRAM instruction; it is advanced whenever an item is output.

(ii) List of program block numbers already used. This is needed to check that no number is used twice; it is updated on each PROGRAM instruction.

(iii) Number of constants currently held in the list. This is increased by the input routine whenever a new constant is put in the list; it is reset to zero whenever the list is cleared by output.

(iv) Object program parameter to be used to label the next list of constants.

(v) Next object program parameter available for use.

Of these, (i) and (iii) are used in other parts of the compiler, and must therefore remain the same when this section is re-written. Because there is a special output routine for orders which refer to the list of floating point constants, no access to (iv) is necessary; and access to (v) is provided through a special routine described below.

There are three print subroutines in the strict sense, which output a digit string corresponding to the contents of a B-line:

(i) Print decimal integer or address. This is the subroutine normally used for any number within an item. Since it is to be used for parameter numbers, it must never put out any spaces, either in place of non-significant zeros or in place of a plus sign.
(ii) Print function part of order from octal form. To reduce transcription errors in writing the compiler, the function part is given shifted two places to the right of where it should be. One initial zero may be suppressed.
(iii) Print halfword in octal. A floating point number will be put out as a pair of octal halfwords, to reduce conversion errors.

The routines described below all deal with one or more complete items; and they therefore call in these open or closed subroutines to deal with the parts of an item.

There are five distinct types of order to be output, depending on the structure of the address part; and thus five entries to the corresponding routine. In each case the function and the two modifier parts are given in B-lines on entry:

(i) Simple numerical address.
(ii) Address involves an Autocode label in this block, and must therefore be just the corresponding parameter. The address is specified by the label number, and is put out as a parameter; in addition the routine must set the marker digit which shows that a reference has been made to this label.
(iii) Address involves an Autocode label in another block. This is the same as (ii), except that on entry two numbers are needed to specify the address.
(iv) Address involves a compiler parameter - that is, a parameter number between 257 and 511. This is the same as (ii), except that there is no marker digit to be set. Note that each parameter appears in just one address.
(v) Address involves reference to list of constants. The address

is now an integer plus the associated compiler parameter, and is put out in this form. Note that only the integer needs to be specified on entry to this routine, since the parameter number is already known.

It is in fact true that Ba = 127 in case (iv), and that Ba = Bm = 0 in case (v); but it may not be expedient to take advantage of this.

The only parameter whose value is set explicitly is (0). The corresponding routine puts out a directive of the form '(0) = ...', obtaining the value from the appropriate B-line into which it has already been set. All other parameters are set by labelling. For a compiler parameter the routine simply puts out the parameter number and a closing bracket. For an Autocode label the routine first checks that the label has not already been used, and sets the corresponding marker digit; it then resets the count of Autocode program lines to count from this label, unless it is not the first label on the line; and finally it puts out the parameter number and a closing bracket. There is also a routine which obtains the number of the next compiler parameter available and alters the corresponding count.

For a switch instruction a jump table is compiled; each item of this table is a pair of halfwords of which the first is the value of an Autocode label and the second is irrelevant and for convenience zero. There is a subroutine to put out such a pair of halfwords; as well as outputting, it sets the marker digit which shows that the label has been used.

There is a routine which puts out the current list of floating point constants as part of the object program; this is entered after compiling an unconditional jump or at the end of a program block. The routine first tests whether the list is empty; if so it exits having taken no action. Otherwise it puts out the object program label currently assigned to the list of constants, followed by the members of the list; for simplicity each of these is put out as a pair of octal halfwords. It then resets the count of floating point constants to zero, obtains a new parameter to identify the list of floating point constants, and exits.

The routine which outputs the start directive also tidies up any loose ends in the compilation, and it hands over to the supervisor all the information it needs about the object program - including the statement whether it is error-free and can therefore be run. It first puts out the start directive; the address of this will always be non-parametric. Then it runs through the Autocode labels, testing if there are any which are unset but to which reference is made. It then hands over to the supervisor the number of parameter rows and the largest parameter number used, and also the current value of (0), which gives the total space needed. Finally, it returns control to the supervisor by an extracode which depends on whether the object program is error-free or not.