UNIVERSITY OF MANCHESTER

DEPARTMENT OF COMPUTER SCIENCE

AN INTRODUCTION TO THE COMPILER COMPILER

by

R.B.E.Napper

University of Manchester

DEPARTMENT OF COMPUTER SCIENCE

AN INTRODUCTION TO THE COMPILER COMPILER

by

R.B.E.Napper

Dec. 1965

(Revised Oct 1966)

This document is designed to enable a person who does not (necessarily wish to use the Compiler Compiler (of R.A.Brooker, D.Morris and J.S.Rohl[1]) to appreciate the ideas behind the system and its potential use.

Some basic experience of programming and general knowledge of compilers theory is assumed in the reader; the first chapter summarises the ground that should already be familiar and so defines the level of knowledge below which there is little point in trying to appreciate the rest of the document. No previous knowledge of the Compiler Compiler itself is required, and no reference need be made to any other literature related to it.

This introduction is not designed as a complete introduction for a user of the system, i.e. it does not replace the existing papers published on the system, which provide the only existing 'user's manual'. It therefore leaves out most of the details of the implementation of the system, and those features that are essentially practical devices for making a compiler written in the system more efficient. It also leaves out those features that have not been required much in practice. It does not fully describe the language of the system, and in particular it does not describe the 'list-processing' machinery, which although interesting and very useful in practice is less novel than the 'language-processing' machinery that is the central feature of the system.

[Occasional notes are made inside square brackets (e.g. as here).

On an initial reading no time should be spent in trying to understand them if their point is not immediately obvious.]

Reference

Brooker, R.A. (et al.)   (1963) 'The Compiler Compiler'

Annual Review in Automatic Programming, Vol. III.

---

[1] I. R. MacCallum was inexplicably omitted from this list.

Index

NOTE : Words in brackets indicate terminology which is introduced and defined within the appropriate sections.

Index

CHAPTER 2

PHRASE STRUCTURE NOTATION

CHAPTER 3

THE STRUCTURE OF A COMPILER PROGRAM : FORMAT ROUTINES

Index

CHAPTER 4

THE STRUCTURE OF THE COMPILER COMPILER ; AUXILIARY ROUTINES

INTRODUCTION

The Compiler Compiler was designed by R.A.Brooker and D.Morris as a special purpose compiler to help the team of compiler-writers in writing the set of systems compilers for the Manchester Atlas computer. A 'compiler' for any particular language (on. Atlas) is a computer program existing in machine code inside the computer. This can be called upon to translate any program written in the designed language into the machine code required to execute the job described by the program. The input data for the compiler program is the 'source program' as produced by the programmer, written in the appropriate language, and its end product is an area of the store filled in with the machine code instructions of the translated program, the 'object code'. When it has compiled these instructions, the compiler removes itself from the store and passes control to the object program.

Similarly the 'Compiler Compiler' is a computer program existing in machine code inside the computer, which can be called upon to translate any program written in Compiler Compiler language into the requisite machine code. In general this program is the description of a compiler, and so instead of obeying it after translation, the Compiler Compiler transfers the machine code of the compiled compiler onto magnetic tape under the control of the Atlas Supervisor. Then whenever the Supervisor gets a program written in this language, it singles out this particular compiler, copies the machine code into the store, and passes control to it. The source program is then translated by the compiler as described in the previous paragraph.

Ch. 1   A SUMMARY OF THE MAIN FEATURES OF A CONVENTIONAL AUTOCODE

<u>**Example of a program**</u>

It is assumed that there is little point in trying to appreciate the Compiler Compiler without some knowledge of programming and compiler theory. The example below of a simple computer program and the discussion following it will serve :

i)   As a reminder of some of the more important features of a programming language (an '<u>autocode</u>');

ii)   As an example of the input data of a compiler, which will be used to give some insight into the job a compiler program has to do;

iii) As a reference against which comparisons will later be made between the structure of a program written in Compiler Compiler language and a conventional program, and also some of the points of implementation.

On the left hand side of the example is given the program written in a simple autocode. On the right hand side is given a covering description of each instruction such as might be the informal verbal comment on its function in the context of the job being programmed, The reader should not be put off by the underlining or 'capital words' used in the covering description, which are used systematically to distinguish the uses of words in different contexts, and will be referred to again later; nor should he be put off by the practice of hyphenating phrases of words, which is used where it is convenient to refer to an object by a self-descriptive phrase intend of a single word.

The input data for this program (which follows the program description) is a set of 'n' dates giving the year 'Y_e' in which a set of successive events occurred; those years are preceded by the number 'n', which is between 3 and 1000. There are constraints on the relation between successive years (see <u>check year</u>) and if a date is below 100 it is assumed to be in the same century as the previous one,

The required output is the three numbers :

$$\frac{Y_n - Y_1}{n} \quad , \quad \frac{\sqrt{\sum_{e=2}^{n}(Y_e - Y_{e-1})^2}}{n - 1} \quad , \quad \frac{\sqrt{\sum_{e=2}^{n}((y_e - y_{e-1})^2 - (Y_{e-1} - Y_{e-2}))^2}}{n - 2}$$

Thus where the '<u>squared-difference</u> between x and y' $\equiv (x - y)^2$, The three values are the 'average difference between successive years', the 'square-root-average of squared-differences between successive years', and the 'square-root-average of squared-differences of squared-differences'.

DATA TYPES

INDEX e, n

CONTROL PARAMETERS : event ≡ number-of-events-so-far, number-of-events ≡ final-event

REAL p, q, x, y

GENERAL VARIABLES : x, y, p ≡ sum-of-squared-differences, q ≡ sum-of-squared-differences-of-squared-differences

INTEGER c

INTEGRAL VARIABLE : century

INTEGER ARRAY : Y[1:1000]

INTEGER LIST : YEAR for each event from 1 up to 1000

ROUTINE FORMATS

add update [INTEGER VARIABLE y]

Update the century or add it to the [INTEGRAL ITEM year].

check year [REAL EXPRESSION f]

Check that the new year is within reasonable range of the previous years (factor of [NUMBER f]).

FUNCTION FORMAT

[REAL a] = sqdf ([REAL a],[REAL b])

[RESULT a] = the squared-difference between [NUMBER a] and [NUMBER b]

MAIN PROGRAM

read n

Read the first number, the 'number of events'.

e = 1

Start with the 1st event.

1: read Y[e]

Read the next number, the YEAR of the event.

add update Y[e]

Update the century or add it to the YEAR of the event.

check year 4sqrt(e)

Check the new year is within reasonable range of the previous years (factor of 4 TIMES the square-root of the number of events so far).

->2 if e = n

Go to (2) if the event was the final event.

e = e + 1; -> 1

Otherwise, consider the next event and return to (1).

2: p = sqdf(y[2],Y[1])

Set p = the squared-difference between the YEAR of the 2nd event and the YEAR of the 1st event.

q = 0

Set q = 0.

cycle e = 3,1,n

REPEAT for each event from the 3rd up to the final-event.

x = sqdf(Y[e],Y[e-1])

Set x = the squared-difference between the YEAR of the current event and the YEAR of the previous event.

y = sqdf(Y[e-1],Y[e-2])

Set y = the squared-difference between the YEAR of the previous event and the YEAR of the last but one event.

p = p + x

Add x to p.

q = q + sqdf(x,y)

Add the squared-difference between x and y to q.

repeat

When each event has been dealt with.

print (Y[n]-Y[1])/n

Print the YEAR of the final-event – the YEAR of the 1st event DIVIDED-BY the number-of-events.

print sqrt (p)/(n-1)

Print the square-root of the number of squared differences DIVIDED-BY 1-less than the number-of-events.

print sqrt (q)/(n-2)

Print the square-root of the sum-of-squared-differences-of-squared-differences DIVIDED-BY 2-less than the number-of-events

stop

END the PROGRAM

ROUTINE

Add update [INTEGER VARIABLE y]   Update the century or add it to the
                                       [INTERGAL ITEM year].

  -> 1 <u>unless</u> y < 100   Go to (1) unless the year is below 100.

  <u>Y = y + c</u>   Add the century to the year.

  <u>return</u>   FINISH.

1: c = 100<u>intpt</u>(y/100)   Set the century to 100 **TIMES** the <u>integral-part</u>
                                       of the year / 100.

ROUTINE

Check year [REAL EXPRESSION f]   Check tape the new year is within
                                           reasonable range of the previous years
                                           (factor of [NUMBER factor]).

  -> 1 <u>if</u> e = 1   Go to (1) if the event is the 1st event.

  -> 3 <u>unless</u> Y[e] > Y[e-1]   Go to (3) unless the YEAR of the <u>current</u> event
                                       is after the YEAR of the previous event.

  -> 1 <u>if</u> e = 2   Go to (1) if the event is the 2nd event.

  -> 2 <u>unless</u>   (Y[e]-Y[e-1]) > f(Y[e-1]-Y[e-2])   Go to (2) if the YEAR of the
                                       <u>current</u> event – the YEAR of the
                                       <u>previous</u> event is more than this
                                       <u>FACTOR</u> times the YEAR of the
                                       previous event – the year of the
                                       <u>last-but-one</u> event.

1: <u>return</u>   FINISH.

2: <u>print</u> Y[e-2]   Print the YEAR of the <u>last-but-one</u> event.

3: <u>print</u> Y[e-1]   Print the YEAR of the <u>previous</u> event.

  <u>print</u> Y[e]   Print the YEAR of the <u>current</u> event.

  <u>caption</u> INCONSISTENT DATA   Print 'INCONSISTENT DATA'.

  <u>stop</u>   END the PROGRAM

FUNCTION

[REAL s] = <u>sqdf</u>([REAL a],[REAL b])   [RESULT s] = the <u>squared-difference</u> between
                                       [NUMBER a] and [NUMBER b].

  s = (a-b)(a-b)   Set s = the <u>square</u> of a – b.

END OF PROGRAM

14
1894
1897
1910
1914
15
17
19
1925
1935
1940
42
45
1952
1965

The main features of an autocode

When a program is being executed, the utilisation of the computer store can be divided into two :

i)   The instruction space. This is an area of the store which contains the machine code instructions of the program. Both the extent of this area and the contents of it usually remain fixed throughout the duration of the program run.

ii) The data space (or 'stack'). This is a separate area of the store which contains the data on which the instructions operate. Both the extent of this area and its contents tend to vary during the execution of the program.

The most important function of an autocode is to deal automatically with the problem of store allocation and reference both for the instruction apace and for the data space. The instructions of the language can again be divided into two classes:

i)   Declarations. These give the compiler information about the structure of the program and its data,

ii) Imperatives. These describe the actual sequence of operations to be carried out on the data of the program in order to execute the required task.

Generally speaking, the compiler does not compile any instructions in the place of declarations, but merely carries out behind-the-scenes operations, making decisions about the organisation of the program and its data, and building up lists of information about it (which are held in the data space of the compiler program). Then for imperative instructions the compiler generates the appropriate machine code to be added to the object program using the information lists to translate from the objects referred to in the source language description into the coded representation of the corresponding numbers or store addresses in the machine.

Note that in this example the compiler is assumed to be a 'one-pass' system, i.e. it compiles the complete instruction as it reads it in on input, and therefore all the requisite information must have been declared before the instruction (in general -- there is an exception in this example in the case of numerical labels).

Declarations

Store allocation There are usually a number of variables used in a program and they have to be allocated storage in a consistent manner. Also variables may be of different 'types', for example, 'integer', 'index' (special form of integer used in modification), 'real' (general floating point number), and 'complex'.

Therefore DATA TYPE declarations have to be made, e.g. INDEX e, n, and REAL p,q, x,y, which tell the compiler to reserve suitable store locations for variables of the appropriate type in the data space of the object program.

A variable is characterized by four main properties :

1). name (or 'identifier') : the symbol of symbol string used to represent the variable in the source program description, e.g. 'e', 'n', 'p' and 'q' (or in another language e.g. 'event' or 'ab4', etc.

2) address : the address of the store location or locations) that has been allocated allocated to the variable, to hold its value. Sots that for example a complex number will be held in two locations to a real's one.

3) value : the current contents of the store location(s) allocated to the variable. This location is usually left empty when compiling, and then it is set and periodically reset to new values when the program is being obeyed.

4) type : the interpretation to be put on the content of the allocated address whenever the variable is referred to. The significance of the type is that variables of different types are held in the store in different ways, and therefore different machine code instructions have to be used for different types to carry out the same operation (e.g. the addition of two like variables), Furthermore, there are restrictions on the possible transformations between variables of different types (e.g. it is not in general possible to set an integer equal to a complex number),

Having recorded the name, address, and type for each variable (in its own data space) the compiler will then know whenever a variable's name is used in an imperative instruction (implying a reference to its value during program execution) where in store it been allocated and how to interpret it.

It is also convenient to have objects which are lists of variables of a certain type, and whose individual elements can (only) be referred to by a double reference ('indexing' or 'modification'). For example INTEGER ARRAY Y [1:1000] defines an array with name 'Y', the 'i'th element of which can be referred to as the variable 'Y[1]'. The declaration gives the name, type, and extent of the array, so that the compiler as well as allocating the array a suitable store address knows how much store to allocate it.

Instruction labels It is often required to refer to one imperative instruction in another instruction, Therefore imperative instructions can be labelled, e.g. 1: or 2:, and they can then be referred to by these numbers,

Division of program into routines A program is usually split up for convenience into a top-level routine (the MAIN PROGRAM) and a set of subroutines, Each routine definition gives the name of the routine followed by the set of instructions associated with this name. A routine can then be called from inside another routine or the main program by an imperative instruction which gives its name, e.g. add update Y[e], or check year 4sqrt(e). During the execution of the program such a 'routine call' will cause the instructions given in the routine definition to be obeyed.

Each routine has an organisation that is mostly independent of the rest of the program, and routines can usually be written, and assembled in the source

program text, in any order.

For example a routine has its own labelling system, and it can declare its own local data space. The effect of a <u>local declaration</u> of data, e.g. (not in the specimen program) REAL a, b, is that two new locations are set up associated with the routine that can be referred to inside the routine as 'a' and 'b'; inside the routine all references to 'a' and 'b' are taken as referring to these local variables and not to any other 'a' or 'b' that may have been declared elsewhere in the program. The compiler will probably arrange to share the space it has allocated to local data of the routine with local space of other routines; so there is no guarantee that values of local variables will be preserved in between successive calls on the routine.

A routine may be created because the same set of instructions is to be obeyed in many different parts of the program, and it is therefore more compact to only state the set once, and then to just use the routine name at all these points. Or a routine may be created for a set of instructions that is only called once, simply for the convenience of splitting the program into manageable units and/or using the local declaration facilities.

[In the case where a unique set of instructions requires some local organisation, but the programmer does not wish to write them separately from the routine they are contained in, he can enclose them in a 'block', e.g. between declaration <u>BEGIN</u> (followed by any local declarations) and <u>END</u>. An example of this is not given in the specimen program.]

In this example of an autocode, the declarations of <u>MAIN PROGRAM</u> and <u>ROUTINE</u>, and the special form of <u>FUNCTION</u> routine, specify the way in which the program has been broken up into routines. Since a routine may be called before the routine definition has been given, declarations of <u>ROUTINE FORMATS</u> and <u>FUNCTION FORMATS</u> can be made at the beginning, in order to give the compiler sufficient information to recognise and deal with any routine call it meets before the routine definition has been given.

The implementation of routines and functions will be described later.

The declaration <u>END OF PROGRAM</u> specifies the end of the source program text. Most programs will have some 'input data' which will follow the program text. This will be read in under the control of the object program when it is being obeyed -- whereas of course the program text is read in by the compiler.

Imperative instructions

<u>The Assignment instruction</u> The key imperative instruction is the assignment statement, in which a new value is assigned to a variable. The value can be a constant, e.g. e = 1, or Y[e] = 25, or the value of a variable e.g. e = n, or c = Y[e], or it may be the value of a 'function', a special form of routine to be described later, e.g. x = <u>sqrt</u>(p).

In general, the value to be assigned to a variable is the value of an '<u>expression</u>', which is a complex of operators and operands set out in the

notation of algebraic formulae, e.g. e + 1, or (Y[n]-Y[n-1])/n, or 100<u>intpt</u>(y/100). In this case the compiler decides from the syntax of the expression in what order and with what operations to calculate the final value in terms of the values of the individual operands. Then it assigns this value to the variable, e.g. setting e = e + 1, or Y[n+i] = (Y[n]-Y[n-l])/n, or c = 100<u>intpt</u>(y/100).

When compiling the requisite machine code the compiler gets the address and type of each variable involved in the assignment from the information set by the data-type declarations. It will also check that the value assigned to the variable is consistent with its type, for example i = b where 'i' is an index and 'b' is complex will be taken as an error,

<u>Control instructions</u> Imperative instructions are obeyed in sequence until a control instruction breaks the sequence, such an instruction is the jump instruction, e.g. -> 1, which causes control to be switched to the correspondingly labelled instruction. More useful are the conditional jump instructions, e.g. -> 2 <u>if</u> e = n, or -> 3 <u>unless</u> Y[e] > Y[e-1], where two values are compared and control only passed to the labelled instruction if a specified condition holds.

Then there are control instructions which link up directly with the general organisation of the program, e.g. <u>stop</u> which means 'finish executing the program', and <u>return</u> which means 'finish' executing this routine and return control to the first instruction after the particular call that has caused entry into this routine.

A more sophisticated control instruction is the '<u>cycle</u>' instruction, This automatically organises the requisite control for the case where a set of instructions has to be obeyed repeatedly in a loop, The extent of the set of instructions is given by the matching instruction <u>repeat</u> following the <u>cycle</u>, Note that the labels 1: and 2: in the main program could have been avoided by writing the sequence :

<u>cycle</u> e = 1,1,n; <u>read</u> Y[e]; <u>add update</u> Y[e]; <u>check year</u> 4<u>sqrt</u>(e); <u>repeat</u>

<u>Routine calls</u> Routines of the program are given names in a consistent manner and they can be called by writing these names as imperative instructions at the appropriate points of the program., During execution the set of instructions defined for the routine will then be obeyed at each such point.

<u>Permanent routines</u> It is convenient for some of the instructions of the autocode to be implemented in the same way as the programmer's routines, For consistency calls on such 'permanent' routines are described in the same manner (format) as programmers' routines.

Examples of permanent routines are the key <u>input instruction</u>, e.g. <u>read</u> n, or <u>read</u> Y[e], which reads the next number of the input data and sets the specified variable to its value, and the key <u>output instruction</u>, e.g. <u>print</u> (Y[n]-Y[1])/n, which prints the value of the specified expression as the next piece of output data. The other permanent routine shown, i.e. <u>caption</u> INCONSISTENT DATA, has an

unconventional format; it prints the symbols specified as the next piece of output data.

Implementation of routines

Cue – Subroutine mechanism A routine allows a set of instructions to be called from many places in the program by using just a routine name. This source program convenience is reflected in the object code in that the set of machine code instructions corresponding to the routine is compiled only once, and it exists as a machine code 'subroutine' in the instruction space. In the case of a programmer's routine, the subroutine will tend to occupy the same space relative to the rest of the program as it does in the source text. For permanent routines the appropriate subroutine is automatically included in the object code by the compiler.

Whenever a routine call is met a 'cue' is compiled at this point. It is the job of this set of machine code instructions to pass control to the subroutine. Also, since the same set of instructions of the subroutine has to be obeyed for all calls on the routine, the cue must carry out any operations that are particular to this call. Most important it has to tell the subroutine to which instruction it must pass control on return. Before jumping to the subroutine it therefore plants the requisite address (a 'link') in a special location associated with the subroutine. Then whenever 'return' is specified in the routine, and after the last instruction, machine code instructions are planted to pick up this link and transfer control back to the particular cue which has called it.

Parameters It is convenient to allow parameters in routine calls, for example the permanent routine 'print…' could have been named just 'print' and it could have arranged always to print out the value of the nonlocal variable 'x' say. In practice this would mean that most print instructions would be preceded by a resetting of 'x', e.g. the print sequence in the main program would be written

    x = (Y[n]-Y[1])/n; print

    x = sqrt(p)/(n-1); print

    x = sqrt(q)/(n-2); print;

But this is tedious, and it would also mean that the programmer could not use 'x' as an ordinary variable in safety as its value would be interfered with whenever he printed a number. So it is convenient to include the expression whose value is to be printed in the routine call, and use a location local to the routine to hold its value.

Note that the two routines used in the specimen program are only called once and so do not require parameters, They could equally well have been defined with just the names 'add update' and 'check year', provided that the instructions defined for the routines had been altered as follows :

    for add update replace each use of 'y' by 'Y[e]'

    for check year replace each use of 'f' by '4sqrt(e)'

However if the program had been more complicated it might have been necessary to call the routines to carry out the same general operations with different variables and values respectively, and so the parametric form would have been more suitable. Note that in fact the function <u>sqdf</u> is called 4 times with different values each time.

<u>Parameter specification</u> The notation for specifying parameters is that the routine format starts with the routine name (e.g. add update, check year) and then follows the specification of the '<u>formal parameter</u>' (e.g. [INTEGER VARIABLE y] and [REAL EXPRESSION f)). If there is more than one parameter they are separated by commas. The specification of a formal parameter comprises the characteristics of the parameter followed by the name by which it will be referred to inside the routine, e.g. 'y' and 'f'. Then every routine call must start with the same routine name and follow it with an '<u>actual parameter</u>' consistent with the characteristics specified for the formal parameter (e.g. Y[e] is an integer variable, and 4sqrt(e) is a real expression). Where there is more than one parameter, the actual parameters are in the same order, separated by commas, and must match the corresponding specifications.

The simple theory of routines with parameters is the '<u>substitution model</u>' that whenever there is a routine call the effect is as if the instructions of the routine had been written in its place with each reference to a formal parameter name replaced by the corresponding actual parameter. Thus <u>add update</u> Y[e] has the same effect as if instead had been written :

        -> 1* unless Y[e] < 100

        Y[e] = Y[e] + c

        -> 2*

    1*: c = 100<u>intpt</u>(Y[e]/100)

    2*: ....

where the labels 1*: and 2*: are independent of the labelling system of the main program, and 2*: is introduced to label the instruction to be obeyed on return, i.e. the next instruction after the call.

However this substitution effect has to be implemented in such a way that the main body of instructions involved, i.e. the subroutine, has a fixed code which is independent of the actual parameters. Whenever a formal parameter is referred to in the routine the code compiled must deal with all actual parameters in the same way. Therefore it is an additional job of the cue to transform each actual parameter into a form that is common to all calls on the corresponding formal parameter.

Leaving aside more sophisticated types of parameters (arrays, routines, and functions) the commonest parameters deal with the simple data objects used by the basic assignment statement, i.e. a 'variable' and a 'value'. If a routine requires to reset the value of a parameter, then each actual parameter must obviously be a variable. Therefore, the syntax associated with the formal parameter

is that of a variable. If the routine only requires the current value of parameter, then the syntax associated with each actual parameter can be an expression. Although of course an actual expression can be just a variable, an assumption is made in this specimen autocode that VARIABLE in a formal parameter specification means that the parameter is to be reset, and EXPRESSION menus that it is just a 'value', which will not be reset.

Different implementations are used to deal with different situations that can arise in implementing parameter calls using a cue-subroutine mechanism. Five of these are listed below, but the last three, which involve the concept of 'side-effects', are given as notes, in each case all the actual parameters matching a formal parameter must be of a consistent type, therefore the data type associated with the formal parameter is included in the specification.

i) <u>Calling an expression by value</u> E.g. [REAL EXPRESSION f]

This is achieved by setting up a local variable, effectively REAL to be associated with the formal parameter, Then each cue calculates the value of its actual expression and puts the value in the location allocated to 'f', Then each reference to 'f' in the routine is taken as a reference to this local variable, i.e. to the value of the actual expression.

Note however that if 'f' is implemented as a local variable (and not a 'local expression' or a constant) an autocode will usually allow 'f' to be reset (thus contradicting the simple substitution model of the routine mechanism). Of course where 'f' is reset the value of the actual expression will be destroyed.

ii) <u>Calling a variable by value</u> E.g. (INTEGER VARIABLE y]

This is implemented in a similar way to (i). A local variable, e.g. INTEGER y, is set up and the cue copies the value of the actual variable, e.g. Y[e], into it. Note however that if 'y' is reset in the routine this will not reset the actual variable, it will only reset the value of the local variable 'y'. Therefore In the case of calling a variable by value the cue arranges that on return from the subroutine the final value of the associated local variable is copied back to reset the actual variable,

iii) <u>Calling a variable by reference</u> E.g. [REAL VARIABLE REFERENCE y]

The implementation of (ii) leaves open the possibility that somewhere in the routine (or in a routine called in it) some of the actual variables of different calls are referred to direct**ly** by their actual name, e.g. Y[e], and not by their common formal name 'y'. If while a particular call on a routine is being obeyed such an actual variable is reset and the formal parameter referred to afterwards, or if the formal parameter is reset and the actual variable referred to afterwards, then at the time of the second reference the actual name and the formal name will be referring to two different values.

Such a '<u>side-effect</u>' can be avoided by arranging for the subroutine to deal direct**ly** with the actual variable each time reference is made to the formal parameter, and not just with a local copy. This can be achieved by setting up a type of local variable, say REAL REFERENCE y. which holds the <u>address</u> of a real variable, and not just the value of the variable itself. The cue passes on the address of the actual variable and not its value, and each

reference to the formal parameter e.g. 'y' is made via this address, i.e. direct to the actual variable. Thus the compiler has to remember that the value of 'y' is not the usual 'contents' of the store location with address associated with name 'y' but the 'contents of the store location whose address is given by the contents of the store location with address associated with y'.

iv) <u>Calling an expression by substitution</u> E.g. INTEGER EXPRESSION NAME x]

Side-effects similar to those described in (iii) can equally happen when calling an expression. It could happen that some of the variables contained in some of the actual expressions matching a formal parameter are referred to by their actual names in the routine (or a routine it calls). And it could therefore happen that the value of the expression at the time of a reference to the formal parameter, e.g. 'x' (as calculated from the current values of the operands) was different from the value calculated in the cue on entry to the subroutine. If the programmer knows that this cannot happen, or if he knows that it may but he still wishes to always refer to the entry value, then he can use implementation (i)

Otherwise, to satisfy the substitution model of a routine with parameters in all cases, it is desirable that the value of the expression should be recalculated on each reference to the formal parameter. In this case two locations are reserved for use by the formal parameter. The cue compiles a calculation of the expression as a 'secondary subroutine' which places the value in the first of these locations. But it does not <u>obey</u> the calculation before entering the subroutine as happens in (1), but it merely passes on (to the second location) the address of the secondary subroutine. Then whenever the formal parameter is referred to in the routine, control is first passed back to the secondary subroutine inside the cue, which calculates the up-to-date value, and then this value is picked up from the first location as required in the usual way.

v) <u>Calling a variable by substitution</u> E.g. [REAL VARIABLE NAME y]

Note that implementation (iii) is still not completely general. It is if all actual parameters corresponding to a formal parameter are simple variables, but if some are array elements, then it is possible for the index to be altered by direct reference to a variable involved, e.g. to 'e' for actual variable Y[e]. In this case complete generality is achieved by compiling a secondary subroutine in the cue which calculates the value of the <u>address</u> of the variable and passes it on to the subroutine, On entry the cue only passes on the address of the secondary subroutine, and each reference to the formal parameter is carried out by transferring control to the secondary subroutine as in (iv), which produces an up-to-date address through which the actual variable is accessed as in (iii).

NOTE : Implementation (iii) is the call-by-simple-name of AA, which does not have implementations of type (iv) and (v); the Call-by-name of Algol is implemented as in (iv) and (v) to achieve the full generality of substitution.

<u>Functions</u>

There is an important special case of the routine mechanism where it is convenient to specify by a calculation a value which is to be an operand in an expression, e.g. the permanent functions <u>sqrt</u>(p), and <u>intpt</u>(y/100). In this case the value of an operand is found by passing control to a subroutine associated with the function routine (as for an ordinary routine), and the function subroutine will pass back the required value, A function can have parameters, but (in this example of an autocode) they must be enclosed in round brackets; also, since it is not usually sensible to wish to reset an actual variable, EXPRESSION is assumed in each parameter specification and so it can be left out. Functions can only be called as operands in the calculation of an expression (i.e. a value), and the main difference between a function and a routine definition is that a parameter must be specified for the function to

pass back the result to the function-cue; also, this parameter must have a specified data-type (which is therefore a characteristic of the function as a whole). In the specimen autocode these two features are specified in the function format by preceding the function name with a specification of this 'result parameter', e.g. '[REAL s] = ...'. (Note that again no variable/expression characteristic is required, but this time the assumption is that a local variable, e.g. REAL s, is set up whose final value in used by the function-cue — there is no presetting of the result parameter on entry.)

Conventions of notation in the specimen program

Various conventions of notation are used in this example of a program (and of an autocode) and in the covering description :

'Underlined Capital Words' on a line of their own head the major declarations;

'Capital Words' are used for data-type and parameter characteristics;

'Capital Letters' are used for array names (e.g. Y);

'Small Letters' are used for simple variable names;

'Underlined Small Words' are used for routine and function names, and for words that are part of the syntax of basic instructions, e.g. if —- note that this underlining is used to avoid confusion with strings of small letters which are simple variables under implicit multiplication;

Square Brackets are used to enclose parameter specifications and array indices;

Round Brackets are used to enclose sub-expressions that are operands of expressions, and to enclose the set of parameters in a function;

';' or a 'new-line' are used to separate instructions.

In the English covering description, conventions differing from these are:
'Small Words' (maybe hyphenated) are used for simple variable names or for parts of a routine or function name;
A Capital Word can also be an array name, and is used to begin control instructions that do not involve explicit label references;
An Underlined Capital Word inside an instruction is an 'operator';
The full range of punctuation is used to separate instructions.

In addition, note the trivial functions used in the covering description but not the program, e.g. previous event ≡ event – 1, current event ≡ event, etc., with obvious meanings.

Ch. 2   PHRASE STRUCTURE NOTATION

Formats

      In order that a programmer can use an autocode he must be given a description of the model (of a computer) constructed for the autocode, a specification of the permitted language, and a description of what operations the language carries out in terms of the model. In the example of the previous chapter a number of types of instruction were categorised according to their function, and examples given of each. From this information an experienced programmer could probably write a simple program in the language, but clearly this description does not adequately specify the autocode,

      In fact the specification of the language is a set of instruction classes, each one with its own particular 'format'; that is, all instructions belonging to a class must satisfy the same syntactical (grammatical) rules. All instructions of a class carry out the same general operation in the context of the autocode model. The set of instruction classes specify the number of different kinds of operation that can be done. Any job required to be done using the autocode must be described using only this set of instructions.

      Thus in a slightly simpler language the only way to do a conditional jump in control might be to use an instruction class with format :

    FORMAT :   -> [LABEL]if[EXPRESSION][= or >][EXPRESSION]

      e.g. '-> 5 if p = 45' or '-> 11 if 5(a+b) > 2xy'

Here any integer could be written in the position of {LABEL] and any two mathematical expressions could be written in place of the [EXPRESSION]s, but the symbols '->', 'if', and '=' or '>' would always have to occur and in their correct place relative to the rest of the sentence. The compiler would interpret this instruction as follows : the two mathematical expressions are to be calculated and their values compared as indicated by the comparison symbol [= or >]; if the test is satisfied, a jump in control must be made to the appropriate instruction in the program, labelled by the same integer; otherwise, control passes to the next instruction as usual.

      It is clear that when talking about instruction classes which carry out a particular generalised task we are forced into the use of class words in 'label', 'variable', 'expression', or 'comparison symbol'. Their interpretations with respect to the autocode model and language are usually independent of the particular instruction class they are being used in, so their definitions can be given once and for all as part of the general description, However it is convenient when describing general operations, and it is necessary when describing formats, to use a clear notation to distinguish between what is a class word and what is an actual object or symbol.

      In the above example the notation used is that a class word is represented by a symbol string (preferably a word in capitals) enclosed in square brackets. With this convention the definition of the permitted format is clear and unambig-

uous once the definitions of the class words have been given. The form of every member of the instruction class is that the string of symbols making up the particular instruction must match precisely the sequence of symbols given in the format; except that at the points where a class word occurs, any symbol string that is a member of this class can occur. The next symbol after a string matching a class word must of course be the same as the next symbol after the class word in the format; or if the class word is followed by a second class word, the next symbol must be the first symbol of a string matching the second class word.

Thus (ignoring spaces) the first symbol of a conditional jump instruction must be a '−' the 2nd a '>', and the 3rd and maybe the next one or two must be a digit; then must come '$\underline{i}$' and then '$\underline{f}$', then an expression (e.g. '45', 'p', '2xy', or '5(a+b)'), then the comparison symbol '=', or '>', and finally another [EXPRESSION].

Therefore the precise specification of a simple language can be given by a set of class word definitions to describe the sub−units of the language, and a set of format definitions, to describe the set of instruction classes, the basic units of the language.

Class word (PHRASE) definitions

A '$\underline{class\ word}$' is a word of the language description language (the 'meta-language') that represents a specific t set of symbol strings in the language being described (the 'source language'). Each such string is called a '$\underline{basic}$ $\underline{phrase}$', i.e. a single source symbol or a string of source symbols.

A '$\underline{phrase\ definition}$' defines a class word and its associated set of basic phrases. Thus :

PHRASE [$\underline{if}$,$\underline{unless}$] = $\underline{if}$, $\underline{unless}$

PHRASE (COMPARISON−SYMBOL) = =, ≠, ≥, <, >, ≤

This says that (only) the symbol strings '$\underline{if}$' or '$\underline{unless}$' are a permissible substitution for the class word [$\underline{if}$,$\underline{unless}$], and that (only) either of the 6 symbols =, ≠, ≥, <, >, ≤ is a permissible substitution for [COMPARISON−SYMBOL].

Thus we could give it slightly more general format for the conditional jump instruction :

FORMAT [LABEL][$\underline{if}$,$\underline{unless}$][EXPRESSION][COMPARISON−SYMBOL][EXPRESSION]

This gives 2 × 6 different ways of expressing a condition instead of the two basic ones '$\underline{if}$ ... =', and '$\underline{if}$ …>'.

The commas in the phrase definition separate the alternative phrases (if any). However the phrases need not be basic phrases. A (non−basic) '$\underline{phrase}$' is a set of basic phrases defined by : a single source symbol, or a class word representing a set of basic phrases, or a combination of symbols and/or class words. (Thus a format is a special case of a phrase, where the set of basic phrases is the set of instructions that are members of an instruction class in a language.)

```
    E.g. :

    PHRASE [EQUALS] = equals, [IS][EQUAL-TO],[IS]

    PHRASE [DOES-NOT-EQUAL] = does not equal, [IS] not [EQUAL-TO], [IS] not

    PHRASE [IS] = is, was, are, were, will be

    PHRASE [EQUAL-TO] = equal to, same as, at
```

These specify 21 permissible basic phrases which satisfy the class word [EQUALS]. That is, 1 from the first alternative phrase 'equals', 5 x 3 = 15 from the 2nd alternative phrase [IS] [EQUAL-TO] (is equal to, is same as, is at, was equal to, was same as, was at, are equal to, ..., were equal to, ..., will be equal to, ...), and 5 from the 3rd. alternative [IS] (is, was, are, were, will be). There are 21 similar basic phrases satisfying [DOES-NOT-EQUAL], (does not equal, is not equal to, is not same as, is not at, was not equal to, ..., is not, are not, ...).

Thus if it was required to specify the permissible ways of saying '=' (or ≠) in a conditional instruction in the English covering description, the phrase [EQUALS] would provide flexible alternatives to match the context of the sentence while still retaining a rigid and unambiguous specification of whet must occur at that point in the sentence to satisfy the syntax of the format.

These definitions could be extended to cover the full range of comparisons as follows :

```
PHRASE [COMPARES-WITH] = [COMPARISON-SYMBOL] [IS] not [COMPARED-WITH], [IS]
                not, [IS] [COMPAPED-WITH], [IS]
PHRASE [COMPARED-WITH] = [GREATER-THAN] or [EQUAL-TO], [LESS-THAN] or [EQUAL-
                TO], [GREATER-THAN], [LESS-THAN], [EQUAL-TO]
PHRASE [GREATER-THAN] = more than, greater than, after, above, over
PHRASE [LESS-THAN] = less than, smaller than, before, below, under
PHRASE [COMPARISON-SYMBOL] = =, =, ≠, ≥, <, >, ≤
PHRASE [IS] = is, was, are, were, will be
PHRASE [EQUAL-TO] =    equal to, same as, at
```

Thus where c is the number of alternative basic phrases of [COMPARISON-SYMBOL] i for [IS], g for [GREATER-THAN], l for [LESS-THAN], and e for [EQUAL-TO], the number of alternative basic ways of writing (COMPARES-WITH) under this

$$c + i(ge+le+g+l+e) + i + i(ge+le+g+l+e) + i = 446$$

For example one of the alternatives is 'are not below or equal to'. And the analysis of this basic phrase with respect to [COMPARES-WITH] can be represented as shown. Note that each digit specifies the 'category-number' of the alternative phrase (below it) in the definition of the class word (above).

```
    [COMPARES-WITH]
          |
       2 |
          |
  [IS] not [COMPARED-WITH]
    |           |
  3 |         2 |
    |           |
are  [LESS-THAN]  or [EQUAL-TO]
       |                |
     4 |              1 |
       |                |
    below           equal to
```
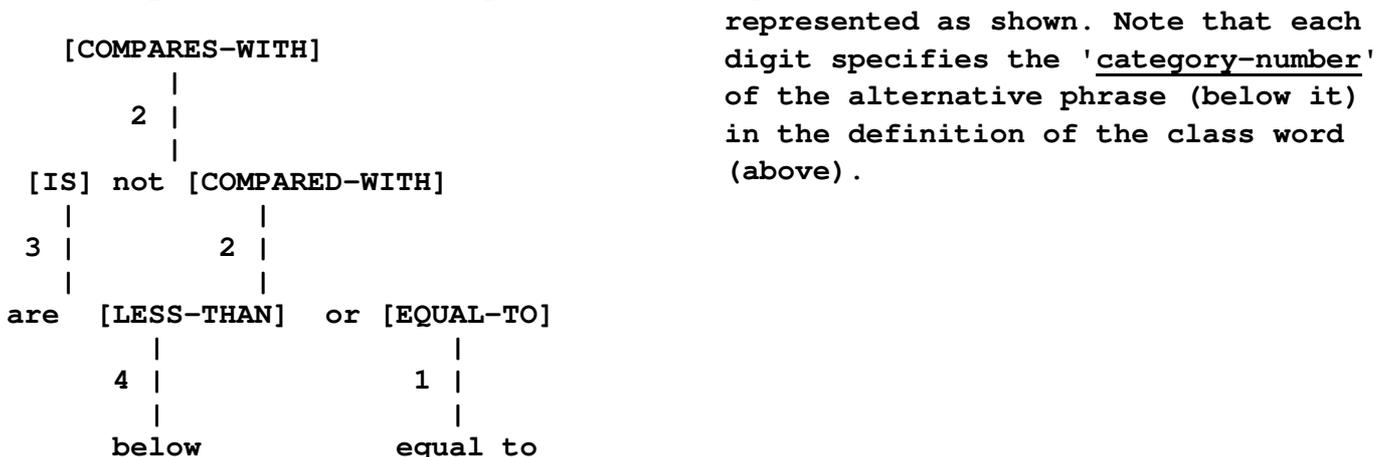
The phrase definitions that have been given so far are simple enough. But to give precision to definitions there are a number of small points to observe, and to achieve the requisite power in defining a language there are a few convenient extensions to the conventions of phrase definitions :

Order of preference

Consider the phrase definition [EQUALS] = equals, [IS] [EQUAL-TO], [IS]. The last two alternatives have the same stem '[IS]'. There is a convention that if one alternative of a class word also occurs as the beginning of another, it is written after it. This is because we make the convention that when matching alternatives of a class word to the head of a symbol string, the alternatives are taken in order from left to right, and as soon as one has been found to match the head of the string we assume that this is the required match for the class word as a whole; we then go on to match the head of the remainder of the source string to the next symbol or class word in the phrase or format in which the class word appeared. So if the stem occurs before the longer alternative the latter can not be recognised,

Consider the example of matching

'go to (3) if p is equal to a(b-c)'

to a conditional jump instruction of the English covering description :

go to ([LABEL]) [if,unless] [EXPRESSION] [EQUALS] [EXPRESSION]

(assuming now that spaces are not being ignored and that an expression cannot contain spaces). When we have got as far as matching 'if' to (if,unless) and 'p' to [EXPRESSION] we then try and match the head of 'is equal to a(b-c)' against (EQUALS). If [IS] occurs before [IS] [EQUAL-TO], than we would match 'is' to [IS] and hence to [EQUALS], and then go on to try and match 'equal to a(b-c)' to (EXPRESSION). Depending on the precise definition of an [EXPRESSION], we would either not recognise it, or, say, match it to 'equal', thus interpreting it as e.q.u.a.l, where '.' denotes multiplication, At the best (nonrecognition) this would be time-wasting – in the case where it was possible on nonrecognition to return to [EQUALS] and pick up trying to match further alternatives of it to 'is equal to a(b-c)' – but note that this implies that each time we recognise an alternative phrase we have to remember which alternative it was and what the head of the symbol string was. At the worst (recognition) this occurrence would cause an unnecessary ambiguity : for even though 'equal' could be interpreted as an expression, it is quite obvious that 'equal to' occurring after an 'is' that is matching (EQUALS) in as sentence is part of [EQUALS].

Repeated phrases (*)

Consider the data-type declaration :   [TYPE][LIST-OF-ELEMENTS]

Where PHRASE [TYPE] = REAL, INDEX, INTEGER, COMPLEX

E.g., 'REAL p, q', or 'INDEX a,b, x,y', or just 'INDEX r'.

Informally a [LIST-OF-ELEMENTS] is defined as a list of names of single

data elements (as opposed to array elements), separated by commas if there is more than one; an (ELEMENT] name is any smell letter.

More formally :

PHRASE [ELEMENT] = a,b,c,d,e,f,g,,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z

PHRASE [LIST-OF-ELEMENTS] = [ELEMENT][FURTHER-ELEMENTS], [ELEMENT]

PHRASE [FURTHER-ELEMENT] = ,[ELEMENT]

This says that a [LIST-OF-ELEMENTS] is either a single [ELEMENT] followed by a number of repetitions of [FURTHER-ELEMENT]s, or just a single [ELEMENT]; where a [FURTHER-ELEMENT] is a comma followed by another [ELEMENT]. Note however that we have not formally defined [FURTHER-ELEMENTS] only a [FURTHER-ELEMENT].

For example

In 'REAL p, q' the first [ELEMENT] is 'p' and there is one [FURTHER-ELEMENT] ',q';

In 'INDEX a,b x,y' the first [ELEMENT] is 'a' and there are 3
                                      [FURTHER-ELEMENT]s ',b' ',x' and ',y'

In 'INDEX r' there is one [ELEMENT] 'r' and no [FURTHER-ELEMENT]s.

The situation where we wish to specify any number of repetitions of a particular class word in a phrase is so common that a special convention is adopted to cover this case :

'*' convention if there is an asterisk '*' before the right-hand square bracket in a class name, this means that a permissible substitution for the class word is any phrase of the (de-asterisked) class word maybe followed by any number of repetitions of phrases of the class word.

Thus for [FURTHER-ELEMENTS] in the above definition we could write [FURTHER-ELEMENT*] and so achieve a formal definition.

Recursion

The formal definition of the '*' convention is

PHRASE [SOME-CLASS-WORD*] = [SOME-CLASS-WORD][SOME-CLASS-WORD*], [SOME-CLASS-WORD]
i.e. any number of repetitions of a class word is formally defined as the class word followed by any number of repetitions of the class word, or just the class word. This is a simple example of 'recursion'.

In general : it is permissible to include a class word inside one or more alternatives of its definition, provided that it is not at the beginning of the alternative, and provided that there is a possibility of the recursion stopping at some point. Note that PHRASE [SOME-CLASS-WORD] = [SOME-CLASS-WORD] is meaningless, and that PHRASE [SOME-CLASS-WORD] = a[SOME-CLASS-WORD], b[SOME-CLASS-WORD] could only 'recognise' an infinite string of 'a's and 'b's,

Thus we could have formally defined

PHRASE [LIST-OF-ELEMENTS] = [ELEMENT][FURTHER-ELEMENTS], [ELEMENT]

PHRASE [FURTHER-ELEMENTS] = [,][ELEMENT][FURTHER-ELEMENTS], [,][ELEMENT]

where [,] indicates a source symbol comma as opposed to a comma used to separate alternatives of the definition.

A general example of recursion will be given later in the definition of [EXPRESSION].

Optional phrase (?)

This last definition of a [LIST-OF-ELEMENTS] and the original one are not very satisfactory. It would be preferable to factor out the [ELEMENT] from the alternatives of the definition, and say :

PHRASE [LIST-OF-ELEMENTS] = [ELEMENT][ANY-FURTHER-ELEMENTS]

or PHRASE [LIST-OF-ELEMENTS] = [ELEMENT][FURTHER-ELEMENTS?]

or PHRASE [LIST-OF-ELEMENTS] = [ELEMENT][FURTHER-ELEMENT*?]

Here the last two definitions obviously mean an [ELEMENT] maybe followed by [FURTHER-ELEMENT]s, i.e. followed by [FURTHER-ELEMENT]s or nothing. And their meaning in terms of [FURTHER-ELEMENT] or [FURTHER-ELEMENT*] is clear.

This again is a common enough and clear enough case for a convention :

<u>'?' convention</u> If there is a query '?' before the ']' in a class word, the definition of e.g. [SOME-CLASS-WORD?] is taken to be either [SOME-CLASS-WORD] or 'nothing', where 'nothing' implies that the class word is automatically recognised without matching any symbol to it.

NIL alternative

Again this '?' convention is a special case of a more general necessary convention, that of specifying 'nothing', or formally, NIL, as the last alternative of a phrase definition, Thus

PHRASE [SOME-CLASS-WORD] = [SOME-CLASS-WORD], NIL

NIL must occur as the final alternative, and it means that if none of the previous alternatives of the class word have been matched to the head of a symbol string, the class word will be recognised automatically without having to match any source symbol to it.

Thus using the explicit NIL alternative and explicit recursion we get the most precise definition of [LIST-OF-ELEMENTS] :

PHRASE [LIST-OF-ELEMENTS) = [ELEMENT][ANY-FUTRTHER-ELEMENTS]

PHRASE [ANY-FURTHER-ELEMENTS] = [,][ELEMENT][ANY-FURTHER-ELEMENTS], NIL

This final form should be compared carefully with the previous forms,

and also with the other neat formal definition :

PHRASE [LIST-OF-ELEMENTS] = [ELEMENT][FURTHER-ELEMENT*?]

PHRASE [FURTHER-ELEMENT) = [,][ELEMENT]

which although more compact, implies knowledge of the '*' and '?'

conventions and implies two further phrase definitions :

PHRASE [FURTHER-ELEMENT*?] = [FURTHER-ELEMENT*],NIL

PHRASE [FURTHER-ELEMENT*]=[FURTHER-ELEMENT][FURTHER-ELEMENT*],[FURTHER-ELEMENT]

Of course, having turned [LIST-OF-ELEMENTS] into a class word with only one phrase in the definition, we can dispense with it in the format for a simple variable data-type declaration and define it simply as :

[TYPE][ELEMENT][ANY-FURTHER-ELEMENTS]

where PHRASE [ANY-FURTHER-ELEMENTS] = [,][ELEMENT][ANY-FURTHER-ELEMENTS],NIL

Then the analysis of e.g. 'INDEX a, b, x, y' with respect to this format is

```
[TYPE][ELEMENT][ANY-FURTHER-ELEMENTS]

2      1          1
|      |          |
INDEX    a        ,[ELEMENT][ANY-FURTHER-ELEMENTS]

                   2          1
                   |          |
                   b        ,[ELEMENT][ANY-FURTHER-ELEMENTS]

                            24          1
                            |           |
                            x         ,[ELEMENT][ANY-FURTHER-ELEMENTS]

                                       25                    2
                                       |                     |
                                       y                    NIL
```
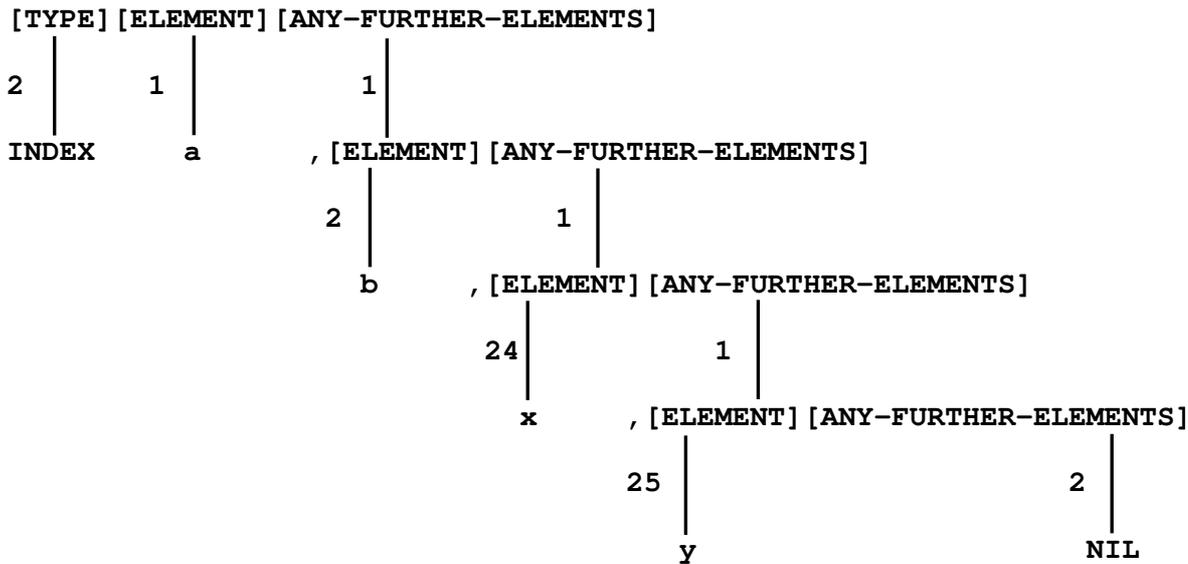
Pseudo-identifiers & Meta-symbols

It will already have been observed in the case of the comma that there can be ambiguity in phrase definitions between symbols occurring in the language-description language ('meta-symbols') and in the language being described (source symbols). This is overcome by the convention that such symbols occurring as source symbols must be represented in phrase definitions by 'pseudo-identifiers' : [SP] for space, [EOL] for end of line, [,] for comma, and [[] for a left hand square bracket – a ']' will be taken as a meta-symbol only if it matches a '['. This convention is chosen to look as if (hence 'pseudo') they are class words specially defined to represent them appropriate source symbol.

Spaces It may have been observed that there has been indecision in the previous discussion as to whether spaces are to be ignored. In fact it is assumed that in the autocode language (and in the meta-language) spaces and underlined spaces are universally ignored; but in the English examples spaces are not ignored. [Note that the English covering description is in fact a formal programming language.]

Thus where [S] represents a general space, i.e. including any number of [SP]s and [EOL]s, and noise words like 'the', the formal phrase definitions for the English examples should be, for example

PHRASE [EQUALS] = equals, [IS][S][EQUAL-TO], [IS]

PHRASE [EQUAL-TO] = equal[S]to, same[S]as, at

   We could also define

PHRASE [EQUALS] = equals, [IS][EQUAL-TO]

PHRASE [EQUAL-TO?] = [S][EQUAL-TO], NIL

But note that we cannot define '[IS][S][EQUAL-TO?]' as an alternative of [EQUALS], since if there was no [EQUAL-TO] the [S] following [IS] would be recognised as part of [EQUALS], which would be inconsistent with the above definitions.

Note also the use here (and later) of the symbol '?' (and *?) in the class word for an optional phrase (or optional repeated phrase) that has been defined explicitly instead of using the '?' convention.

BUT NOT

It is sometimes convenient to be able to specifically exclude sub-alternatives from a phrase definition. This can be achieved by finishing the definition by BUT NOT and then the prohibited phrases. This effect could of course generally be achieved by redefining the phrase, but this could well mean that one could not then use the structure of subphrases which has been used in defining the main body of the class word set.

For example, note that [DOES-NOT-EQUAL] contains the 4 ungrammatical phrases 'will be not[EQUAL-TO?]'.

This could be corrected without disturbing the subphrases by redefining it as :

```
PHRASE [DOES-NOT-EQUAL] = does[S]not[S]equal, [IS][S]not[EQUAL-TO?],
          will[S]not[S]be[EQUAL-TO?], BUT NOT will[S]be[S]not[EQUAL-TO?]
```

Definition of an expression

As a final example of a phrase definition, consider the formal definition of a mathematical expression

E.g. 4, e + 1, q, Y[e], q + 2(Y[e]−Y[e−1])/2−(Y[e−1]3−Y[e−2])/5),
     sqrt(p)/(n−1), ab(b+c)(c+d), Y[e] − 2sqrt((p+q−r)/5)/n+(p−1)(p−2)/3

An informal analysis of the syntax of an expression in the language of the specimen program is an follows :

An expression is a complex of operators and operands. Two operators cannot follow each other; two operands can, this being treated as implicit multiplication, An operand is something with a value, and can be either : a specific number, e.g. 14, 1, 3: the value of a variable, e.g. e, Y[e], Y[e−2]; or the value of an expression in brackets, e.g. (b+c), (n−1) (Y[e]−Y[e−1]), or (p+q−r/5), or the value of a function, e.g. sqrt (p).

The formats of these individual operands are :

1) [CONSTANT] = [DlGIT].[INTEGER]α[+?][INTEGER], [INTEGER].[INTEGER],[INTEGER]
   e.g. 5.3 α−4, 1.2934α321; 485.35, 0.00004; 8, 439725

2) (VARIABLE] = [ELEMENT], [ARRAY-BASE] [[] [EXPRESSION]]
   Where an [ARRAY-BASE] is a capital letter.
      e.g. x, a; Y[e], Y[e−1], P[a+bc]

3) ([EXPRESSION]) i.e. an expression in round brackets

4) [FUNCTION] = [small-letter*] ([EXPRESSION][ANY-FURTHER-EXPRESSIONS])
Where [ANY-FURTHER-EXPRESSIONS] = [,][EXPRESSION] [ANY-FURTHER-EXPRESSIONS],
NIL
 e.g. sqrt(p), sqrt((x+y)/2), sqdf (Y[e]−Y[e−1], (Y[e−1]+Y[e−2])/2)

Note that full recursion with respect to an [EXPRESSION] is possible in alternatives (2), (3), and (4).

Note that the form of these 4 basic operands is such that it is immediately clear from the first symbol of a string which of the 4 forms it is, and further, the extent of the string specifying the operand is unambiguously defined. Thus :

1) Starts with a [DIGIT], then any further digits, maybe including a '.', may be followed by 'α[INTEGER]' to give a decimal exponent.

2) Is just a small letter; or it starts with a capital letter, then '[' and

    then all symbols up to a matching ']'

3) '(' then all symbols up to a matching ')'.

4) An underlined small letter, then maybe some more, then '(' then all further symbols up matching ')'.

    The formal definition of an expression is therefore :

```
PHRASE [EXPRESSION] = [+?][OPERAND][OPERATOR-OPERAND*?]
     Where
PHRASE [OPERATOR-OPERAND*?] = [OPERATOR][OPEAND][OPERATOR-OPERAND*?],NIL
PHRASE [OPERATOR] = +,-,×,/,NIL
PHRASE [OPERAND]  = [CONSTANT],[VARIABLE],([EXPRESSION]),[FUNCTION]
PHRASE [+?]       = +,-,NIL
PHRASE [CONSTANT]=[DIGIT].[INTEGER]α[+?][INTEGER],[INTEGER].[INTEGER],[INTEGER]
PHRASE [VARIABLE] = [ELEMENT], [ARRAY-BASE][[][EXPRESSION]]
PHRASE [FUNCTION] = [small-letter*]([EXPRESSION][ANY-FURTHER-EXPRESSIONS]),
                   [small-letter*]        ((i.e. a parameterless function))
PHRASE [ANY-FURTHER-EXPRESSIONS] = [EXPRESSION][ANY-FURTHER-EXPRESSIONS],NIL
PHRASE [ELEMENT]   = a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
PHRASE [ARRAY-BASE] = A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
PHRASE [SMALL-LETTER] = a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
PHRASE [DIGIT]     = 0,1,2,3,4,5,6,7,8,9
PHRASE [INTEGER]   = [DIGIT][INTEGER],[DIGIT]  ((i.e. [INTEGER]≡[DIGIT*]))
```

    The formal analysis of e.g. 'a+(pq-Q[j]/2)' with respect to the class word [EXPRESSION] is given in the appendix. Note that in this example the basic symbols in the analysis have been picked out in quotes. Note also that the basic phrase matching a sub-class-word is sometimes shown to the right of the category number. Note of course that at any such point in the analysis, the sub-tree below the point is the formal analysis of this basic phrase with respect to the class word above it, independent of the rest of the tree.

Formal specification of a language

    The class word definitions involved in the definition of an [EXPRESSION] cover many of the class words involved in the language. The list can be continued as follows :

```
PHRASE [if,unless] = if, unless
PHRASE [COMPARISON-SYMBOL] = =,≠,≥, <, >, ≤
PHRASE [;] = ;, [NEWLINE] ((this is the standard instruction separator))
PHRASE {NEWLINE] = [EOL][NEWLINE],[EOL]   ((i.e. one or more newlines))
PHRASE [LABEL] = [INTEGER]
PHRASE [OUTPUT-SYMBOL] is the set of symbols that can be printed in 'caption ..'
```

And to specify the formats of definitions the following class-words are
required:
PHRASE [DATA-SPECIFICATION] = [TYPE][ELEMENT][ANY-FURTHER-ELEMENTS][NEWLINE],
                    [TYPE]ARRAY : [ARRAY-BASE][[][INTEGER]:[INTEGER]][NEWLINE]
Where PHRASE [TYPE] = REAL, INDEX, INTEGER, COMPLEX
 and PHRASE [ANY-FURTHER-ELEMENTS] = [,][ELEMENT][ANY-FURTHER-ELEMENTS], NIL
PHRASE [ROUTINE-FORMAT] = [small-letter*][NEWLINE],   ((parameterless routine))
             [small-letter*][PARAMETER-SPEC][ANY-FURTHER-PARAMETERS][NEWLINE]
Where PHRASE [PARAMETER-SPEC] = [[][TYPE][CALL?][ELEMENT]]
 and PHRASE [CALL] = VARIABLE REFERENCE, EXPRESSION NAME, VATIABLE NAME,
                                            VARIABLE, EXPRESSION
 and PHRASE [ANY-FURTHER-PARAMETERS] =
                          [,][PARAMETER-SPEC][ANY-FURTHER-PARAMETERS], NIL
PHRASE [FUNCTION-FORMAT] = [PARAMETER-SPEC][=] [small-letter*][NEWLINE],
                 [PARAMETER-SPEC][=] [small-letter*]([PARAMETER-SPEC]
                                 [ANY-FURTHER-PARAMETERS])[NEWLINE]

Where PHRASE [=] = =
 Also PHRASE [S] = S

Then the list of formats used in the specimen program in order of appearance is
as follows :

FORMAT : DATA TYPE [S?][NEWLINE][DATA-SPECIFICATION*]
FORMAT : ROUTINE FORMAT [S?][NEWLINE][ROUTINE-FORMAT*]
FORMAT : FUNCTION FORMAT [S?][NEWLINE][FUNCTION-FORMAT*]
FORMAT : MAIN PROGRAM [NEWLINE]
FORMAT : read [VARIABLE][;]
FORMAT : [VARIABLE] = [EXPRESSION];
FORMAT : [LABEL]:
FORMAT : [small-letter*][EXPRESSION][[ANY-FURTHER-EXPRESSIONS][;]   ((routine call))
FORMAT : -> [LABEL][if,unless][EXPRESSION][COMPARISON-SYMBOL][EXPRESSION][;]
FORMAT : -> [LABEL][;]
FORMAT : cycle [VARIABLE] = [EXPRESSION],[EXPRESSION],[EXPRESSION][;]
FORMAT : repeat[;]
FORMAT : print [EXPRESSION][;]
FORMAT : stop[;]
FORMAT : ROUTINE [NEWLINE][ROUTINE-FORMAT][DATA-SPECIFICATION]
FORMAT : return [;]
FORMAT : caption [OUTPUT-SYMBOL*][NEWLINE]
FORMAT : FUNCTION [NEWLINE][FUNCTION-FORMAT][DATA-SPECIFICATION*?]
FORMAT : END OF PROGRAM[EOL]

Thus if the specimen autocode only provided those facilities that have been illustrated in the specimen program, then this list of formats would be a sufficient specification of the syntax of the language. To fully specify the language there would also need to be a description of the computer model defined for the language, and a description of what each instruction class does in terms of this model.

The following brief descriptions will indicate the sort of job the compiler has to do on recognising certain key phrases and formats of the language.

[DATA SPECIFICATION] : [TYPE][ELEMENT][ANY-FURTHER-ELEMENTS]

For each [ELEMENT] :

Add the name of the [ELEMENT] to the list of variables declared for the current routine (or the main program), and note that its type is the specified [TYPE]. Allocate an appropriate location of the data space (stack) to this [ELEMENT].

Thus when subsequently the compiler finds the [ELEMENT] referred to in an imperative instruction, it can look it up in the list of names to get its address and type. Knowing from the syntax of the imperative what general operation it wants to perform on the variable, it can then compile the appropriate machine code instructions.

Imperative instruction : [VARIABLE] = [EXPRESSION][;]

Compile the appropriate set of machine code instructions to calculate the value of the [EXPRESSION]. If the [VARIABLE] is an [ELEMENT] set this value in the location that has been allocated to the [ELEMENT]. If the [VARIABLE] is of the form [ARRAY-BASE][[][EXPRESSION/2]], first compile instructions to calculate [EXPRESSION/2], add this index to the [ARRAY-BASE] and finally put the value of the original [EXPRESSION] in the location specified by this address.

Declaration :   [LABEL]:

Add the [LABEL] to the list of labels for the current routine (or the main program), together with the address of the next machine code instruction to be compiled in the object program,

Imperative : ->[LABEL][if,unless]
                         [EXPRESSION/1)[COMPARISON-SYMBOL][EXPRESSION/2][;]

Compile the appropriate machine code instructions to calculate the value of [EXPRESSION/1] - [EXPRESSION/2], and then compare it with 0 as indicated by [if,unless] and the [COMPARISON-SYMBOL], making a jump to the instruction with this [LABEL] if the test is satisfied.

The compiler can get this address by looking up the [LABEL] in the label list. However the label may not have occurred yet in the routine; in this case the address of the machine code instruction containing the jump address is added to another list, together with this [LABEL] number. Then at the end of the routine the two lists are matched up and all the label references are filled in.

Ch. 3   THE STRUCTURE OF A COMPILER PROGRAM : FORMAT ROUTINES

Recognition of Formats

An ordinary one-pass compiler program successively recognises the instructions of a source program and interprets them in the context of the autocode it is designed for. So the top-level routine (MAIN PROGRAM) starts with an initialisation procedure to prepare to compile. Then in the main section of the program it loops round a procedure to recognise the next instruction of the source program and then interpret it appropriately. Finally, at the end of the source program, it ties up all the loose ends, removes itself from the computer store, and passes control to the object program it has compiled.

It is clear that the natural unit of recognition is the source instruction; so the compiler program successively matches the head of the symbol string that is its input data against the list of formats in the language. Having found the format it then has to interpret it. As the general interpretation is the same for each source instruction satisfying a format, this suggests that the natural routine structure for the compiler program is a routine associated with each different format, The MAIN PROGRAM recognises which format the source instruction belongs to and then passes control to the appropriate 'format routine'.

As is continually being realised in the field of computer languages, as soon as one sets down information in a reasonably formal and systematic way, it is possible to write a computer program to recognise such a formal description and interpret it appropriately. There is no need to write a special set of instructions in a more basic language to deal with each special situation so described. Thus in a conventional program there is now no need to write out a set of basic instruction to calculate the value of an expression. One merely writes out the expression in a formal language, and the compiler will recognise it and interpret it correctly in the context of the rest of the program.

So with the Compiler Compiler it is sufficient to write out the formal definition of the source language in class words and formats – similar to declarations of the compiler program – for the Compiler Compiler to be able to organise the whole process of format recognition automatically. To be precise, it can insert a set of instructions in the compiler program, the 'analysis routine', which together with the information it stores about the syntax of the source language (in the instruction space of the compiler program) will recognise the next instruction of the source language and pass control to the appropriate format routine.

The MAIN PROGRAM of the compiler program is therefore automatically included in the compiler program by the Compiler Compiler, and does not appear in the compiler program's description. This top-level routine will be referred to as the 'master-routine', and its chief constituent is the analysis routine. The compiler-writer can specify an initialisation routine for it to obey before it starts to recognise the source program, and he will arrange to terminate the compiler program in his END OF PROGRAM format routine.

Principal declarations

The principal declarations of the complier program are therefore the PHRASE and FORMAT declarations, which define the syntax of the source language, and the ROUTINE declarations which introduce the format routines that define the meaning of the source language.

For simplicity, it can be assumed that the compiler program description (in Compiler Compiler language) starts with the set of PHRASE and FORMAT declarations, and that these are followed by the format routines.

The form of the PHRASE and FORMAT declarations is virtually the same as that given in the description of PHRASE STRUCTURE NOTATION.

A phrase definition starts with the word PHRASE, then the class word name then '=', then the alternative phrases of the definition, separated by commas. For example :

PHRASE [ANY-FURTHER-ELEMENTS] = [,][ELEMENT][ANY-FURTHER-ELEMENTS], NIL

PHRASE [OPERAND] = [CONSTANT], [VARIABLE], (([EXPRESSION)), [FUNCTION]

The conventions described previously were consistent with the Compiler Compiler language except that there are some restrictions on the symbol string that can be chosen for a class word name.

The specification of formats in the Compiler Compiler is much as before, except that as more than one format class is allowed, the class word name must be included in the FORMAT format. Built. in to the system is the format class [SS] (Source Statement), and it is this format class that the master-routine scans each time it tries to recognise the next instruction of the source program.

Thus for example

FORMAT [SS] = <u>ROUTINE FORMAT</u> [<u>S</u>?][NEWLINE][ROUTINE-FORMAT*]

FORMAT [SS] = <u>MAIN PRORGRAM</u>[NEWLINE]

FORMAT [SS) = [VARIABLE]=[EXPRESSION][;]

FORMAT [SS) = ->[LABEL][<u>if</u>,<u>unless</u>][EXPRESSION][COMPARISON-SYMBOL][EXPRESSION][;]

Formats must be listed in order of preference.

[Note that this does <u>not</u> mean that e.g. -›[LABEL][;] must be listed after -> [LABEL][<u>if</u>,<u>unless</u>]... since the instruction-separator [;] ensures that the unconditional jump format will not recognise the first part of the conditional jump. This is a commons device to avoid order-of-preference difficulties (i.e. to end a phrase or format with a 'separator' common to all the alternatives).

Note however that all the permanent routines which have the same format as that of the general routine call, e.g. print [EXPRESSIONS][;] will have to be listed before the format of the general routine call.]

Imperative instructions

Format routines are made up of imperative instructions of the Compiler Compiler language, to describe the operations to be carried out as appropriate to recognising an instruction of the source program.

The Compiler Compiler allows conventional variables of only one type, 24-bit integers ('index'). The permitted form of the names is as follows :

A1, A2, ... A28,...., etc. for variables local to a format routine

B1, B2, ... B17, ...etc. where there is a practical limit of 40, for variables global to the program (nonlocal to each routine),

There is therefore no need for any declarations for those variables; the type is always 'index', A and B indicate 'local' and 'global' respectively, and the integer, N say, gives the address, i.e. the Nth, location on the list of such variables in the local or global data space.

No array declarations are provided. Again the use of lists is indicated implicitly in the notation. There is only one list, the whole computer store under the interpretation of type 'index', and the notation for an array element is not e.g. P[i], Y[e], or P(i), Y(e), that is [ARRAY-BASE][[][EXPRESSION]] in the specimen language, or [Identifier] ([EXPRESSION]) as in Atlas Autocode but just ([ADDRESS]), e.g. (B1), (A1+7), or (B5+A2), where [ADDRESS] is a limited form of combination of variables and integers.

The language of the Compiler Compiler includes the conventional types of instruction : e.g. assignment, jump, and conditional jump, in terms of the above variables and a labelling system.

E.g. A1 = B4, B35 = B11 + A2, B12 = (A1+2) × B11, (A1+2) = B12/B11,

-> 4 IF B11 + A2 ≥ B5, -> 24 UNLESS (B16> > (B16+1), - > 6

The form of an expression is severely limited, particularly as there is no recursion allowed. Note that of course (A1+2),(B16), and (B16+1) refer to the contents of the store address A1+2, B16, and B16+1 respectively„ The values of the variables A1 and B16 must therefore be absolute store addresses.

The basic language of the Compiler Compiler also provides some instructions for list processing.

Of the other standard features of a programming language, there is a 'print' instruction but no 'read', as all input is read automatically by the built-in analysis routine. There is also a subroutine mechanism that will be described in the next chapter, and there are some special instructions for handling symbol strings, to be described below.

The data space associated with the variables and the other special facilities, and the reference to it, is organised automatically by the Compiler Compiler, However any special data space required by the compiler-writer, e.g. to hold the name-type-address st list for the variables of the source program, he must organise himself, using the rudimentary array variables and absolute store addresses, Such private data space must of course be kept separate from the instruction space and the data space of the compiler program, and from the instruction space of the source program which it is compiling.

Phrase Variables (Phrase identifiers)

It should be evident from the previous chapters that the natural parameters of both the definitions of the language syntax and the interpretation of instructions in terms of the autocode are the class words. As we talk about general operations

we refer to class words, on the understanding that in any particular occurrence of the language or its interpretation there will be associated with each class word a symbol string which specifies the particular object ('variant') we are currently dealing with

Consider

FORMAT : -> [LABEL][if,unless][EXPRESSION][COMPARISON-SYMBOL][EXPRESSION][;]

For example :

a)   -> 5 if p = 45

b)   -> 11 if 5(a+b) > 2xy

or, from the specimen program :

c)   -> 3 unless Y[e] > Y[e-1]

d)   -> 2 if (Y[e]-Y[e-1]) > f(Y[e-1]-Y[e-2])

Its interpretation bas been discussed in terms of the [LABEL], [if,unless], the [COMPARISON-SYMBOL], and the two [EXPRESSION]s, maybe giving particular examples of the associated symbol string.

But obviously the general interpretation is independent of the particular matching symbol string :

	[INTEGER]	[if,unless]	[EXPRESSION]	[COMPARISON-SYMBOL]	[EXPRESSION]
a)	5	if	P	=	45
b)	11	if	5(a+b)	>	2xy
c)	3	unless	Y[e]	>	Y[e-1]
d)	2	if	(Y[e]-Y[e-1])	>	f(Y[e-1]-Y[e-2])

Equally clearly much of the subroutine structure of the compiler program will refer to class words, for example :

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF [EXPRESSION]
COMPILE INSTRUCTIONS TO SET [VARIABLE] = CONTENTS OF ACCUMULATOR
FIND ADDRESS AND TYPE FOR [VARIABLE]

Etc...

Therefore the Compiler Compiler introduces a new type of variable into the formal language specially for dealing with these associations between class words and symbol strings (i.e. basic phrases). These are called 'phrase-identifiers' in the Compiler Compiler. However since this name word is used in the Compiler Compiler to refer to class words, to ensure precision these variables will be called 'phrase-variables' in the following discussion.

It will be remembered that in a conventional autocode there are conventional variables of the type 'real', 'integer', 'complex', etc., which are all numbers, or combinations of numbers. Each variable has : a name, i.e. a string of symbols by which it is referred to systematically in the program description; an address, i.e. the location in the data space which it has been allocated; a value, i.e. the contents of the allocated store address(es), which vary as the program is being executed; and a type, which tells how the value must be interpreted

whenever it is operated on. A variable can be global (i.e. declared with the main program), or it can be local, in which case its name can be chosen. independently from the rest of the program, and during the execution of the program the address and value only exist while the routine in question is being obeyed (i.e. the value may be destroyed between successive calls of the routine).

A 'phrase-variable' is a special non-numerical, variable whose value is a symbol string. Phrase-variables therefore cannot usually be involved in the same instructions as conventional variables, and there is a special set of 'phrase-handling instructions' in the Compiler Compiler language for handling them.

Phrase-variables are always local to a format routine,

Although the value of a phrase-variable is a symbol string (e.g. a source symbol string that has been matched to a class word in a format), it becomes clear that, in interpreting the value of a string of symbols, we are more concerned with its relation to the definition of the class word it is associated with than the code of each symbol in it. For example (at its simplest) if we require to set a conventional parameter A5 to represent the information that 's' is the value of an [ELEMENT], it is easier to identify the value of 's' as being no.19 in the definition of an [ELEMENT] (and so set A5 = 19, the category number) than to calculate this number from the symbol code for 'a', even though this should be a straightforward calculation,

Therefore the type of each phrase-variable is different for variables associated with different class words; the type is not interpreted in the Compiler Compiler as just a string of symbols — in which case the values of all phrase-variables would be interpreted as ordered lists of numbers interpreted as symbol codes. For similar reasons symbol strings are not represented in store as such by the Compiler Compiler, but by 'analysis records' whose structure reflects the analysis trees shown in Chapter 2 (also see the appendix), However it will be convenient to continue to refer to the value of a phrase-variable as a symbol string (or 'phrase value'),

Because the interpretation of a phrase-variable is carried out relative to the associated class word definition, we can regard the phrase definitions as being declarations of new types of objects. That is, as well as giving information to allow the analysis routine to recognise source instructions, they give the Compiler Compiler the requisite information to interpret the values of phrase-variables when they occur in instructions of a format routine.

It will, be remembered that the type and address of the conventional variables in the Compiler Compiler language is implicit in their names — there is no need to declare variables. The same procedure is used for the special phrase-variables. The name of a phrase-variable must be the associated class word name, maybe followed by a '/' and an integer, all in square brackets. Thus regarding the class word itself as '/0', each different phrase-variable of the same type is a routine and must be numbered(named) differently relative to the class word, e.g. [EXPRESSION],[EXPRESSION/1],[EXPRESSION/2], etc..

The formal parameters of format routines

As the analysis routine of the compiler program recognises an instruction as being a member of a particular format, it builds up a list of associations between the class words in the format and the particular symbol strings matching them. These are the natural parameters of the format routine, since they specify the particular source instruction matching it.

So the form of the ROUTINE heading declaration is the same as the corresponding format except that the class words in the format are replaced by phrase-variable names in the routine heading. The routine heading thus serves to introduce the names of the formal parameters (which must all be different).

For example :

```
ROUTINE [SS] ≡ ROUTINE FORMAT [S?][NEWLINE][ROUTINE-FORMAT*]

ROUTINE [SS] ≡ MAIN PROGRAM[NEWLINE]

ROUTINE (SS) ≡  [VARIABLE] = [EXPRESSION][;]

ROUTINE [SS] ≡ ->[LABEL][if,unless]
                           [EXPRESSION/1][COMPARISON-SYMBOL][EXPRESSION/2][;]
```

Each routine heading is followed by the set of imperative instructions to be carried out each time it is entered. As part of the entry mechanism the phrase-variables in the routine heading will all have had values assigned to them, i.e. the particular sub-strings of the particular source instruction causing entry to the routine.

Phrase-variables other than those occurring in the routine heading can be used in the routine.

Any phrase-variable (including the formal parameters) can have its value reset any number of times during the execution of a routine (the special phrase-handling instructions that carry out these operations will be described below).

Thus the situation with respect to phrase-variables and format routines is exactly the same as for conventional variables and conventional routines, except that there can only be local phrase-variables and there need be no declarations as the types are implicit in the phrase-variable names.

[NOTE : To translate this section into a description of a conventional routine, where necessary read 'value' for 'symbol string', 'variable' for 'phrase-variable', and 'type' for 'class word', and remember that entry to a conventional routine is done by the routine call cue, which sets up the current values of the actual parameters in the call as appropriate to the call, and hands them on to the routine.

The similarity between the form of a format and of a routine heading sometimes causes confusion in the Compiler Compiler. This would be resolved relative to conventional autocodes if e.g. Atlas Autocode used the notation of leaving out the names from the routine specification, and the Compiler Compiler required that all phrase-variable names should have explicit numbers.

For example, in Atlas Autocode

routine spec print (real, integer, integer)

routine print (real e, integer b, integer a)

this being a routine say to print out the value of an expression, specifying the number of digits before and after the decimal point.

And in the Compiler Compiler :

```
FORMAT [SS] = ->[LABEL]
          [if,unless][EXPRESSION][COMPARISON-SYMBOL][EXPRESSION/2][;]

ROUTINE [SS] = ->[LABEL]
          [if,unless/1][EXPRESSION/1][COMPARISON-SYMBOL/1][EXPRESSION/2][;/1]
```

Phrase-handling instructions

In some format routines the operation to be carried out on recognition is not varied by any formal parameters, for example stop[;] and return[;]. where it does not matter whether the separator [;] was a ';' or a [NEWLINE].

However in many routines the operations required may vary in places according to the particular symbol string values associated with the formal parameters. Therefore there are some basic 'phrase-handing instructions' provided. These can convert information from symbol string form into conventional variables, and can generally manipulate the phrase-identifier variables, reassigning values to them, and testing them etc..

In order to describe the syntax of these phrase-handling instructions (and the language of the Compiler Compiler in general), a 'meta-meta-language' notation will be used to describe the classes of phrase-variables and conventional variables, etc.. These will be distinguished by underlining the square brackets round class names. This convention is not adopted by the Compiler Compiler; the corresponding class words and formats of the Compiler Compiler will also be given as a reference.

[AB] is a conventional variable [A] or [B], e.g. A1, A7, A12, B4, B36, etc..
[N] is an integer e.g. 5, 45, 12, etc..
[ABN] is [AB] or [N]; these are called [ABN] [AB], and [N] in the Compiler Compiler.

[PHRASE-VARIABLE] or [PI] in the Compiler Compiler :
    This refers to a phrase-variable to which a value is to be assigned by the
    instruction in question
    (e.g. the equivalent of [VARIABLE] is '[VARIABLE] = [EXPRESSION]').
[PHRASE-VARIABLE-VALUE] also [PI] :
    This refers to a phrase-variable to which a value is to be used as an
    operand in the instruction in question
    (e.g. the equivalent of a [VARIABLE] occurring inside an [EXPRESSION].

[PHRASE-EXPRESSION-VALUE] or [GENERATED-P] in the Compiler Compiler :
    This refers to a phrase with [PHRASE-VARIABLE-VALUE]s instead of class
    words. This defines a symbol string value formed from the symbols of the
    phrase with the current [PHRASE-VARIABLE-VALUE]s substituted in. Or the
    phrase can be a basic phrase itself, without any phrase-variables in it.
    This is the equivalent of an [EXPRESSION] in the conventional assignment
    instruction.
[PHRASE-EXPRESSION] or [RESOLVED-P] in the Compiler Compiler :
    This refers to a phrase with [PHRASE-VARIABLE]s instead of class words (if
    any) — the distinction between this and [PHRASE-VARIABLE-VALUE] will
    become apparent in context later.

[NOTE : There is normally no distinction made between a conventional [VARIABLE] and a [VARIABLE-VALUE], i.e. between a variable whose value is being set and one whose value is being used as an operand. There is a slight difference in practice, in that the former need not have a current value assigned to it when it is encountered during program execution, but the latter must have, Most autocodes will not check for example that variables being used in an expression have had values assigned to them since the beginning of the program, or since the routine was reactivated in the case of local variables that are not formal parameters. If this happens (usually due to a programmer's error), depending on the particular compiler and whether or not the variable is a local variable, the value may have been preset at 0, or it may be the final value of the variable in the previous activation of the routine, or it could be any arbitrary number - e.g. in the case where local storage space is shared between routines. However with the Compiler Compiler a check is always made that a phrase-variable that should have a current value does.

Also note that there is a slight difference in the permitted form of a [PHRASE-VARIABLE] and a [PHRASE-VARIABLE-VALUE] name in the Compiler Compiler, due to rarely used facilities that will not be discussed here.]

1)   Conversion from symbol string value to conventional number

[AB] = CATEGORY OF [PHRASE-VARIABLE-VALUE]

                    or in the Compiler Compiler : [AB] = CATEGORY OF [PI]

E.g. Al = CATEGORY OF [ELEMENT]

or   A7 = CATEGORY OF [COMPARISON-SYMBOL/1]

or   B23 = CATEGORY OF [OPERAND]

   This sets the conventional variable [AB] to the category number of the phrase-variable value with respect to the corresponding phrase's definition.

   Thus if [ELEMENT] was currently an 's', A1 would be set to 19, or if 'c', to 3, etc..

   If the [COMPARISON-SYMBOL/1] (defined as =,≠, ≥, <, >, ≤) was currently '≠' A7 would be set to 2, or if '>' to 5.

   B23 would be set to 1 if the [OPERAND] was a [CONSTANT], 2 if it was a [VARIABLE], 3 if an ([EXPRESSION]), and 4 if it was a [FUNCTION].

2)   Resolving

   The previous instruction does not give any information about subphrases of a phrase, For example if we had a [VARIABLE] on hand e.g. f, P[e], or R[ab+c], we could find out from the category number if it was an [ELEMENT] (cat, 1) or an array-element (cat. 2), but we could not find out any further information until we had broken up the [VARIABLE] into its constituent subphrases.

   To accomplish this there is the 'resolve' instruction :1

RESOLVE [PHRASE-VARIABLE-VALUE] INTO [PHRASE-EXPRESSION]

   or LET [PI] ≡ [RESOLVED-P]

E.g. RESOLVE (VARIABLE/1] INTO [ELEMENT]

or   RESOLVE [VARIABLE] INTO [ARRAY-BASE] [[] [EXPRESSION/3]]:

Here the [PHRASE-EXPRESSION] must be an alternative (or subalternative)

phrase of the definition of the class word associated with the [PHRASE-VARIABLE-VALUE] – except that the class words in the phrase are [PHRASE-VARIABLE]s.

This has the effect of setting up new values for the [PHRASE-VARIABLE]s in the [PHRASE-EXPRESSION], which values are the appropriate sub-symbol-strings of the current [PHRASE-VARIABLE-VALUE].

The [PHRASE-VARIABLE-VALUE] remains unchanged.

For example, if [VARIABLE/1] was currently an 's' (or a 'k', say)

RESOLVE [VARIABLE/ 1] INTO [ELEMENT]

would reset the value of the [ELEMENT] to 's' (or 'k'), Then the instruction [AB] = CATEGORY OF [ELEMENT] would yield the number 19 (or 11).

If the (VARIABLE] was currently 'Y[e]' or 'R[ab+c]'

RESOLVE [VARIABLE] INTO [ARRAY-BASE][[][EXPRESSION/3]]

would set up a new value for [ARRAY-BASE] 'Y' or 'R' as appropriate, and is new value for [EXPRESSION/3] 'e' or 'ab+c'.

Note that the resolve instruction carries out the same kind of operation as the routine heading implies on entry to the routine. In that case a symbol string that is a member of the format class e.g. [SS] is automatically split up to give the initial values of the principal phrase-variables.

Note that the resolve instruction can not be used unless it is already known that the current value belongs to the appropriate category of the phrase definition. Thus RESOLVE [VARIABLE/ 1] INTO [ELEMENT] if the current value was 'Y[e]' would be an illegal instruction that would bring the execution of the compiler program to a halt.

3)   Testing

It is frequently required to carry out different instructions in a format routine depending on the particular alternatives of a class word that a phrase value belongs to. This can be achieved indirectly by getting the category number of the phrase-variable. Frequently, having established which alternative the symbol string belongs to, it is then required to resolve the phrase-variable in order to get at the subphrases.

These two operations are combined in the phrase testing instruction :

-> [N] [IF,UNLESS] [PHRASE-VARIABLE-VALUE] IS OF THE FORM [PHRASE-EXPRESSION]

           or -> [LABEL] [IU] [PI] ≡ [RESOLVED-P]

E.g. -> 1 UNLESS [VARIABLE/1] IS OF THE FORM [ELEMENT]

Or   -> 8 IF [VARIABLE] IS OF THE FORM [ARRAY-BASE][[][EXPRESSION/2]]

Here the symbol string of the current [PHRASE-VARIABLE-VALUE] is compared with the [PHRASE-EXPRESSION]. And if it matches the phrase : control is changed (IF) or not changed (UNLESS) to the appropriate instruction as indicated by the label; and if the [PHRASE-EXPRESSION] contains any [PHRASE-VARIABLE]s the symbol

string is then resolved into the [PHRASE-EXPRESSION], as described above.

    The [PHRASE-VARIABLE-VALUE] remains unaltered.

    If the condition holds and there are [PHRASE-VARIABLE]s in the [PHRASE-EXPRESSION], they are assigned new values as appropriate, But if the condition does not hold, no new values are assigned, and a jump is made if the condition is UNLESS.

    Note that the [PHRASE-EXPRESSION] need not be a principal alternative

    of the class word definition; it can equally well be a subalternative,

    for example

-> 23 IF [VARIABLE/5] IS OF THE FORM [ARRAY-BASE] [[] [ELEMENT/4]]

where one might want to deal directly with the special (common) case where the array modifier is just an index. This is equally true of the resolve instruction, but it is less likely to occur since it must be known in advance that the [PHRASE-VARIABLE-VALUE] is a member of the [PHRASE-EXPRESSION] class.

    Note that the [PHRASE-EXPRESSION] need not contain any [PHRASE-VARIABLE]s

for example :

    -> 18 IF [if,unless] IS OF THE FORM if

4)   Generating

    It is sometimes required to assign values direct to phrase-variables instead of indirectly via a routine heading, a resolve instruction, or a successful test instruction. This can be done by :

SET [PHRASE-VARIABLE] = [PHRASE-EXPRESSION-VALUE]

    or LET [PI] = [GENERATED-P]

E.g. SET [VARIABLE/1] = [VARIABLE]

    where it is required to makse a copy of the value of the [VARIABLE]

or   SET [COMPARISON-SYMBOL/2] = >

or   SET [COMPARES-WITH] = [IS][S]not [COMPARED-WITH?]

    Here the [PHRASE-VARIABLE] is assigned a new value, being the string defined by the [PHRASE-EXPRESSION-VALUE], i.e. by the symbol string of the phrase with the current string of any [PHRASE-VARIABLE-VALUE] in the [PHRASE-EXPRESSION-VALUE] substituted in.

    For example if the [IS] and [COMPARED-WITH?] had been assigned values e.g. by a successful test instruction :

-> 3 UNLESS [COMPARES-WITH] IS OF THE FORM [IS][COMPARED-WITH?]

and it was required to switch the test implied by [COMPARES-WITH], then the above instruction would reassign the switched value. If [COMPARES-WITH] had been 'was greater than or equal to', the test instruction would have resolved it, setting [IS] = 'was' and [COMPARED-WITH?) = 'greater than or equal to', and then the generate instruction would have reset [COMPARES-WITH] to 'was not greater than or equal to'.

<u>**Examples of the use of phrase-handling instructions**</u>

<u>**Example A**</u> **Consider the format routine for the conditional jump format is a compiler for the covering-description language :**

```
ROUTINE [SS) ≡ go to ([LABEL]) [if,unless] [EXPRESSON/1] [COMPARES-WITH]
                                                [EXPRESSION/2][PAUSE]
```

We would first of all :

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF [EXPRESSION/1] – [EXPRESSION/2]

And then we would have to compile a machine code instruction that tested the accumulator and executed a conditional jump. Say that the function code of the machine code instruction is a certain Code F for '=', F+1 for '≠',F+2 for'≥', F+3 for '<', F+4 for '>' and F+5 for '≤'.

The relevant phrase definitions are :

```
PHRASE [if,unless] = if, unless
PHRASE [COMPARISON-SYMBOL] = =, ≠, ≥, <, >, ≤
PHRASE [COMPARES-WITH] = [COMPARISON-SYMBOL], [IS] [S]not[COMPARED-WITH?],
                                                [IS][COMPARED-WITH?]
PHRASE [COMPARED WITH?] = [S][COMPARED-WITH], NIL
PHRASE [COMPARED-WITH] = [GREATER-THAN][S]or[S][EQUAL-TO],
        [LESS-THAN][S]or[S] [EQUAL-TO], [GREATER-THAN], [LESS-THAN], [EQUAL-TO]
```

Etc., (as on P.16)

We require to set A1 to the correct relative code (0 to 5) for the comparison on which (if successful) we require to change control. Thus

```
    A2 = CATEGORY OF [if,unless]                 (( 1 for 'if', 2 for 'unless'))
    -> 1 UNLESS [COMPARES-WITH] IS OF THE FORM [COMPARISON-SYMBOL]
    A1 = CATEGORY OF [COMPARISON-SYMBOL]
    A1 = A1 – 1             (( A1 is now set correctly for [if,unless] = if ))
    -> 5
1)  -> 2 IF [COMPARES-WITH] IS OF THE FORM [IS][COMPARED-WITH?]
    RESOLVE [COMPARES-WITH] INTO [IS][S]not[COMPARED-WITH?]
    A2 = 3 – A2    ((in the case of 'not' this effectively switches [if,unless]
                      so that the following analysis on [COMPARED WITH?] applies
                      to both the 2nd and 3rd alternative of [COMPARED-WITH] ))
2)  -> 4 IF [COMPARED-WITH?] IS OF THE FORM [S][COMPARED-WITH?]
3)  Al = 0        ((Otherwise set Al for '=' as there is no [COMPARED-WITH?] ))
    ->5
4)  Al = CATEGORY OF [COMPARED-WITH]     ((We now require to operate on Al
    -> 3 IF Al = 5                         as follows : 1->2, 2->5, 5->0))
    -> 5 IF Al ≥ 3                        ((3 and 4 are already correct))
    A1 = A1 + 1
     -> 5 IF A1 = 2
    A1 = 5
```

```
5)   -> 6 IF A2 = 1          ((We now require to switch the code in the case of
                                      'unless' : 0 <->. 1, 2 <-> 3, 4 <-> 5 ))
     Al = Al ≢ 1             ((happens carry out this switch direct ))
6)   .... Etc.              (( A1 is now set to 0 to 5 as required for all the 892
                              alternative forms of [if,unless] ... [COMPARES-WITH]))
```

Example B A classic use of resolving, testing, and resetting values occurs
when dealing-with repeated phrases. And in this case we get a loop of
instructions, so that different executions of the same instructions carry out
the successive manipulations.

Consider the situation of wanting to set a conventional variable, say A5, equal
to the number represented by an [INTEGER]

Where PHRASE [INTEGER] = [DIGIT*]

```
     RESOLVE [INTEGER) INTO [DIGIT*]
     A5 = 0
1)   -> 2 UNLESS [DIGIT*] IS OF THE FORM [DIGIT][DIGIT*]
     A12 = CATEGORY OF [DIGIT]          (( 1 for '0', 2 for '1', 3 for '2', etc.))
     A5 = A5 + Al2 - 1
     A5 = A5 × 10
     -> 1
2)   RESOLVE [DIGIT*] INTO [DIGIT]
     A12 = CATEGORY OF [DIGIT]
     A5 = A5 + A12 - 1
```

Thus in the case of [INTEGER] = the symbol string '4392', on successive
executions of the instruction labelled (1) the following changes will take
place in the values of :

Time	Values of [DIGIT]	[DIGIT*]	A5	((for A5 are given the changes in the instructions immediately following))
Initially:	Unassigned	'4392'	0	
1st time	'4'	'392'	40	
2nd time	'3'	'92'	430	
3rd time	'9'	'2'	4390	
4th time	no change as condition was not satisfied, so go on to (2)			
(2)	'2'	unchanged	4392	

     It will be remembered that the '*' and '?' notations do not always provide
the most precise definitions. Consider instead :

PHRASE [INTEGER] = [DIGIT][ANY-FURTHER-DIGITS]

Where PHRASE [ANY-FURTHER-DIGITS] = [DIGIT][ANY-FURTHER-DIGITS], NIL

This would give the more compact sequence :

1) RESOLVE [INTEGER] INTO [DIGIT][ANY-FURTHER-DIGITS]

    A5 = 0

2) A5 = A5 × 10

    A12 = CATEGORY OF [DIGIT]

    A5 = A5 + A12 - 1

    ->2 IF [ANY-FURTHER-DIGITS] IS OF THE FORM [[DIGIT][ANY-FURTHER-DIGITS]

    In this case successive values after obeying the test instruction are :

Time	Values of [DIGIT]	[ANY-FURTHER-DIGITS]	A5
Initially:	'4'	'392'	0
1st time	'3'	'92'	4
2nd time	'9'	'2'	43
3rd time	'2'	'NIL'	439
4th time	condition not satisfied, no change		4392

    [NOTE : that the relative clumsiness of the first version is because for [DIGIT*], in order to resolve the final [DIGIT] e.g.'2', we are forced to take a different path from the usual resolution of [DIGIT*] into [DIGIT][DIGIT*].
    Note also that there is the same problem in using [DIGIT][DIGIT*?] for [INTEGER] as it is not possible to resolve [DIGIT*?] into [DIGIT][DIGIT*?] since [DIGIT*?] is not defined as [DIGIT][DIGIT*?]
but PHRASE [DIGIT*?] = [DIGIT*], NIL   ((? convention))
and PHRASE [DIGIT*] = [DIGIT][DIGIT*], NIL   ((* convention))]

Finally, consider a section of a routine dealing with the array declaration
[TYPE] ARRAY : [ARRAY-BASE][[][INTEGER/1]:[INTEGER/2]]

    We could have used the set of instructions above as a subsequence to convert the values of both [INTEGER/1] and [INTEGER/2], using as a link to the subsequence A11 say, so that the subsequence is followed by the switch
    -> A11   ((i.e. go to the label given by the current value of A11)).
    Then we could have 'called' the subsequence and set the required value of the first integer in A8 as follows :

    SET [INTEGER] = [INTEGER/1]

    A11 = 4

    -> 1

4) A8 = A5   ((Thus setting A8 as required))

        And then somewhere else in the routine :

    SET [INTEGER] = [INTEGER/2]

    A11 = 14

    -> 1

14) A9 = A5   ((Thus setting A9 = the value of [INTEGER/2]))

Ch. 4   THE STRUCTURE OF THE COMPILER COMPILER :   AUXILIARY ROUTINES

The previous chapter described the structure of a compiler program as written in Compiler Compiler language. The master-routine is built in, and the format routines are the principal routines of the program. There are special phrase and format declarations to describe the syntax of the source language.

The imperative instructions of the compiler program were described, and particular reference was made to the new class of phrase-variables introduced into the Compiler Compiler language, which are the parameters of the format routines.

No reference has been made to the structure of the Compiler Compiler, and no comment made on the way it compiles a program in its language. Nor has any description been made of any conventional routine mechanism for the compiler program (as opposed to the conventional routine mechanism the compiler will provide for the source language).

Integration of Compiler Compiler and compiler program

Of course the Compiler Compiler is itself a compiler program. When it is presented with a compiler program description it has to recognise it and translate it into the requisite machine code for translating any program written in the source language. The permitted language of the Compiler Compiler can equally be expressed in phrase structure form, e.g. using the phrases [AB], [N], [IF,UNLESS].

Therefore the machinery that the Compiler Compiler requires to do its own recognition and translation of a compiler program is much the same as it plants in the compiler program to recognise and translate source program.

Because of this the Compiler Compiler has been carefully designed so that a considerable part of its organisation can be used both when it is compiler-compiling a compiler program and when the compiler program is compiling a source program.

Therefore the master-routine and other powerful machinery that is useful during compiling is not in fact planted in the compiler program by the Compiler Compiler; it is part of the Compiler Compile machinery itself, which is taken over by the compiler program.

The situation is more complicated even than this. The source program translation machinery and the compiler program translation machinery can also be used when writing the Compiler Compiler itself, to generate the less basic section, (e.g. the equivalent of 'permament routines') in terms of the more basic machinery.

Bootstrapping technique in the. Compiler Compiler

Given the problem of writing a Compiler Compiler, without the aid of a Compiler Compiler compiler, there are two obvious ways : write it entirely in machine code (i.e. using a primitive assembly language), or write it using some existing autocode. The former is very hard work and difficult to revise; the latter generates an inefficient program compared with machine code.

However, particularly since the same mechanism is useful after the Compiler Compiler has been written, the solution chosen is neither of these, but a 'bootstrapping' technique. The overall design of the system being decided in advance, the basic core organisation is written in primitive assembly language to the stage where it can recognise language and compile the appropriate code (albeit primitively). Then the core machinery takes over to do the recognition and translation of the further material to be added to the system. As this new machinery is added on (e.g. routines to deal with new formats), the language the system can recognise gets more and more powerful. So that by the time the Compiler Compiler is ready for use, the last sections to be completed have been written using virtually the full power of the Compiler Compiler that is available to the compiler-writer.

Since the Compiler Compiler will be kept on magnetic tape, and will be called by the Atlas Supervisor as a compiler when required, it is convenient during development to be able to store the current version on tape in between runs, as if it were a working compiler, and call it down at the beginning of each run. Then the input of each development run will merely comprise some additional new machinery (and/or corrections to the existing machinery), together with a test on it, or a request to update the current version if the new machinery is already tested,

The structure of the Compiler Compiler is therefore such that it can easily add new program material to itself, and such that it can easily replace old sections (e.g. routines) with new ones, In order to permit this, all the machine code and all the other information stored with the Compiler Compiler is stored in 'relocatable', form.

[To achieve this, every *item* (e.g. a stored form of a phrase definition or a routine) is given a serial number. Then the only fixed part of the Compiler Compiler is the first 1024 (in fact) locations, which contain the current address of the item with the corresponding serial number (if there is one). Then within an item, all references to addresses within the item are coded relative to the base address of the item (or in the case of instructions containing addresses, relative to the instruction itself); all references from one item to another are made via the serial-number-address list. Items can then be moved around in any fashion provided that the address on the item list is altered appropriately.]

A powerful effect of this replacement facility – which is primarily required for making corrections while developing the system – is that routines can be written in a more primitive language earlier on, maybe only doing a limited job, and they can then be replaced later on by a version in a more powerful and/or efficient language, maybe doing a more powerful job, using new facilities that the earlier version has itself helped to provide.

Routine mechanisms of the Compiler Compiler ; compiling versions

In order to achieve the working system there are a number of different routine mechanisms, some very primitive, some very sophisticated. The variety is a convenience, so that the Compiler-Compiler-writer or the experienced compiler-writer can in the interests of economy cut out some of the organisational machinery involved in using the more sophisticated routine forms. The simpler routines do not

have any parameters, Some are only required for the earliest-stages of the bootstrapping.

The most powerful and general machinery is an adaptation of the format and format routine machinery described in the previous chapter. There are two for the 'compiling version' (or 'primary compiling routines') and the ordinary routine mechanism.

The ordinary routine mechanism will be discussed later, as it applies equally to the compiler-writer as well as the Compiler-Compiler-writer. It can be regarded as analogous to the conventional routine mechanism whereby an autocode programmer can specify a routine of instructions and a format by which it can be called. Then he can use this call in any other routine, and the instructions of the routine will be obeyed whenever the cue is encountered in the execution of his program (i.e. relative to the compiler-writer, when his compiler is translating source program),

When the source programmer uses a basic instruction of an autocode he expects that the requisite machine code instructions will be compiled on the spot. When he uses a routine call he knows that this will not happen, but instead instructions will be compiled ending in a jump to another section of the program, to the appropriate subroutine which carries out the required task. If he wants to specify a new format of the language, and specify what instructions must be compiled on the spot to execute the task, he cannot do it in terms of the autocode - the whole object of the Compiler Compiler is to provide the compiler-writer with a convenient language to describe how to translate from an actual variant of a format into the requisite machine code instructions.

In general the specification of how to carry out a particular task is different from the specification of how to compile the requisite machine code instructions to carry out the task. They are only the same when the task can be expressed entirely in terms of a set of more basic instructions in the given language, and then of course the routine mechanism can be used instead (see note on P.52). The source programmer using an autocode, and the compiler-writer using the routine mechanism of the Compiler Compiler, are only concerned with the former type of specification. However the Compiler-Compiler-writer has to be able to describe the latter operation, and in his own language.

Therefore a key mechanism of the Compiler Compiler which is not generally used by the compiler-writer is the 'compiling-version'. This is a routine which on recognition of a particular format in the Compiler Compiler language, e.g. [AB] = [WORD], e.g. A1 = B1, or B8 = B1 + 4, is entered immediately to compile the appropriate machine code instructions to be added to the routine of the compiler program (or Compiler Compiler) in which it occurs. Of course this code must be consistent with the conventions of the Compiler Compiler, e.g. it must be relocatable.

As will be illustrated (P.44) it is not always possible for the Compiler Compiler to compile the requisite code on recognition of a basic instruction, and

it has to use a routine mechanism. Therefore two routine forms must be associated with each basic format of the language, an ordinary routine version and the compiling-version.

Development of a compiler program

The following discussion will now be true not only for the <u>later</u> stages of the writing of the Compiler Compiler but also for the writing of a compiler program. Because of the complete integration of the two processes, the writing of the later stages of the Compiler Compiler is the same process as the writing of a compiler program using the Compiler Compiler; and equally the compiler program is added to the existing Compiler Compiler as if it was an official part of it.

As is the case when developing the Compiler Compiler, it is convenient for the compiler-writer store the current version of the compiler program on magnetic tape, only including additions or alterations to the system in the input for any particular development run. So the process of growth during development is identical in the two cases, except that the Compiler Compiler stops growing at a point where it is of general application, whereas the compiler program, starting from a copy of the Compiler Compiler, grows on to it in a particular way adapted to the autocode it is designed to implement.

Thus apart from the fact that there are no [SS] formats in the Compiler Compiler, the structure of the compiler program during development is the same as that of the Compiler Compiler.

When a compiler program is fully developed, a special declaration can be used, END OF PRIMARY MATERIAL, to eliminate from the final version of the compiler-cum-Compiler-Compiler those sections of the Compiler Compiler that are not required for the translation of source program (e.g. compiling versions for the basic language).

The languages of the Compiler Compiler system

The analysis machinery of the Compiler Compiler, which recognises the format of an instruction in terms of phrase and format definitions, is used both when compiler-compiling a compiler program and when compiling a source program.

Since there are differences in the conventions and organisation relevant to the two processes, the machinery operates in two distinct modes

1) Compiler Compiler Mode

(The translation of the compiler program by the Compiler Compiler)

2) Compiler Mode

(The translation of a source program by the compiler)

The usual development run will therefore start with additions and corrections to the current version of the compiler program, to be read and translated in Compiler Compiler mode. This will then be followed either by a 'DEFINE COMPILER ...' declaration to update the compiler, or by a declaration 'END OF MESSAGE' to

indicate a switch to compiler mode, the subsequent material being some piece of source program to test the new machinery. Often it will then be required to <u>obey</u> the piece of compiled source program; in this case, under the control of (say) the format routine for 'end of program', the compiler will pass control to the compiled program, maybe removing itself from store first, and instructions will then be obeyed in '<u>Source Mode</u>'.

When the system is in compiler mode, it is only required to recognise the source language. When it is in Compiler Compiler mode, it is required to recognise the language of the Compiler Compiler.

As has been mentioned, the routine mechanism of the Compiler Compiler is an adaptation of the format routine machinery. Here, instead of using a preset routine format, e.g. [<u>small-letter*</u>] (EXPRESSION][ANY-FURTHER-EXPRESSIONS] as in the specimen language, the compiler-writer can define the format of his subroutines in phrase structure notation, and the parameters of the associated routines are phrase-variables as for (SS) format routines.

To keep the source language, the basic Compiler Compiler language and the language of these routines apart, there are 4 format classes built in to the system :

1) [<u>MP] Master Phrases</u> These are the principal declarations of the Compiler Compiler language, e.g. PHRASE, FORMAT, and ROUTINE.

2) [<u>BS] Basic Statements</u> This is the class of formats that represents the basic imperative language of the Compiler Compiler. They can only be recognised inside format routines in Compiler Compiler mode. In general, on recognising such an instruction, the Compiler Compiler will pass control to the corresponding compiling-version routine which will then compile the requisite machine code instructions to be added to the compiler program (or the Compiler Compiler), according to the Compiler Compiler conventions (e.g. relocatable programming).

3) [<u>AS] Auxilliary Statements</u> This is the class of formats representing the routines of the Compiler Compiler language. Some of these are already included in the Compiler Compiler ('permanent routines') and the compiler-writer will add more as convenient for the subroutine structure of his program. They can only be recognised in Compiler Compiler mode. In general, on recognising such an instruction, the Compiler Compiler will plant a cue, so that when the routine it is contained in is being obeyed, in compiler mode, a jump will be made to the associated format routine.

4) [<u>SS] Source Statements</u> This is the class of formats specifying the language of the source program. It is only this class that is recognised in compiler mode, and on recognition control is passed to the appropriate format routine, which is then obeyed – in the case of an imperative instruction, this will then compile machine code instructions to add to the source program, according to the source program conventions (which will in general be quite separate from the compiler).

Source statements can also be recognised in Compiler Compiler mode, in which case a cue will be compiled into the compiler program, so that when the instruction is encountered in compiler mode, control will be passed to the associated format routine – i.e. [SS] instructions in Compiler Compiler mode are treated exactly the same as [AS] routine calls.

Note that the division of the Compiler Compiler language itself into [BS] and [AS] formats roughly marks the point where in practice the completion of the Compiler Compiler can be achieved using its own machinery at very near full power. In practice most of these [AS] formats also have compiling-versions.

Note that although in general it is only compiling-version routines that are obeyed in Compiler Compiler mode, it is possible for an ordinary format routine to be obeyed in Compiler Compiler mode if a compiling-version calls it.

Auxiliary routines

The most general routine mechanism of the Compiler Compiler language is the auxiliary statement. This is a class of instructions that is defined in phrase structure notation, in the way described in previous chapters.

The conventions are exactly the same as for source format definition – there is no 'meta-meta-syntactical' convention like underlining square brackets. Therefore class word names must be kept distinct from the names of the basic language (e.g. [PI], [N], [AB]) and those of the source language.

Auxiliary phrases and formats can use phrases of the source language both auxiliary and source formats can use phrases of the basic language,

Phrase and format definitions of both the source language and the auxiliary language can be included anywhere in the program description (e.g. in between format routines), provided they are defined before they are used in format routines and before their own format routines.

It is possible to use an instruction in a format routine before its own format routine has been given. An instruction can occur in any format routine, including its own (i.e. it can be used recursively).

The format routine mechanism of an [AS] format is exactly the same as for an [SS] format, except that it can never be entered direct from the master-routine of the compiler program.

Any instruction occurring in a format routine can contain phrase-variables, provided that the instruction is the formal form, or a formal subform, of the format definition.

For example, consider the basic statement [AB] = [WORD]

(where the definition of a [WORD] has not been given).

Permissible instructions satisfying this format are for example

B5 = A18, A23 = (B1+2), A8 = A7 + 15, [AB/1] = B17, A36 = [WORD],

[AB] = [AB/2] + [N], [AB] = [WORD/1].

Where [AB] + [N] is a formal subalternative of [WORD].

Note however that e.g. [AB] = [AB/2] + [INTEGER] would not be recognised as a

member of this format. For although [INTEGER] is defined to recognise the same set of symbol string as [N], it is not a formal alternative phrase or subphrase of the definition of [N] – obviously so in this case [N] is a name of the basic language, and so was defined first and independently.

This example demonstrates the case where a compiling-version of a [BS] statement cannot be used, it will be used for the first 3 examples, but the other 4 all contain phrase-variables whose values will not be known until the instruction are obeyed in compiler mode. Therefore for the <u>parametric</u> uses of the instruction cues must be planted to the 'ordinary' format routine associated with the format, so that the instruction can be carried out interpretively during compiler mode when the values of the phrase-variables are known.

Remembering that a format is merely a special class of phrase the effect of a routine cue on execution in compiler mode is exactly the same as if it was regarded as a [PHRASE-EXPRESSION-VALUE] in the routine it is contained in. That is, when it is obeyed it defines a unique phrase-value (instruction), being the symbol string of the instruction with the current values of any [PHRASE-VARIABLE-VALUE]s in it substituted at the appropriate places. The cue therefore generates this symbol string, which defines the unique variant of the format required to be obeyed at this point, and then resolves it into the routine heading of the corresponding format routine, thus setting up the initial values of the principal phrase-variables of the format routine (see example 6).

The cue mechanism for parametric forms of the basic instructions, and for all calls on auxiliary routines (except for nonparametric calls for [AS] routines which have compiling versions), and for all source statements occurring in format routines, thus simulates the familiar mechanism described in the previous chapter where the master-routine, having recognised a symbol-string of the input stream as corresponding to an instruction of the source language, notes the symbol strings matching the class words in the format and passes control to the associated format routine with these symbol strings as the initial values of the formal parameters.

On return from the format routine (via an END directive) the next instruction after the cue in the routine it was contained in will be obeyed. This might be a basic set of machine code instructions corresponding to a nonparametric form of a basic instruction (i.e. which was compiled using a compiling-version), or it may be another routine cue.

<u>Examples</u>

There are a number of different ways in which routines and phrase-variable parameters can be used. The following examples demonstrate some of these usages:

1) <u>Use of source symbol-string phrase-variable as parameters</u>

One of the simplest examples of the use of auxiliary routines is where a routine is required to process a phrase-variable of the source language in some consistent manner.

For example, where [SEP] is a separator in Compiler Compiler mode :

FORMAT [AS] = FIND ADDRESS AND TYPE FOR [VARIABLE] [SEP]

FORMAT (AS) = COMPILE INSTRUCTIONS TO SET ACCUMULATOR =

VALUE OF [EXPRESSION][SEP]

FORMAT [AS] = COMPILE INSTRUCTIONS TO SET [VARIABLE] =

CONTENTS OF ACCUMULATOR[SEP]

Here the format, routine heading, and routine call will all tend to look the same (except for numbering in the phrase-variable names), and the routine call will just pass on the current value of the phrase-variable to the appropriate format routine.

For example the call : FIND ADDRESS AND TYPE FOR [VARIABLE/1]

to :

ROUTINE [AS] ≡ FIND ADDRESS AND TYPE FOR [VARIABLE] [SEP]

-> 1 IF [VARIABLE] IS OF THE FORM [ARRAY-BASE] [[] [EXPRESSION]]

RESOLVE [VARIABLE] INTO [ELEMENT]

A1 = CATEGORY OF ELEMENT

B28 = (B4+A1)

B29 = (B5+A1)

B30 = 0

END ((i.e. 'return'))

1)   ...... etc.

> B4 & B5, say, are the positions in the private store of the lists for the address and type for each of the 26 possible element names. 'B30 = 0' says that the [VARIABLE] is not an array element.

Note that the phrase-variable is not the same as that in the routine call; it is in effect a copy of it (i.e. a call-by-value). Any reassignment of the [VARIABLE] in the routine would not affect the value of [VARIABLE/1] in the routine calling it.

Note that the routine passes on the required information using the global variables B28, B29, and B30.

2) **Use of non-source _phrases to sub-specify instructions**

Special class words can be defined for [AS] formats to distinguish variations on the job to be done in a routine.

For example in FIND ADDRESS AND TYPE FOR [VARIABLE] the compiler-writer may decide that he wants to carry out some extra operations if the variable (of the source program) is being assigned rather than its value referred to, and that he may sometimes require the routine to compile the necessary instructions to calculate the array index in the case where the variable is an array element and the index is an expression. He might then define (where 'M' stands for 'mode') :

PHRASE [M5] = REFERENCE, ASSIGNMENT

PHRASE [M6?] = (FOR IMMEDIATE ACCESS), NIL

Then :

FORMAT [AS] = FIND ADDRESS ([M5]) AND TYPE FOR [VARIABLE] [M6?][SEP]

This format could be called by e.g. :

FIND ADDRESS (REFERENCE) AND TYPE FORM [VARIABLE]

FIND ADDRESS (ASSIGNMENT) AND TYPE FOR [VARIABLE/3]

FIND ADDRESS (REFERENCE) AND TYPE FOR [VARIABLE/1] (FOR IMMEDIATE ACCESS)

FIND ADDRESS (ASSIGNMENT) AND TYPE FOR [VARIABLE] (FOR IMMEDIATE ACCESS)

Then in the format routine, the 4 subspecifications could be isolated by e.g. :

-> 6 IF [M5] IS OF THE FORM REFERENCE

-> 21 UNLESS [M6?] IS OF FORM (FOR IMMEDIATE ACCESS)

etc.

3)   Handling conventional variables of the Compiler Compiler language

There are no facilities for using conventional variables as parameters of auxiliary routines. However the same effect can be achieved by using their names as phrase-variable parameters.

For example we might like to turn into a subroutine the sequence given at the end of the previous chapter for converting a symbol string [INTEGER] into its value in conventional variable form :

FORMAT [AS] = CONVERT [INTEGER] INTO [AB] [SEP]

Called by e.g. :

CONVERT [INTEGER] INTO A5

CONVERT [INTEGER/2] INTO B21

With routine :

ROUTINE [AS] ≡ CONVERT (INTEGER] INTO [AB] [SEP]

RESOLVE [INTEGER] INTO [DIGIT][ANY-FURTHER-DIGITS]

    Al = 0

1) Al = Al × 10

    A2 = CATEGORY OF [DIGIT]

    Al = Al + A2 − 1

    -> 1 IF (ANY-FURTHER-DIGITS] IS OF THE FORM [DIGIT][ANY-FURTHER-DIGITS]

    [AB] = Al

END

The instruction that sets the conventional variable to its desired value is [AB] = Al, a parametric form of a basic assignment instruction. In the two calls shown above, the effect of this instruction will be A5 = Al and B21 = Al respectively,

Note that as it stands the instruction A5 = Al is ambiguous, as the [A]s are local variables and, as in this case, can belong to different routines. Therefore the Compiler Compiler makes a special case in recording the symbol-string

value of the phrase-variable [A] (and also of a label) so that it is possible to tell which routine out of the stack of routines currently being obeyed the actual symbol-string occurred.

Note that the call could have been

CONVERT [INTEGER] INTO [AB/1]

where [AB/1] was itself a conventional-cum-phrase-variable parameter of the routine containing the call; in this case the relevant local variable would be coming from an even higher level routine.

Note that again the phrase-variable [AB] in the routine is only a local copy. [AB/1] in the call is not reset. This is obvious in the previous calls where there is an explicit variant A5 or B21 in the instruction. However the effect of the 'CONVERT ...' routine relative to conventional parameters is a call-by-substitution. This is true of all variables and expressions relative to conventional parameters, i.e. the values of expressions and the addresses of variables are recalculated on each reference to the formal parameter.

4)   General routine calls

In all the examples so far the relation between format and routine call has been simple. That is, either the call has handed on a source phrase-variable, e.g. [VARIABLE] or [INTEGER], or it has contained a basic form of a non-source class word, e.g. (FOR IMMEDIATE ACCESS), or it has contained a basic or parametric form of the conventional variable [AB].

However any routine call can contain any subform of the defined format, provided it is a formal subform.

In the examples given above, variations on the calls using different subforms are not likely to arise in practice :

e.g. FIND ADDRESS AND TYPE FOR e

is not likely to occur, as no general compiler is likely to want to deal specially with a particular element.

And CONVERT 74 INTO A5

would be an extravagant way of carrying out the basic instruction A5 = 74.

However situations can arise when the routine call contains nonbasic subphrases of class words.

For example in the routine

ROUTINE [SS] ≡ [INTEGER] [if,unless][EXPRESSION/1][COMPARISON-SYMBOL]
                                                    [EXPRESSION/2][;]

Where the first thing to do might be

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF
                                        ([EXPRESSION/1]) - ([EXPRESSION/2])

Note here that '[EXPRESSION/1] - [EXPRESSION/2]' is not a formal subphrase of [EXPRESSION]/ Instead we must write '([EXPRESSION/1] - [EXPRESSION/2])'.

5) <u>Resetting phrase-variables</u>

None of the uses of auxiliary routines so far described have affected the values of any phrase-variables involved in the routine cue. There are however some restricted ways in which this can be done.

A formal 'call-by-reference' facility has not been implemented in the Compiler Compiler. There is no particular reason why this has not been done, except that it has not been required much in practice and it can be 'got round' by experienced users using existing informal facilities.

As there is so little that can be done with the existing formal facilities, the following description will be given only as a note, Note however that it would be fairly easy to add one or two formats to the Compiler Compiler (to simulate calling a variable by value) that could be used formally by inexperienced users to achieve greater power than at present.

The phrase-variable identifying parameter [PI] (i.e. [PHRASE-VARIABLE] or [PHRASE-VARIABLE-VALUE]) can itself be used in a fomat, to give a means of 'calling by reference', i.e. dealing directly with the phrase-variable in the call instead of a copy of it. There is a special phrase-handling instruction not described in the previous chapter that can then reset this variable, i.e. [PI] = [AB], but this requires fairly expert knowledge of the implementation of the Compiler Compiler to use safely.

The basic phrase-handling instructions for assigning phrase-values as given in the previous chapter are :

```
LET [PI] = [RESOLVED-P][SEP]
        i.e. RESOLVE [PHRASE-VARIABLE-VALUE] INTO [PHRASE-EXPRESSION]
-> (LABEL) [IU][PI] ≡ [RESOLVED-P][SEP]
        i.e. -> [ABN][IF,UNLESS] [PHRASE-VARIABLE-VALUE] IS OF THE FORM [PHRASE-EXPRESSION]
LET [PI] = [GENERATED-P][SEP]
        i.e. SET [PHRASE-VARIABLE] = [PHRASE-EXPRESSION-VALUE]
```

In each of these, the Compiler Compiler on recognising the instruction uses the phrase definition of the particular [PI] to analyse the phrase-expression following it, It must know explicitly which [PI] it is, and the phrase-expression must (therefore) follow it.

Therefore (PI) cannot be used parametrically in a format routine unless the phrase following it is itself the parametric form of a phrase-expression. i.e. [RESOLVED-P] or [GENERATED-P]. Equally, [RESOLVED-P] and [GENERATED-P] can not be used in the place of [PI] (i.e. on the left hand side of the instruction).

Therefore [PI] cannot be involved in one of these phrase-handling instructions unless the [RESOLVED-P] or the [GENERATED-P] occurs in the routine heading, and therefore in the format, because they cannot be formed inside the format routine in any other way. Therefore a phrase-variable cannot be reset by a nonbasic instruction (except by the unorthodox method referred to above) unless the values involved in the resetting occur explicitly in the instruction, i.e. the instruction contains a [PHRASE-VARIABLE-VALUE] followed by a matching [PHRASE-EXPRESSION] into which it can be resolved, or a [PHRASE-VARIABLE] followed by a [PHRASE-EXPRESSION-VALUE] to which it can be reset.

Therefore routines can only be defined for fairly simple variations on the existing phrase-handling instructions.

For example, to make the conditional instruction as used in the previous chapter recognisable by the Compiler Compiler :

```
FORMAT  [AS] = -> [LABEL] [IF,UNLESS] [PI] IS OF THE FORM [RESOLVED-P][SEP]
ROUTINE [AS] ≡ -> [LABEL] [IF,UNLESS] [PI] IS OF THE FORM [RESOLVED-P][SEP]
    LET [IU] = IF
    -> 1 IF [IF,UNLESS] ≡ IF
    LET [IU] = UNLESS
    1) -> [LABEL][IU][PI] ≡ [RESOLVED-P]
    END
```

Note that [LABEL] (and not e.g. [ABN]) must be used as this is a special phrase-variable where the Compiler Compiler arranges to remember the particular routine the actual symbols occurred in (therefore [LABEL] should <u>not</u> in fact be used as a class word of the source language description).

It is possible to follow a [PI] by more than one phrase-expression, so if we wanted to be more ambitious we could define

```
FORMAT  [AS] = -> [LABEL][IF,UNLESS][PI] ≡ [RESOLVED-P] OR [RESOLVED-P][SEP]
ROUTINE [AS] ≡ -> [LABEL][IF,UNLESS][PI] ≡ [RESOLVED-P/1] OR
                                               [RESOLVED-P/2][SEP]
        -> 1 IF [IF,UNLESS] ≡ UNLESS
        -> [LABEL] IF [PI] ≡ [RESOLVED-P/1]
        -> [LABEL] IF [P1] ≡ [RESOLVED-P/2]
        END
    1)  -› 2 IF [PI] ≡ [RESOLVED-P/1]
        -> [LABEL] UNLESS [PI] ≡ [RESOLVED-P/2]
    2)  END
```

6)   [SS] instructions in format routines - formal macros

Consider a situation where it was required that the <u>add update</u> routine of the specimen program should be part of the specimen language. And further, it is required that all instructions should be compiled on the spot instead of using a cue-subroutine mechanism, i.e. as an '<u>open routine</u>' or a '<u>macro</u>'.

We could define :

```
FORMAT  [SS] = add update[VARIABLE][;]
ROUTINE [SS] ≡ add update [VARIABLE][;]
-> 1 unless [VARIABLE] < 100
[VARIABLE] = [VARIABLE] + c
return
1:  c = 100intpt([VARIABLE]/100)
    END
```

Note that 'return' still means 'jump to the first instruction to be obeyed after this routine has been obeyed' but the code compiled is different as an open routine does not require a link.

If the instruction 'add update Y[e]' were recognised this format routine would be entered with phrase-variable value [VARIABLE] a 'Y[e]'.

Therefore the successive symbol strings generated by the 5 cues in the format routine would be :

```
-> 1 unless Y[e] < 100
Y[e] = Y[e] + c
return
 1:
c = 100intpt(Y[e]/100)
```

   The effect of this format routine is therefore the same as if these 5 instructions (4 imperatives, 1 declaration) had been written in the program in the place of each call. As each of the sub-format-routines is entered the same resolution takes place in each case, E.g. for the first [VARIABLE] = [EXPRESSION] the format routine would have been entered with the formal phrase-identifier parameter [VARIABLE] set to the actual value 'Y[e]' and [EXPRESSION] = 'Y[e] + c'.

   To be more precise, the effect of the format routine is exactly the same as that shown on P.10 to illustrate the simple 'substitution' theory of routines, in the case where the actual parameter was again 'Y[e]'. In that case the effect (in execution) of the cue-subroutine mechanism is the same as if the instructions of the routine had been written down with each occurrence of a formal parameter, e.g. 'y', replaced by the corresponding actual parameter, e.g. Y[e]. In this case the actual compilation is the same as if each instruction of the format routine had been written down with each occurrence of a formal phrase-identifier parameter, e.g. [VARIABLE], replaced by the actual phrase-variable variant, e.g. 'Y[e]'.

   Therefore in general, a format routine which is written down as the corresponding conventional routine but with each formal parameter replaced by the corresponding phrase-variable compiles an open sequence of instructions unique to the actual parameters of the call; and this sequence is exactly that postulated in the simple substitution theory of routines, and the routine carries out the same action as if each parameter of the routine call had been implemented by substitution (using a cue-subroutine mechanism).

   Such a routine can be called a <u>formal macro</u>.

   [To be even more precise, the effect of a formal macro (and of the substitution model of a routine) is the same as if the routine call had been replaced by the instructions of the corresponding routine <u>enclosed in a block</u>, e.g, <u>BEGIN</u>; ....; <u>END</u>, with each occurrence of a formal parameter replaced by the corresponding actual parameter. This overcomes the problem of local declarations (including labels) and 'return" is interpreted as 'Jump to the <u>END</u> of the block'.]

7)   [SS] <u>instructions in format routines – informal macros</u>
   Consider a format routine for the English autocode :

FORMAT [SS] = choose random [ITEM] [NUMBER] ([NUMBER]) or [NUMBER] ((NUMBER))
                                    [orNUMBER-WEIGHT<u>*?</u>][PAUSE]
Where PHRASE [orNUMBER-WEIGHT<u>*?</u>] = [S]or [NUMBER] ([NUMBER])
                                    [orNUMBER-WEIGHT<u>*?</u>], NIL
E.g.choose random direction left(5) or right(5) or forwards(8) or backwards(2).

[Note that [S]s are being left out except where they occur at the beginning of phrases. [ITEM] is a variable in this language and [NUMBER] an expression; 'direction', 'left', 'right' etc. are simple variable names.]

The purpose of the routine is to set the [ITEM], e.g. 'direction', equal to one of the given alternative values (in this case 4 : 'right', 'forwards', and 'backwards') at random with probability weighted in the ratios given in brackets (e.g. 5:5:8:2). [See the next page for the source routine required for this example.]

The corresponding format routine is :

```
ROUTINE [SS] ≡ choose random [ITEM] [NUMBER/1] ([NUMBER/2]) or [NUMBER/3]
                                    ([NUMBER/4])[orNUMBER-WEIGHT*?][PAUSE]
    SET [NUMBER] = ([NUMBER/2]) + ([NUMBER/4])
    SET (orNUMBER-WEIGHT*?/1] = [orNUMBER-WEIGHT*?]          ((i.e. make a copy))
1)  -> 2 UNLESS (orNUMBER-WEIGHT*?/1] IS OF THE FORM
                            [S]or[NUMBER/5] ([NUMBER/6])[orNUMBER-WEIGHT*?/1]
    SET [NUMBER] = ([NUMBER]) + ([NUMBER/6])
    -> 1
2)  LOCAL GENERAL VARIABLES : limit, random-number
    generate a random-number between 0 and this (NUMBER].
    set the limit = [NUMBER/2].
    go to (1) unless the random-number is below this limit.
    set the [ITEM] = [NUMBER/1].
    FINISH.                                                  ((i.e. 'return'))
    A1 = 1
    ASSIGN VALUE A1 TO [N]
3)  [N]:                                        ((i.e. source program label = A1))
    -> 4 IF [orNUMBER-WEIGHT*?] IS OF THE FORM NIL
    A1 =  A1+1
    ASSIGN VALUE A1 TO [N]
    add [NUMBER/4] to the limit.
    go to ([N]) unless the random-number is below this limit.
    set the [ITEM] = [NUMBER/3].
    FINISH.
    RESOLVE [orNUMBER-WEIGHT*?] INTO [S]or[NUMBER/3]([NUMBER/4])
                                                    (orNUMBER-WEIGHT*?]
    -> 3
4)  set the [ITEM] = [NUMBER/3].
    END
```

If this format routine is entered after meeting the instruction :

'choose random direction left(5) or right(5) or forwards(8) or backwards(2).'
the initial settings of the formal parameter phrase-variables are :

    [ITEM] = 'direction'
    [NUMBER/1] = 'left'
    [NUMBER/2] = '5'
    [NUMBER/3] = 'right'
    [NUMBER/4] = '5'
    [orNUMBER-WEIGHT*?] = ' or forwards (8) or backwards (2)'
    [PAUSE] = '. '

Then the sequence of SS instructions complied is in effect :

```
LOCAL GENERAL VARIABLES : limit, random-number
generate a random-number between 0 and (((5)+(5))+(8))+(2).
set the limit = 5.
go to (1) unless the random-number is below this limit.
set the direction = left.
FINISH.
1: add 5 to the limit.
go to (2) unless the random-number is below this limit.
set the direction right,
FINISH.
2: add 8 to the limit.
go to (3) unless the random-number is below this limit.
set the direction = forwards,
FINISH.
3: set the direction = backwards.
```

Note that ASSIGN VALUE [AB] TO [N] is an [AS] instruction of the given Compiler Compiler language. A label in the English autocode is [N] (i.e. [N], an integer). 'Generate a random-number ...' is an [SS] instruction of the English autocode.

Note in this example that the format of the routine contains a class word, i.e. [orNUMBER-WEIGHT*?] that is not one of the net of class words corresponding to permitted formal parameters for routine headings (e.g. (ITEM], a variable, or [NUMBER], an expression); in fact this class word has been created specially for this format. Similarly the instructions of the format routine do not consist of [SS] instructions only, but include [BS] and [AS] instructions as well.

This routine is therefore not a 'formal macro', but an informal macro.

[Note that the example of the formal macro illustrates the remark made on page 40 :

The specification of how to carry out a particular task is only the same as the specification of how to compile the requisite machine code instructions to carry out the task if the task can be expressed entirely in terms of a set of more basic instructions in the given language.

It should be observed that an instruction e.g. '-> 1 unless y < 100' occurring in a source program routine describes what action is to be taken on meeting this instructions when the routine is being obeyed. The same instruction occurring inside a format routine does not mean that 'go to (1) unless y < 100' is to be obeyed when this instruction is reached during the execution of the format routine, but instead 'COMPILE INSTRUCTIONS TO -> 1 unless y<100'.

In the case of example (6) this distinction is blurred, but the situation is much clearer in the case of (7) since the format routine contains many [BS] and [AS] instructions of the Compiler Compiler language. These instructions obviously describe actions which are obeyed when the format routine is being obeyed, and so in this context it is clear that the [SS] instructions are not obeyed when the format routine is being obeyed, but when the corresponding open routine is obeyed at execution tine. Any possible confusion can be avoided if we always remember to mentally preface each [SS] instruction occurring in a format routine with e.g, 'COMPILE INSTRUCTIONS TO ...'. Then every instruction in the format routine reads correctly as an operation to be carried out at compile time.

In fact if it happened that a source language required the same syntax for certain instructions as that of the Compiler Compiler language, then such an [SS] instruction occurring in a format routine would be ambiguous. If it had been thought likely in designing the Compiler Compiler that such occurrences would be common, then it would have been decided to prohibit [SS] instructions

from format routines, and instead a new [AS] format would have been provided to cover this case, e.g. :

FORMAT [AS] = COMPILE [SS]

Therefore a formal macro description and a source routine description look the same (apart from the formal parameters) simply because the 'COMPILE' before each [SS] instruction in the format routine can be left out. Conversely because whether or not we should preface each [SS] instruction with 'COMPILE' is implicit in the type of routine it is contained in (i.e. a format routine or a source program routine) it is possible for the body of the routine to be the same in both cases. That is, in the case of a formal macro the two specifications as stated in the opening paragraph are the same.]

Uses of format classes other than [SS] [AS], and [BS]

A format class is an alternative form of phrase definition. It has the special property that its alternatives can be defined in separate declarations and that a format routine can be associated with it.

It can be handled exactly like a phrase, i.e. it can be included in phrase and format definitions, it can be used as a phrase-variable, and it can be involved in phrase-handling instructions,

But there is an exception in that CATEGORY OF [PHRASE-VARIABLE-VALUE] will not yield the normal number, 1, 2, ..., but the system serial number that has been allocated to it, whose only property relative to the imagined category number is that it steadily increases.

However the 'format-variable', has the property of being immediately resolvable into the associated format routine. It is therefore classically used in the situation where each phrase of a set of phrases has just one specific operation associated with it, which is independent of any others of the set. The most obvious candidate for a format class in the previous discussion is the [FUNCTION].

The permanent functions have the property that they cannot be recognised on their own as a single instruction of the source program, but otherwise they are distinct operations of the language, e.g like the permanent routines.

So in the specimen autocode it may be more convenient to define :

FORMAT CLASS [FUNCTION)

                        ((this declaration introduces a new format class))

FORMAT [FUNCTION] = sqrt ([EXPRESSION])

FORMAT [FUNCTION] = cos ([EXPRESSION])

FORMAT [FUNCTION] = intpt ([EXPRESSION])

FORMAT [FUNCTION] = radius ([EXPRESSION][,][EXPRESSION])

etc.

FORMAT [FUNCTION) = [small-letter*] ([EXPRESSION][ANY-FURTHER-EXPRESSIONS])

This last format is the 'catch-all' to recognise any program-defined functions.

Then for example in the routine :

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF [EXPRESSION]

if any [OPERAND] in the [EXPRESSION] turns out to be a [FUNCTION], we could obey :

CALL R [FUNCTION]

which would call the appropriate format routine e.g.

ROUTINE [FUNCTION] ≡ radius ([EXPRESSION/1][,][EXPRESSION/2])

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF

<u>sqrt</u> (([EXPRESSION/1])((EXPRESION/1) + ([EXPRESSION/2])([EXPRESSION/2]))

END

Other uses of the Compiler Compiler

There is a special instruction the Compiler Compiler language for planting a machine code instruction in a location of the store. This has not previously been referred to. It in no way requires elaborate facilities of the Compiler Compiler, nor is the Compiler Compiler specially oriented to its use.

So if there is any other job than writing a compiler for which the main structure of a compiler program is convenient, then the Compiler Compiler can be used as an ordinary compiler with which to write the program.

A suitable Job might be any data-processing job which has a complex input data structure expressible in phrase structure notation. In the field of computer languages, it could equally well be used to translate source programs written in a particular language into another language (e.g. the assembly language of a particular computer), producing an output document that can be used as input to another compile.

Appendix
NOTES ON IMPLEMENTATION OF PHRASE VARIABLES
Observations on analysis trees

The analysis of a basic phrase with respect to a class word can be represented by an 'analysis tree' as shown in Chapter 2. A more complex example is given on P.60, of 'a+(pq-Q[j]/2)' with respect to the class word [EXPRESSION]. [The basic symbols have been picked out in quotes, and alongside some of the category numbers is given (as a reminder) the substring associated with the class word above.]

In such a representation it will he noticed that the subtree of each class word, e.g. the lower half of the tree, stemming from [OPERATOR-OPERAND*?] 'Q[j]/2', is 'independent' of the rest of the tree. That is, if the rest of the tree apart from that stemming from the class word was stripped away, we would be left with a formal analysis of the associated symbol string with respect to the class word.

In this example then the tree for 'a+(pq-Q[j]/2)' contains 37 independent analysis (sub)trees (including itself), that is one corresponding to each class word in the tree. 16 of the trees are 'degenerate', i.e. the tree ends immediately in a basic phrase (or NIL). The other 21 trees contain further (sub)trees If the alternative of a class word (given by the category number) is a basic phrase its tree is degenerate. Otherwise the tree branches at the next stage into a separate subtree for each class word in the alternative (non-basic) phrase.

Note for example that the branch point under '-Q[j]/2' which matches [OPERATOR-OPERAND*?] corresponds to a resolution instruction e.g.:

RESOLVE [OPERATOR-OPERAND*?] INTO [OPERATOR][OPERAND][OPERATOR-OPERAND*?]

```
         │
   1     │ -Q[j]/2
         │
         │
[OPERATOR][OPERAND][OPERATOR-OPERAND*?]
   │        │              │
   2  │ -    2 │ [Q[j]    1 │ /2
   │        │              │
```

This resolves the value associated with [OPERATOR-OPERAND*?] i.e. the subtree stemming from [OPERATOR-OPERAND*?] above '-Q[j]/2', to set up 3 new values corresponding to the subtrees stemming from [OPERATOR],[OPERAND], and [OPERATOR-OPERAND*?], representing values '-', 'Q[j]/2' and '/2' respectively.

When [EXPRESSION] is being dealt with, e.g. in :

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF [EXPRESSION]

in order to extract all the information about it, at each of the 21 branch points it will be necessary to have a resolve instruction (or a successful test instruction) and for each of the 16 degenerate points it will be necessary to use a 'CATEORY OF' instruction (or an unsuccessful test for the NIL alternatives).

Note however that although some of these 37 instructions may be obeyed as separate instructions in the format routine, many will be carried out in subroutines, or by the same instruction obeyed repeatedly, or by the same instruction in a recursive use of the routine, e.g. for sub[EXPRESSION/1] 'pq-Q[j]/2', to :

COMPILE INSTRUCTIONS TO SET ACCUMULATOR = VALUE OF [EXPRESSION/1]

Representation of a phrase-variable

Consider now the problem of representing a phrase-variable. Since the analysis tree so closely represents the information required in the phrase-handling instructions, it will be expected that the representation in store will reflect the same tree structure. The representation of an analysis tree in store is called an 'analysis record'.

As is the general practice with variables, associated with each phrase-variable is an address (fixed relative to the local data space) defining a certain number of locations to hold its value. In the case of a degenerate tree the only information we require is the category number, and this could be held in just one location at this address. Remember that each phrase-variable is of a different 'type', i.e. each phrase-value is interpreted in relation to the corresponding class word definition; therefore there is no need for example to store the system serial number of the class word with the category, or to store the actual symbol or symbol string of the basic phrase as this is implicit is the category number.

However for a non-degenerate tree it is clear that (e.g. where recursion is being used) there is no limit to the amount of information that can be contained in an analysis tree, and so it is impossible to allocate a set number of locations
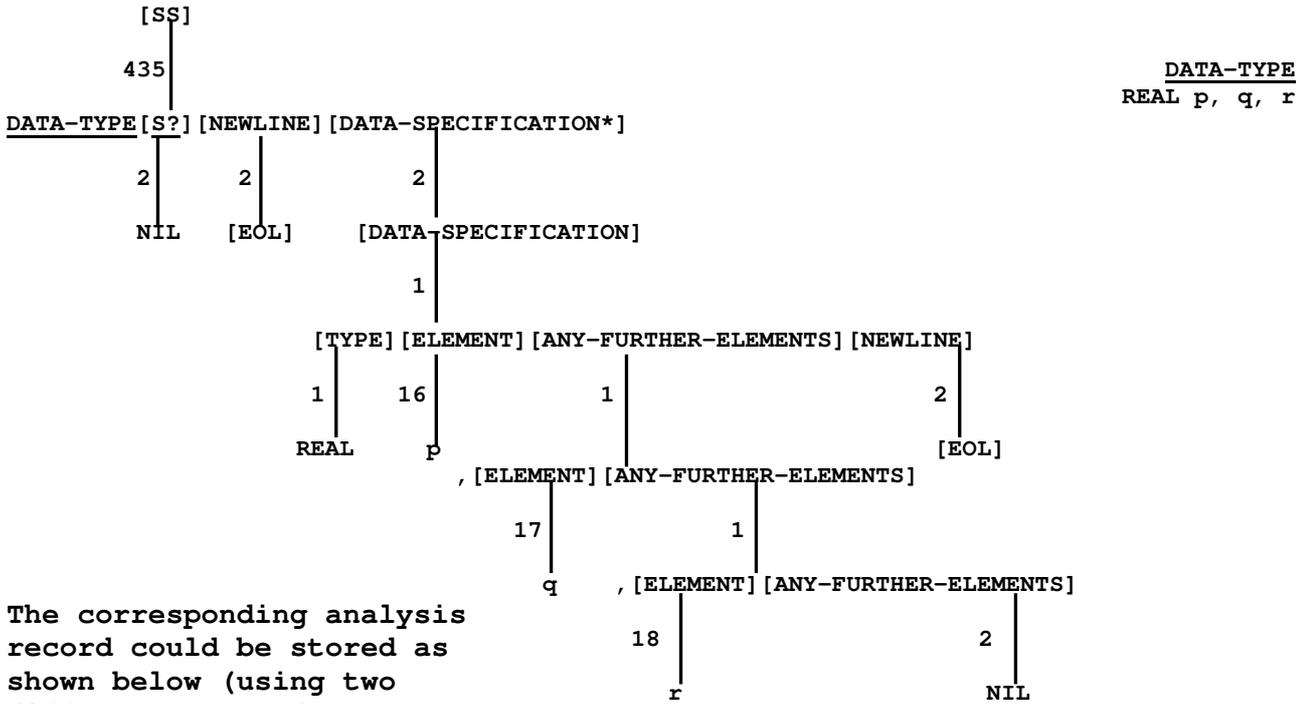
in advance to a phrase-variable. Therefore it is necessary to adopt the standard procedure in this situation of allocating just one fixed location at the address associated with the 'name' of the phrase-variable, and this location always contains the address of the (rest of the) 'value' of the phrase-variable; this value can then be allocated storage as convenient each time a new value is assigned to the phrase-variable. The fixed location(s) associated with a variable can be called the 'name-location'.

For generality the same procedure is adopted for all phrase-values. Thus even for a degenerate tree the store representation (i.e. the analysis record) is an address in the name-location which 'points' to another location anywhere in store which contains the category number.

Address	Contents
S0	Information
S1	necessary for
S2	'stack'
S3	organisation
S4	"
S5	Address of tree for [SS}
S6	e.g. S'4 " [S?]
S7	e.g. S'5 " [NEWLINE]
S8	[DATA-SPECIFICATION*]
S9	S'8 [DATA-SPECIFICATION]
S10	Address of tree for [TYPE]
S11	S'14 " [ELEMENT]
S12	" [ANY-FURTHER-ELEMENTS]
S13	" [ELEMENTS/1]
S14	" [INTEGER]
S15	" [ARRAY-BASE]
S16	" [INTEGER/1]
S17	" [INTEGER/2]
S18	" [ANY-FURTHER-ELEMENTS/1]
S19	Value of A0
S20	" A1
S21	" A2
S22	" A3
S23	" A4
S24	" A5
S25	" A6
S26	" A7
S'0	.
S'1	.
·	·
·	·
S'4	2
S'5	2
·	·
·	·
S'8	1
·	·
S'14	16
·	·
·	·

Consider the local data space ('stack') organisation, for example of the ROUTINE[SS] ≡ DATA-TYPE[S?][NEWLINE][DATA-SPECIFICATION*]

Here $S[N]$ represents the $[N]^{th}$ location in store after the base address S0 for the local data space of the routine and $S'[N]$ is a location in another area of store S' containing the category numbers, etc.

Assume that the phrase-variables used by the routine are as shown on the list opposite (in order of appearance), and that the highest local conventional variable $A[N]$ used is A7.

The examples of degenerate trees whose full analysis record is shown are those for [S?], [NEWLINE], and [ELEMENT], with category numbers 2, 2 and 16 respectively, which imply the symbol string value NIL, [EOL], and 'p'.

Consider now the problem of storing the value of a non-degenerate tree, e.g. for [DATA-SPECIFICATION] with the first alternative phrase [TYPE][ELEMENT][ANY-FURTHER-ELEMENTS][NEWLINE]. An analysis tree for this phrase (without the [NEWLINE]) is given on P.20. The first thing to be contained at the address (e.g. S'8) pointed to by the name-location is clearly the category number still. Then the most obvious thing to do to is place the analysis records of the set of (e.g. 4) sub-trees it branches into in the locations following the category number.

We are now faced with the sub-problem of representing N analysis records in store starting from a fixed location (e.g. S'9) where N is the number of class words in the alternative phrase associated with the category number in the corresponding class word definition. But this is exactly the same problem as we have started from, in representing the values of the set of (e.g. N'=14) phrase-variables that are used in that format routine in the given area of store from S5 onwards.

We therefore use the same solution as before. That is, we follow the category number by N locations each giving (in order) the address of the (rest of the) analysis record of the corresponding class word in the alternative phrase. The rest of the analysis record can then be positioned anywhere in store.

This process can of course be continued indefinitely : all the locations pointed to by this sublist or addresses will contain category numbers; each one which corresponds to a non-basic subalternative will be followed by its own sub-sublist of addresses, etc..

For example consider the analysis tree of with respect to [SS], the source language. [Note that since [SS] is a format and not a class word the 'category number', e.g. 435, is the system serial number of the format and its associated format routine.]

DATA-TYPE
REAL p, q, r

```
          [SS]
           |
      435  |                                                      DATA-TYPE
DATA-TYPE[S?][NEWLINE][DATA-SPECIFICATION*]                      REAL p, q, r
      2|       2|         2|
      NIL     [EOL]     [DATA-SPECIFICATION]
                             1|
              [TYPE][ELEMENT][ANY-FURTHER-ELEMENTS][NEWLINE]
                 1|    16|          1|                    2|
                REAL    p                                [EOL]
                          ,[ELEMENT][ANY-FURTHER-ELEMENTS]
                            17|           1|
                             q          ,[ELEMENT][ANY-FURTHER-ELEMENTS]
                                          18|              2|
                                           r              NIL
```

The corresponding analysis record could be stored as shown below (using two different ways of representing the same storage).

Conventionally :

Addr.	Contents
S5	S'0
S'0	435
S'1	S'4
S'2	S'5
S'3	S'6
S'4	2
S'5	2
S'6	2
S'7	S'8
S'8	1
S'9	S'13
S'10	S'14
S'11	S'15
S'12	S'24
S'13	1
S'14	16
S'15	1
S'16	S'18
S'17	S'19
S'18	17
S'19	1
S'20	S'22
S'21	S'23
S'22	18
S'23	2
S'24	2

```
(S5) = S'0

                435 —S'4 —S'5 —S'6
                    /      /      /
                 2     2     2 —S'8
                                /
                   1—S'13 —S'14 —S'15 —S'24
                       /      /      /
                     1     16      2
                        1 —S'18 —S'19
                           /      /
                        17     1 —S'22 —S'23
                                  /      /
                               18     2
```
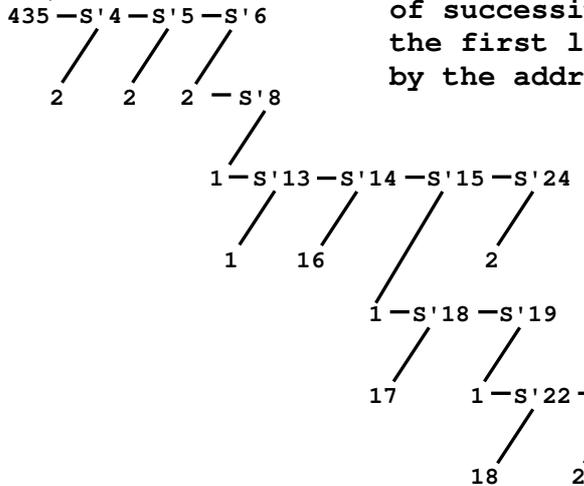
Or alternatively :
Here strings of hyphenated numbers; (or single numbers) represent the contents of successive locations; the address of the first location in the set is given by the address pointing (down) to it.

Note that although in this example the storage of subtrees has been systematic, this is not essential to the representation; treating each category number together with any addresses immediately following it as a unit, these units could have been placed anywhere in store.

Consider now the instruction

-> 5 UNLESS [DATA-SPECIFICATION] IS OF THE FORM
                    [TYPE][ELEMENT][ANY-FURTHER-ELEMENTS/1][NEWLINE]

The name-locations associated with these phrase-variables are S'9,S'10, S'11, S'18, and S'7 respectively, or where the system B-variable B72 gives the address of S0, the base of the local data space, B72+9, B72+10, ...B72+7.

Say that at the time the instruction is obeyed, [DATA-SPECIFICATION] has the value 'REAL p, q, r[EOL]', with analysis record held in the same locations as in the appropriate subtree above (stemming from S'8) :

| [DATA-SPECIFICATION] | (S9) = S'8 | 'REAL p, q, r[EOL] |

```
                                            (S9) = S'8
                                              /
                                    1 ─ S'13 ─ S'14 ─ S'15 ─ S'24
                                       /      /      /       /
                                      1      16            2
```

The required operation can be achieved by the following set of basic instructions (where B91 is a 'working-space' B-variable) :	1 ─ S'18 ─ S'19
	  / /
	17 1 ─ S'22 ─ S'23
	        / /
	       18 2

B91 = (B72+9)	Set B91 to the contents of the name-location S9 (e.g. = s'8).
-> 5 IF (B91) ≠ 1	Go to (5) if the category number in B91 is not 1 (it is).
(B72+10)=(B91+1) (B72+11)=(B91+2) (B72+18)=(B91+3) (B72+ 7)=(B91+4)	Set the name-location for [TYPE] to point to the address given in the location after the category number; set {ELEMENT} to point to that given in the next location, [ANY-FURTHER-ELEMENTS] to the next and [NEWLINE] to the address given in the 4th location after B91.

This sets up the new phrase-values :

```
e.g. : [TYPE]            [ELEMENT]        [ANY-FURTHER-ELEMENTS/1]     [NEWLINE]
      (S10)=S'13         (S11)=S'14          (S18) = S'15            (S7) = S'24
          /                  /                   /                       /
         1                  16          1 ─ S'18 ─ S'19                 2
                                          /        /
                                         17      1 ─ S'22 ─ S'23
                                                 /       /
                                                18      2
```
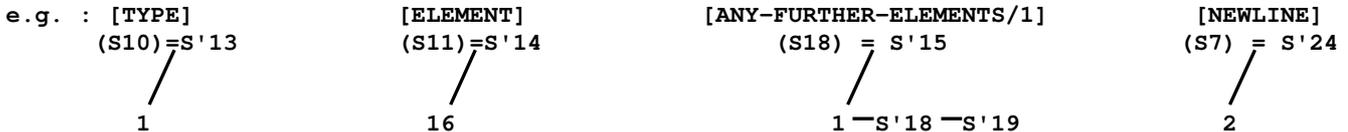
By reference back to the analysis tree on the previous page, it will be seen that [TYPE] is now set to 'REAL', [ELEMENT] to 'p', [ANY-FURTHER-ELEMENTS/1] to ',q,r' and [NEWLINE] to '[EOL]'.

   Note that, although a resolve instruction resets the value of each phrase-variable, no new locations are required to hold the new values. They automatically share the space used for the phrase-variable being resolved.

   A resolve instruction is implemented in the same way as a test, except that there is a jump-to-monitor if the category number is not correct.

   In place of the instruction e.g. A1 = CATEGORY OF [DATA-SPECIFICATION] would be compiler-compiled the basic sequence : B91=(B72+9), A1=(B91).

   Consider now a generate instruction,
E.g. SET [DATA-SPECIFICATION] = [TYPE][ELEMENT]
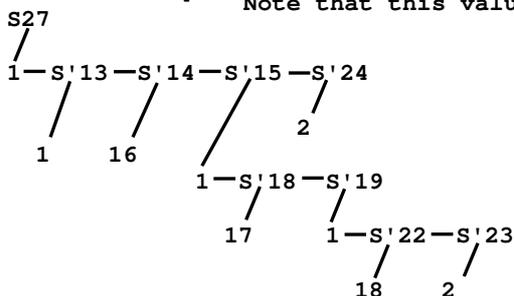   [ANY-FURTHER-ELEMENTS/1][NEWLINE]
Assume, say, that the value of the [DATA-SPECIFICATION] has been altered since the test instruction shown above was obeyed, but that the new values that were created then have not been; it is required to reset [DATA-SPECIFICATION] to its former value.

   This would be achieved by :

(B72+9) = B90 (B90) = 1 (B90+1) = (B72+10) (B90+2) = (B72+11) (B90+3) = (B72+18) (B90+4) = (B72+7) B90 = B90+5	Where B90 is the end of the local data space (e.g. S27) for the routine, we are going to use this area for the new space required for the new value. The first location contains the category number, the next the address in the name-location of [TYPE] (e.g. S'13), the next that for [ELEMENT], the next that for [ANY-FURTHER-ELEMENTS/1] and the 4th that for [NEWLINE]. Finally we add 5 to the end-of-stack to preserve the new space used.

```
[DATA-SPECIFICATION]
(S9) = S27
        /
     1 ─ S'13 ─ S'14 ─ S'15 ─ S'24
        /      /      /       /
       /      /      2
      1      16     /
                   1 ─ S'18 ─ S'19
                     /        /
                    17      1 ─ S'22 ─ S'23
                            /       /
                           18      2
```

This sets up the new analysis record shown. Note that this value is identical to the previous value of [DATA-SPECIFICATION], since the two trees have the same structure and each pair of corresponding category numbers match; but the area of store occupied by the analysis record is partially different, i.e. any exploration of the tree is now 'routed' through the 5 new locations S27-S31 instead of S'8-S'12 as in the previous analysis record.

Top-level [SS] routine implementationm

While the analysis routine is trying to recognise an (SS] instruction (in Compiler mode) it builds up an analysis record in a fixed location, say S'0; this is no trouble as the structure of an analysis tree exactly reflects the method of recognition.

At the stage where it has recognised an [SS] instruction it carries out certain organisation pertinent to the associated format routine, e.g. setting up the local data space at SO following the last location of S' it required for the analysis record (e.g. at S'25). Initially all the values of the A[N] variables and the contents of the name-locations are cleared to 0. B90 is set as appropriate (e.g. = S0 + 27). Then S'0 is set in the name-location of [SS], and the format routine is entered with a sequence resolving into the formal parameters,

E. g. for

ROUTINE [SS] ≡ DATA TYPE[S?][NEWLINE][DATA-SPECIFICATION*]

with analysis record as shown on P.57, the initial values set will be [S?] in S6 = S'4, [NEWLINE] in S7 = S'5, and [DATA-SPECIFICATION*] in S8 = S'6.

As always in the case of a resolution, the analysis records of the formal parameters therefore share the storage space of the original analysis record.

Subroutine [AS] implementation

The implementation of format subroutines (i.e. [AS] routines, and sometimes [SS] or e.g. [FUNCTION] routines) is closely analogous. When the analysis routine (in Compiler Compiler mode) recognises a routine call it stores the analysis record of the instruction with respect to its class in the cue. When the cue is obeyed (in Compiler mode) the analysis record is copied into the stack and the data space for the routine is set up underneath it. Then (e.g.) [AS] is set to point to the copy and the routine is entered with a resolution of the format class into the principal parameters.

However there is a difference from the top-level recognition machinery, because it is possible for an instruction to contain phrase-variables. If this happens the analysis record is truncated at each such point (i.e. where on trying to match a class word to the head of the symbol string it finds a phrase-variable of the appropriate type). Special information is planted at each truncation point so that when the analysis record is copied the appropriate subtree is attached to the main analysis record by filling in at this point the current address in the name-location associated with the phrase-variable.

Note that when a phrase-variable is called from the routine above in this way it is a call-by-value U a call-by-reference, A resetting of the phrase-variable in the subroutine will not affect the current setting of the corresponding phrase-variable in the routine above. The fact that the name-location of a phrase-variable always contains an address is not relevant; it contains a 'value address' not a 'variable address'. For a call-by-reference it would be necessary to have a special form of address which pointed to a name-location and not to a category number as is the usual case.

Note that in a conventional compiler there is rarely any need to use the SET ... instruction except to make copies of phrase-variables. All settings of phrase-variables tend to be done by resolutions. Therefore in the case of source material there is never any need to acquire further storage space as illustrated above. All the material of all the analysis records except for the name-locations themselves is therefore contained in the original analysis record of the [SS] instruction, Therefore in the DATA TYPE example all the 12 phrase-values that are created in the execution of the format routine will comprise addresses pointing to the appropriate area S' of the original analysis record. This is equally true if a source phrase-variable is passed down to a subroutine.

Note that it is not possible in the formal language to reset the contents of store locations involved in analysis records except the name-locations of phrase-variables. Therefore the area S' remains inviolate. If this area is interfered with (as is possible using informal facilities) then this is done at the compiler-writer's risk. Such an alteration may alter the current values of a number of phrase-variables (i.e. all those using this value as a subvalue). Such as arbitrary alteration contradicts the fundamental property of a variable, that it can only be altered by referring directly to its name (e.g. is an assignment statement, or as an actual parameter of a routine which resets the parameter).

It is of course a novel feature of the phrase-variable that different phrase-variables share the same store without mutual interference. This is an

economic advantage, and the fact that the store outside name-locations cannot be
altered in the formal language allows the implementation to take advantage of the
'sharing' property of analysis trees. [Note that if phrase-values had been represented
as symbol strings instead of analysis records it would still have been possible to
share storage space. E.g. S' would comprise the original [SS] string and name-locations
would now be pairs of addresses pointing to the first and
last symbols of the phrase-value. However in this case
every phrase-handing instruction except SET ... would
require entry to the analysis routine to re-analyse the
symbol-string involved, and SET instructions would not be
able to share existing space (in general.]

Analysis of 'a+(pq-Q[j]/2)

w.r.t [EXPRESSION]

```
                    1 |
[+?] [OPERAND] [OPERATOR-OPERAND*?]

3 |      2 |a            1 |+(pq-Q[j]/2)

 NIL   [VARIABLE]    [OPERATOR] [OPERAND] [OPERATOR-OPERAND*?]

         1 |          1 |        3 |                  2 |
      [ELEMENT]       '+'     '('[EXPRESSION]')'       NIL

       1 |                     1 | pq-Q[j]/2
       'a'
                 [+?] [OPERAND] [OPERATOR-OPERAND*?]

               3 |      2 | p        1 | q-Q[j]/2
                NIL   [VARIABLE]   [OPERATOR][OPERAND][OPERATOR-OPERAND*?]

                       1 |          5 |        2 |q          1 |
                    [ELEMENT]       NIL    [VARIABLE]

                     16 |                     1 |          q-Q[j]/2
                     'p'                   [ELEMENT]

                                            17 |
                                            'q'

            [OPERATOR] [OPERAND] [OPERATOR-OPERAND*?]

          2 |-        2 |Q[j]          1 | /2
          '-'       [VARIABLE]     [OPERATOR][OPERAND][OPERATOR-OPERAND*?]

                      2 |            4 |      1 |              2 |
          [ARRAY-BASE]'['[EXPRESSION]']'  '/'  [CONSTANT]       NIL

          17 |            1 |j                 3 |
          'Q'    [+?][OPERAND][OPERATOR-OPERAND*?]      [INTEGER]

               3 |    2 |         2 |                2 |
               NIL [VARIABLE]    NIL             [DIGIT]

                     1 |                          3 |
                  [ELEMENT]                       '2'

                    10 |
                    'j'
```