Studies in the Design and Implementation of Programming Languages for Symbol Manipulation

D.J.Rees

Ph.D.

University of Edinburgh

August 1969

INDEX

1. INTRODUCTION

Compared with the development of computing hardware, the development of programming languages has followed a different course. Hardware innovations such as the use of transistors and integrated circuitry have resulted in machines with very substantially improved capabilities, making older machines and even comparatively modern machines obsolescent. The programming languages currently in most widespread use, however, remain those which were already in use as many as ten years ago, namely FORTRAN, ALGOL 60, and COBOL. Nevertheless, considerable improvements can be made to these languages. The reasons why no improvements were made appear to be primarily twofold. Firstly, they are regarded as 'standard' languages, which in order to facilitate transferability of programs, has made them virtually immutable. Secondly, they can be employed in almost all programming situations without the need for change.

Instead, very many other languages have been designed and implemented with particular objectives in view, but which almost invariably limit their application to a narrow field. Only recently have attempts been made to unify some of the developments under the cloak of a single language (PL/1 and ALGOL 68). Data structures are a particular example of what features have been incorporated. There are still considerable omissions however. For instance, neither language has incorporated list processing or symbol manipulation facilities within its basic framework.

The latter seems to be most surprising. With the increased capabilities of modern computers and the consequent broadening of their range of application, techniques involving symbol manipulation are becoming increasingly important. Natural language processing such as the analysis of texts for authorship and mechanical translation, and formal manipulations, such as those involved in mechanical theorem-proving and algebraic formula manipulation are some obvious applications. The last mentioned, that of algebraic manipulation of formulae, is one of the most important applications. Several systems, notably FORMAC, have been developed for this purpose. With the advent of multi-access computing systems a much greater interaction between man and machine is becoming possible, where the advantages of algebraic manipulation and mathematical assistance packages are felt the greatest. This, further, demonstrates the need for symbol manipulation facilities to be available together with normal arithmetic facilities in a programming language, for not only must the formulae be manipulated but also they must be evaluated in normal arithmetic terms.

This combination has not completely satisfactorily been acheived in any languages developed in the past. The present investigation is an attempt to overcome this deficiency. A language called ASTRA has been the result. Before discussing the design and implementation of ASTRA, several existing languages are examined in order to discern the desirable properties of a language for symbol manipulation. It is the belief of the present author that the features of ASTRA described herein represent an advance on previous languages. The methods used in the ASTRA compiler are also described.

2. FEATURES OF SYMBOL MANIPULATION LANGUAGES

A fundamental consideration in the design of a programming language must always be the type of data which it is intended to manipulate. Thus numerically orientated languages have the means for handling integer quantities, real quantities and frequently also complex quantities, both for their storage and for operations on them such as addition, multiplication and so on. They also cater for scalar quantities on the one hand and arrays of scalar quantities on the other. These are highly suitable for a large proportion of numerical calculations, but may not necessarily be so for any other type of calculation. It is perhaps unfortunate that almost all languages are sufficiently general-purpose that they can be used for manipulations on any type of data. This has meant that there has been a tendency to ′make do′ with the existing facilities rather than to design and implement languages with more suitable facilities. It has been particularly true in the case of problems which can be formulated and solved by methods involving the manipulation of symbols. Any language with facilities for manipulating integer quantities can be used for this purpose by regarding the set of symbols concerned as a mapping onto the integers, or a subset of them. It is most likely, however, that the manipulations thus made available will not be suitable for operations on symbols - the ability to add the symbol ′A′ to the symbol ′B′, for instance, is of doubtful benefit. The operations which are usually required tend to concern a number of symbols considered together as a unit and not as single items. This being the case, probably the main reason why other languages have been used is because of their array type of data structures which can be organised in fairly simple ways to hold these groups of symbols. The further consideration that most languages have some means, however rudimentary, of inputting and outputting symbolic data, for alphanumeric headings in the case of output for instance, has also obscured the need for better and more suitable languages.

A simple view of the requirements of a language intended to be used for symbol manipulatilon is that it should have as a basic unit of data, upon which operations are performed, an ordered set of symbols. It is intuitively clear that it is a set of symbols rather than a single symbol which is required because the information content of a single symbol is relatively small when selecting from those conventionally used, say letters, digits, punctuation marks and a few others perhaps - a choice of around a hundred at most. This is as opposed to a single integer, for instance, when the information content may be considerably greater. The range available might be say -2**47 to +2**47 . Thus an integer is by itself a useful entity whereas a symbol is most probably much less useful. The further fact that integers are often grouped together in arrays for many problems makes it clear that a group of symbols (which could of course consist of one symbol if required) is likely to be the most useful basic entity. The ordering of the group is a further obvious asset in most applications. To be general purpose the entity should at least be capable of being used in an ordered form even if that facility is not used in particular cases.

By noting that a large class of problems can be approached by means of manipulations on symbols, it should not be forgotton that an even larger class deals with numbers and arrays of numbers. It would therefore seem extremely probable that a combination of symbolic and numeric working would be of great value. There seems no superficial reason therefore why the data entity should not also be capable of holding numerical data i.e. an ordered set of symbols or numbers. It could be regarded as a means of increasing the set of symbols available or alternatively as a grouping together of essentially different types of components. To compare these two possible views take the example of a polynomial expression. It could be treated as a collection of symbols some of which may be digits, namely the coefficients, whereas it is almost certain to be more useful to consider the coefficients as complete entities i.e. numbers, rather than collections of digits and to group these together with the symbols for the variables and operators. Taking the first view would create difficulties in this case as the numerical value of a coefficient might well be confused with the numerical equivalent used for internal storage of symbols. It would appear, therefore, that if numerical values are to be grouped with symbols, and this could certainly be useful, the second view ought to be taken, namely that the two types should be distinguishable.

The other single most important property that the data entities being considered can usefully have is a measure of internal structure - in addition to the simple ordering property. One of these groups of items i.e. symbols and/or numbers, may well be considered to have a subgroup of items which form an entity within the larger entity. Taking the example of algebraic formulae, the subgroup might consist of a bracketed sub-expression within the expression. Clearly, a group might have a number of sub-groups and sub-groups might equally well also have sub-groups within them. This form of structure can be superimposed, by means of programming conventions if necessary, onto unstructured groups of items but some built-in structuring mechanism is to be preferred if only to remove the onus of conventions. Furthermore, built-in structuring may remove restrictions of use caused by the conventions. For example the sub-groups might be delimited by using the left and right bracket symbols. Such a convention would imply a restriction on the use of those symbols as elements of a group.

While considering the form of data to be represented within the language, it is also necessary to consider the method of internal representation that is to be used. The most commonly favoured methods involve some form of list, in other words, an arrangement whereby each element of the group comprising the list is linked to other members of the group by means of addresses or pointers of some sort associated with each element. Sub-groups are easily represented in this kind of structure by having extra links as part of the main list, linking to other lists i.e. sub-lists, to give a tree structure. The variations possible on this sort of theme are legion, some of which will be discussed in succeeding chapters. The particular variation chosen clearly depends on the form of data and manipulations on it which the language is designed to cater for. For this reason also, lists may not be necessary at all - ordinary arrays of contiguous locations may be sufficient.

It is therefore necessary to consider what manipulations are commonly required on these units of data. The first essential is the ability to construct units. One method is to input from an external source a complete unit using conventions of some sort to delimit the constituents of the unit. More simply, a single constituent could be the form of input. Instead of input, literals, that is, the equivalent of sequences of digits etc. used to represent values of numerical quantities, can be provided in the language to form units having that value. In the case of symbols it will be necessary to delimit them in some way in order to avoid confusion and ambiguities. The form of literal will be most useful if it can represent the whole range of forms of data structure catered for by the system. For the inclusion of values of numerical constituents, a method of distinguishing between these and sequences of symbols which happen to be digits would be necessary. Similarly, the method of representing sub-groups of constituents must not conflict with the constituents themselves, for instance, surrounding them with brackets.

New data units will also be required having the value of those already in existence as well as from literals. As an example, a data unit might be required consisting of constituents having the value of an existing unit followed by that of a literal possibly followed by those of further units. Another capability should be the ability to form new units with sub-groups having the value of other units or literals or combinations of them.

An additional facility could be the ability to include in new units the value of just part of existing units, say one of its sub-groups or perhaps the first few of its constituents.

Almost all languages have a facility for defining functions whether the values produced are numerical or otherwise. If the language provides functions which have as their values these data units, these should be available as operands just as existing units are.

Having the ability to form new units is only the first stage. In certain types of problem, it is necessary to be able to alter the value of an existing unit. For example, in the case of dictionaries; when new entries are made it should be possible to make the neccessary additions without creating a completely new dictionary on each occasion. This is one of the reasons why list proscessing techniques are popular since the insertion of extra items or deletion of items can be performed at very low cost simply by altering the linking information between items. When arrays of consecutive locations are used, insertion and deletion can be time-consuming and wasteful of space unless great care is taken. This method is only likely to be used therefore where such processes are infrequently occurring.

The remaining facility is that of the examination and testing of the values of data units. Having formed a unit or altered a unit in some way, it will be necessary to compare either all of it or parts of it with other units to control the future course of action of the program. Just as it is convenient in the formation of units to be able to select parts of existing units, the first constituent for example, for inclusion in the new unit, the course of the program is very likely to be determined by the value of just part of a unit, the first constituent again for example, and not necessarily by the value of the whole unit. In effect, the requirements of operands in expressions for the formation of new units are precisely the same as those required for testing purposes.

The way in which the manipulations are represented and the structure of the language in general must next be considered.

When a functional approach is used, as in LISP, discussed in the next chapter, the data units manipulated are the values of functions which are defined in terms of a basic set of operations and other functions themselves defined in the same way. This can be set against the kind of approach typified by SLIP, also discussed in the next chapter, where in addition to functional values, data units may have an independent existence of their own, the values of which can be manipulated and examined by a sequence of operations each of which are essentially independent. It is usual to assign names to such data units, although in certain systems the idea of an unnamed 'workspace' is introduced e.g. COMIT, into which units are loaded for operations to be performed on them and removed when the operations are complete. However, even in these systems, the backing-area from which units are loaded and to which they are returned, consists of named locations. In either case, it is very useful to be able to give some mnemonic significance to the names in order to aid the program writing and to this end the use of fixed names or a choice from a fixed set of names is less useful, although not prohibitively so. In particular, the ability to choose names with some mnemonic significance helps greatly in the process of getting an overall view of the problem without having to remember such details as the fact that this location contains what I am using for that and so on.

Two methods of storage of data units have been mentioned - lists and arrays. There remain, however, many alternative schemes for controlling the total storage of all units. Some languages, such as ALGOL, have a block structure which offers a convenient method of storage control using a stack for holding both scalars and arrays. Data units containing symbols can be of variable length depending on the number of constituents they contain. This is one of the main difficulties in systems using arrays rather than lists to hold the data units. If sufficient space is allocated for the maximum size of each unit, either by the language system or by the programmer stating the size explicitly, most of the space is likely to be wasted most of the time. Alternatively, amounts less than the maximum required can be allocated with the consequence that the storage may have to be rearranged every so often. When list processing techniques are used, a bank of unused storage, itself a list, called an Available Space List or Free List, can be used, from which cells may be taken to form part of data units. As regards returning cells to this list when they become free, they can either be returned immediately they become free or alternatively left until the free list becomes exhausted - if ever - and then all collected up into a new Free List - a process known as 'garbage collection'.

Recursive facilities are likely to prove extremely useful in view of the inherently recursive nature of many of the problems which recommend themselves to solution by symbol manipulation techniques. Methods of acheiving this are often combined with a stack for storage control as in ALGOL for instance. Programs for certain systems are allowed to modify not only the data units but also the program itself. Indeed the program may be a data unit of the same form and in these types of system the facility is easily included. Whether this is altogether desirable is a matter for comment. It may be that the capabilities of the system are substantially improved with this facility. On the other hand, it may seriously detract from the comprehensibility of the program with the consequent difficulties of debugging and modification of the program.

However great the capabilities of a language, it must always be possible to use it easily and conveniently so that the overall picture of the problem is not lost. This usually implies the use of concise notations, but not to the point of destroying clarity. The greatest degree of latitude consistent with unambiguity is desirable rather than any fixed framework. Although the semantic content of the program is the most important, which the expressiveness of the language is designed to cater for, the syntactic details are not trivial in practice and relaxation of strict rules can pay dividends if only in the lesser degree of frustration in the programmer. Errors and blunders in programs are the inevitable consequence of human fallibility and the more the language system takes note of this fact by way of providing useful error messages and diagnostic facilities, the happier the problem solver is likely to be - with the attendant beneficial effects on his project. For instance, when a program fails, the position of failure relative to the original source text and as much information as possible relevant to the existing situation should be given. Fortunately the days of core-dumps are numbered - they are singularly difficult to abstract useful information from when list processing is involved, since this necessitates continual references to widely separated locations in the store.

Other contributing factors to the ease of using any system are the subsidiary facilities available. The usefulness of arithmetic facilities has already been mentioned, the provision of which extends the range of problems that can be solved to beyond those of a purely symbolic nature. For problems which are large in relation to the storage space available some form of backing store and facilities for using it conveniently are clearly valuable. Therefore the language and its structure should be designed bearing this in mind.

A remaining consideration is the efficiency of the system. However fast and powerful the machine being used, the more efficiently it is used the more problems and longer problems it will be available for. Inevitably, fast compilation and fast running are virtually mutually exclusive, so that a choice usually has to be made between the two. Which is concentrated on will depend on the use to which the system is to be put and the mix of programs presented to it. In an experimental and research environment, in which symbol manipulation systems are mostly used at present, such as a University, there is likely to be a large amount of program development and testing and less running of production programs over a long period, although problems involving large-scale search procedures such as theorem-proving might tend to contradict this. At the one extreme are the minimal compile-time but usually slow running interpretive systems while at the other are the multi-pass and time-consuming compilers which aim to produce optimum code. Production of code with an optimum performance in a symbol manipulation system may well be more difficult than in, say, a language for numerical calculations in view of the variability in type and size of the data units and the many types of problem to be solved. One particular problem may be amenable to array storage techniques while another may only be suitable for list processing. To cater for such widely different mechanisms would be beyond all but the most highly sophisticated systems. A midway course is likely to satisfy the largest proportion of users.

3. EXISTING SYMBOL MANIPULATION LANGUAGES

The discussion of the previous chapter can be divided into a number of areas which also form a useful basis for consideration of some existing languages designed for symbol manipulation work. These areas can be summed up under the general headings:

1. Form of data representation
2. Manipulations available
3. Program structure
4. Ease of use and debugging
5. Subsidiary facilities
6. Efficiency

The languages or systems which may be considered the most significant either historically or in terms of common availability of use or features present are discussed below. These are IPL-5, SLIP, LISP 1.5, COMIT, and SNOBOL.

IPL-5

IPL-5 is the fifth, but only significant, member of a series of Information Processing Languages, developed by Newell and Tonge of the RAND Corporation around 1960, as a result of their desire to apply computers to heuristics and the simulation of cognitive processes.

1.

The language is designed to manipulate lists and list structures of which the constituents are ´IPL symbols´. These latter can be chosen by the programmer, subject to certain rules. For example, the ´regional symbols´ take the general form of a letter or punctuation mark followed by a positive decimal number. These are the equivalent of ´identifiers´ of languages such as ALGOL, FORTRAN etc. They can either be used as data elements in their own right, as names of sub-lists, or as names for locations which may hold data in the various permissible forms - alphabetic, integer, floating point or octal.

For instance,

NAME	SYMB	LINK
X1	0	
	X2	
	X3	0

represents a list named X1 having the constituents X2 and X3. Structured data elements are available and are represented by the use of sub-lists, e.g.

NAME	SYMB	LINK
X1	0	
	S1	
	X2	0
S1	0	
	X3	0

in which the list X1 consists of a sub-list named S1, which contains the single constituent X3, and X2. Locations are set aside for data terms i.e. specific values, by making use of two more fields in the programmers representation - the 'P' and 'Q' fields of the list:

NAME	P	Q	SYMB	LINK	
NO	0	1		5	integer value 5
T2	2	1		XYZ	alphabetic value 'XYZ'

This is the only way in which literals can be introduced into a program.

2.

The manipulations which can be carried out on these lists fall into two groups. Firstly, primitive operations of a simple nature such as duplicating the first item in a list or removing the first item. The list to be operated on can be specified either directly or with levels of indirectness i.e. either taking the list to be the one named or taking the list to be a sub-list of the one named, or a sub-list of that. The remaining and much larger class of operations, known as J-processes, enable more complicated manipulations to be carried out. These include list processing operations such as the insertion of items at specified points of a list e.g. either before or after a particular symbol, or at the end of a list; deletion of symbols; replacement of items in a list; erasure of lists and list structures; and copying of lists. Arithmetic J-processes also exist for performing addition, subtraction etc. on constituents of lists.

3.

A program is written as a list in exactly the same form as those used for data. Each constituent of a list is a single instruction, specified by the P, Q and SYMB fields. The LINK field is used to indicate the next instruction to be executed. For control to pass on to the next instruction in the list, the field can be left blank and an implicit name of the next constituent inserted automatically by the system. A branch out of the normal sequence is indicated by the name of the instruction to be jumped to. Subroutines are represented by sub-lists, the name of which appears in the SYMB field as normal. Control branches to the sub-list named by the SYMB field when the P and Q fields are both zero, and returns to the instruction following when the end of the sublist is reached. The primitive instructions are represented by the contents of the P and Q fields, which are numerical and in the range 0 to 7, thus giving a theoretical 64 possible operations, operating on an operand given in the SYMB field. The J-processes are indicated in the same way as subroutines i.e. can be regarded as built-in subroutines. In addition to the data lists used by the program, the system provides a number of standard names, H0, H1, .. W0, W1, .. which are used for special purposes. For example, H5 is a list cell which can have either the value + or -, and is set by certain of the J-processes when conditions arise in relation to those processes. There is then a primitive instruction, that having the P field equal to 7, which branches control to the position indicated in the SYMB field instead of the normal continuation indicated in the LINK field taken when the value of H5 is -. Similarly, H0 is a cell which is used to communicate with the J-processes i.e. parameters are placed in this list before calling the J-process. The J-process also leaves its results there, for the program to examine.

4.

IPL-5 is a fairly low level language, sometimes called a pseudo-code, and thereby suffers from some inherent disadvantages. Notably, there is difficulty in being able to picture the method of solution of the problem without grovelling in the minutiae of the program. The subroutine structure can only be useful to a limited extent in this direction. The form of notation for the program, that of lists of instructions which are indicated, at least in the P and Q fields, numerically, is not particularly conducive to the ease of understanding the program, unless the programmer is very familiar with the language. For this reason, both program writing and debugging tend to be difficult, although tracing facilities are available to help with debugging.

5.

Arithmetic has already been mentioned as available through J-processes, but this can only be regarded as providing minimal facilities. There is no facility for evaluating expressions, as might be found in ALGOL. Input-output and backing store operations are also available via J-processes.

6.

IPL-5 invariably runs under the control of an interpreter rather than a compiling system and this has the effect of slowing the running speed considerably.

SLIP

A Symmetric LIst Processor system can be built into most existing high-level languages, as it consists of a set of subroutines which are called using the normal mechanism of that language. The original version was written to be embedded in FORTRAN, by J. Weizenbaum of M.I.T., and it has since been embedded in others such as MAD.

1.

The SLIP system defines a particular type of list and list structure. These lists are made up of constituents in a form in which one field of every constituent is a datum field. What this datum may consist of will depend on the language within which the system is embedded. Typically, it can be an integer or real number or a symbol or small group of symbols if the size of the field permits it. Sub-lists provide a structure for the data object.

The type of list used is one in which there is no preferred orientation (hence Symmetric), in other words, the location of both a constituent´s predecessor and it´s successor are stored with the datum (known as LNKL and LNKR). In addition, there is with each constituent an identification field (known as ID), which indicates the type of constituent:

LNKL	ID	LNKR
	DATUM	

Each list has a header cell (ID=2) which does not hold any datum but instead a count of the number of lists of which this list is a sub-list, and possibly a reference to a ´description list´ for this list. A sub-list is indicated by setting ID to 1 and the datum to a reference to the header cell of the sub-list. An ordinary datum cell has ID equal to 0, and the remaining ID type, 3, indicates a ´reader´ cell, which is used when scanning down a list and its sub-lists. The lists are circular in that the last cell is linked up to the header (and also the header linked to the last cell, for symmetry).

2.

A comprehensive set of routines are available for manipulating the SLIP lists. Lists are created either by copying an existing list or by calling a routine which sets up an empty list and then inserting items into it using other routines provided. For instance, items can be inserted at the top or bottom of a list, or to the right or left of an item in the list. To create list structures, the list intended to be a sub-list can be treated just as an ordinary datum and the name of the list inserted in the required position. An equivalent set of routines provide for removing items from a list. Substitution routines are also available as are facilities for examining and testing the contents of a list. Other routines test, for example, whether two lists are identical, whether a list is empty, or whether an item is a list or a datum. The testing facilities of the embedding language would be used for individual items after they had been extracted. Items of a list can be examined, either directly, if their exact position in the list is known, or by using a ´reader´ mechanism. A ´reader´ is a special cell which is used to traverse a list either to the left or to the right until a particular condition arises. This may be just to move one cell along, or to move to the next datum other than a sub-list, for example, and to retrieve the value held in the cell where it terminates. Two types of traverse are available, either straight along the list in sequence, passing over sub-lists without traversing them, or to traverse the list and all its sub-lists as they are encountered, known as 'linear' and 'structural' advances respectively.

3.

Program structure naturally follows that of the embedding language. It is the type of language such as FORTRAN or ALGOL for which SLIP was designed, depending as it does on routine calls and the ability to manipulate data arrays which provide the list storage space. In common with other languages, an Available Space List is used to maintain a stock of free cells to be called up whenever a new item in a list is created. The system is organised in such a way that the programmer is relieved of most of the burden of controlling his available space. When all references to a list are removed - this is the purpose of the count held in the header cell of each list - the list is automatically returned to the Available Space List. It is still the job, however, of the programmer to erase any list he has finished with, rather than to leave the reference in existence, if the program is liable to run out of space.

4.

SLIP is as easy to use as the language in which it is embedded and the extra list processing facilities are sufficiently straight forward as to be able to be grasped by a programmer already familiar with the basic language fairly quickly. Similarly, the effort of debugging a SLIP program can be eased by whatever facilities are available in the embedding language system. Although embedded in a high level language, the SLIP list manipulations tend to be of a low-level nature and there remains the danger that in constructing the list of precisely the right format by means of numerous routine calls, the overall structure of the solution may be lost in details of a non-significant nature.

5.

One of the very great benefits from embedding a system within an existing language framework is that all the features of that language remain and can be used when required. Thus, if the solution of a problem lends itself partly to the use of list processing techniques and partly to arithmetic techniques, then both these can be used without paying the heavy penalty that might be imposed when a language primarily intended for non-numerical work is used. A not insignificant gain from the technique of embedding is that the computing installation has to make no special provisions for incorporating yet another language into its operating system with the consequent gains to all concerned.

6.

This type of system is one where a compiler is normally used, and therefore the efficiency of running programs may be very good, depending on the degree of optimisation included in the compiler. The general housekeeping of list processing systems implies a certain reduction in speed from a purely numerical type of program, but this is unavoidable.

LISP 1.5

The LISP language was designed principally by J.McCarthy, at M.I.T., to be used for symbolic data processing in the field of artificial intelligence. It was first described in Comm. A.C.M. April 1960: 'Recursive Functions of Symbolic Expressions and their Computation by Machine'.

1.

All data takes the form of 'symbolic expressions' or 'S-expressions', which are defined either as an 'atomic symbol', consisting of up to 30 letters and digits and starting with a letter, or a sequence (S-expr . S-expr). In other words, the data can be regarded as a binary tree with atomic symbols at the terminal points. A simple list of elements is therefore one in which the first item in an S-expression is an atomic symbol and the other a similar S-expression, except for the last item. Such a simple list with the last element 'NIL', the null atomic symbol, for example :

```
( A . ( B . ( C . NIL ) ) )
```

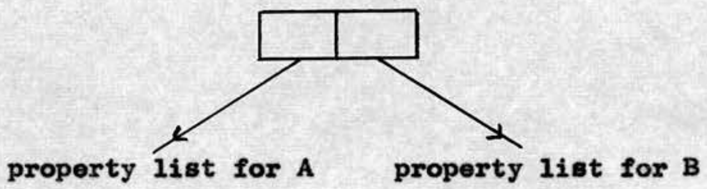
can also be written

```
( A B C )
```

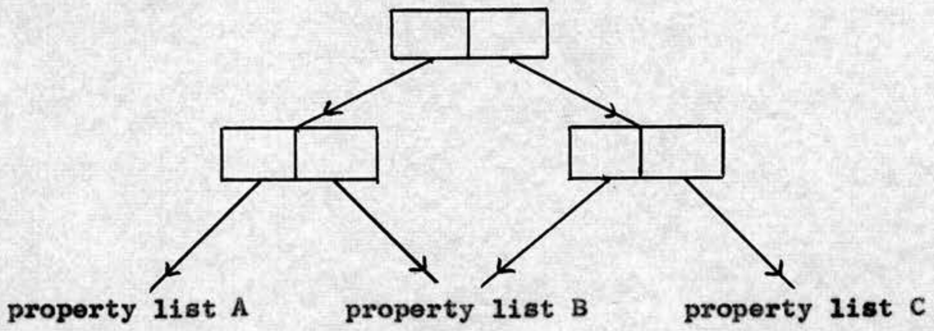
It is also possible to have numbers in fixed, floating point or octal form, in place of atomic symbols.

Internally, a compound S-expression (i.e. not an atomic symbol) is represented by a cell divided into two parts, the first part associated with the first component of the S-expression and the other with the remaining component. In both cases, if the component of the S-expression is an atomic symbol, then that part contains a reference to a property-list which is recognisable as such and holds information about the atomic symbol. If the component is a compound S-expression then it contains a reference to the cell representing that compound S-expression. E.g.

(A . B)



((A . B) . (B . C))



For numerical components the property-list contains a cell holding the value.

2.

Lists to represent an S-expression are created in a way suggested by the manner in which S-expressions are defined, that is, by making a copy of the list representing the first S-expression and also for the second, if they are not atomic, and placing the references either to the copy or to the property-list of the atomic symbol in a new cell. This is acheived by use of a function named ′cons′ (for CONStruct) which has as its two arguments the two components of the S-expression being formed. This is the only function which creates lists. Modification of lists is usually performed by making a new copy containing the desired changes, but an existing list can be modified using one of two other functions, ′rplaca′ and′ rplacd′, which replace the first and second reference components, respectively, of a list cell with a reference to a new list. These latter are, however, only recommended for use with caution as they can affect definitions and other basic information with possible undesirable consequences.

The first component of an S-expression is called the ′car′ part and the second the ′cdr′ part. Functions having these names enable lists to be examined, their values being the ′car′ and ′cdr′ of the list which is the argument. If the structure of the list is known, any component of it can be examined by a succession of ′car′ and ′cdr′s. Their value is undefined if that component does not exist, in other words, if the argument is an atomic symbol. E.g.

car[(A . B)] = A

car[car[(A . B)]]   is undefined.

The only form of testing of the structure of a list is by the use of two functions - ′atom′ and ′null′. ′atom′ has the value true if its argument is an S- expression consisting only of an atomic symbol and false otherwise. Since all lists are binary trees, this is sufficient to determine the whole structure, by repeated application of the function. ′null′ is true if its argument has the value NIL and false otherwise. There is a further predicate, ′eq′, which tests the equality of two atomic symbols and is undefined for non-atomic arguments. Similarly, the equality of two non-atomic lists can be tested by testing their atomic components, extracted using car, cdr and atom.

These few elementary functions form the basic tools for manipulations on lists in LISP. In general, when a non-elementary operation is required, the programmer combines existing functions to define a new one. Several functions of this character which are widely used are invariably built into any LISP system, such as ′list′ which creates a list containing all the arguments in the function call in sequence.

3.

A LISP program consists of a series of definitions of functions, followed by calls to evaluate functions with given sets of arguments. The emphasis is on an entirely functional approach to programming, unlike most languages which require a sequence of independent statements which are executed in this sequence until a transfer of control to some other statement takes place. In the LISP functional approach this is acheived by the heavy use of recursive techniques.

The method of Church′s lambda-calculus is used to define new functions. E.g.

f = λ[[x;y];cons[car[x];cdr[y]]]

which defines a function named f with arguments x and y, having the value cons[car[x];cdr[y]]. Predicate functions have already been mentioned. These can be used as arguments to the built-in function 'cond' which is the LISP equivalent of a conditional statement in other languages. Its use takes the form :

```
cond[[p1;e1];[p2;e2];.....]
```

i.e. a series of pairs, [predicate;expression]. Working from the left, the value of 'cond' is that of the expression paired with the first predicate having the value true.

Unfortunately, the notation used above is only a 'meta-language' and is not that used when presenting programs to be run. Instead, this form must be transliterated (if used at all) into a form which is a LISP list itself. As an example,

```
cons[car[x];cdr[y]]
```

would become

```
(CONS(CAR X)(CDR Y))
```

Where an atomic symbol is used as a literal, rather than a variable, say X, it has to be written (QUOTE X) to avoid ambiguity.

The reason for this notation lies in the fact that the interpreter that accepts LISP programs is also a LISP program (for the most part) and therefore can only act on data in the form of lists. This has the effect of producing a consistent structure in which the program may, if it wishes, modify itself and parts of the interpreter.

4.

Although producing a consistent structure, it also introduces practical difficulties such as the task of correctly controlling the proliferation of brackets, (and), in the program. Once syntactic difficulties have been ironed out, the question of ease of use reduces to whether the programmer finds it easy to think of his problem in functional and recursive terms. In many cases this is so and LISP will be a convenient language to use. Debugging in a recursive context is liable to be difficult and the heavily recursive methods which are necessary in LISP programs accentuate this. Error messages are provided, but the exact situation within the recursion is more difficult to locate. Trace facilities are available to help in this respect, but at the usual risk of not being sufficiently selective and producing large amounts of output. Individual functions can be traced in such a way that print-out occurs whenever the function is entered, giving its name and the values of its arguments.

5.

Arithmetic functions are provided but are very low level and inefficient. There are also facilities for using magnetic tape as backing store in some implementations.

6.

LISP is usually run as interpretive system and runs slowly as a consequence. Arithmetic in particular comes out very poorly. However, it is possible to compile particular functions which then run very much more efficiently, but at the cost of making them immutable. A program being interpreted is closely bound to the interpreter and a knowledge of how the interpreter works can have a profound effect on efficiency. It is normally preferable, however, from the programmers point of view, not to have to have an intimate knowlegdge of the system in order to be able to write efficient programs.

COMIT

The COMIT system was developed by the Mechanical Translation group and the Computation Center at M.I.T.

1.

A data element consists of an ordered set of 'constituents'. A 'constituent' can either be a 'symbol' alone or a 'symbol' with subscripts, where by a 'symbol' is meant a string of one or more characters, as convenient for the program. Since certain characters have special significance the character set is augmented by 'double characters', the first of which is an asterisk. When the symbol has subscripts these can consist of one numerical subscript and any number of 'logical' subscripts, which have the same form of name as a 'symbol'. Logical subscripts may, further, have one or more values associated with them and consisting of the same form of name again, but not numerical. Numerical values can only be represented as the subscript of a 'constituent'.

The three effective levels of structure - 'symbol', subscript names, and subscript values - are the only structuring of the data possible. Constituents are represented in the program with ´+´ as separator :

JOH + SEB + BACH

Subscripts are separated from the symbol by ´/´ :

BACH / .1685, OCCUPATION ORGANIST COMPOSER

´ORGANIST´ and ´COMPOSER´ are the values of the subscript ´OCCUPATION´. There is no ordering significance in subscripts and subscript values.

Internally, the constituents are represented by linked pairs of store locations the first of each pair containing some or all of the characters of the ´symbol´ and the second flags indicating the type of data in the first, for example, whether the characters start, are within, or terminate the ´symbol´, together with a link to the next pair of locations.

2.

The equivalent in COMIT of ´variables´ of other languages are ´shelves´. There are a fixed number of them, 127, identified by number rather than names chosen by the programmer himself. Manipulations do not take place upon the data while they are in a shelf, but in an unnamed ´workspace´, which can be filled from a shelf and emptied back onto a shelf. The shelves can be used as a pushdown store i.e. when data are transferred from the workspace, the previous contents of the shelf remain intact and can be accessed again when the more recently entered data are removed. An operation on the data in the workspace consists of matching a given pattern of data with some part of the workspace and then transforming it in some way. The pattern may consist of a constituent or a number of constituents in a given order. Suppose the workspace contains

JOH + SEB + BACH

then

* SEB = SEBASTION *

will find the constituent SEB and replace it with SEBASTION. Constituents matched on the left hand side before the ´=´ are identified by numbers 1, 2, 3, etc. for reference on the right hand side :

* JOH + SEB + BACH = 3 + 1 + 2 *

reorders the constituents into :

BACH + JOH + SEB

When the constituent is unknown or immaterial a dollar sign is used :

* $ + BACH = 1 *

deletes BACH from the workspace. If the number of constituents is known, however, then a number can be written after the $:

* $1 + BACH = 2 *

will delete the single constituent 'SEB'. Subscripts of a constituent can be inserted, deleted and 'merged'.

* BACH = 1 / .1685 *

inserts the numerical subscript value 1685. Similarly,

* BACH = 1 / OCCUPATION ORGANIST *

inserts the subscript 'OCCUPATION' with the value 'ORGANIST'.

* BACH = 1 / -OCCUPATION *

deletes that subscript. If the constituent already has a subscript with that name, then 'merging' of the values takes place. If there are no values in common with those already there, the new values are substituted, otherwise just those which are in common. For example, if the workspace contained :

BACH / FORENAMES JOHANN SEBASTIAN

then the rule

* BACH = 1 / FORENAMES JOHANN CHRISTIAN *

would result in the common subscript JOHANN being retained and the others discarded :

BACH / FORENAMES JOHANN

The rule :

* BACH = 1 / FORENAMES CARL PHILIPP EMANUEL *

has the effect of replacing the subscript values since there are none in common :

BACH / FORENAMES CARL PHILIPP EMANUEL

It is also possible to carry subscripts over from one constituent to another.

The pattern matched in the workspace can consist of subscript values. E.g.

* $1 / OCCUPATION COMPOSER =

which will only match a constituent having a subscript with that name and at least that value. When it is required to match any one from a number of patterns, instead of attempting to match each one in succession, a device known as a 'list-rule' is available which orders the patterns lexicographically so that matching can be performed more efficiently.

3.

A COMIT program consists of a sequence of 'rules', simple examples of which have been used above to illustate how the workspace can be manipulated. The same 'rule' may be repeated until the left hand side fails to find a match by replacing the surrounding asterisks with an arbitrary name. In general, the left hand name acts as a label for the rule and a name on the right acts as a jump instruction if the rule succeeds. If the pattern fails to match the workspace, the next rule is executed. Looping can also be controlled by using the numerical subscript of a constituent. E.g.

```
* BACH / .L15 = 1 / .I1 L
```

The rule finds a match if the numerical subscript of 'BACH' is less than 15. It then increments it by 1 and goes to rule L. If the name on the right hand side is $ then control passes to the rule having the subscript name of the first constituent as its name. Most other features of the language are included by means of more or less mnemonic code letters and numbers, following a double oblique slash // in the rule, called the 'routing section'. E.g.

```
* $ + BACH = 1 + 2 // *S6 1 *
```

means store the workspace before BACH in shelf 6.

Different letters are used for refilling the workspace and so on. The shelf number can be taken from a subscript. Input-output is acheived using the same mechanism with other letters.

A rule can have a number of 'sub-rules'. E.g.

```
THERE   .       .       .
WHERE   LEFT    .       .       THERE
        RIGHT   .       .       THERE
        UP      .       .       THERE
        DOWN    .       .       THERE
```

Only one of these sub-rules is executed, the choice depending on the setting of the 'dispatcher'. If unset, a random choice is made. It can be set by using the routing section : e.g.

```
*       .       .       // WHERE UP    *
```

will set the dispatcher to execute the sub-rule 'UP' when rule 'WHERE' is reached. In other words, this corresponds to the switch of ALGOL.

Subroutines can be defined, but there is a certain amount of difficulty in handling return addresses. It is up to the programmer to use a shelf as as pushdown store for this purpose and then to use the $ go-to. Essentially, there is no formal mechanism for subroutines and certainly not for parameters.

Storage allocation is handled automatically. An available space list is used to which spare cells are returned by the system whenever they become unused.

4.

The idea of pattern-matching with the contents of the workspace is a technique easily assimilated by the programmer and is very likely to prove a conceptually easier way of viewing his problem. Referencing the matched constituents by number is also an easy and quite convenient solution. Where the language tends to fall down is from that point onwards - in the structure of control. It is clear that there are sufficient facilities for most purposes, but at a lower level than might be expected from its quite high level pattern-matching capabilities.

There are numerous error comments both during compilation and dynamically to aid debugging. Trace facilities during execution can be obtained by slightly altering appropriate rules.

5.

Backing store facilities in the form of writing to and reading from magnetic tape can be used by having various routing section instructions. Arithmetic is restricted to manipulating numerical subscripts in rather inconvenient ways.

6.

The COMIT system uses a partial compilation and interpretation of the intermediate code produced. The speed would therefore be expected to be intermediate also - beteween full compilation such as SLIP and complete interpretation such as LISP. In practice, speed is lower than this in view of the process of pattern- matching, which can be slow if care is not taken in programming. Such a reduction of speed may well be acceptable if fewer debugging runs are required.

SNOBOL

SNOBOL is a string manipulation language implemented on the IBM 7090 and is the work of Farber, Griswold and Polonsky of the Bell Telephone Laboratories.

1.

The strings which SNOBOL uses consist of sequences of symbols. Any symbol letters and digits etc. can be used. Thus, a string might have the value 'THE FIRST RAY OF LIGHT'. There is no provision for structuring the string at the system level i.e. the lists used to represent a string cannot have sub-lists. However, the bracket symbols, (and), when items of a string, can have a special significance which imparts a structure to the string when certain manipulations are carried out. Numerical data can only be included by breaking down the number into some symbolic equivalent. For instance, the number twelve would be represented by the symbol '1' followed by the symbol '2'.

2.

The concept of named variables familiar in ALGOL-type languages is used in SNOBOL, except that the values of the variables are strings of symbols. The names themselves can be invented and used without special declaration and consist of a string of characters (letters, digits, periods and record marks) of arbitrary length. The literal form of a string consists of the characters enclosed in quotation marks e.g. 'LIGHT'. Assignment is the familiar type :

```
PP1 = 'THE FIRST'
```

which forms a string named PP1 containing the value 'THE FIRST'. The same effect is produced by :

```
PP1 = 'THE' 'FIRST'
```

which concatenates 'THE' and 'FIRST'. Concatenation is denoted by the space between the literals. Variables can be introduced similarly :

```
PP2 = PP1 'RAY OF LIGHT'
```

creates a string named PP2 containing 'THE FIRST RAY OF LIGHT'.

The three main operations considered essential by the creators of SNOBOL were the creation of strings, mentioned above; the examination of contents of strings; and the alteration of strings depending on their contents. The last two are acheived by a pattern-matching system not unlike that in COMIT. The name of the string to be scanned is followed by the pattern :

```
PP2 'RAY' *X* 'LIGHT'
```

If the literals 'RAY' are found followed later by the literals 'LIGHT', then a new string named X is formed containing what appeared between the two literals within PP2. The use of bracket symbols imparts structure to a string when a 'balanced variable' is indicated in the pattern. E.g.

```
EX = 'X*(Y+Z/(A-B))+Y'
EX 'X' *(EY)* 'Y'
```

The *(and)* around the name EY indicate that only a balanced string should be deemed to match i.e. one with a) balancing every (, and no) ocurring before its corresponding (. A further useful attribute is that it should be non-null. Thus EY will contain '*(Y+Z/(A-B))+' and not just '*('.

A fixed numbers of characters in a pattern can also be indicated. E.g.

```
PP2 'F' *Z/3* 'T'
```

would form a match only if there existed an 'F' separated from a 'T' by three characters somewhere in the string. String Z would then take the value of these three characters. The values of existing strings can also be used to indicate the pattern. E.g.

```
PP2 PP1 *Y* 'OF'
```

Similarly, when a partial match is found and a value assigned to a string, such as Y above, this string can be used to indicate the future pattern to be matched. E.g.

```
PP2 PP1 *Y* 'OF' *Z* Y
```

succeeds only if the characters found after an occurrence of the value of PP1 and before an 'OF' occur again later in the string.

Strings are modified by a replacement indicated after a pattern has been matched. E.g.

```
PP2 'RAY' *X* 'LIGHT' = 'LIGHT'
```

deletes 'RAY' and everything up to 'LIGHT' from PP2. Only that part of the string which was matched is replaced.

If the pattern has to match from the first character of the string, 'anchored' mode must be used. The mode can be set anchored or unanchored for all pattern matches, but the mode can be changed for just one match by writing the appropriate mode as the first element of the pattern.

3.

A program consists of a sequence of statements, each of which is a rule of the type indicated above optionally preceded by a label and followed by a 'go-to'. A 'go-to' takes the form of labels to which control is to be passed, either unconditionally or conditionally on the success or failure of the pattern-matching in the rule. E.g.

```
        .     .       / (PAPERS)
```

indicates an unconditional transfer to the statement labelled PAPERS, and

```
        .     .       / S(PICK) F(WICK)
```

for which control goes to PICK on success and to WICK on failure of the pattern match. An indirect form of control is available, e.g.

```
        LABEL = 'L' I            / ($LABEL)
```

Instead of using a label named $LABEL, the system takes the contents of LABEL to be the name. Thus control will pass to L1,L2,L3,etc. depending on the contents of I, '1', '2', '3', etc.

There is a subroutine facility included in the system which allows both string valued functions and predicates to be defined e.g.

```
        DEFINE( 'SIN(X)' 'L3' 'Y' )
```

defines a function named SIN with a parameter X, which starts at label L3 and has a local variable Y. The return from the function is handled automatically when the label RETURN is used. This can be used conditionally, when the function acts as a predicate, or unconditionally, as required. E.g.

```
        L3    X = ...
              Y = ...
        . . . . . . . . . . / (RETURN)
```

A number of functions are predefined into the system, such as certain input-output functions and predicates such as EQUALS and UNEQL which compare two strings. Storage and manipulation of free lists is handled automatically.

4.

The pattern-matching design of SNOBOL, as in COMIT, may prove a very useful tool in designing the solution for a problem and it is sufficiently high-level for convenient use. Debugging in such circumstances is likely to be easier than expected in comparison with the FORTRAN/ALGOL type of language. Trace facilities are provided by built-in functions.

5.

Arithmetic is available by use of strings containing the symbolic values of numbers. Thus if X contains ′12′ and Y ′-3′, then (X+Y) will have the value ′9′. Magnetic tape is available as a backing store.

6.

The relative slowness of execution of pattern-matching systems should be compared with the gain in programming and debugging time for a given problem. This applies to SNOBOL just as it does to COMIT.

4. THE ASTRA LANGUAGE

It has been noted in the previous chapter that the search for better and more convenient techniques of symbol manipulation has led to the development of a number of systems over recent years. Each has features which can be particularly useful in certain circumstances and which may also make another system more useful in other circumstances. The lines of development from early low level systems can be traced and the kind of facilities that are required can be discerned with more certainty. It seems reasonable at this stage, therefore, to contemplate an attempt to make further steps forward. Those features of other systems which have proved useful should, if possible, be retained in some form whilst also exploring other possibilities which may or may not prove to be so useful. It is doubtful if a completely new approach not making use of at least some existing techniques would be of value if the result is to be used for problems of a similar nature.

It is worth noting also the developments which have been made in general purpose languages, FORTRAN, ALGOL and lately PL/1. Only one system, SLIP, has taken advantage of these, the remainder being highly specialised systems set completely apart. The result of this is that SLIP is widely available - to any installation capable of running FORTRAN or similar languages, whilst the others have only been implemented on a small number of machines with the consequent lack of availability. Although SLIP is embedded in high level language systems, its symbol manipulation facilities are of a lower level, since it is primarily a system for handling lists, albeit in quite comprehensive ways. The use of subroutines does not confer the degree of expressiveness which might be desired, but has this considerable advantage of transferability of the system.

Another approach can, however, be made which, whilst not retaining quite the transferability of SLIP, allows as high a level of expressiveness as any of the other languages. This is the technique of using an existing language system as a basis and extending it in the required direction. It will be shown below that satisfactory extensions can be made for the specific task of symbol manipulation. As to transferability, it cannot be acheived simply by transferring a deck of subroutines, but it is likely to be significantly easier than creating a complete system of an equivalent order of comprehensiveness. Almost all installations are equiped with high level language systems such as FORTRAN, ALGOL, MAD etc. The ease of extensibility of a particular system will vary considerably, depending on the design of the language itself, but perhaps mostly on the methods of implementation used in the compilers for the languages and also the type of operating system in use - for instance, list processing techniques involving random access to memory may be frowned upon in a 'paged' environment. On the other hand, certain language systems, of which MAD is an example, have built-in facilities for extending the language in a number of ways. For installations using in-house produced compilers the required knowledge of the implementation techniques and advice will usually be easily available and extensions quite quickly incorporated. When depending on outside documentation, the task may not be quite so easy but still quite conceivable.

Although it has not been the case to date with symbol manipulation systems, it is possible to design systems with transferability in mind. For instance, a compiler written in the language it compiles can be transferred from machine to machine, within bounds such as core space and backing store facilities, by providing a machine code generation phase for the new machine and recompiling itself. This could have been done (and still can be) for the existing special purpose systems and to this extent there must also be other reasons for extending a basic language to provide symbol manipulation facilities other than transferability alone.

A much more important reason is that the technique allows all the features of the basic system to remain available and to be used where these are more suitable than symbol manipulation. The obvious example here is that of arithmetic. This may well have been secondary in the minds of some designers with particular problems in mind who therefore preferred to develop their systems without giving it a large proportion of their time and consideration. In LISP and SNOBOL, for example, the arithmetic is extremely slow and cumbersome. Clearly, the design objectives required nothing else; but should these have been the design objectives, thereby unduly constricting that proportion of the problem-solvers who find arithmetic necessary ? Some users will have pressed on, accepting the penalty, whilst others will have turned to systems similar to SLIP. Extending a language with already good facilities overcomes these difficulties completely. It should be regarded as a mistake to assume that problem-solvers will find all the tools they need in one line of development of languages. The development of symbol manipulation languages and FORTRAN / ALGOL-type languages should not therefore be separate as they have been in the past - each has much to offer the other. With the amount of effort being absorbed with the development of general purpose languages, this cannot be ignored by workers in the symbol manipulation field. By having an extension towards symbol manipulation in one direction, all other extensions will be added bonuses to the symbol manipulation workers, at no cost.

The ASTRA language is such an extension. It is based upon Atlas Autocode and has extensive string manipulation facilities as an addition. Atlas Autocode is a language developed by R.A.Brooker at Manchester University originally for the Atlas computer. In extending a language, its basic philosophy should be borne in mind. In other words, the extensions should be designed to fit in as far as possible so as to avoid conflict. This approach is not necessarily restrictive, as is demonstrated in ASTRA, and it results in a cohesion of the extended language which will be of great value. Before examining the extensions which constitute ASTRA it is necessary to consider Atlas Autocode for a moment.

It is similar to ALGOL 60 in many ways, notably in its block and fully recursive routine structure and most types of source statement. In a few respects it is simpler than ALGOL, without serious detriment to the language and on occasion with considerable gains in the efficiency possible without undue optimisation effort. A program consists of a sequence of ʹsource statementsʹ where a source statement is taken to be a sequence of characters terminating with a ʹseparatorʹ, which may be either a newline character or a semi-colon. (The newline can be overridden with a continuation marker for long statements). All spaces are ignored and therefore may be inserted to improve layout. This illustrates the point of avoiding conflict, syntactically in this case. The method of SNOBOL and other systems of using spaces as significant separators would clearly not fit in with Atlas Autocode well. Phrase structure notation is used as a convenient way of representing the syntax of Atlas Autocode and it is also used for ASTRA. Consider some of the alternatives from the class of source statements 'SS'.

1. [TYPE][NAME LIST]

[TYPE] stands for <u>integer</u> or <u>real</u>. For example :

```
                    integer i, j, k
                    real x, y
```

Declarations of variable names prior to use is obligatory as in ALGOL 60. Names may consist of a string of letters optionally followed by a string of digits and a string of primes and may contain any number of characters.

Arrays of scalar quantities are declared similarly. For example :

```
              integerarray A, B(1:10), C(0:2*n-1,1:2)
```

Arrays may be of any dimension and the bounds for each dimension may be any integer expression, i.e. expressions involving only integer-type operands, evaluated dynamically at run-time.

2. [NAME][APP] = [EXPR]

[APP] is the Actual Parameter Part, for instance array subscripts. The assignment statement may be exemplified by :

```
                    i = j+k/(1+2*m)
```

Expressions may be of any complexity involving the operators +, -, *, /, **. Only integer expressions may be assigned to integer variables (no rounding or truncation is defined in this context), but both integer and real expressions may be assigned to real variables. Integer and real operands may be mixed in real expressions; however, only an integer operand may follow the exponentiate operator **.

3. -> [N] and [N] :

Numerical labels are used for transfers of control, the jump instruction being the label number prefixed by the symbols '->' . For example :

```
                    . . . . .
                 3: . . . . .
                    . . . . .
                    -> 3
```

Switches are also incorporated. For example :

```
                switch sw(1:3)
                -> sw(i)
         sw(1): . . .
         sw(2): . . .
         sw(3): . . .
```

4. [iu][CONDITION] then [UI]

[iu] stands for if or unless, and [UI] may be an assignment, jump, or any of a number of other types of statement. The [CONDITION] clause allows general conditions to be specified. The basic Simple Condition or SC is defined as :

```
[EXPR][COMP][EXPR],
[EXPR][COMP][EXPR][COMP][EXPR],
([CONDITION]);
```

where [COMP] is any one of the comparators =, -=, >, >=, <, <=. The full Condition is built up from Simple Conditions :

```
[SC]
[SC] and [SC] and [SC] . . . .
[SC] or [SC] or [SC] . . . .
```

For example :

```
if x>y then x=y
if ( a=b and c-=d ) or e=1 then ->1
```

No precedence is defined between ands and ors, so that bracketing has to be used to avoid ambiguity.

5. cycle [NAME][APP]=[EXPR],[EXPR],[EXPR] and repeat

Loop statements are cycle etc. and repeat. The [NAME] must be an integer variable and the three expressions must be integer expressions which represent the initial, increment, and final values of the variable respectively. The values are evaluated once only, when the cycle is first entered, unlike ALGOL 60 where completely dynamic evaluation takes place. The group of statements to be looped around are closed by the repeat statement. cycles and repeats may be nested to any depth.

6. [RT][NAME][FPP]

[RT] stands for routine, [TYPE]fn, or [TYPE]map and [FPP] is the Formal Parameter Part, either a list of parameters or null. Block and routine (procedure) structure is similar to ALGOL 60 in terms of scope of identifiers but with differences in the types of parameters to routines. Value types remain the same but name types have a different effect. Instead of the full substitution demanded in ALGOL 60, there is a call by reference or 'simple' name in which the reference to the actual parameter involved is evaluated only once - on entry - and which remains fixed throughout the lifetime of that invokation of the routine. Whilst making 'Jensen's Device' impossible, this has extremely beneficial effects on efficiency, since 'thunks' are not necessary. For example :

```
routine CARL(integer i, j, realname x)
. . . . .
return
. . . . .
end
```

The dynamic exit from a routine is denoted by return. From functions (fn) and maps (map) this is denoted by result = :

```
realfn xyz(integerarrayname A, routine R)
. . . . .
result = x
end
```

A map in Atlas Autocode results in the calculation of an address, rather than a numerical value and is used mainly for storage compression. For example, a map would allow access to a symmetric matrix as if all the elements were present while only storing one triangular section of it :

```
realmap M(integer i, j)
if i>=j then result = addr(A(i*(i-1)/2+j))
result = addr(A(j*(j-1)/2+i))
end
```

return and result statements may also be made conditional.

Routines and functions may be recursive to any depth. For example :

```
integerfn fact(integer i)
if i>1 then result = i*fact(i-1)
result = 1
end
```

Atlas Autocode already has some facilities for symbol manipulation, making use of integer variables. A single symbol between quotes is a valid operand and it may be used in integer expressions since it takes the integer value of the numerical character code of that symbol. Input-output routines are available for reading and printing single symbols. These two features make possible worthwhile symbol manipulation processes and can be used with list processing facilities in terms of integer locations, as in SLIP, to provide reasonable basic capabilities. E.g.

I = ′*′

<u>if</u> J=′X′ <u>then</u> K=′Y′

We now discuss the ways in which Atlas Autocode has been extended to provide symbol manipulation facilities. The first decision was whether to extend the limited facilities which already existed, making use of integer-type variables, or to branch out in a completely new direction. Clearly, an integer location can hold more information than one single symbol; on Atlas and KDF9 six symbols can be stored in one of their 48-bit words. For longer symbol strings integer arrays can be used. While this may be useful on occasion, it confers no great advantage over storing single symbols, apart from that of space minimisation. The problems of defining operations on these data elements remain as before. Another approach is to use the integer location to contain a ′reference′ or ′pointer′ to a data area set aside to contain symbols. The essential difficulty inherent in both schemes is that of distinguishing between the different uses to which the integer location is put. On the one hand they contain ordinary numerical data and on the other they may contain symbols or pointers. Operations defined for one form will not necessarily be valid for the other. It can be left to the user to program carefully knowing that if the uses are confused chaos is liable to ensue with little diagnostic help available. The alternative is to pass around tags with the integer to indicate its current form of usage. This, though, would lead to unacceptable burdens on efficiency of ordinary integer arithmetic by having to test on every access. Where and how to store the tags is a further problem. The approach of using integer locations as pointers to a string value has been investigated by De Morgan and Rutovitz, who produced a modified Atlas Autocode compiler with string facilities at Manchester University. Their manipulations are by use of integer functions and routines with integer parameters. The keeping of distinctions between addresses and normal values is left to the programmer. This meant that only minimal changes had to be made to the compiler.

A more attractive approach, but one which necessarily means more modifications to the compiler, is that used in ASTRA - that of having a completely separate type of data object - string. This overcomes the ambiguities of using integer variables noted above and has a certain precedent in that there are already the types real and complex for data of non-integral form. This does not imply that different types of variables cannot be used in the same expression, for example, implicit type conversion can be done. It does, however, in Atlas Autocode, impart a useful degree of checking, in that a fault can be registered if a real value is assigned to an integer variable, say. The same applies to string variables. String operands can be mixed, in expressions, with other types and conversions defined where useful and checking on assignment can be made.

In common with the other forms of declaration, we have now proposed that of string. E.g.

```
string r,s,t
```

which declares the variables r, s, and t such as to take string values. Further justification for the incorporation of a new type can be found. Not only should variables containing specific data forms be available for any new type, but also the other situations in which a data type can be involved, such as functions producing results of the new form and the use of the type in parameters, should be possible, so as to retain the cohesion of the extended language and not to leave the impression of bits tacked on here and there. This is possible in the case of a string type. String-value-producing functions are quite consistent. E.g.

```
stringfnspec fn (integer i)
```

string parameters are also consistent with the existing Atlas Autocode language. Value-type parameters can be regarded as declarations at the level of the routine or function with their values preassigned by the value of the corresponding actual parameter. E.g.

```
routine AB (string S)
```

Name type parameters as defined in Atlas Autocode also create no difficulties, being a pure reference to a variable of the specified type, which is given as the actual parameter. E.g.

```
routine CD (stringname T)
```

Arrays of numerical values are an essential feature of Atlas Autocode and similar languages, whether the values are integer or real. Although string values cannot be represented in general by a single location, this does not render string arrays impossible. All it means is that they will have an extra 'invisible' dimension to hold the values of the strings in each position in the array :

<u>stringarray</u> ST (1:100)

Whether it is, in practice, an extra dimension of locations depends on the method of representing the string values - whether arrays of locations are used or list processing techniques. Virtually all symbol manipulation systems use list processing for the basic data representation, but arrays should not lightly be discarded. They have the great advantage that the components of the string can be accessed simply by incrementing a pointer, thereby facilitating their examination and such operations as copying and concatenation. One of their main disadvantages is that efficient store management is difficult. This makes itself felt in various ways. Firstly, how big is the array set aside to hold the string value to be ? If too little is set aside then when a value too long occurs, a new set of locations big enough must be found or some shuffling around of other string values must occur. Then, of course, the string value contracts and the space is wasted. Systems of this sort have been tried, with some success, such as the 'rolls' of the Digitek Corporation's 'POPS' system, but in a somewhat different context, where the number of variable data areas may be quite small, e.g. 20 to 30. A program using string manipulations is likely to have many more string variables or elements of string arrays than this, when their systems become less efficient. In this situation, the gain in efficiency over list processing is nullified.

An alternative is for the programmer to specify the maximum size of each string and if he exceeds his limits to wind up the program, indicating a fault. This use of arrays of a programmer specified size has been used in IMP, another extension of Atlas Autocode, in this case with the special purpose of providing the software implementation language for a large multi-access system and several compilers. For the kind of character string manipulations envisaged - that of handling input-output and providing alphanumeric titles of files in a file handling package, arrays were thought to be superior to lists, especially in view of a very useful set of machine instructions for the particular machine involved - the ICL 4-75 - those manipulating byte arrays in Store to Store operations. For these purposes, string variables are regarded as declarations of arrays of byte-long (8 bits) locations :

<u>string</u> U(10),V,W(20)

In essence, these IMP facilities form a subset of those to be found in ASTRA and therefore will not be discussed at length here.

Store management of arrays is also difficult in the situation when part of the value of a string is to be changed. For instance, if a group of components in the middle of the string are to be replaced by a different value, the new value may take more space. Finally, structuring of strings is difficult when using arrays. A degree similar to that of SNOBOL can be introduced by having the manipulations take account of the symbols in the string, e.g. brackets to indicate the extent of sub-strings. Scanning down the string cannot omit examining substrings, however, unlike a sub-list which can be passed over without inspecting its value.

A list processing system was chosen for ASTRA to avoid these limitations and is described in detail in the next chapter.

What should be the data form of this new string type ? The concept of an ordered set of symbols is quite general and can be regarded as sufficiently basic to symbol manipulation processes to be made use of formally. This is the conclusion reached by the authors of COMIT, SNOBOL, and LISP, although in the last case it is slightly disguised by their use of binary tree forms. A LISP list with each car branch atomic is of precisely this ordered set form. The primary data form for ASTRA was chosen to be of this kind - a sequence of characters e.g.

NEW1011

a*b+c*d

Slight variations are possible at this point - in essence, the answer to the question : what is the ´atom´ of the form (´atom´ in the LISP sense of the most basic component of the string)? Each character of a string may contribute equally in the manipulation of the string or it may be that groups of characters form a more natural basis, for example, complete words rather than letters. COMIT, in particular, was written by a group of workers in the field of natural language processing and for their needs a word or syllable could be regarded as indivisible. This led them to the system described above in which the atom is a group of characters, thereby enabling them to pack the characters more closely into word locations. They still, however, incorporated a facility for ´splitting the atom´ and recombining them in different groups when occasion demanded. This approach may be able to save space, but it was felt that for a general purpose system such as ASTRA is intended to be, a retention of the most basic system, that of treating single characters as the atoms, would be preferable. It would avoid the inelegancies of splitting and recombining atoms as in COMIT. The advantages of being able to group characters together are well recognised, but this need not be at the atomic level.

The facility for being able to group characters together or to impose a structure on the string has quite far-reaching effects in simplifying algorithms for many problems. In this respect, ASTRA goes much further than COMIT or SNOBOL. It makes use of the ´list structures´ in list processing i.e. lists with sub-lists also with sub-lists to any depth, where the sub-list is equated with the substring or group of characters that are associated. Thus substrings may have substrings and so on to any depth. This generality is much to be preferred and is akin to that of LISP. If we use brackets temporarily to delimit substrings, this allows us as possible ASTRA strings :

```
NEW ( ington ) 1011
( x+y ) * ( z/ ( u-v ) )
```

The desirability of incorporating numerical values in strings has been mentioned. ASTRA, as currently implemented, only has minimal facilities in this direction, but the language has been designed and implemented in such a way that it would be possible to incorporate more facilities in future versions of the system. This is discussed further below.

LITERALS

Atlas Autocode has the means of representing a single character by enclosing it in quotes, e.g. ´x´ . The quotes are necessary to avoid conflict with identifiers. For the format-effectors, space and newline, special actions have to be taken since spaces are discarded everywhere on input and a newline terminates a source statement. Instead, the symbols ´underline´ (_) and ´tilda´ (-) are used. This mechanism can clearly be generalised to include a number of characters between quotes, without at all departing from the spirit of the language. This is therefore the form of literal used to represent strings of characters in ASTRA. E.g.

´NEW1011´

´x+y*z´

This creates the difficulty of representing quotes as part of the literal characters. ASTRA solves this by representing a quote by two adjacent quotes and terminates the literal by a quote not followed by a further quote. There seems to be no convenient form of literal with which to represent strings containing substrings. It was felt undesirable to use any characters with a ´meta´-significance to delimit substrings since these characters are then effectively removed from the character set. Brackets, the obvious choice, are sufficiently often used in their own right as to make their removal a serious loss. The desired effect is in any case available using operations detailed below, and so the absence of substrings in literals is no material restriction. Indeed, the resultant form is probably clearer than any use of meta-characters in a literal would be.

STRING EXPRESSIONS

The method of assigning a value to a variable (apart from input from external sources) in Atlas Autocode is by a statement of the form :

[variable] = [expression]

The rules concerning what may constitute the expression for different types of variable are fairly strict. Unlike PL/1, say, where almost any type of value can be assigned to any type of variable with the necessary type conversions carried out implicitly, the only conversion involved with integer and real types is from the former to the latter. In particular, an expression which must have an integer value, such as when assigning to integer variables, must involve only integer types, either variables or constants, and not even produce intermediate non-integral values. Any expression involving real variables, constants or intermediate values, can only be assigned to a real variable.

With this precedent, it is sufficient to treat expressions which produce a string value i.e. string expressions, separately from other types. Certainly, no conversion from string to real or integer need be defined, except in restricted cases such as a single character string to an integer value, which is occasionally useful.

The simplest form of expression is one consisting of a single operand, which could either be a single variable, array element, or function, or a literal of the required type. E.g.

```
string r,s
r = ´NEW1011´
s = r
```

the result of which is to set the string variables r and s both to the value ´NEW1011´. For more complex expressions, operators are involved. The arithmetic operators +, -, *, etc. play no useful part in string expressions, although they could be defined as in SNOBOL, on strings of a restricted form i.e. only containing digit symbols. Where the normal arithmetic variables are available this seems superfluous. The only operator which has a useful part to play in the formation of string values is that of concatenation. The character chosen for this operator was the full stop, ´.´. It is unambiguous in this context and gives a neat appearance to string expressions. E.g.

```
string r,s,t
r = ´NEW´ . ´1011´
s = r . ´Ext.6298´
```

after which r takes the same value as before - ´NEW1011´ (purely as an illustration, since there is no advantage in splitting the literal in this way) and s the value ´NEW1011Ext.6298´. Note that the full stop in ´Ext.6298´ causes no conflict since it is part of a literal. In arithmetic expressions of Atlas Autocode, the number of operands is not restricted, and the same is true of string expressions. E.g.

```
t = ´Telephone´ . s . ´Edinburgh´ . t
```

This example illustrates a further point. The variable t appears on the right hand side as well as the left hand side and therefore, as in the arithmetic assignment, :

```
i = i + 1
```

the value used in the expression should be the value before any assignment has taken place. This need not so obviously be the case with strings as it is for integers, since the first part of the expression ´Telephone´ etc. could be assigned to t before the operand t was discovered, thus changing the value of t to be concatenated. This possibility arises because string values are not scalar. In ASTRA, however, it is treated in the desired fashion, comparable to the arithmetic case. Hence, in the example above, the effect is to prefix a string to the value already held in t.

We now consider substrings. With only one string operator - concatenation - there is no question of precedence between operators and therefore no need to bracket expressions to indicate the order of evaluation, as in arithmetic expressions, to circumvent the defined precedence rules. Brackets can therefore be used to surround those parts of a string expression which are to be substrings. E.g.

```
string r,s,t
r = ´NEW´ . ( ´ington´ ) . ´1011´
s = ( ´x+y´ ) . ´*´ . ( ´z/´ . ( ´u-v´ ) )
```

r would then take the value ´NEW´ followed by a substring with the value ´ington´ and terminated with ´1011´. Similarly, s will contain two substrings and a sub-sub-string.

Expressions also occur in other contexts, for example as the actual parameter corresponding to value-type parameters of routines and functions. This carries through to string expressions quite naturally : e.g.

```
routinespec RT (string s)
RT( ´VALUE´ )
```

Effectively this is a declaration of s when the routine is entered and an immediate assignment of the value of the actual parameter. The other main instance of expressions is in conditional statements when string values are compared. This usually takes the form of testing the value of a variable, e.g.

```
        if s = ´NEW´ then ->1
if s = ´NEW´.(´ington´).´1011´ then line = ´University´
```

The type of comparison required, string or arithmetic, is easily found by considering the types of the operands in the expressions being compared, with one very minor exception, which is :

```
if ´x´ = ( ´x´ ) then . . .
```

Although this would not sensibly be written, the operands could be taken as either of type integer, as in existing Atlas Autocode usage, or as type string. Regarding them as integer values, the test succeeds since bracketing does not affect the value, but as strings the bracketing causes (´x´) to be regarded as a substring and hence a different value from ´x´. There would be no ambiguity, of course, if any operator or variable appeared. This has been overcome by regarding single characters within quotes first and foremost as of type string - as seems most natural. The test, if ever written, would therefore be false in ASTRA. The forms of condition allowed in Atlas Autocode and kept in ASTRA are quite general. The ʻSimple Conditionʻ Phrase [SC] takes either the form :

[EXPR][COMP][EXPR]

or

[EXPR][COMP][EXPR][COMP][EXPR]

where [COMP] can be any of the comparators =, ¬=, >, >=, <, <=. For instance :

$a < x <= b$

is an example of the second form. These Simple Conditions can be grouped to form more general conditions : e.g.

([SC] <u>and</u> [SC]) <u>or</u> [SC] <u>or</u> ([SC] <u>and</u> [SC])

For string expressions the effect of the comparators = and ¬= is clear. The test of equality should include the values and the positions of all substrings. The effect for the others >, >=, <, <= is not so clear. When the string consists of purely alphabetic components, then ordinary dictionary ordering can be used i.e. ʻAʻ<ʻBʻ<ʻBCʻ etc. For other symbols, including digits, the ordering relationship implicit in the character set codes, can be used to generalise the dictionary ordering. Thus :

ʻA1ʻ < ʻB2ʻ < ʻB234ʻ

As any ordering of symbols such as +, -, ;, > is in any case rather arbitrary, there seems no drawback in using the ordering provided by the character codes. This incidentally has the effect, in the ISO code used in ASTRA, of putting digits before letters. The question of substrings is not solved by reference to codes. Clearly,

ʻAʻ . (ʻBCʻ) . ʻDʻ

should precede

ʻAʻ . (ʻCDʻ) . ʻDʻ

but should

ʻAʻ . (ʻBʻ) . ʻCʻ

precede

ʻABCʻ

or follow it ? The sensible choices to avoid confusion are that substrings should either rate higher than all symbols or lower than all symbols. ASTRA has arbitrarily chosen the latter. Thus :

(′A′) < ′A′ < ′A′.(′B′) < ′AB′

One further use of expressions is to assign the result of a function. For a string function, the result would be a string expression, e.g.

```
stringfn SF
. . . .
result = r . ′and′ . s
end
```

Only two string operands have been used so far - literals and names. Variations and extensions of these are possible and indeed essential. Firstly, there is the representation of a literal having a null value, used, for example, to initialise a string variable onto which values are to be concatenated. The symbol _ has been chosen for this purpose being clearer than ′′. E.g.

```
string s,t
s = _
s = s . t
if s = _ then stop
```

By having names as operands, the values of string variables can be examined and tested. But this is not adequate. Since strings can be regarded as vector quantities it is highly desirable to be able to examine just part of a string and not the whole of it. For instance, the first character of a string may control a process without depending on the remainder. One solution is to split the string into other strings and then examine these. This process of splitting constitutes a very important feature of ASTRA and is discussed in a moment. A simpler, though less complete solution, as will be seen, is to be able to index the components of the string in a similar fashion to array indexing. This does have the advantage, though, of making possible immediate examination rather than performing a preliminary splitting operation. Thus, the nth component of a string can be obtained by indexing with the value n. On a notational point here, the ordinary round brackets ought not, if possible, to be used to enclose the index to avoid confusion with arrays :

```
stringarray A(1:10)
. . . A(n)
```

Since A is an array, A(n) is still a complete string value.

```
string B
. . . B(n)
```

to pick off the nth component of B would be confusing. If the programmer had intended B to be an array and used it accordingly, as shown, it would not be faulted as it would if it were an integer or real variable; the program would be treated as valid incorrectly and this would presumably cause trouble. There are two obvious alternatives to solve the difficulty. One is to have a special function whose value was the required component. E.g.

```
item(n,B)
```

The other is to use a different form of bracket. Since square brackets are available and are not used anywhere else to cause ambiguities, they are the obvious choice. The use of square brackets to enclose the index is more convenient and appears more legible than the functional notation. This is the form adopted :

```
string B,C
if B[1] = 'A' then -> 1
C = C . B[n]
```

For strings without substrings, the index accesses the appropriate character, i.e. B[n] is the nth character of B. In common with ordinary array access, the indexes can be any integer expression and also in common with arrays, any attempt to index a non-existent element is invalid. I.e. if the value of the index is less than or equal to zero or greater than the number of components in the string, then a fault is monitored. When substrings are present in the string, the question arises of how many components, for the purpose of indexing, a substring consists of. The two most obvious possibilties are to count every character of each substring, sub-substrings etc. as a component or alternatively to count a substring possibly containing further substrings as a single component no matter how large. The second alternative was chosen for the following reasons. It is frequently the case that the structure of a string, that is, the number and position of substrings is either known or known to follow a definite pattern, whereas the actual contents or value of any particular substring, in particular its size, tends to be unknown. This being so, it is much more convenient, as examples will show, to be able to index over a substring by counting it as one unit than to have to find the size of substrings before being able to index. This is not to deny that to be able to index irrespective of structure might be useful in certain circumstances. However, it was felt unnecessary to incorporate both systems, thereby complicating the syntax to the programmer somewhat, as long as the means was available to examine the values of substrings in some other way. This is the ´splitting´ process described below. This reasoning also renders unnecessary the possibility of further depths of indexing such as :

s[3[2]]

i.e. the second component (character or substring) of the 3rd component (presumed to be a substring) of string s.

Having established the principle of indexing the components of a string it is possible to extend the notation not only to give a single component but also groups of components by specifying two indexes. E.g.

s[3:5]

which would have the value s[3].s[4].s[5] . The use of the colon in this way is quite consistent bearing in mind the form of array declarations, where the lower and upper bounds for each dimension are separated by a colon. Again, the indexes can be integer expressions and if any of the indexed components do not exist, as before, a monitor is caused. An extra frill which is sometimes useful is to be able to specify the value of the string from a certain index right to the end, without having to compute the length of the string. This is done by replacing the second index by an asterisk, which is syntactically unambiguous in that position. E.g.

s[4:*]

which is the string consisting of s without its first three components. It should be pointed out that consistency is maintained when substrings are indexed. Consider

s = ´NEW´ . (´ington´) . ´1011´

Then s[4] has the value (´ington´) i.e. still one component, rather than ´ington´ with six components. The brackets can be stripped off and the value examined by the ´splitting´ process.

It was decided not to incorporate a possible extension of the use of indexing as typified by :

s[3] = ´OLD´

s[4:6] = ´HAM´

The intention is to change only that part of the string which is referred to by the indexing, the remainder being unchanged. The effect of this type of statement can instead be achieved by the techniques of ´resolution´ and ´replacement´, which are described below.

RESOLUTION

The process of ´splitting´ or separating a string into components is called ´resolution´ after Brooker et al. in the Compiler Compiler.

The process of indexing over groups of components can be regarded as splitting the string. E.g.

```
s[1:4]    s[5:7]    s[8:*]
```

The string is only effectively split for the duration of the expression in which the operand specifying the part appears. If the division is conceptually of a more permanent nature, then to avoid recalculating the indexes and separating the group of components each time it is used, it ought to be assigned to a variable :

```
x = s[1:4]
y = s[5:*]
```

This can be condensed and incidentally made more efficient by the simplest form of resolution statement. The salient point is the index of the break point, here between components 4 and 5. In other words, the first four components of s are to be assigned to x and the remainder to y. This is written as :

```
s -> x[4] . y
```

In fact, the statements are not precisely equivalent, but the differences are discussed later.

This class of statement in which multiple assignments take place has no equivalent in Atlas Autocode. It was therefore felt unnecessary to place the names of the variables being assigned values on the left-hand-side of the statement as is conventional when a single variable is assigned to. It is consistent from the syntactic viewpoint, having a single name on the left hand side as in assignment statements. Semantically it is also preferable, since the process starts with the variable containing the string and then resolves it into its parts in a start to end scan, which is consistent with the order of writing the variables in the statement from left to right. The same reasoning could also apply to assignments of course. In order to distinguish the statement from an assignment something other than ´=´ must be used. The choice of ´->´ is somewhat arbitrary but has the convenience of being short, neat and easy to assimilate. The full stop is used, not to indicate concatenation, but to indicate separation in this case. Some operator is needed as is the case in string expressions and the full stop causes no ambiguity, so it is used again.

This use of indexes to resolve the string is available in a general form. Further examples are :

```
s -> x[4] . y[3] . z
```

which assigns the first four components of s to x, the next three to y and the remainder to z.

```
s -> x . y[4]
```

assigns all but the last four components of s to x and those to y. As in all situations where indexing is used, if a non-existent component is indexed, a fault ´failure to resolve´ is monitored. ´failure to resolve´ can also occur with a statement such as :

```
s -> x[3] . y[4]
```

if the number of components of s does not happen to be seven. It could be argued that if the string happened to be longer than seven, the remainder could be ignored. The view is taken in ASTRA that the terminal checking that nothing is left over may well be of use in debugging if a string takes on a value longer than expected. This corresponds to the ´anchored´ mode of SNOBOL. If the programmer is not concerned whether there is a remainder, he can always append a variable for the rest to be assigned to :

```
s -> x[3] . y[4] . z
```

There remains the slight argument that some time is bound to be wasted in making the final unwanted assignment. To overcome this, a dummy is introduced - the symbol - . This symbol can be used in place of any variable name in a resolution statement, when that group of characters which would otherwise be assigned to the variable does not need to be referred to again. E.g.

```
s -> x[3] . y[4] . -
```

There is no ambiguity with the symbol - used to denote a newline since that always appears between quotes. Another example might be :

```
s -> -[3] . y[4] . z
```

i.e. ignore the first three components of s and resolve the remainder into y and z.

The real importance of resolution is not the abbreviation of a number of statements to split a string into parts, as have been the only examples so far, but the ability to provide 'pattern-matching' facilities. We regard this pattern-matching, exemplified in COMIT and SNOBOL, as of prime importance in the language. The gain in clarity over systems without it such as LISP and indeed over the facilities so far described of ASTRA is considerable. It also has the effect, very often, of compressing the program, which may explain the clarity, by replacing a number of primitive comparisons with one comprehensive comparison. In the context of ASTRA, what we mean by the pattern includes the structure of the string, its substrings and sub-substrings etc., in addition to the symbols it contains. In other words, the presence of a substring constitutes a pattern just as much as the presence of a particular character or sequence of characters does.

To take first the case of matching a character. Suppose it is required to find out if the character is contained in the string and if so where, i.e. what comes before it and what after it. Consider :

```
e -> a . '*' . b
```

This resolves the string e into two parts. a assumes the value of the components occurring before '*' and b the value of those after it. A number of comments are needed. As with indexing, this form of resolution takes no account of the contents of substrings. It only attempts to match with an asterisk any symbols not in substrings. Substrings, even containing asterisks, would be passed over. (They can be found by a different form of resolution statement). For instance, if e had the value :

```
( 'x*y' ) . '*' . ( 'a+b' )
```

then a would assume the value ('x*y') and b the value ('a+b') . If e had contained no asterisk apart from those in substrings, the instruction would fail and cause the monitor 'failure to resolve' to occur. The pattern-matching, in this case scanning the string for an asterisk, takes place from left to right i.e. from beginning to end of the string. Clearly the process is ambiguous, since the string may contain a number of asterisks, unless some rule of this sort is introduced. A left to right scan seems the most natural rule. One could incorporate a number of rules for different methods of scanning but for the sake of simplicity, ASTRA keeps to one rule. The effect of the opposite order, that of scanning from right to left, can be acheived by reversing the order of the components of the string (a simple operation) and using the left to right scan.

It follows that if the first component of e was an asterisk, then a would assume a null value and similarly, if the only asterisk was the last component, then b would assume a null value. Alternatively, the resolution could specify these cases. Take the former :

e -> ´*´ . b

This resolution will only succeed if e starts with an asterisk. The remainder will be the value of b.

e -> a . ´*´

only succeeds if the asterisk in e is the last component.

The literal that specifies the pattern to be found in the string being resolved is of quite general form i.e. any number of characters between quotes. E.g.

r -> s . ´THE´ . t

Suppose r was ´THITHER THEY WENT´ , then s would become ´THI´ and t ´R THEY WENT´.

A number of literals may be included to specify the pattern more completely and to split the string into any number of parts, in the obvious generalisation. E.g.

r -> s . ´T´ . t . ´TH´ . u . ´THE´ . v

With the same value of r, s becomes null, t becomes ´HI´, u becomes ´ER´, and v becomes ´Y WENT´. Here again, as in all forms of resolution, any string variable which is to be assigned a value by the resolution can be replaced by the - sign, where the value is not required :

r -> - . ´T´ . t . ´TH´ . u . ´THE´ . -

As mentioned above, the pattern of a string includes substrings. To match the first substring in a string we can write :

y -> a . (b) . c

a then takes the value of all the components appearing before the first substring, b takes the value of the contents of the substring, including sub-substrings etc., and c the remainder of the components of y. For example, suppose y has the value

´NEW´ . (´ington´) . ´1011´

then a becomes ´NEW´, b becomes ´ington´, and c ´1011´. The same rules of ´anchoring´ apply when the pattern is formed of substrings :

y -> (b) . c

only succeeds if y starts with a substring. Similarly,

y -> (b)

only succeeds if y consists of just one component which is a substring.

This would have the effect of stripping the brackets from around y and assigning the value to b.

This form of pattern can be generalised, e.g.

```
y -> a . ( b ) . c . ( d . ( e ) . f ) . g
```

The resolution starts as before matching the first substring; it continues by matching a further substring, the contents of which must also include a substring. E.g.

```
'2*'.('u+v').'*'.('u-v').'*'.('w/'.('u*v'))
```

a then becomes '2*', b 'u+v', c '*'.('u-v').'*', d 'w/', e 'u*v', f and g null.

There remains a further requirement in the specification of a pattern; when the pattern is held in a string variable. In other words, instead of specifying a literal such as '+' or '-' in the resolution, we wish to specify a variable which may contain either '+' or '-' depending on the circumstances and attempt to match the current value. In order to distinguish this use of a variable from the form already used, it is surrounded by two pairs of quotes. E.g.

```
e -> a . ''s'' . b
```

Suppose e was 'x+y-z', then if s was '+', a would become 'x' and b 'y-z', or if s was '-', a would become 'x+y' and b 'z'. This form of pattern specification can go further than individual literals since the variable may contain substrings which would also be matched. The quotes could have been put around the variable being resolved into and omitted from this last type of pattern specification in order to become perhaps more consistent with string expressions, but the gain would be doubtful since variables taking resolved values are much more frequent and extra quotes scattered about would destroy some of the clarity.

The names inside the double quotes can be any which have a string value. In other words, they may be string array elements, string functions and string maps in addition to single string variables. Parts of the value can also be selected. E.g.

```
''A(1)''     ''A(1)[2:*]''     ''f(x)''
```

A point which arises when the value of a variable is used for pattern-matching, is illustrated by :

```
r -> s . '*' . t . ''s'' . u
```

Which value of s should be used as the pattern - the value before the statement was encountered or the value of the first components of r up to the first asterisk ? The latter course is useful in many circumstances and is perhaps the more general. It can, however, be something of a double-edged weapon; in at all complex resolutions it is not always obvious to the programmer how the resolution will proceed even knowing the value being resolved. This method also seems to conflict to a degree with the philosophy of the language being built on - Atlas Autocode - in that the precedents are to perform any evaluations once only and then to proceed using the fixed value. This is the case with cycle statements, where the initial, increment and final values are computed before entry to the loop and also with the name-type parameters where the address is calculated once only before entry to the body of the routine - unlike ALGOL 60. In view of this and for the sake of simplicity, ASTRA uses the former method. In the example therefore, the original value of s would be used for matching, but the value of s will (or may) be different after executing the statement.

A more general form of pattern specification has also been incorporated, where the pattern is to be the result of a string expression. Consider the expression :

```
´*´ . s . ´*´
```

Two approaches to resolution for this pattern are already available. Firstly :

```
r -> t . ´*´ . ´´s´´ . ´*´ . u
```

and secondly a preevaluation :

```
ss = ´*´ . s . ´*´
r -> t . ´´ss´´ . u
```

The second would be slightly more efficient, since in the resolution there is now effectively only one pattern to be matched, ss, instead of three, ´*´, s, and ´*´.

In order to avoid the extra preevaluation statement, an expression can be directly specified in a resolution by a statement of the form :

```
r -> t . ss[´*´ . s . ´*´] . u
```

The pattern to be scanned for is that contained within the square brackets and as such is a natural extension of the use of indexes within square brackets. In this context too, the value of the expression, in other words the pattern which was matched, is available after the resolution, as the value of ss, in the example. This ´naming´ of the matched pattern has important consequences in respect of ´replacement´ which is discussed below. In particular, the expression can consist of a single operand, for example :

```
r -> s . t['NEW'] . u
```

t would then have the value 'NEW' after the resolution, if successful.

If a dummy name had been used :

```
r -> s . -['NEW'] . u
```

this would have been exactly equivalent to the normal resolution :

```
r -> s . 'NEW' . u
```

Similarly :

```
r -> s . -[ss] . u
```

is exactly equivalent to :

```
r -> s . ''ss'' . u
```

The remaining form of resolution further broadens the patterns which can be matched. It is frequently necessary to be able to resolve a string scanning for not one string value but for any from a set of string values. For instance, in processing an arithmetic expression, it might be required to locate the first operator whether it is a plus, minus, multiply or divide sign. With the facilities so far described this sort of operation is rather inconvenient. The obvious solution was to extend the form of resolution just described so as to provide for a statement of the form :

```
r -> s . t['+','-','*','/'] . u
```

where the expressions between commas (only single literal operands in this case) are the alternative patterns to be scanned for. The naming of the string matched is also important here, since it then allows inspection of its value after the resolution in order to determine which pattern from the alternatives was matched.

This obvious solution was not adopted for the reason that the number of alternative patterns is fixed by the statement. It might well be the case that on some occasions only plus or minus are to be scanned for, or just the multiply and divide on others, in addition to scanning for all four. In order to overcome this difficulty and to be more flexible, the following method was chosen. All the alternative patterns to be scanned for form parts of the value of a single expression. When evaluated dynamically therefore, any number of alternatives can be included. The individual alternatives are delimited simply by forming them as the values of substrings. In other words, the format of a string to be used for this form of resolution is :

```
(pattern 1) . (pattern 2) . . . . (pattern n)
```

For situations in which no variability of numbers of alternatives is required, the expression would be of the form :

```
(´+´) . (´-´) . (´*´) . (´/´)
```

The task of distinguishing between expressions intended to be matched as a single unit and those of this special format used as alternatives is overcome, arbitrarily, by prefixing the multi-alternative form expression with the symbols -: . For example :

```
r -> s . t[-: (´+´) . (´-´) . (´*´) . (´/´) ] . u
```

Again, the string actually matched is the value of the specified variable, in this example t. Use of this form of resolution with varying numbers of alternatives could be illustrated by :

```
ops = _
if i=1 then ops = (´+´) . (´-´)
if j=1 then ops = ops . (´*´) . (´/´)
r -> s . t[-: ops ] . u
```

Although the various forms resolution can take have been described separately and the examples used have only shown the particular form under discussion, any of the forms may be combined in one statement. For instance, the indexing and contextual forms might be combined in order to ignore the first n components of a string and then to match the remainder with a literal :

```
r -> -[n] . s . ´*´ . t
```

Further examples might be :

```
r -> - . ( s . ´´t´´ . u ) . -
a -> b . ( c ) . d[3] . -
a -> - . ( - . ( - ) . - ) . - . ´*´ . b[1] . -
```

In the last example, the resolution proceeds by scanning past the first substring with a sub-substring and up to the next asterisk, taking the next component as the value of b.

Resolution as so far described has taken the form of imperative statements with a fault monitor and termination of the program occurring if the resolution is impossible. Clearly, there must be a way of testing whether a resolution is possible without causing termination if it is not. In ASTRA, this is easily accomplished by adding resolutions to the forms of ´Simple Condition´ which may be inserted in conditional statements. E.g.

```
if r -> a . ´*´ . b then ->1
```

Since Atlas Autocode allows quite complex conditions built from conjunctions and disjunctions of simple conditions, it follows that multiple resolutions or resolutions and arithmetic comparisons may appear in the same way. E.g.

```
        stop unless i=1 or r -> -.′*′.-
    if ( a->′*′.b or r->s.′*′) and j=k then return
```

When a resolution forms part of a condition, there arises the question of whether any resolution should take place when the resolution is possible or whether it should remain purely a test. There is no question in arithmetic conditions of there being anything more than a test. No assignment of new values to variables can be implied, apart from side-effects of functions evaluated in the course of comparison. This is not the case with a resolution condition, however, where the resolution and assignment of values could be implied. We could take the view that it should remain a pure test and make an imperative resolution later if required. The crucial point is that last. It turns out in practice that the resolution is virtually always required if it is possible. This choice is also much the more efficient (in the absence of optimisation) since duplication of pattern-matching is avoided. The final argument in favour of this view is that the programmer can always insert dummy names in his resolutions to avoid assigning new values to variables. ASTRA makes this choice.

When the resolution fails, the values of the variables on the right-hand-side remain unaffected. In multiple conditions, the testing is carried out from left to right as far as necessary to determine the value of the condition (not necessarily all of it, as in ALGOL 60). Any resolutions which succeed are carried out, even though a later part of the condition may fail. Perhaps it would be more desirable for resolution not to take place if the whole condition is false, but the difference is only marginal and this has not been incorporated as a matter of practical convenience in the implementation.

In the discussion of string expressions and resolutions up to this point nothing other than the values involved - the values of the string expressions or the values of variables after resolutions - has been mentioned. The fact that these values will have a physical representation introduces a number of alternative interpretations of these values, irrespective of the method of representation. The existence of possible variations can be made clear by considering even arithmetic scalar variables. E.g.

$$x = y + z$$

Apart from the familiar meaning, this could mean, for instance, 'wherever the value of x is required, take it to be the sum of the current values of y and z'. ALGOL 60 name type variables can be used in a very similar way to this. With string variables :

$$r = s . t$$

could be given the same sort of interpretation. However, we feel this would be departing from the principles of Atlas Autocode too far, since the equivalent arithmetic statement is not interpreted in this way. Other possible interpretations cannot arise with scalar variables. The representations of s and t could be physically linked by the same mechanism that links individual components of s and t to create the representation for the value of r. If we assume that we do not wish the values of s and t to be destroyed (a possibility however) then either copies must be made of the representations of s and t or r, s and t must use all or part of the linked representation as their values. This latter is discussed further in relation to resolutions. For string expressions, however, the straightforward uncomplicated approach of making copies of the representations of the operands and linking them together in the way required by the format of the expression is most attractive. The disadvantages of creating and holding duplicate copies, namely speed and space, are balanced by the simplicity, both in implementation and understanding by the programmer. The programmer's understanding will be greatly helped by the directly comparable approach to arithmetic expressions of Atlas Autocode with no further considerations to cloud the issue. He is also freed from the responsibility of coping with representations unnecessarily and can concentrate his attention on the values of his variables and expressions.

Since resolutions have no real equivalent with arithmetic operations, we feel more free to incorporate alternative interpretations such as have been described above, where there are advantages to do so. The potential advantages of using a single representation for a number of variables are those of speed and space. When a variable has the same value as another variable, space is minimised if they both use the same representation. The case in resolution is that variables on the right-hand-side take values which are parts of the value of the variable being reolved. Those parts of the representation could therefore be used to represent the values of the right-hand-side variables. The alternative is to copy the relevant parts of the representation and assign those to represent the values of the right-hand-side variables, thereby incurring the speed penalty of copying. The speed penalty may not in practice be quite so severe as indicated, since using one representation for a number of variables will involve extra complications to maintain the identity of variables' values. To reiterate these two alternatives of resolution, consider a variable r with the value 'VOLTS*AMPS', then

r -> s . '*' . t

will give s the value 'VOLTS' and t the value 'AMPS'. In terms of representations, starting with r :

$$\underbrace{\boxed{\text{VOLTS*AMPS}}}_{r}$$

we could either make further copies for s and t :

$$\underbrace{\boxed{\text{VOLTS*AMPS}}}_{r}$$

$$\underbrace{\boxed{\text{VOLTS}}}_{s} \qquad \qquad \underbrace{\boxed{\text{AMPS}}}_{t}$$

or use the original representation for all :

$$\boxed{\text{VOLTS*AMPS}}$$
$$\underbrace{\underbrace{\phantom{\text{VOLTS}}}_{s}\underbrace{\phantom{\text{*AMPS}}}_{t}}_{r}$$

together with a means of distinguishing the individual values of s and t.

We now recall the requirements of string processing in general. An important ability to have is to be able to modify a string. We can already do this with the facilities of ASTRA already described, namely creating the new value by means of evaluationg a string expression. The more components the value has however the less efficient this method of modification is liable to become because of the copying implied. The greatest advantage of using the same representation for a number of variables now becomes apparent - it provides a facility in high level language terms of modifying values. In effect, it makes use of the fact that if one representation is used, variables have a 'position' in the representation as well as a value. Hence, the name of such a variable can be used to change the value of that part of the representation, without affecting the remainder or causing the remainder to be copied. E.g.

```
r -> - . ( s ) . -
```

Using one representation, s now effectively 'names' the contents of the first substring of r. Some form of assignment to s could then be taken to signify a change in the value of that substring within r. A statement such as :

```
s = 'WATTS'
```

would normally be understood to create a representation of 'WATTS' and assign it as the value of s, without thereby changing the value of other variables. It is still necessary to have this facility to create 'clean' values, free of interaction with other variables. Therefore, another form of assignment is required. A different assignment operator is sufficient to make the distinction clear and to avoid inadvertent misuse.

Unfortunately, the ability to modify part of a string in this way brings with it other complications. For example, we can carry out the resolution :

```
r -> s[4] . t[4]
```

followed by a further resolution of the same string :

```
r -> -[3] . u[2] . -[3]
```

Using the same representation for r, s, t and u, the result will be that u overlaps s and t. If now we replace the part of r named by u, something must happen to s and t. We do not wish to restrict the value forming the replacement. Restricting the replacement to the same number of components as the replaced value, it would be possible to modify the values of overlapped variables so that they kept the same number of components. Such a restriction is extremely inconvenient in many cases and should be avoided if at all possible. Some other method of circumventing the difficulty has therefore to be found. A different form of restriction could be not to allow as valid resolutions those which produce overlapping variables or to allow replacement of a variable which overlaps others. This again is not really satisfactory being inconvenient for the programmer. The best compromise seems to be to allow the replacement with unrestricted values, but to clear any value from overlapped variables and leave them unspecified in value and position. This appears to be the only difficulty with this form of resolution. There are no problems, for instance, in the use of the variables having resolved values at later stages. They can be further resolved as and when required and the parts thus delimited replaced.

This system of using one representation for a number of variables after a resolution was chosen for ASTRA. The methods of representation and means of overcoming the implicit problems are described in the following chapter. No modifications in the foregoing description of resolution are required and the extra assignment operator for replacement was chosen to be ´<-´ . To illustrate its use; with r having the value :

```
´x*´ . ( ´y+y+y´ )
```

after :

```
r -> - . ( s ) . -
```

s will have the value :

```
´y+y+y´
```

and a position as the substring of r .

```
s <- ´3*y´
```

will then produce a value of :

```
´x*´ . ( ´3*y´ )
```

for r. s, incidentally, retains the same position and also assumes the new value. After :

```
s = ´SOMETHING ELSE´
```

r would still retain the same value, but would no longer share its representation with s, which would have a ´clean´ representation no longer associated with r.

Using the replacement operation, assignments of the form :

```
s[3] = ´OLD´
s[4:6] = ´HAM´
```

can be achieved by :

```
s -> -[2] . t[1] . -
t <- ´OLD´
s -> -[3] . t[3] . -
t <- ´HAM´
```

The resolution :

```
r -> s . ´NEW´ . u
```

does not allow the ´NEW´ part of r to be replaced since there is no reference by a string variable to it. Using the resolution :

```
r -> s . t[´NEW´] . u
```

however, ´NEW´ may now be replaced :

```
t <- ´HAVEN´
```

Similarly in resolutions with multi-alternatives :

```
r -> -.[-:(´+´).(´-´).(´*´).(´/´)].-
s <- ´.OP.´
```

There is one class of situations where a restriction of full generality was felt to be worthwhile. This is those resolutions which use the same name on both left-hand and right-hand sides. For example :

```
r -> s . ´*´ . r
```

If r had originally been formed as a result of a resolution, say :

```
t -> - . ( r ) . -
```

no problems will arise. If, however, r had originally been formed by a normal assignment, say :

```
r = ´a*b´
```

there will be certain consequences. After a resolution, the names on the right-hand side refer to parts of the original string which is the value of the string which is resolved. When the same name is used on both right- and left-hand sides, the original string will not be the value of any string variable and parts of it, the ´*´ after :

```
r -> s . ´*´ . r
```

for example, will be in ´limbo´. A solution would be to treat this as a normal assignment to s, splitting the representation in two and returning the ´*´ cell to the Free list. This was felt to be sufficiently inconsistent as to be worthwhile ´outlawing´ this type of resolution.

Most inconveniences this restriction might cause are eliminated by making a preliminary resolution :

```
r -> t
t -> s . ´*´ . t
```

after which there are no problems since the string r still exists.

The check on whether the name on the left-hand side is the same as that of any of the names on the right-hand side must be performed dynamically at run-time, as evidenced by the following examples :

```
A(i) -> s . ′*′ . A(j)
```

where A is a string array and i and j may or may not have the same values at run-time. Similarly :

```
. . . . .
RT(r)
. . . . .
routine RT(stringname s)
. . . . .
r -> -.′*′.s
. . . . .
end
```

To assist ASTRA programmers, there are several routines and functions built-in to the system which may be freely used in programs without the need for declaration or specification. These are :

<u>routinespec</u> read item (<u>stringname</u> s)

′Take the next symbol from the input data stream and assign it as the value of s′.

<u>stringfnspec</u> next item

′Inspect the next symbol in the input data stream and assign it as the value of the function without removing it from the stream′.

<u>routinespec</u> skip item

′Pass over the next symbol in the input data stream′.

<u>routinespec</u> read string (<u>stringname</u> s)

′Pass over symbols in the input data stream up to the first left bracket and take the following symbols up to the corresponding right bracket as the value to be assigned to s, regarding any inner brackets as denoting substrings′.

<u>routinespec</u> print string (<u>string</u> s)

′Print out the value of s, with brackets to denote substrings′.

<u>stringfnspec</u> length (<u>string</u> s)

′Assign as the function value the number of components of s, counting substrings as 1′.

<u>stringfnspec</u> itos (<u>integer</u> i)

′Create a string with one component having the integer value i′.

integerfnspec stoi (string s)

'Supply the integer value of the one-component string s'.

The functions itos and stoi provide the only method of inserting and extracting numerical information from a string. This is because there is no implicit type conversion defined in integer and string expressions. It could, however, be incorporated. For example, numerical values could be incorporated in string expressions :

```
integer i,j,k
string r,s,t
r = s . i . t
```

when a single component having the integer value of i would be concatenated with s and t. Similarly :

```
i = j + r + k
```

when the numerical value of string r could be defined either as the value of the single component of string r or as a conversion of a string of digit symbols.

A meaning could also be defined for mixed-type resolutions. For example :

```
r -> i . r
```

which would be equivalent to :

```
r -> s[1] . r
i = stoi(s)
```

There is no reason why real quantities should not be treated in the same way as integer quantities in these respects.

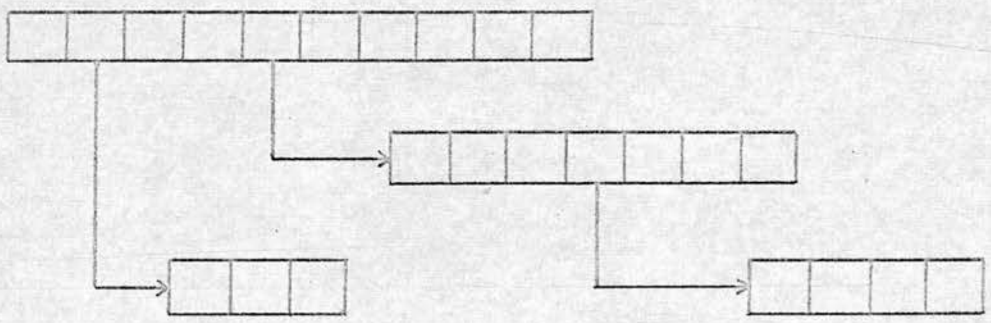
Routines providing a magnetic tape backing-store facility are also available.

The foregoing pages describe the ASTRA language as implemented for the KDF9. Examples of programs appear in Appendix A.

5. INTERNAL REPRESENTATION OF ASTRA STRINGS

It was decided to base the design of the ASTRA string facilities on the use of list processing techniques to represent string values. The justification for this as a general approach was described in the previous chapter. At this point, having described the string facilities of ASTRA, it is possible to detail the particular features for which a list processing scheme is most natural. The arguments which led to the particular form of list chosen to represent ASTRA strings are then discussed followed by a description of the manipulations of the lists which implement the ASTRA facilities.

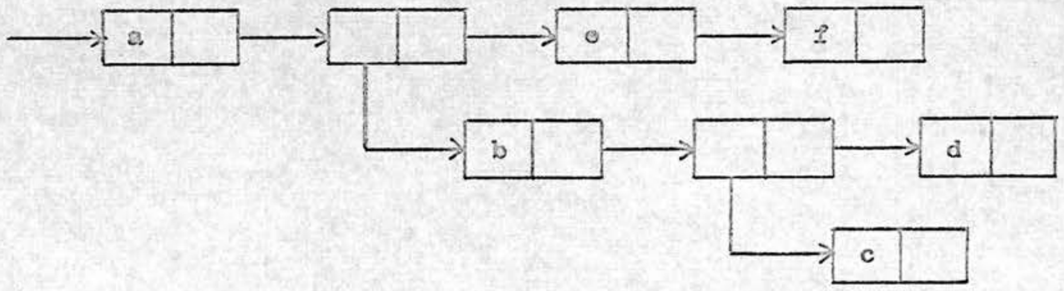
The structuring, which provides one of the most valuable features of ASTRA strings, is naturally associated with structuring in a list. In other words, substrings correspond to sublists, sub-substrings with sub-sublists and so on. This correspondance automatically brings with it the further requirement of ASTRA that substrings should count as one component when indexing down a string. The indexing process itself is undoubtedly more suited to array representations, but is complicated when substrings are present and counted as one component in length. If arrays were used, this useful feature would probably have to be foregone in favour of taking the total number of components of the substring and all its substrings as the value of the count. A halfway stage between lists and arrays could be used, using separate arrays for substrings instead of the same array, in order to retain the indexing convenience :



The problems of efficient store management become quite severe, however, and the ability to replace parts of a string remains difficult. This latter is another situation where list processing is clearly superior, since replacements only involve changing links to lists. Since indexing is relatively unimportant in ASTRA when compared with resolution, the less convenient scanning necessary with lists is not considered too severe a disadvantage. It was also felt that list processing provides a more flexible approach bearing possible developments of the language in mind.

The ASTRA programmer is provided with high level facilities to manipulate his strings. The precise form of the representation of these strings is therefore of no concern to him. (This statement might have to be modified slightly for a programmer wishing to optimise the efficiency of his programs, when he would have to know the relative efficiency of various operations, but in principle no knowledge of the form of representation is necessary) . There was, therefore, complete freedom in the choice of the type of list to be used for the representation, the only consideration being the requirements of the language, ease of manipulation from the point of view of the implementor and tolerable efficiency in use. The last two considerations conflict to a certain extent in that it is very desirable to implement as much as possible in high level language terms for the same reasons of ease and convenience that apply in the general use of high level languages. The ASTRA compiler calls heavily on the philosophy and is, in fact, written in ASTRA itself. In order to attain a reasonable degree of efficiency in list processing operations, however, a lower level view, which allows the use of facilities peculiar to a particular machine, will almost always give radical improvements over high level methods. Before the choice can be made, the likely degree of improvement must be ascertained. This consideration is therefore postponed until the requirements of ASTRA string representations have been examined.

The simplest form of list, that of pairs of cells, one of which contains a link to the next pair of cells in the list (or zero for the last pair of the list), the other containing either the piece of information or a link to a sublist, can be considered initially. Such a form of list can be shown diagramatically in the following way :



This could represent the ASTRA string :

´a´ . (´b´ . (´c´) . ´d´) . ´ef´

The tree structure of this form of list is quite consistent with the requirements for ASTRA strings. There is no need, for example, for common sublists of the SLIP type. The other important feature of SLIP, that of the lists being symmetric, in other words having links forwards and backwards, is also not required since all scanning of strings is defined to be from left to right in ASTRA. This is the case both in indexing and in resolution of strings and it is never necessary to proceed from right to left. The occasions when, even so, it might be more efficient to scan from right to left, such as :

```
r -> s . ´*´
r -> - . s[2]
```

occur sufficiently infrequently as to imply that the extra cost of a symmetric system would be uneconomic. Note that resolutions such as :

```
r -> - . ´*´ . s . ´*´
```

must still take place from left to right in order to locate the first ´*´ in r, in spite of having to match the final component.

A list of the form shown above can be identified solely by the location of the first cell. It would therefore be possible to store this address in precisely the same way that values of scalar variables are stored, using just one location on the run- time storage stack. Apart from the provision of an area of free cells, this would enable string variables to be treated by the compiler just as integer and real types as far as addressing is concerned. This capability is quite valuable in that the additions and changes to the compiler are on a relatively small scale and do not necessitate its redesign in any major way.

The factor which renders this type of representation insufficient is the way in which resolution is defined. After a resolution, a number of variables will share the representation of the variable which has been resolved, using parts of it to represent their values. Suppose string r has the value :

´a´.(´b´.(´c´).´d´).´ef´

After :

```
r -> s[2] . t
```

it is clear that the location of the first cell is insufficient to determine the value of s, although it would be sufficient for t. The final cell must also be defined in some way, so that the value of s is not taken as that of the whole of r, but only the first two cells and any sublists of those cells. The two obvious solutions are either to hold the locations of both first and last cells of the representation or to hold the location of the first cell and the number of components which represent the value. In the latter case it is only necessary to count the components at the highest level of the string, in other words, with substrings counting as one component, since the values resulting from a resolution always include all the value of any substrings. A value such as a number of constituents at one level and part but not all of the constituents of a substring cannot be acheived using ASTRA operations. Neither of these two methods conflict with the ability to store the locating information for the representation of the value in the same fashion as integers and reals. Either the two pieces of information can be packed into one location or a constant two locations can be used.

The first method suggested was, in fact, chosen for reasons connected with the effects of replacing parts of strings using the '<-' operator. It was argued in the previous chapter that the best way to overcome the difficulty of defining the values of variables, parts of whose representations had been replaced - the situation when there was an overlap such as that produced by :

```
r -> s[3] . t
r -> -[2] . u[2] . -
u <- 'NEW'
```

- was to clear the value from the affected variables, which would be s and t here. This implies that it must be possible to determine which variables are using the part of the list which is being replaced as part of the representation of their values. A number of solutions are available all tending to reduce the efficiency of processing, but some much less than others. ASTRA has chosen what is believed to be the most efficient of these.

The first possible method is to maintain a record (which will change dynamically as processing proceeds) of the cells of the representation of all string variables currently assigned a value. The first and last cells are sufficient. Whenever part of a representation is discarded, which will occur when it is being replaced, the cells can be inspected. If any such cells correspond with either the first or last cells of a string's representation, found by scanning the record of first and last cells, then that string's value is being interfered with and, as we suggest, any value can be cleared from that string. We take the view that where total overlap occurs, such as :

```
r -> s[5] . -
r -> -[1] . t[3] . -
t <- 'REP'
```

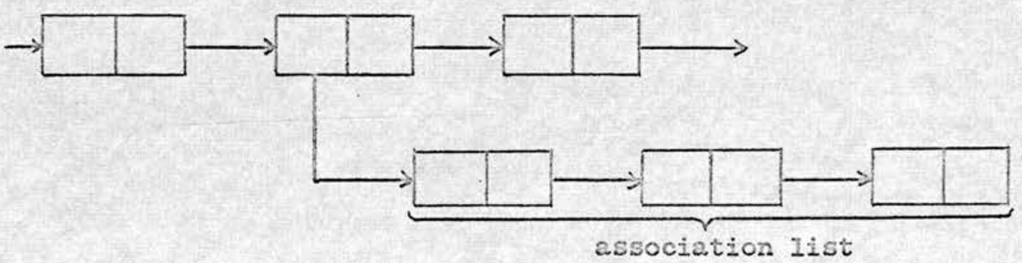
where s totally overlaps t, the value of s in this case should be modified as required by the replacement, rather than cleared of any value. In the reverse situation :

```
s <- 'LACE'
```

t would lose its value, since this is no more than a compounding of the two other situations both front and end overlap.

This method can clearly be improved by subdividing the record of first and last cells into groups associated with a single representation. This also requires, however, that the correct group must be identifiable when replacement is taking place from the variable involved. The scanning for matching cells is thereby much reduced.

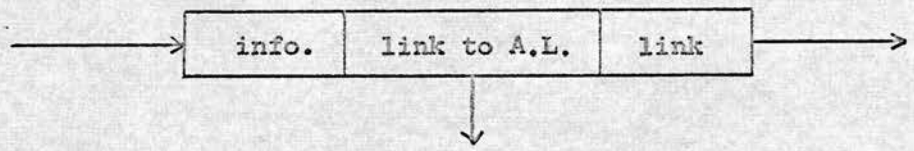
A more elegant solution to the problem than either of these two methods will now be described, but this is dependent for its efficiency on the way in which each cell of a representation can be constructed. The object is to remove the scanning implicit in the other methods. This implies that each cell being discarded should itself be able to supply the information concerning its multiple use. Again, only the cells which either start or terminate a variables representation need supply the information. The information needs simply to be a record of the identity of the variables which either start or terminate at that cell. This is best acheived by having what may be termed an 'association list' attached to the cell :



This association list must be made independent of other cells in the representation. In this system, there are now three distinct pieces of information associated with one cell; the information symbol; the link to the next cell in the representation; and the link to the association list, if any. The question postponed from above now becomes relevant - that of using facilities of the basic design of a particular machine not available within the data structure types of a high level language. If three pieces of information are associated to form a cell of the representation, it becomes more compelling to minimise the space occupied by a representation by means of packing the information as closely as possible in store.

The machine upon which ASTRA was to be implemented was the Engish Electric KDF9. In this case, the basic 48-bit word is split into three 16-bit fields for many basic machine operations including addressing and indexing. In particular, list processing using such three field cells is very convenient and can be very efficient when the basic operations of KDF9 are used; more efficient in fact than if separate words were used for information and links, in terms of speed as well as space. The strength of these considerations left hardly any other practical course open for a KDF9 implementation.

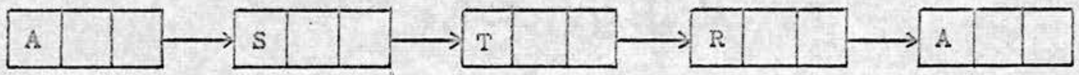
The last suggested method of overcoming the overlap problem is therefore very feasible, taking a 16-bit field for each of the three pieces of information :

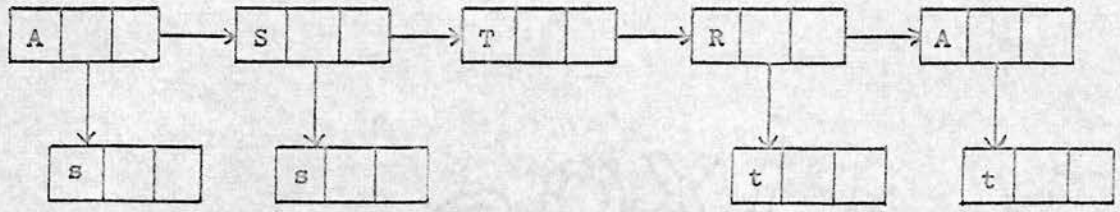


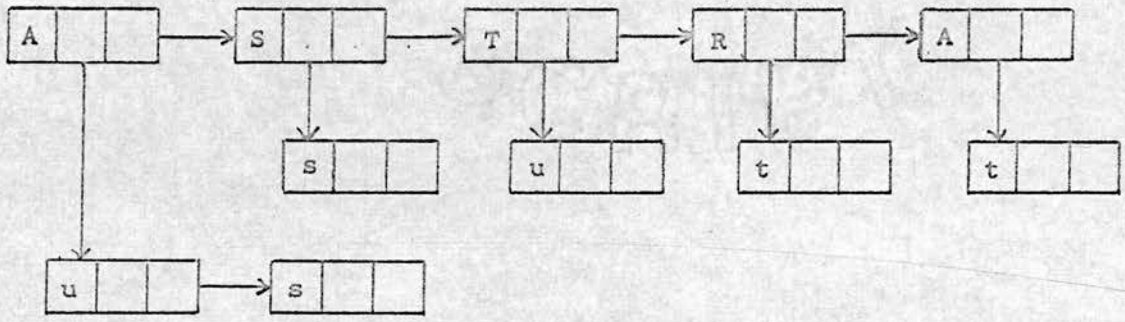
The kind of situation envisaged can be illustrated with an example, for further clarification :

```
r = 'ASTRA'
r -> s . 'T' . t
r -> u[3] . -
```

The successive modifications to r's representation would be :







This system of lists with three-field cells was chosen as the basis of the lists used in ASTRA.

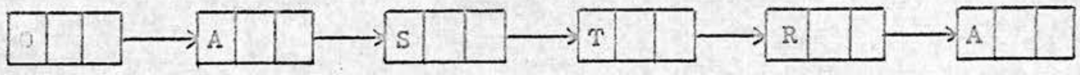
Some refinements and additions are necessary to cope with unusual situations and for programming convenience. A case such as :

```
r = 'ASTRA'
r -> s . 'A' . -
```

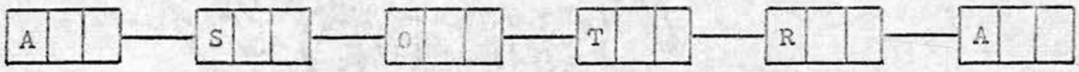
is slightly awkward. s will have a null value, but still a position before the first constituent of r. If s were to be replaced with a value, this value should prefix that of r. There are, however, no cells before that containing the first 'A' to give a 'position' for s. The same also applies with :

```
r = 'ASTRA'
r -> 'AS' . s . 'TRA'
```

The method of solution adopted introduces the concept of a dummy cell. That is, one linked in with the list in the normal way, but containing no information - a null cell. In the first example above, the resolution would cause such a cell to be prefixed to the representation :

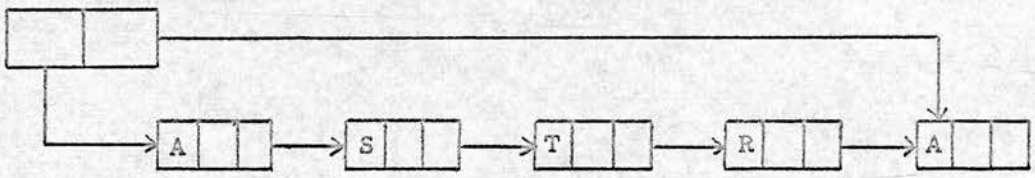


s can now be given a postion ahead of the symbols 'ASTRA' in r. In the second example, a dummy cell would be inserted :

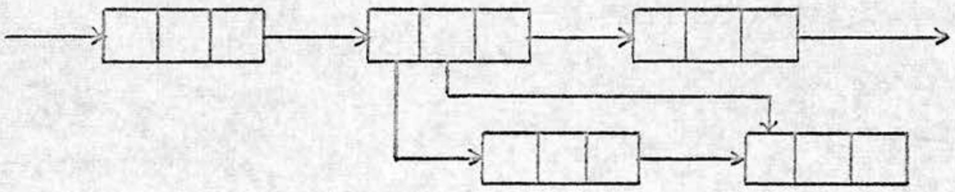


All operations on this list, such as indexing and scanning for a pattern, must allow for the presence of dummy cells and pass over them when encountered.

The use of a location for a string variable on the normal run-time stack was discussed above and it was decided to store the first and last cells of its representation, packed together :



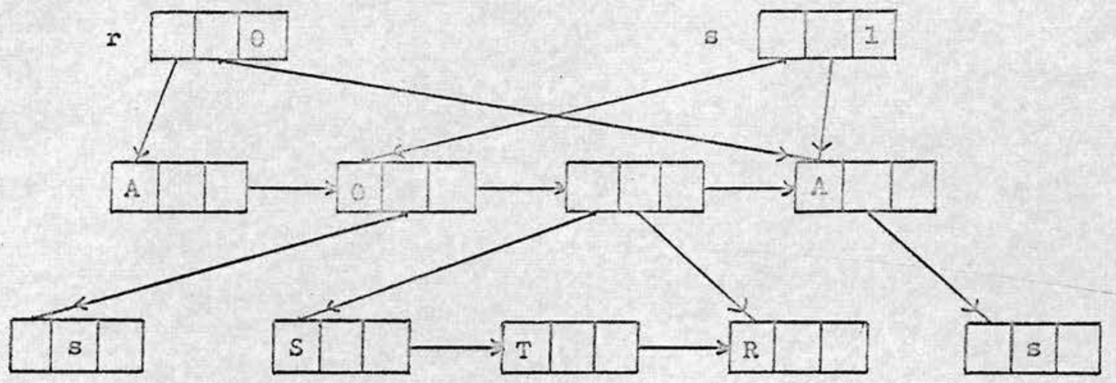
From the programming point of view, it is extremely convenient to be able to treat substrings in the same way as highest level strings. For the sake of consistency, then, the first and last addresses of substrings ought to be stored in those cells which act as substring pointers :



With three-field cells, however, this can only be done by displacing the link to the association list from the cell. The dummy cell enables this to be done. When an association list is needed for a substring pointers cell, which may be relatively infrequent, for it only occurs in situations such as :

```
r = ´A´.(´STR´).´A´
r -> -[1].s
```

a dummy cell can be inserted before the substring pointers cell to which the association list can be linked :



The third field of the location in the run-time stack for each string variable, not needed for a link, is used as a marker to record the form of assignment - by '=' operator or by resolution. It is necessary to distinguish between the two in certain circumstances, discussed below. One difference between variables assigned in the two ways is that the variable assigned by the '=' operator can never be overlapped by other variables, since other variables may only be a sub-part of the original. In particular, association lists are not required to refer back to this original variable from the first and last cells of the total representation, since these are only used to guard against the overlap and replacement condition.

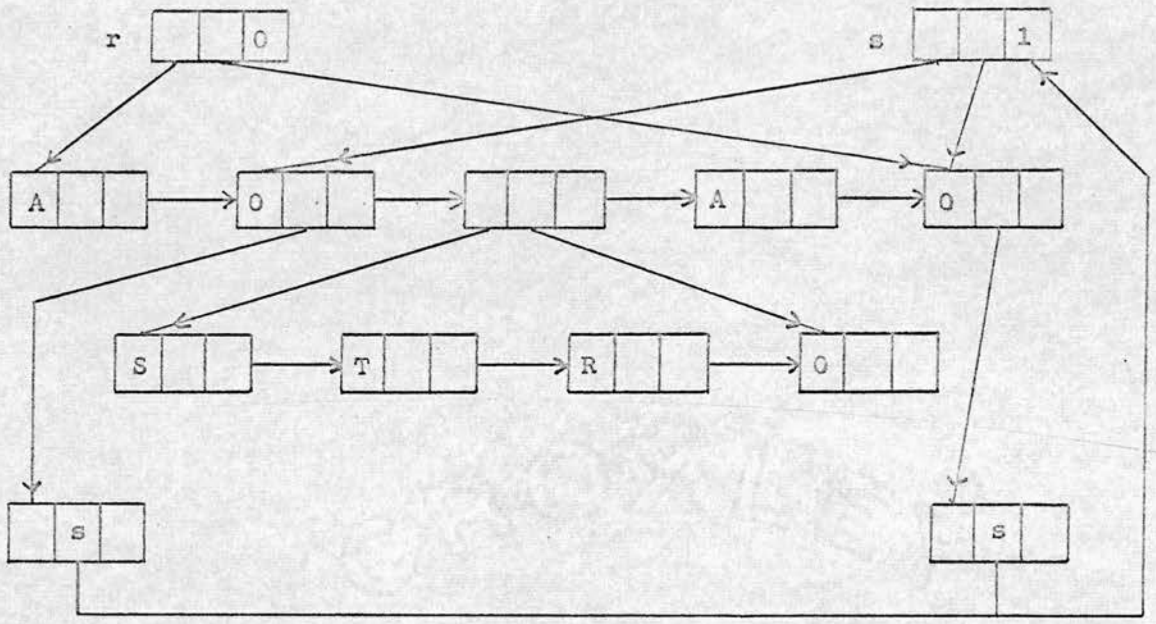
Some confusion could arise between substring pointer cells and cells containing information and a link to an association list. In the practical implementation, the two are differentiated between by negating the links to association lists.

There is a further modification to the list representations as so far described which takes advantage of the existence of dummy cells, purely to ease manipulation of the lists. It was found to be convenient to have the locations of the first and last cells of a representation distinct. This would not be the case for strings with only a single constituent or null value using a dummy cell. It was decided, therefore, to take the location of the first cell exactly as so far described, but in place of the location of the last cell, to take the location of the cell following the last significant cell and store those two locations to identify the string. This arrangement is particularly of value in the implementation of resolution. It implies that a dummy cell should be appended to every string or substring. The situation after :

r = ´A´.(´STR´).´A´

r -> -[1].s

in the actual implementation would therefore be :



Other examples of list structures for typical string values are given in Appendix B.

6. IMPLEMENTATION OF ASTRA STRING FACILITIES

This chapter is concerned with the operations involved in string manipulation from the run-time point of view. Compile-time aspects are dealt with in subsequent chapters.

Evaluation of expressions and resolution can be broken down into a number of basic types of operation, such as concatenating two strings or scanning down n components of a string. Since the representations of strings are list structures of a fairly complex nature, even these basic operations may require relatively substantial pieces of program to perform them. This is quite different from the basic operations of evaluating an arithmetic expression, say, which are almost always built-in hardware functions, such as 'add a number into an accumulator', and so on. In order to control the size of compiled programs, the basic string operations, in other words, basic list manipulating operations, must be in the form of subroutines, which are called upon with parameters varying to meet the particular circumstances at the point of call. This fortunately is no departure in principle since certain functions such as run-time stack control are already performed by such a mechanism. The remainder of this chapter is a description of the basic list manipulating operations required for the ASTRA string facilities and how the general forms of expression evaluation and resolution are broken down into these basic operations. Expression evaluation is described first.

The types of operand allowed in string expressions are :

1. the name of a string variable, string function, or element of a string array, possibly with selection of part only of its value by means of indexes,
2. literal string constants,
3. substring expressions.

The definition of string evaluation requires that the resulting string has a new representation, independent of those of the operands. In other words, the representation must consist of concatenated copies of the representations of the operands. For string function operands, however, no representation is in existence until the function is evaluated, so that no copying is necessary. In this case, the one representation of the value can be used directly for concatenation. The same is also true for literal string constants in this particular implementation. The symbols of a literal are stored as closely packed as possible on a fixed stack in a similar way to that in which numerical constants are stored. At the time of evaluation of a string expression with a literal, a list of the correct form for the literal is produced and this again can be concatenated directly without further copying, to form part of the complete representation. The alternative of storing the literal in the form of a list, to be copied whenever the expression involving that literal is evaluated is quite unnecessarily wasteful of space.

Three basic operations are apparent from the description up to this point :

1. Produce a copy of the representation of a string variable's, or element of a string array's, value.

2. Produce a representation for the value of a literal constant.

3. Concatenate two representations to produce one single representation.

These would be the operations involved in evaluating the string expression :

s . 'ASTRA'

A further basic operation is very similar to 2., namely :

4. Produce a representation for a null string.

This would be required before an assignment such as :

r = _

The selection of parts of string values by means of indexes introduces other basic operations. In the case of variable names, the part of the value required in the expression must be selected before a copy is made. In the case of functions, those parts of the representation not selected must be discarded. Although three types of selection by index are defined in ASTRA, those exemplified by :

r[3]

r[3:4]

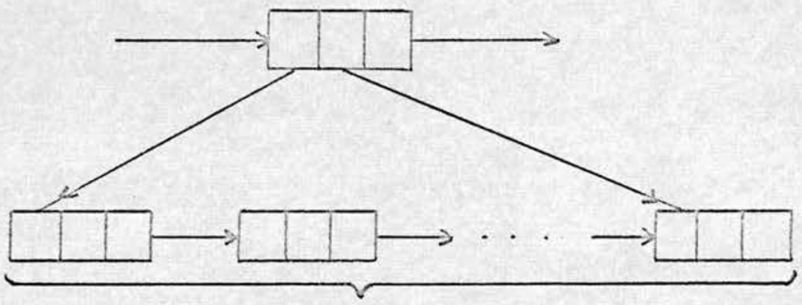
r[3:*]

the first can be regarded as a shorthand form of :

r[3:3]

resulting in two basic types of selection. The distinction between variables, which already hold a value which must be preserved, and functions therefore results in a further four basic operations.

The remaining type of operand, a substring expression, only requires one additional basic operation. After evaluation of the expression, all that is required is an operation to transform the representation produced into that of a string consisting of one constituent which is a substring pointer cell :



representation for expression

This can then be concatenated in the ordinary way.

These nine basic operations are all that is required for the evaluation of any expression. Some examples of the break-down of expressions are presented here to illustrate the process.

Example 1

s . ´ASTRA´
Copy rep. of s
Form rep. for ´ASTRA´
Concatenate two reps.

Example 2

s . ´ASTRA´ . t
Copy rep. of s
Form rep. for ´ASTRA´
Concatenate two reps.
Copy rep. of t
Concatenate two reps.

Example 3

```
s[3:6]
Select 3:6 part of s
Copy rep. selected
```

Example 4

```
fn[5:*] . A(1)
Discard 1:4 part of value of fn
Copy rep. of A(1)
Concatenate two reps.
```

Example 5

```
( _ )
Form null string
Form substring from rep.
```

Example 6

```
s . ( t[1] . 'SUB' )
Copy rep. of s
Select 1:1 part of t
Copy selected part
Form rep. for 'SUB'
Concatenate last two reps.
Form substring from rep.
Concatenate two reps.
```

These examples are intended to illustrate the principles involved in the break-down of expressions. Other operations are also involved in some. The function must be called and evaluated as another basic step of the fourth example. The location of the copy of s must be preserved whilst the subexpression is evaluated in the last example. In all cases, the parameters for the subroutines which implement these basic operations must be set up. Complete details of the parameters and operations are given in Appendix C.

Resolutions may now be considered. Clearly, many of the basic operations already described in connection with expression evaluation will be needed again. Resolution statements are considerably more complex than expressions. Although the process is defined to be from left to right, the possibility of partial backtracking exists. Consider for instance :

```
r -> s . ( t . 'LIT' . u ) . v
```

The first step is to scan string r for a substring; upon finding such the whole process goes down a level, so to speak, to scan for 'LIT' in the substring. If this value is not located within the substring, it does not necessarily imply that the resolution is impossible. Some backtracking must take place so that the scan at the main level can continue in order to locate the next substring, which may then contain 'LIT'. It is also convenient to have backtracking available in another situation. Take :

```
r -> s . 'STR' . 'ING' . t
```

This is a somewhat artificial example to illustrate the point. The straightforward approach is to scan for 'STR'. The components which follow may or may not be 'ING'. If not, then it is necessary to backtrack to locate the next 'STR'. In this case the need is easily removed by forming the single value 'STRING' to scan for, but it is more efficient not to produce a single value in cases such as :

```
r -> s . ''t'' . 'STRING' . u
```

where a copy of t would first have to be made instead of utilising the existing representation for t. Another example where backtracking is unavoidable is :

```
r -> - . ( - . 'SUB' . - ) . 'STRING' . s
```

The other main consideration in the design of the methods of resolution is the requirement that the values of variables intended for the resolved parts should not be affected if the resolution is unsuccessful.

These two considerations, backtracking and unchanged variables, lead inevitably to the process of resolution being a three stage one :

1. preliminary evaluations
2. the scanning process
3. assignment of resolved parts and garbage collection.

1.

Functions can be called within a resolution. For example :

```
r -> s . ''fn1'' . t
r -> A(fn2) . 'ING' . -
```

where A is a string array indexed by the value of a function. If a function appears once in a statement it must only be called once. Where there may be backtracking over a function call it must not be re-evaluated in case of possible side-effects. It is of course much more efficient to perform such evaluations only once and the same applies to many of the other types of operand in a resolution statement. Parts of strings should only be selected once, as in :

```
r -> s . ''t[3:6]'' . u
```

Indexes should only be evaluated once :

```
r -> s[n*m+3] . -
```

Representations for literals should only be produced once, from the close packed form :

```
r -> s . 'RES' . t
```

All this implies a preliminary evaluation stage prior to the actual scanning process. Duplication is thereby avoided. The parameters to the basic operations of resolution are evaluated prior to the scanning, unlike expressions where the parameters can be evaluated as evaluation of the expression proceeds. The result of this first stage is therefore an array containing the parameters which are then used by the basic operations of the second stage. As will be described in the next section, the resolution is broken down into a basic operation per operand, so that each location in the array refers to one operand.

At this stage we introduce terminology for the identifying information of a string. We write :

A(r)

to mean the three-field value :

```
address of first cell / address of last cell / 0
     in rep. of r    /       in rep. of r   / or 1
```

which identifies the representation of r. We write

@r

to represent the address of the variable r on the run-time stack. Clearly, the information at @r is A(r).

Some examples of the information stored in the first stage array for various resolutions are now given :

Example 1

```
                    r -> s . 'SCAN' . t
```

for which the information is :

```
                              @s
                              A('SCAN')
                              @t
```

Example 2

```
              r -> s[2].t.u['SCAN'].v.''w[6:*]''.'CANS'.x
                              2 / @s
                              @t
                              A('SCAN') / @u
                              @v
                              A(w[6:*])
                              A('CANS')
                              @x
```

Where an index follows a variable indicating a scan down the specified number of components the value of the index is stored alongside the address of the variable, i.e. packed into the same word as the address. Similarly, when a value to be scanned for is 'named', A(value) is packed with @variable into the same word. Where a substring is to be scanned for, no evaluation of parameters is involved, but a location in the array is taken and left empty for the sake of consistency. It is also convenient for the second stage for a location to be taken and left empty for the end of a substring :

Example 3

```
                  r -> s . ( t . 'SUB' . u ) . v
                              @s
                              0
                              @t
                              A('SUB')
                              @u
                              0
                              @v
```

Example 4

```
          r -> ('JOHN'.(s[n].'JAMES')).'JERRY'
                              0
                              A('JOHN')
                              0
                              n / @s
                              A('JAMES')
                              0
                              0
                              A('JERRY')
```

Zero is stored in the array for dummy names - in place of the address.

Example 5

```
          r -> - . s[-:('+').('-')] . -
                              0
                          1 / A(('+').('-')) / @S
                              0
```

A tag bit is set with A(expression) when a multi-alternative form is present.

2.

Since assignment of resolved parts cannot be made during the course of scanning, for the reason that it may fail at a later stage in which case all variables including those which had a possible value to be assigned must be left unchanged, the obvious solution is to build up a second array containing the information necessary to make the correct assignments when the resolution has been found to be successful i.e. at stage 3. This is also satisfactory for cases of backtracking. When backtracking causes a change in the value of the part to be assigned, all that is necessary is to overwrite the information stored in the array with the modified value. The reason for the extra locations allowed for in the stage 1 array is so that the positions can correspond one for one with the postions of this array produced in stage 2.

In practice, the arrays of stages one and two are interlaced, one location for parameters from stage 1, one location for information produced by stage 2, and so on, as this arrangement is more convenient to handle. Since the space for these arrays is only required during the execution of the resolution statement, no permanent area has to be set aside for each resolution and the normal temporary working area of store, beyond that taken by declared variables on the run-time stack, can be used. (This is also used during evaluation of arithmetic expressions for storage of partial results for instance.)

Resolution can be regarded as a process of matching the operands on the right- hand-side with contiguous parts of the string being resolved. This holds for all types of operand - names, literals, and substrings. The result of the second stage in the resolution process is this matching. The method of identifying a string, namely the pair - location of first cell and location of cell following last cell - provides a simplification of the information which has to be stored to identify the matching. The significant feature is that the second location in the pair identifying one operand is the same as the first location in the pair identifying the following operand. The only information which has to be stored by stage 2 therefore is the location of the first cell for each matched operand. The second location in the pair identifying the operand can be found from the next position in the array, when all matching has been completed, i.e. at stage 3. Two examples should clarify this :

Example 1

```
            r -> s[2] . t
      stage 1 array   stage 2 array
         2 / @s
                          @r[1]
         @t
                          @r[3]
         0
                          @r[*]   i.e. address of last cell of r.
```

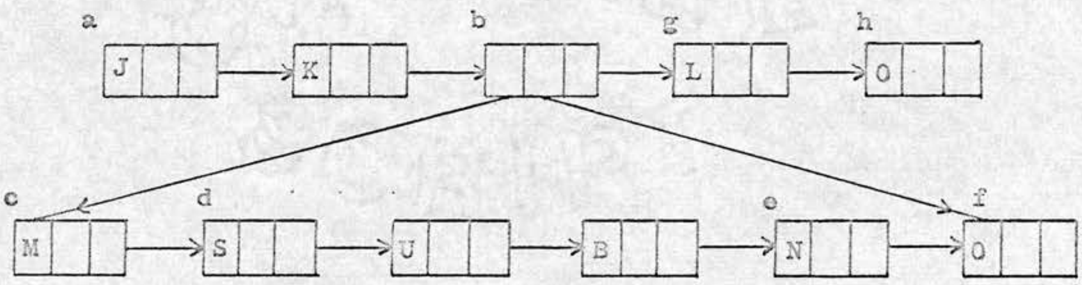
The required pairs for s and t are :

(@r[1] , @r[3]) and (@r[3] , @r[*])

Example 2

```
        r -> s . ( t . 'SUB' . u ) . v
        stage 1 array   stage 2 array
            @s
                            a
            0
                            b
            @t
                            c
            A('SUB')
                            d
            @u
                            e
            0
                            f
            @v
                            g
            0
                            h
```

where a-h might be locations on a representation for r such as :



The required pairs for s, t, u, and v are respectively :

(a,b) , (c,d) , (e,f) , (g,h)

The basic operations for which permanent subroutines are to be provided fall easily into place. By and large, there will be one for each operand in stage 2. These are now described :

a. Assign location of current cell to the array. (By 'current cell' is meant the cell in the representation of the string being resolved up to which matching has so far progressed.)

In the resolution :

r -> s . ′NEW′ . t

this operation is all that is required for operands s and t. It is, in fact, so simple as not to require being made into a subroutine, but to be effected by in-line code.

b. Assign the location of the current cell to the array, then scan along n components of the representation.

This type of operation would be required for string s in :

r -> s[n] . -

The parameters to this and subsequent operations are the current position on the representation of the string being resolved from which matching is to take place, the last cell in that string or substring (used to check when the end is reached) and some of the information in the array produced by stage 1. The addresses of string variables in the stage 1 array are not needed in stage 2, but only in stage 3 when assignments are being made. The scanning operations assign to the array and exit with values of the parameters set for the next operation. In the example :

r -> s[n] . -

n is taken from the stage 1 array, and the other two parameters for the scanning operation are @r[1] and @r[*] . This operation would assign the location of the current cell i.e. @r[1] to the array and exit with the parameters modified to @r[n] and @r[*] , for the next operation.

c. Scan along from the current position for a literal value and assign the location of the first cell matched to the array.

For example :

r -> s . ′OLD′ . t

This operation is also used for operands of the type ″name″ :

r -> s . ″fn(m,n)″ . t

and both variations :

r -> s . t[′NEW′] . u

r -> - . s[-:(′+′).(′-′)] . -

d. Compare a literal with the current position in the string without scanning and assign the location of the first cell to the array if successful.

For example, the 'ING' operand in :

```
r -> s . 'STR' . 'ING' . t
```

or much more likely - 'LIT' in :

```
r -> s . ''fn(m,n)'' . 'LIT' . t
```

and also 'REST' in :

```
r -> s . ( t ) . 'REST' . u
```

In each case, no scanning is involved, since the literal must appear immediately following the final position of the previous operand if the resolution is to succeed.

This is also used for the forms exemplified by :

```
r -> s . ( t ) . u['REST'] . v
r -> - . 'TH' . s[-:('A').('E').('I').('O').('U')] . u
```

e. Scan along from the current position for a substring and assign the location of the substring pointer cell to the array.

For example :

```
r -> s . ( t ) . u
```

f. Check that the current position is a substring pointer cell without scanning and assign the location to the array.

This operation is in the same relation to e. as d. is to c., and would be used in resolutions such as :

```
r -> s[3] . ( t ) . u
r -> - . 'PRE' . ( s ) . -
r -> - . ( s ) . ( t ) . -
```

The remaining operations, with one exception, are concerned with the terminations of strings and substrings. In situations such as :

```
r -> - . ( s . 'WOW' . t ) . -
```

when 'WOW' has been located, there is no more scanning or checking to be done, the location of the last cell of the substring must simply be stored in the array, so that the complete pair for t can be ascertained in stage 3. For situations such as :

```
r -> - . ( s . 'WOW' ) . -
```

after 'WOW' has been found, a check must be made that those cells matched actually terminated the substring. If not, then some backtracking must take place in order to scan for a further 'WOW', which may terminate the substring.

As mentioned above, the scheme of resolution described uses input parameters for the basic operations consisting of current and last locations of the representation, which are modified by one operation in preparation for the next. When a substring is involved in the resolution and is located in the main string level, these parameters must be preserved whilst the resolution process for the substring is in progress and reinstated after it. Operations e. and f. can conveniently be used for preserving the main level parameters and setting up parameters for the substring resolution i.e. the first and last cells of the substring, and in fact are arranged to do so. The vacant location on the stage 1 array at the start of each substring is used for this purpose. Restoring the parameters for the higher level resolution to proceed can be made a function of the operations called at termination of substrings.

The particular division of functions of the terminating operations is to a large extent arbitrary. They are, in the implementation, as follows :

g. Check for end of substring and exit from substring (i.e. restore parameters) if successful.

h. Exit from substring after failure to resolve.

i. Exit from substring after success (no checking needed).

For example :

```
r -> - . ( s ) . -
```

j. Check for end of string (highest level, not substring).

For example :

```
r -> s . 'WOW'
```

k. Assign last cell of representation to stage 2 array (highest level string).

For example :

```
r -> s
```

k., like a., is simple enough not to be a subroutine and is purely in-line code.

The remaining operation which requires a subroutine is that concerned with backtracking. There is a certain amount of clearing up to be done when backtracking becomes necessary. The following are some more examples of situations in which backtracking must be catered for :

```
r -> s . ( t ) . 'POST' . -
r -> s . 'PRE' . ( t . 'SUB' ) . -
r -> s . t[1]
```

The code involved is collected into a subroutine purely in order to minimise the length of compiled code. Hence the further operation :

1.   Backtrack.

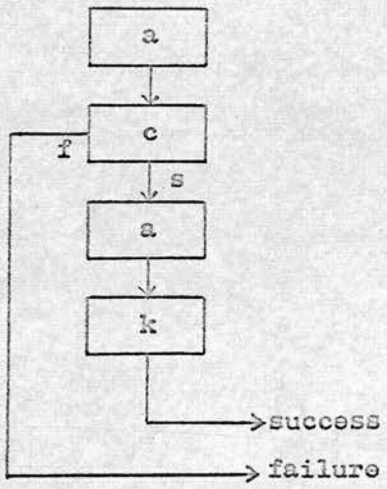
To clarify the use of these basic operations, some examples are now given. Although it has not been emphasised in the preceding discussion, these basic operations will show up any failure to resolve as well as proceeding in an orderly fashion when successful. The operations illustrated in the following examples, therefore, show two exit routes, one for success and the other for failure, where applicable. In fact, there are really three classes of exit. Success, complete failure at the current substring, necessitating return to the higher level, or at the main string level, and partial failure at the current level, i.e. where backtracking for a further attempt is possible. These three cases are denoted by s,f and b in the diagrams.

Example 1

r -> s . ´LIT´ . t

stage 1 array   stage 2 operations

```
@s

A(´LIT´)

@t

0
```



When a resolution is successful, there is just one exit point from the sequence of basic operations. There may be a number of possible failure exits, since several basic operations may possibly fail. They are all, however, for economy, directed to a common point for further action. This action will depend on whether the resolution is part of a condition or an unconditional instruction or a substring and so on. The particular actions needed are discussed in connection with stage 3.

Example 2

r -> s[2].t.´VAL´.u.´´w[2:4]´´.´GIN´.x

stage 1 array   stage 2 operations

2 / @s

@t

A(´VAL´)

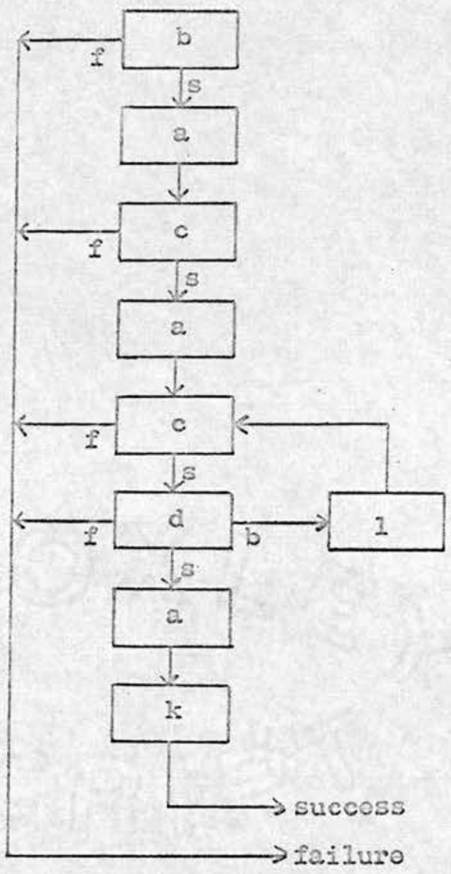
@u

A(w[2:4])

A(´GIN´)

@x

0



There are all three possible exit routes from operations of type d, i.e. matching ´GIN´ without scanning. Route s is taken if ´GIN´ is found; route b if it is not found and the end of the string has not been reached i.e. the pattern may occur further down the string; route f is taken if ´GIN´ is not found and the end of the string has been reached when, clearly, no match is then possible. In fact, the failure exit from type c operations is also caused by the end of the string being reached before the pattern was found.

Example 3

r -> s . (t . ´WINE´ . u) . v

stage 1 array   stage 2 operations

@s

0

@t

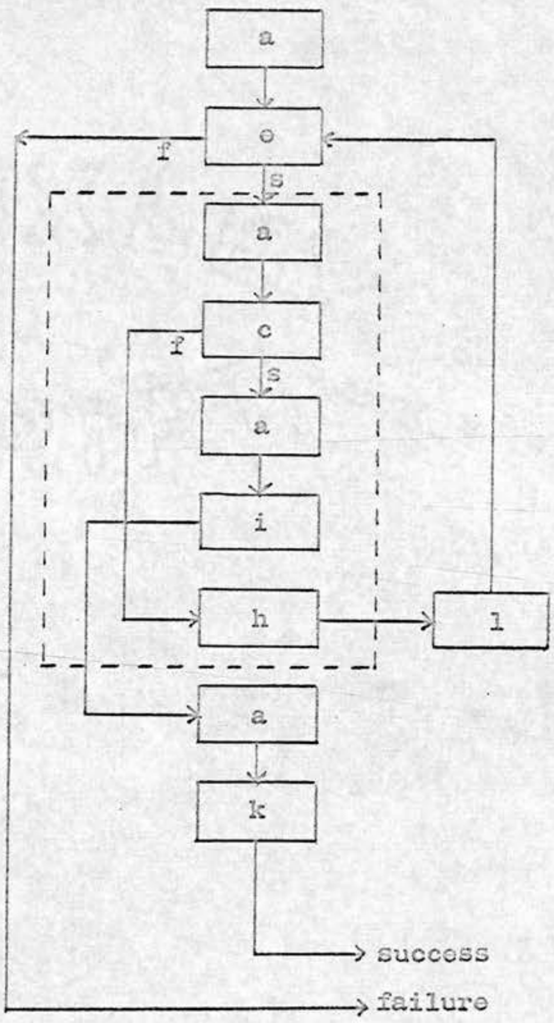
A(´WINE´)

@u

0

@v

0



The operations enclosed in the dotted box are those involved with the substring. The less than optimum flow of control shown in this diagram results from the attempt to produce a consistent scheme for all, including highly complex, resolutions. One or two redundant transfers of control in a resolution hardly affect the overall performance however.

Example 4

r -> s . 'X' . (-[2]) . t

stage 1 array   stage 2 operations

@s

A('X')

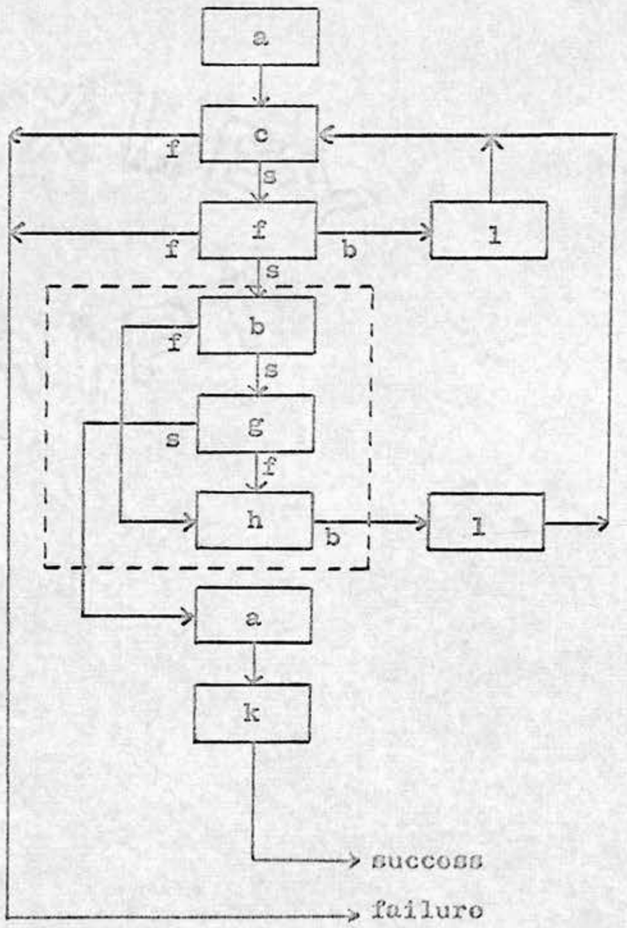
0

2 / 0

0

@t

0

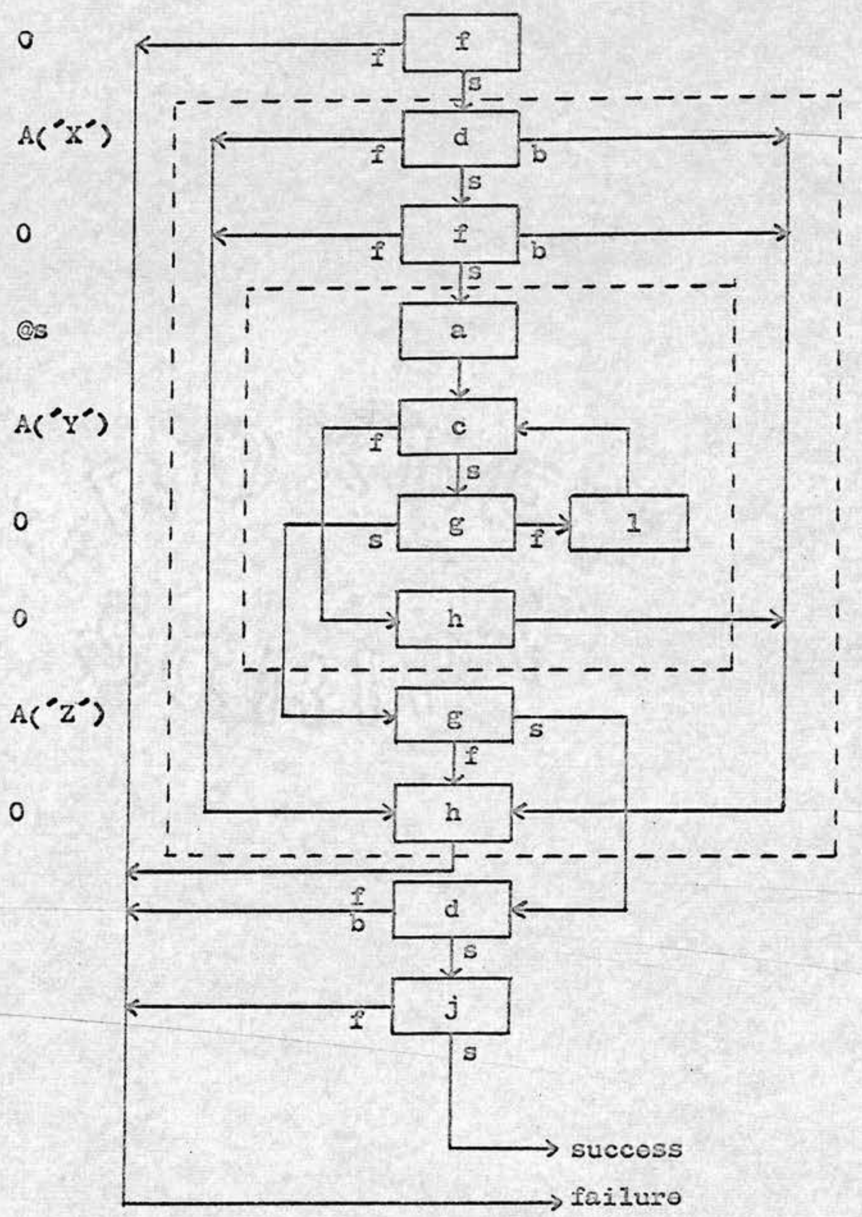


The exit from the substring resolution on failure, operation h, leads to a backtrack operation 1, but this is outside the dotted lines, since the fact that backtracking is possible can only be determined at the higher level.

Example 5

r -> ('X' . (s . 'Y')) . 'Z'

stage 1 array   stage 2 operations



In this case, there are possible backtrack exit routes from the d and f types of operation, but context demands that they both be treated as 'complete failure' exits. In other words, the exit routes are defined completely by the particular basic operation. Rather the exit 'point' is defined and the route from this point defined by the context. This fixing of routes is discussed further in connection with the compilation of resolutions.

3.

The final stage of resolution, assignment of resolved parts and garbage collection, takes slightly different forms in the two contexts of resolution - unconditional and conditional statements. The straightforward case is that of an unconditional resolution statement. The result of the first two stages is to pass control either to the success point or to the failure point, as indicated in the diagrams above. Upon failure, in the unconditional case, the whole program has to be wound up, so the action at the failure point has to be to jump off to the controlling monitor indicating the fault. Upon success, the assignments of the resolved parts must take place and the literals and string values used for matching during the resolution, or rather their representations, must be cleared where appropriate. In certain cases, there is no garbage to be cleared, for example :

r -> - . ''s'' . -

The representation used for matching is that of s itself and so must not be destroyed. In cases such as :

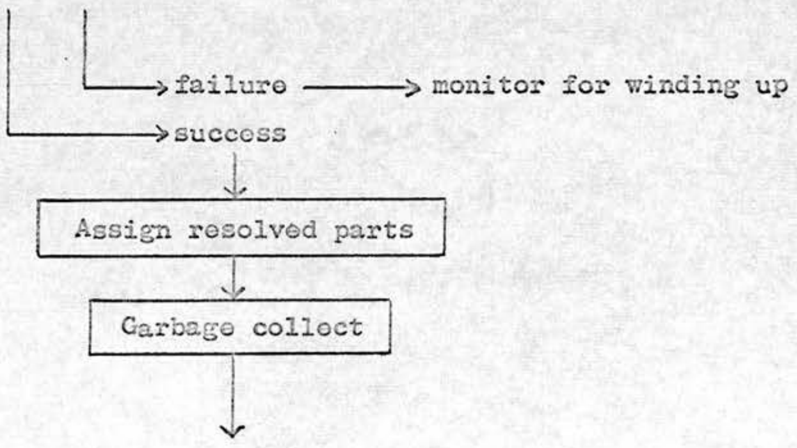
r -> - . ''fn'' . -

r -> - . 'HUB' . -

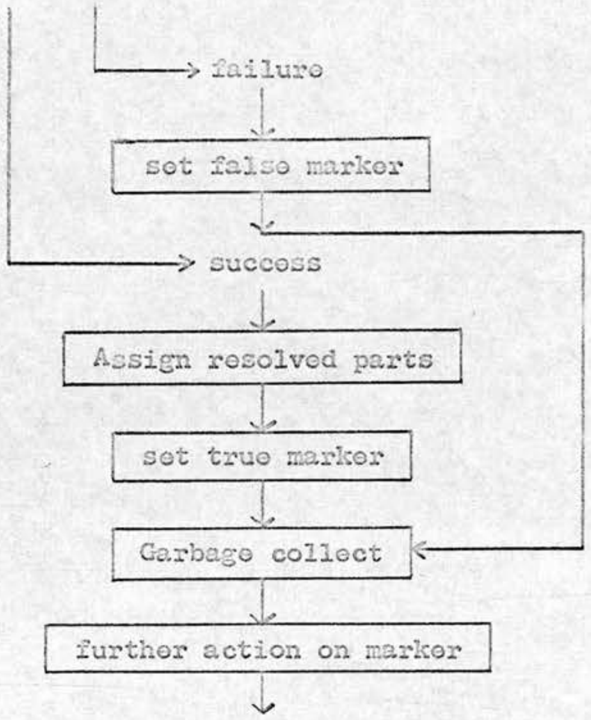
where fn is a string function, the garbage must be dealt with. This distinction will be noted again when the compiling algorithms are discussed.

When the resolution is part of a condition in a conditional statement, different action has to be taken. At the failure point, the program is not to be wound up, but merely to indicate the falsity of the condition, allowing the program to proceed. No assignments of resolved parts are required, but there will still be the garbage involved in the matching to be cleared. For this reason, the two forms of action are separated. In terms of block diagrams, the arrangement is :

Unconditional :



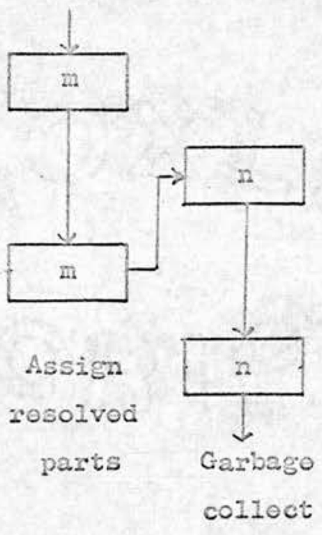
Conditional :



Both the actions are broken down into basic operations, either for each variable to be assigned a resolved part, or for each representation used in matching to be cleared. The array produced by stage 1 containing addresses of variables etc., used by stage 2, is also used for this third stage. E.g.

r -> s . ´STR´ . t . ´ING´

```
stage 1 array
    @s
    addr1
    A(´STR´)
    addr2
    @t
    addr3
    A(´ING´)
    addr4
    O
    addr5
```



The value to be placed in the location set aside for variable s is, in the three fields :

addr1 / addr2 / 1

The addr1,...addr5, were placed in the array by the stage 2 operations. The ´1´ in the right-most field indicates that the string is a resolved part; whereas the value in the location for variable r is :

```
addr1 / addr5 / 0
```

For all assignment operations, ´m´, the address of the variable to be assigned to is taken from the stage 1 array, and the value to be assigned is the conjunction of the contents of the two positions, one on and three on from the address. In addition to the assignment, the back references (to the location of the variable) must be pushed down from the first and last cells as described previously. Slight complications may arise in doing this and more dummy cells are sometimes necessary as mentioned in the previous chapter. For instance :

```
r -> s . ( - ) . -
```

A dummy cell has to be inserted before the substring pointer cell and the back reference pushed down from that, since all the fields of such a substring cell are already occupied. In practice, the new cell has to be inserted after the substring cell and the substring links copied into it, since the lists are only linked from front to end, implying that the penultimate cell cannot be accessed directly without scanning from the front.

It is also necessary to insert dummy cells where the string has a null value in order not to violate the convention adopted of having the locations of first and final cells distinct. For example :

```
r = ´ASTRA´
r -> ´ASTRA´ . s
```

This use of dummy cells is somewhat inelegant but is a simple way out of the various difficulties.

Operations of type n highlight the general questions of garbage collection and storage management. With any list processing scheme, there has to be a pool of ´free´ cells of some sort to be drawn upon as required. the usual method is to set aside an area of store and link the locations together as cells of an ´Available Space List´. Such an area of store can hardly be accommodated on a dynamic stack which programs with block and routine structure require, and which expands and contracts as blocks and routines are entered and left. The method adopted in ASTRA is to set aside a static amount of storage from which all string representations take their cells. In fact this area is set aside after the fixed constants required by the program and before the dynamic stack. Its size can be set by the programmer by a statement such as :

<u>**asl**</u> = 10000

If no such statement appears,

<u>**asl**</u> = 5000

is assumed. Garbage collection can be dealt with in basically two ways. The method adopted by most LISP implementations illustrates one of these. When a list is no longer in use, its cells are not immediately reclaimed, but left unchanged in store until the Available Space List can no longer supply the demands of the program. Only at this stage are all the free cells collected together into an Available Space List for the program to proceed. The free cells are identified by scanning down all existing lists which are in use and marking their cells, so that a simple scan of the area set aside can pick up those that are not marked and hence not in current use.

The alternative approach is to reclaim any cells which are released as soon as they are released and attach them to the Available Space List for immediate re-use. Whether this is possible or convenient, depends to a large extent on the language involved. In the SLIP system, a halfway stage is used, under programmers control, in which the sublists are not detached when a list is returned to the Available Space List. Only when cells are taken from the Available Space List are they inspected for a link to a sublist. If such a sublist is found and not in use as a common sublist of another list it is detached from the cell and attached to the end of the Available Space List itself.

The design of ASTRA is such that it is very convenient to do all garbage collection as processing proceeds, and this method was therefore adopted. The slight advantage of the SLIP and LISP systems for programs where the turnover of cells is small and a LISP type garbage-collect may not ever be necessary, was felt to be relatively unimportant. Cells are available to be reclaimed and returned to the Available Space List whenever an assignment is made to a string variable :

r = ´STRING´

This statement indicates that a new value and representation is to be assigned to r. The old value is to be discarded and therefore its representation can be reclaimed. The marker in the third field of each string variable´s location on the stack indicates whether the variable was assigned by resolution or not. If it was assigned by resolution, only the cells containing the back references attached to the representation of the main string's value, which have to be removed, can be returned to the Available Space List and not the cells of the representation of its value since these are still required as part of the main string's representation. E.g.

```
r = 'ASTER'
r -> 'AS' . s
s = 'STARE'
```

If the variable was assigned by an ordinary assignment statement, the whole representaion can be returned to the Available Space List.

```
r <- 'REPLACEMENT'
```

In this instance, that part of the representation which is being replaced has to be returned to the Available Space List. The remaining instance is that of leaving the block or routine in which string variables are declared. E.g.

```
begin
string r, s, t
stringarray A(1:10)
. . . .
end
```

Since the strings r, s, t, A(1), . . A(10) can no longer be accessed i.e. have been 'undeclared', their representations can be dispensed with (there are no own variables in KDF9 Atlas Autocode or ASTRA).

In returning a string representation to the Available Space List, all cells must be inspected for the back references, in order to clear the variables with resolved values which overlap. It is therefore convenient to unlink all the sublists at this point, rather than to postpone the process as in SLIP. When a cell with a sublist containing back references is encountered, the reference indicates the location assigned to the variable on the stack. Whether the cell with the back reference is that of the first cell or the last cell of the resolved variable's representation, the corresponding last cell or first cell can be found from the information stored in the variables location on the stack. The back reference in the sublist linked from this cell then must also be removed. This is necessary when dealing with a 'replaced' string, when overlaps can occur. Finally, the resolved variables location is set to zero i.e. unassigned.

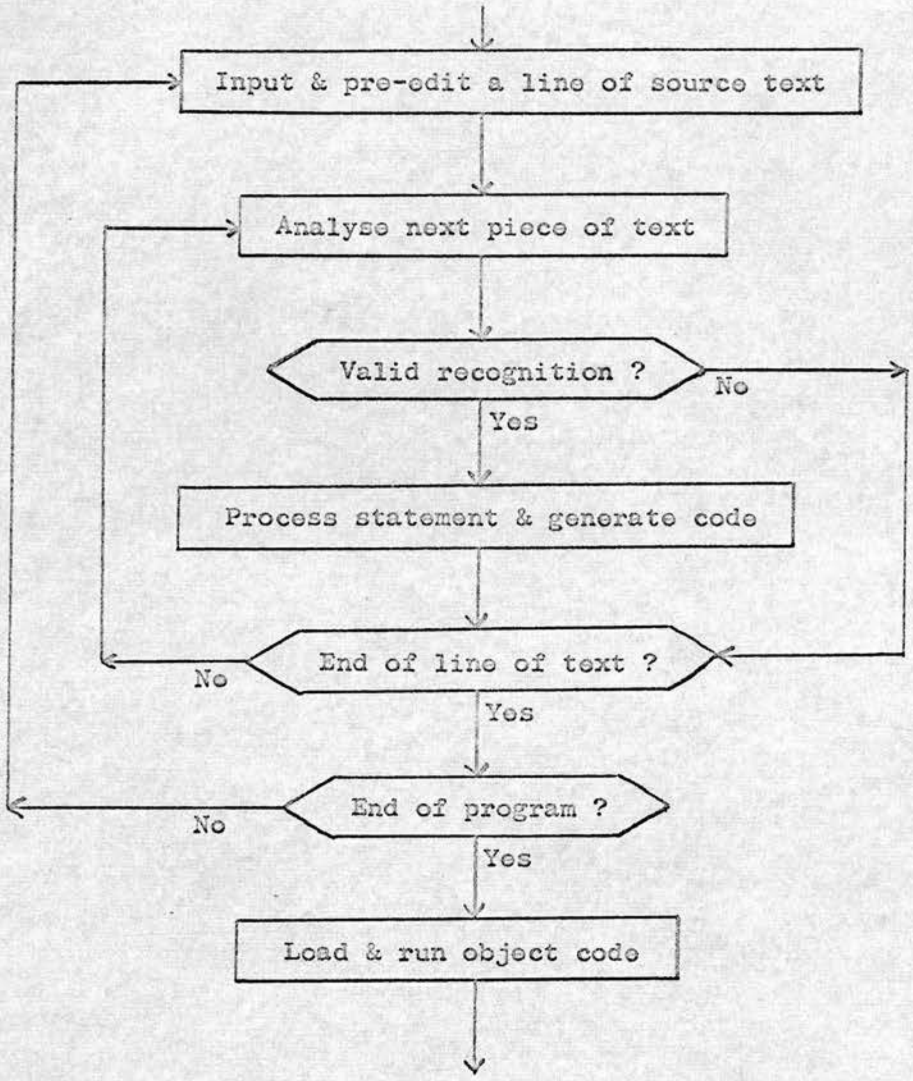
7. COMPILING TECHNIQUES

The string facilities which have been added to Atlas Autocode to produce ASTRA are a built-in feature of the new language. In other words, the compiler for Atlas Autocode had to be modified, rather than, say, using a prepass over the source program text to convert string statements into some equivalent Atlas Autocode form and then using the standard Atlas Autocode compiler. Before the particular compiling algorithms for the string facilities can be described, however, the techniques used in the Atlas Autocode compiler, adhered to in making the modifications, must be presented.

The compiler is ʼsyntax-directedʼ. That is, tables containing a syntactic description of the language are used and the comparison of the souce text with this description ʼdirectsʼ the semantic processing and generation of machine code. The processing proceeds source statement by source statement and for each source statement is separated into the two stages, syntactic analysis or recognition, and semantic processing, which can be discussed separately.

The present author was a member of the team which wrote the original Atlas Autocode compiler for KDF9 and in particular was charged with the development of the analysis stage, but was also concerned with the semantic stage together with the other members of the team : Paul Bratley, Harry Whitfield and Peter Schofield.

The scheme of operation of the compiler can be shown best in a diagram :



Input & pre-edit a line of source text
Analyse next piece of text
Valid recognition ?
No
Yes
Process statement & generate code
End of line of text ?
No
Yes
End of program ?
No
Yes
Load & run object code

The pre-edit of the source text consists of removing redundant space characters and partially identifying the language's key-words e.g. integer, string, begin, etc. In program preparation, the forms %INTEGER, %STRING, etc. are used, the per cent sign acting as a kind of visible shift character. The pre-edit modifies the codes of the characters (all letters) in the shift mode, for convenience of subsequent analysis. The shift sign is used to avoid ambiguity with program identifiers. Any non-letter character cancels the shift mode.

The inner loop of the diagram illustrates that more than one statement may appear on one line when the separator semi-colon is used.

ANALYSIS

The method of syntactic description of the language is that of phrase structure. Phrases may be defined by statements such as :

```
P[DIGIT] = '0','1','2','3','4','5','6','7','8','9';
```

Phrase names are enclosed in square brackets and literals, which refer to the actual characters appearing in the source text, are enclosed in quotes. The general form of definition of a phrase is :

```
P[phrase name] = (alternative 1) , (alternative 2) , . . . . . ;
```

The phrase thereby defined may take any of the alternative forms. The alternatives themselves may contain any number of literals and phrase names. For example :

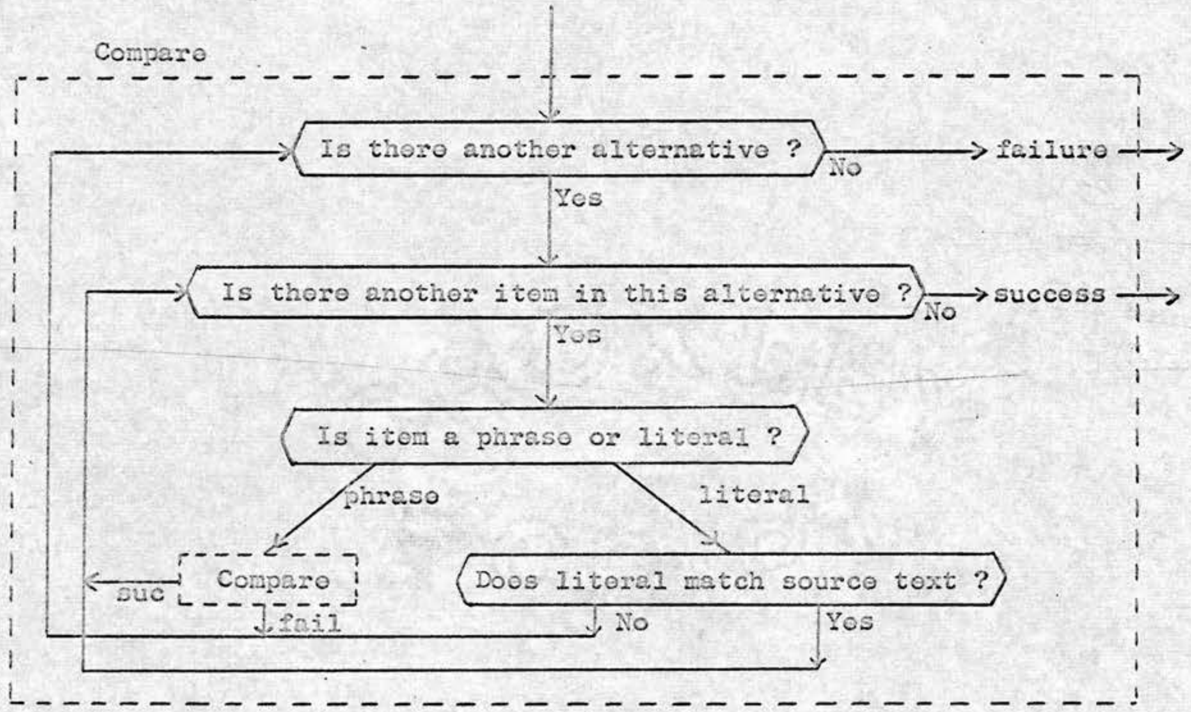
```
P[N] = [DIGIT][N] , [DIGIT] ;
```

This illustrates the possibility of a recursive defintion of a phrase N to consist of one or more digits. The final alternative in any phrase may be a 'null', written 0, when 'nothing' may constitute a valid occurrence of the phrase. There are some restrictions on the order in which alternatives may appear and phrases and literals within alternatives, which are a result of the particular comparison algorithm used, discussed below.

The compiler operates upon one source statement at a time. The phrase definitions are therefore based upon the source statement as the basic unit. One phrase, SS, is defined which has as its alternatives the various forms of source statement which are permissible. All other phrases are subsidiary to this. Complete Phrase Structure definitions of ASTRA are given in the next chapter. For the moment, as an illustration, a very much abbreviated set is given here :

```
        P[SS] = [UI][S],
                'if'[COND]'then'[UI][S],
                [N]':',
                [TYPE][NAME LIST][S],
                'begin'[S],
                'endofprogram'[S];
        P[UI] = [NAME]'='[EXPR],
                'caption'[TEXT],
                '->'[N];
        P[S] = ';','-';
        P[TYPE] = 'integer','real','string';
        P[NAME LIST] = [NAME]','[NAME LIST],[NAME];
    etc.
```

For comparison of source text with the phrase structure description i.e. phrase SS, a right-linear recognition algorithm is used. The algorithm is given in the following diagram :



Compare
Is there another alternative ?
No
failure
Yes
Is there another item in this alternative ?
No
success
Yes
Is item a phrase or literal ?
phrase
literal
Compare
suc
fail
Does literal match source text ?
No
Yes

The dotted box indicates a recursive reincarnation of the comparison algorithm for the subsidiary phrase which is the next item in some alternative of some phrase. The word another in 'another alternative' and 'another item' used in the diagram is intended to mean the next component to the right in the written phrase structure description -

hence right-linear recogniser. This restricts the phrase structure to being right-linear also, though this is hardly a restriction in practice. It means that a phrase such as N must be defined in the order :

```
P[N] = [DIGIT][N] , [DIGIT] ;
```

and not :

```
P[N] = [N][DIGIT] , [DIGIT] ;
```

or :

```
P[N] = [DIGIT] , [DIGIT][N] ;
```

In the first incorrect case, a recursive loop in the comparison routine would occur, and in the second, a single digit would be recognised as a valid occurrence of N, instead of all the digits there may be. These two factors can inhibit a valid recognition of source text, but there is also a third factor which, while not inhibiting valid recognition, has a profound effect on the efficiency of the operation of the algorithm. The factor referred to is that of backtracking. The example, phrase N, illustrates this. The last digit of each N will be matched twice ; firstly as the first item in the first alternative and again as the second and valid alternative when the second item of the first alternative [N] has failed to match i.e. not found another digit. Although in this case, relatively little time will be wasted, if larger sequences of text are involved with perhaps many phrases matched, the waste can become prohibitive. Fortunately it is very easy to avoid backtracking by redefining the phrase in a different way, so long as the necessity is recognised. For phrase N, the following definitions would suffice :

```
P[N] = [DIGIT][DIGITS];
P[DIGITS] = [DIGIT][DIGITS] , 0;
```

Each digit is thereby only ever recognised once. As an illustration of the saving in a practical case, phrase structure for conditions may be examined. At an early stage of the development of the compiler, before the importance of avoiding backtracking was fully recognised, the following definitions were in use :

```
P[COND] = [SC]'and'[COND],[SC]'or'[COND],[SC];
P[SC] = [EXPR][COMP][EXPR][COMP][EXPR],
        [EXPR][COMP][EXPR],'('[COND]')';
```

Phrase [EXPR] indicates an arithmetic expression and phrase [COMP] represents the comparators =, ¬=, <, . . etc. Take as an example of a condition :

x=0 <u>or</u> y=0

The SC x=0 is matched twice and the SC y=0 three times before a match is found. To recognise x=0 as an SC requires x, =, and 0 to be matched twice so as to allow for the possibility x=.=y say. The same is true for y=0 . In other words, x, =, and 0 would be matched four times and y, =, and 0 six times. With bracketed sub-conditions, the amount of repetition rises steeply. One particularly long and complex condition used as a test case took 76 seconds to be recognised. The same condition analysed using the same program but more efficient phrase structure then took 0.35 seconds to be recognised - a factor of over 200 ! The better phrase structure in use was :

```
P[COND] = [SC][REST OF COND];
P[REST OF COND]='and'[REST OF COND],'or'[REST OF COND],0;
P[SC] = [EXPR][COMP][EXPR][REST OF SC], '('[COND]')';
P[REST OF SC] = [COMP][EXPR], 0;
```

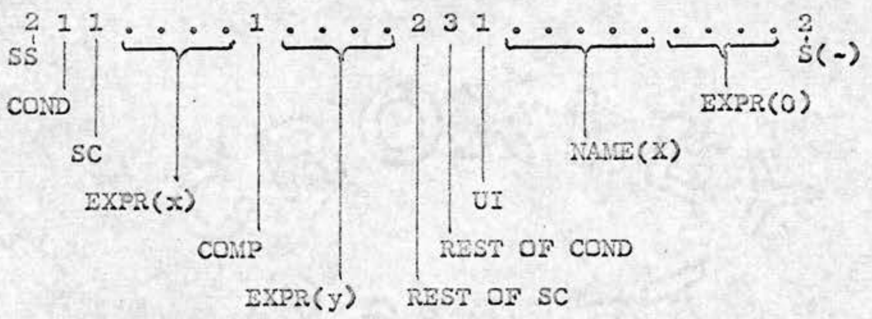
Efficiency was also found by a judicious choice of order of alternatives within phrases, where there was no problem of an invalid recognition resulting. In particular, the basic phrase, SS, was ordered so that the most commonly occurring types of source statement appeared as the first alternatives. For example, assignment statements occur much more often than endofprogram and so appear earlier in the alternatives for SS. In order to determine the best possible order, a number of programs, both long and short, from different programmers (to overcome varying styles of programming) and intended for different purposes so as to be a fairly representative batch, were examined and an overall frequency of the types of statement found.

When a statement has been found to be a valid member of the class of statements SS, the analysis tree or 'analysis record' relating to it is passed on to the semantic phase. If the statement is invalid, no further processing of it is attempted and the analysis proceeds, after a suitable fault message with the next statement. This class of faults corresponds fairly closely with those caused by faulty preparation of the program input medium - cards or paper tape. As such the errors are usually very easy to locate, it was not felt necessary to attempt to pinpoint the precise position of error in the statement since this would degrade the performance of the analysis routine. It is not at all obvious, in fact, how to locate the error, since the algorithm automatically backtracks over the text when no match is found and it is necessary to try another alternative in any phrase definition. However, when the phrase structure is designed to obviate as much backtracking as possible, a good indication of the position of error can be found by maintaining a continuous record of the furthest position attained along the line of text during analysis. As stated above though, this was felt to be unnecessary and was not incorporated.

The 'analysis record' produced by a valid recognition is a precise specification of the particular statement in terms of the given phrase structure. Basically, it consists of a single array containing the numbers of the successful alternatives in the phrases involved in the valid recognition. For example, using the definitions of SS above, and the backtrack-eliminating definition of COND, the statement :

<u>if</u> x=y <u>then</u> x=0

would produce an analysis record :



It is clear from this example, that no record of the tree structure of the analysis is retained. Effectively, all the nodes of such a tree have been compressed into a linear format which keeps the ordering relationship, however. Another feature of the record is the absence of any information relating the alternative numbers to the phrases concerned. This was found to be unnecessary for the purposes of the semantic phase since the record always starts with an alternative of SS and the thread can be followed onwards throughout the record, simply by referring to the tables of phrase structure in use. The final notable absence is any information relating to the literals which would form the terminal points of an analysis tree. The reason for this is the same as that concerned with the omission of phrase identifiers from the record, namely, that no extra information results.

There are certain classes of object which it is necessary to make into exceptional cases. These are objects such as NAME, CONSTANT etc. There are two reasons for treating these exceptionally. Firstly, the efficiency i.e. speed of recognition, can be markedly improved in certain cases by using special purpose procedures rather than the general purpose syntactic analysis procedure. Secondly, it is convenient to build up tables of various kinds which are used by the semantic phase in addition to the analysis record.

An example of the first kind is given by phrase DIGIT above. To recognise that a character, say that contained in 'text(i)' i.e. position i of an array 'text' , is a digit requires that it be tested successively against '0', '1', . . and so on. If a special purpose procedure could be invoked, the valid recognition of a digit could be reduced to the success of a condition such as :

'0' <= text(i) <= '9'

This is so because it is known that the internal character codes have consecutive equivalent integer values for the digits (even though they are not 0, 1, 2, . .). The gain would clearly be even more spectacular for a phrase 'LETTER'.

The second form of exceptional case is exemplified by names. For efficiency, these must have a special purpose procedure for their recognition. This being so, it is very convenient also at the same point to enter the name into a table of names encountered in the program so far and assign an identifying number to it instead of passing the alphanumeric characters forward any further.

The exceptional cases are called 'built-in' phrases, that is, phrases which are built-into the compiler in the form of program statements and not into the phrase structure tables.

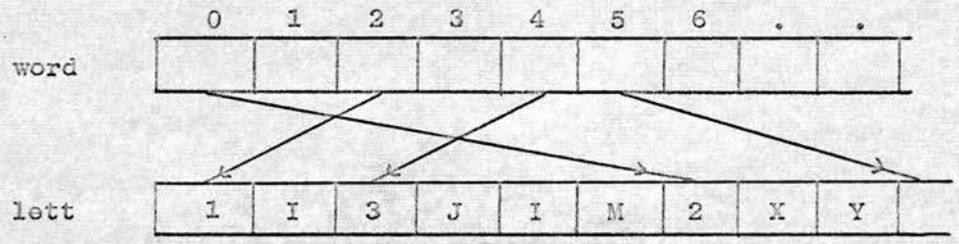
The built-in phrases in use are the following :

1. NAME
2. CONSTANT
3. TEXT
4. CAPTION TEXT
5. SET MARKER 1
6. SET MARKER 2

1. NAME

The function of built-in phrase NAME is to recognise names in all the contexts in which they appear in source text and to enter them into a table of names. A unique identifying number is attached to each name as it is recognised and it is this number which is placed in the analysis record array in place of an alternative number. Repetitions of any name already in the table cause the existing identification number for that name to be supplied. This is irrespective of any redeclaration of the same name in inner blocks or routines of the program. This discrimination is the function of the semantic phase of the compiler.

The dictionary system for storing the names has to be two-way. In other words, from a name an identifying number must be produced and from an identifying number the alphanumeric characters of that name must be available - this latter for the purpose of producing legible program maps and diagnostics. The system employed uses two arrays therefore :



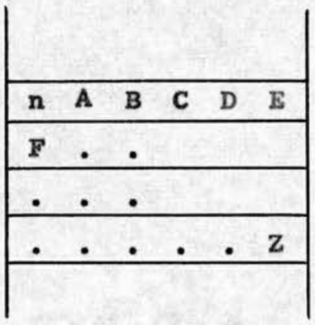
The larger array 'lett' contains the characters of the names, each prefaced by the number of its characters. The identifying number for the name is given by the index of the location in the array 'word' which contains the pointer to the position of the actual characters in 'lett'. Thus in the diagram name number 0 is XY , number 2 is I and number 4 is JIM . The array 'lett' can be used as a stack, filling in names from the bottom and progressing up the array. (There is no need ever to remove names from the dictionary. Although the range of activation of some names may be small, it is the common practice to declare the vast majority of names at the outermost level of block structure which implies that they are active throughout the program. The extra complication involved in removing names from the dictionary and dealing with the consequent holes was not felt to be warranted). It is most efficient not to use the array word as a stack however. If this is done, when a name is encountered and an attempt is made to match it with one of the names already in the dictionary (in order to get the unique identifying number) a linear scan through the existing names has to be made, with the consequent inefficiencies of that process. Instead, a ´hash´ system is better. In this system, an ´approximate´ identifying number is calculated in some fairly arbitrary manner from the characters of the name. The only criterion in the choice of method of calculation is that the numbers produced from a collection of names should span the indexes of the array word in an even a way as possible - trying to avoid grouping. Since the choice of names is a particularly personal characteristic of programmers, no algorithm can be expected to give perfect results for all programs. The approximate number is used as a starting position in word , from which a cyclic scan can begin. Before compiling begins, all the locations of word are given a recognisable tag to indicate ´not yet in use´. Upon scanning from the starting position, if a position ´not yet in use´ is encountered this implies that the new name is not yet in the dictionary and can therefore be inserted at this point. If a position is in use, the name there can be compared with the new one for equality. If it matches there is no insertion to be done and the identifying number is that of the existing name. If there is no match, the next (cyclic) position in the array is considered. Whenever the complete cycle is performed without finding either a match for the new name or a ´not yet in use´ tag, the dictionary is already full.

2. CONSTANT

This coding recognises all forms of constant, integer types and string types. E.g. 10, 2@3, ´ASTRA´. A table of the values so recognised is also built up and this table later forms the first part of the running programs data area. (Switch vectors and caption texts are also stored in the same table). Such is the design of KDF9, that small integer constants (less than 2**15) can be incorporated as ´immediate operands´ within the object code. Constants are therefore divided into three classes :

1. Small integers
2. String constants
3. Larger integers

Two positions in the analysis record are used to identify constants. the first gives the type of constant, 1, 2, or 3 as above, the second depending on the value of the first. For small integers, class 1, the number itself is put in the second position. For string constants and large integers, the position in the table of constants is given. In the table, large integers occupy a single location but string constants may be of any length. The layout is :



where six characters are packed per word with the number of characters in the first position. As mentioned above, it might be quicker for statements such as :

<u>if</u> s = 'ASTRA' <u>then</u> ->1

if the string constant were stored in the list form required for comparison with s, but since there are other situations such as :

s = 'ASTRA'

when a new copy of the value 'ASTRA' has to be made for the assignment, it was considered better policy to minimise the space occupied by packing in a consistent way.

3. TEXT

The only practical way to ignore the text of a comment statement without modifying the recognition algorithm is to have special coding i.e. a built-in phrase which skips along the text until a separator (; or ~) is found.

4. CAPTION TEXT

As with built-in phrase TEXT, this also skips along the text until a separator is found. It also has the function, however, of storing the characters in the table which also contains constants so that the captions are available for output during the running of the object program. The format of storage is the same as that for string constants, and the position in the table is placed in the analysis record.

5. & 6. SET MARKERS 1 & 2

These two built-in phrases are exceptional in that they perform no recognising function and do not leave anything in the analysis record. They are purely to ease the semantic processing in two situations. For example, the first alternative of SS is :

[UI][SET MARKER 1][REST OF SS]

This is designed to cater for the situations exemplified by :

x = 0

x = 0 <u>if</u> x = y

The [UI] should only be recognised once i.e. backtracking should be avoided and to this end the REST OF SS is defined as :

[<u>if</u> or <u>unless</u>][COND][S] , 0 ;

If the statement is conditional, code to evaluate the condition must be planted before that for the UI . The position of that part of the analysis record relating to the COND must therefore be located first and indeed the fact that the statement is conditional must be determined. This could be done by scanning the analysis record since the phrase structure for UI is known, but it is much quicker to have the position directly marked in some way. This is the function of SET MARKER 1. When this phrase is executed, the current position in the analysis record i.e. after that part relating to UI, is set into a global variable (marker1) which can be inspected by the coding for the semantic phase. SET MARKER 2 is used in a very similar situation, the first alternative of UI being

[NAME][APP][SET MARKER 2][REST OF UI]

where

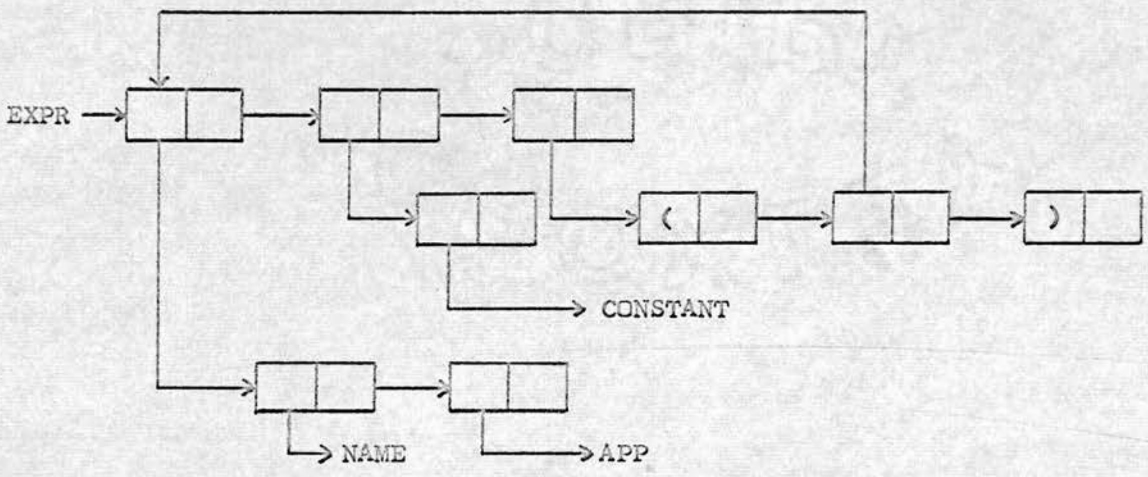
P[REST OF UI] = ′=′[EXPR],0;

APP stands for ′Actual Parameter Part′. In other words, the distinction is made between routine calls and assignment statements.

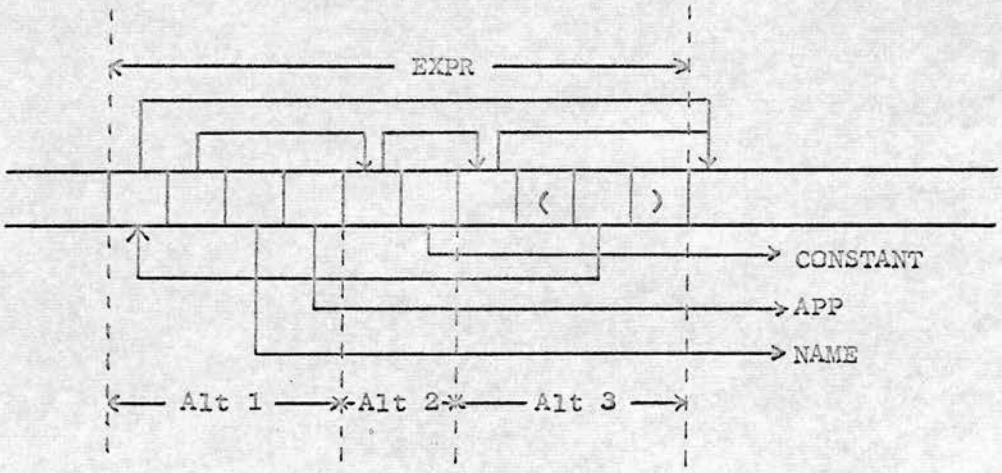
These were the only two situations in which this exceptional method was felt to be useful.

An obvious way in which to store the tables of phrase structure is in the form of a list structure. Take a defintion of EXPR as an example :

P[EXPR] = [NAME][APP] , [CONSTANT] , ´(´[EXPR]´)´ ;

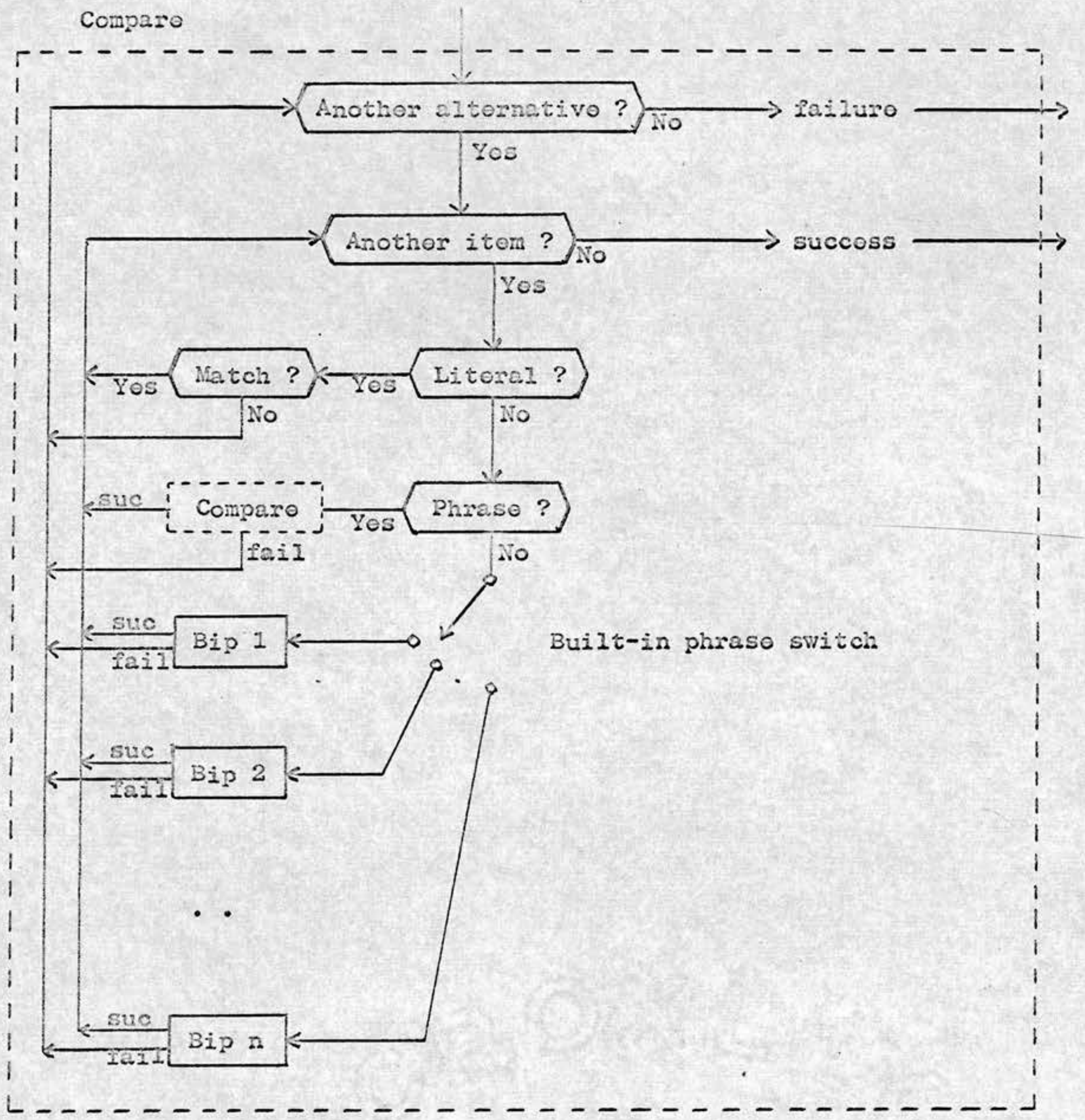


The complete tables would be just one list structure, that for phrase SS with all subsidiary phrases such as EXPR sublists of the main list for SS. This method of representation was experimented with but a form of representation using a linear array was found to give faster operation and was therefore adopted. The equivalent linear representation to that illustrated above would be :



The locations from which pointers emerge contain indexes of positions in the same array indicating where that phrase is defined. By judicious choice of bounds for the array, literals and built-in phrases can be distinguished from true phrases by the range in which the value lies.

The analysis algorithm is hardly affected by the addition of built-in phrases.



The compilers, both Atlas Autocode and ASTRA, are written in the same language that they compile. By this means, all the advantages of high level languages were available in writing the compilers and in almost all respects they can be treated as ordinary programs. When a new version of the compiler is required, the suitably modified program which is the new compiler is compiled by the existing compiler to produce the object code for the new compiler.

After a statement has been recognised as valid syntactically, the resulting analysis record is passed on for semantic processing and code generation. The layout of this second phase (which is a routine named cSS for ´compile Source Statement´) consists of sections of coding each of which deals with one of the alternative forms of source statement, together with a number of routines which process commonly occurring objects such as expressions.

The first number in the analysis record indicates the type of source statement to be dealt with i.e. which alternative of phrase SS was matched. This is used immediately to switch to the appropriate section of coding :

```
         routine cSS
         -> sw(A(1))
sw(1):   comment [UI][SET MARKER 1][REST OF UI]
         . . . .
         return
sw(2):   comment cycle . . .
         . . . .
         return
         . . . .
         end
```

The array named A contains the analysis record. The most important subsidiary routines supporting cSS are named cSEXP, cNAME, cUI, cCOND which deal with those objects which are defined by the phrases EXPR, NAME, UI, COND. A global variable named p is used as a pointer to the analysis record array A and, by convention, whenever a routine such as one of these four is called the value in p should indicate the position relating to that phrase. When the routine is left the value in p should indicate the position immediately following the entries relating to that phrase. A global variable is used in preference to a parameter purely for the sake of efficiency.

Two examples to demonstrate the scheme of processing are now presented. They are slightly simplified from actual compiler versions.

Example 1

Consider a declarative statement :

<u>string</u> r, s, t

We may suppose that the alternative of SS relating to this type of statement is :

[TYPE][NAME][REST OF NAME LIST]

where

```
P[TYPE] = 'integer' , 'string' ;
P[REST OF NAME LIST]=',' [NAME][REST OF NAME LIST],0;
```

NAME is the built-in phrase which leaves an identifying number in the analysis record for each name. If this alternative is the sixth of SS, say, the analysis record corresponding to the statement above would be :

```
6  2  id(r)   1   id(s)   1   id(t)   2
```

where 'id' stands for 'identification number of'.

The purpose of the section which deals with this type of statement is to store information relating to the name i.e. its 'tags' for future reference and to assign a unique 'stack relative address' to each. The appropriate section of coding would be :

```
sw(6):  comment scalar declarations
        type = A(2)
        p = 2
        comment n = value of next stack relative
        comment       address to be assigned
61:     test name set twice(A(p))
        store tags(type,n,A(p))
        n = n+1
        p = p+2
        if A(p) = 1 then ->61
        return
```

The conditional statement tests the alternative number of each manifestation of REST OF NAME LIST until the list of names is exhausted and the tags have been stored for each of them. The two subroutines 'test name set twice' and 'store tags' are made such since the same action is required when dealing with other types of statement e.g. arrays, routines etc. in other sections of coding.

Example 2

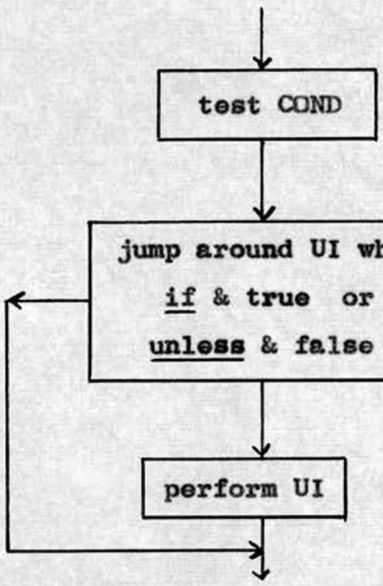
Conditional statements i.e. those corresponding to the fifth alternative of SS :

[iu][COND]´then´[UI]

where

P[iu] = ´if´ , ´unless´ ;

The format of the object code required is :



An analysis record for this type of statement is of the form :

$$5\begin{pmatrix}1\ \underline{\text{if}}\\2\ \underline{\text{unless}}\end{pmatrix}\underbrace{.\ .\ .\ .\ .\ .\ .\ .\ .}_{\text{relating to COND}}\quad\underbrace{.\ .\ .\ .\ .\ .\ .\ .\ .}_{\text{relating to UI}}$$

The section of coding to deal with this illustrates the way in which routines are used to perform major functions :

```
sw(5):  comment Conditional statements
        p = 3
        cCOND            ;! i.e. dump code to test condition
        plant jump(A(2))
        cUI              ;! i.e. dump code to perform UI
        set jump label
        return
```

By observing the conventions concerning the global pointer p, having set p correctly for entry to cCOND, cCOND itself should leave p correctly positioned for entry to cUI (since the literal then takes no space in the analysis record).

cSEXP, cNAME, cCOND

Full use is made of the recursive structure of the language in which the compiler is being written. Since the phrase structure makes extensive use of recursive definitions, it is natural to make the processing routines recursive in the same way. This is not slavishly followed, however, as the processing of the recursive phrase REST OF NAME LIST above shows. It greatly simplifies a large number of situations however. Take phrase COND as an example. COND is defined in terms of Simple Conditions, SC, which is itself defined in terms of expressions and COND. cCOND therefore has a subroutine cSC which is called upon as the main coding separates off the simple conditions of the total condition. cSC calls on cSEXP to compile expressions for it as they are encountered in [EXPR][COMP][EXPR] etc., and calls on cCOND when the simple condition is the third alternative ′(′[COND]′)′.

Expressions provide another illustration of this technique of gradually breaking down the complex object and dealing with each simple part one at a time. Consider the expression :

A(p+1) - B(p*q,r)

cSEXP will be called to deal with this whole expression. At this level it is broken down into operands. Since both operands here happen to benames, two calls on cNAME will be made. cNAME deals with the whole operand incuding actual parameter part and so for A(p+1) will call on cSEXP back again to compile p+1 and for B(p*q,r) will call on cSEXP twice more for p*q and for r. Again, since these expressions involve names cSEXP will call upon cNAME several more times. Eventually the most basic constituents will have been located and dealt with.

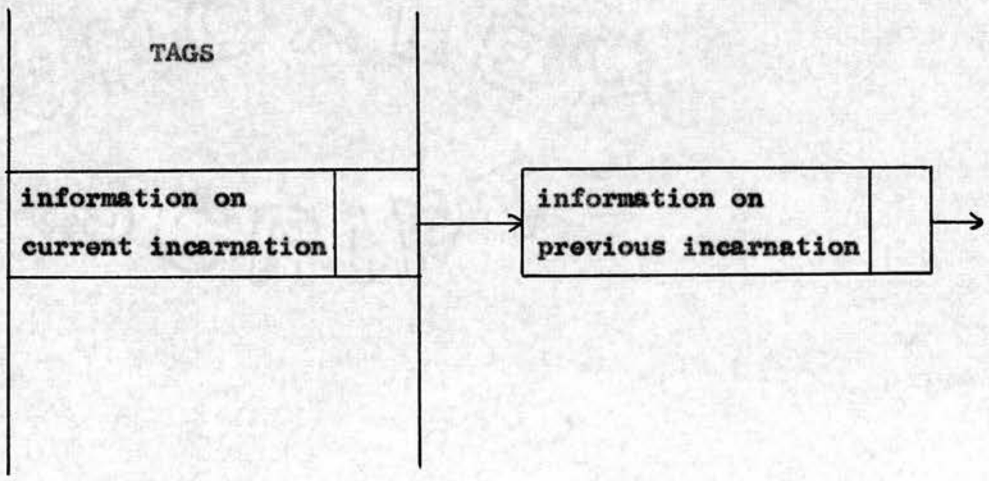
This method of processing implies that care must be taken when designing the routines in respect of overwriting contents of variables by recursive calls. In other words, local variables must be used rather than global variables for sensitive information. This amount of care never became troublesome during the writing of the compilers.

The function of the routines cSEXP and cCOND is quite clear, but that of cNAME justifies some amplification. The only use of it so far implied is in the compiling of names which appear in expressions. This is far from the case however. It was found in dealing with names, that even such apparently diverse contexts such as routine calls and assignment statements called for very similar processing of the name. For this reason, cNAME was given a wide variety of functions, controlled by a parameter on call.

cNAME(0) :	treat name as a routine call
cNAME(1) :	assign a value to name
cNAME(2) :	pick up a value from name
cNAME(3) :	get machine address of name

By this means, every appearance of a name in the program results in a call on cNAME, with the parameter varying with contexts.

The identifying numbers of names assigned by built-in phrase NAME lie in the range 0 to 255 (256 different allowed names has been found to be sufficient). Since names play such a predominant part in the language in their various guises, the most widely used storage array in the compiler is that which stores the information concerning the names. This array is called TAGS and ranges from 0 to 255, so that the information concerning a particular name is stored in the position of TAGS indexed by its identification number. When the same name is redeclared at an inner level of the block structure, the information in the TAGS position relating to the original declaration must be preserved somewhere else, since the same identifying number will be supplied regardless of the name's redeclaration. The preserved information need not be immediately accessible because all use of the name now refers to the new declaration - a basic feature of block-structured languages. The preserved information has to be restored to TAGS when the block in which the name was redeclared is left, as the original declaration then becomes valid again. This preserving of information is accomplished by a list processing scheme which uses each position in the TAGS array as the head cell of a pushdown list :


TAGS
information on current incarnation
information on previous incarnation

When a name is redeclared, a cell is taken from an Available Space List and the old TAGS information copied into it. The information relating to the new declaration is placed in the TAGS array together with a link to the pushed-down cell. The same type of cell and list processing scheme is used in a number of contexts throughout the compiler, some of which will be mentioned later.

The information which defines the current usage of a name consists of four items which are packed together for storage in TAGS. They are :

1. type
2. level (of declaration)
3. dimension
4. address

8 bits	4 bits	4 bits	16 bits	16 bits
type	level	dim.	address	link

1. type

Each type of use of a name is assigned a 'type number' to distinguish it. These are :

0	:	name not set
1	:	string
2	:	integer
3	:	string array (name)
4	:	integer array (name)
5	:	switch
6	:	routine
7	:	stringfn
8	:	integer fn
9	:	string map
10	:	integer map
11	:	string name
12	:	integer name
13	:	-
14	:	addr

The array and array name types are assigned the same type number since the way they are dealt with is identical.

2. level

This field holds the textual level at which the name was declared. This is quite distinct from the recursive level at which the running object program may declare the name. Fifteen levels are therefore quite sufficient as provided for by the 4-bit field.

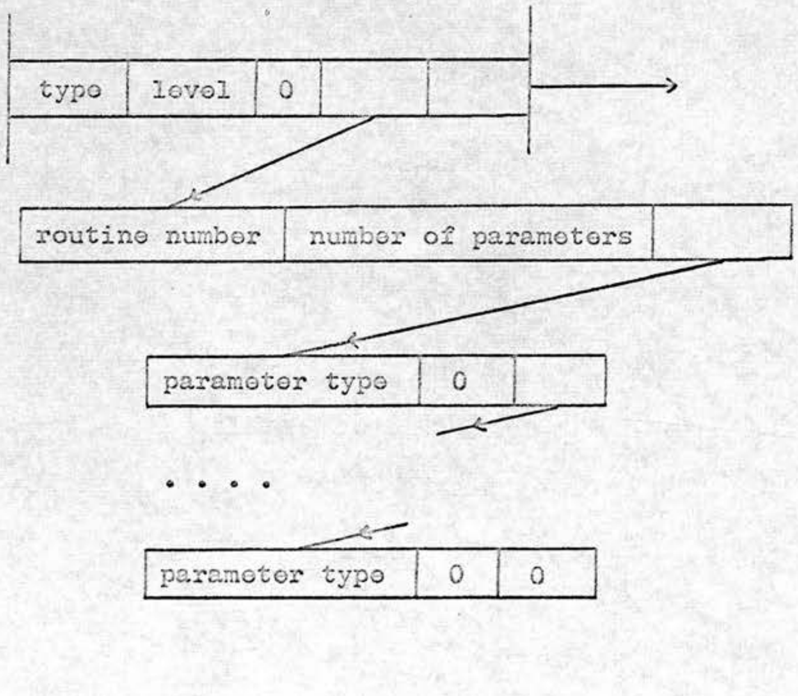
3. dimension

When the name is an array, this field holds the dimensionality. For scalar variables it holds 0.

4. address

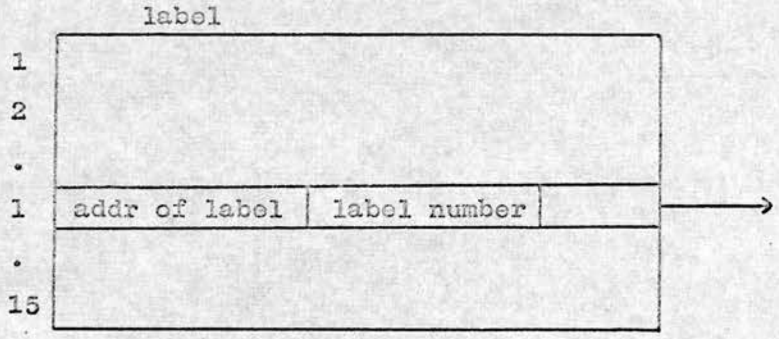
Each scalar variable and array is assigned a location of storage on the run-time stack. These are addressed relative to a position on the stack which represents the start of the locations for variables declared in that block. This relative address for each variable is stored in the address field.

In the cases of routines, functions and maps, which are not assigned locations on the run-time stack, the address field is used as a further list link. The sublist of cells contains the information on the parameters for the routine, function or map. The format is :



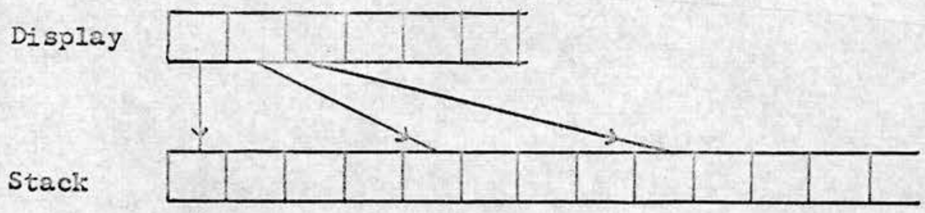
A list processing scheme which packs the information closely, as this does, leads to slightly reduced speed of operation because of the amount of unpacking to be done, but has the essential advantage of minimising the total amount of storage used, which was at a premium on KDF9.

The same list processing scheme was used for other purposes such as dealing with labels, jumps, and cycle statements. The headcells on these occasions relate to the textual level in the program and are therefore arrays declared from 1 to 15. Whenever a label is encountered, information relating to it is pushed down onto the list corresponding to the current textual level. A similar action is taken for jump instructions. The two corresponding lists are matched, in order to relate the references and the lists popped up when the end statement of the current level is encountered. cycle and repeat statements are also local to the same textual level as are labels and jumps. In this case, however, a cell is pushed down whenever a cycle is found and the top cell popped up when a repeat is found. This deals with the nesting of cycles and repeats. As a matter of convenience, there is also a pushdown list for the names declared at any textual level. The appropriate list is popped up at the end statement in order to undeclare the names i.e. pop up the TAGS lists. This is more efficient than scanning the TAGS array for variables declared at the current level. For example, the label list :



Stack Management System

For a block-structured language such as Atlas Autocode or ASTRA, the scheme proposed by Dijkstra for addressing variables on the run-time stack is very convenient. As has been mentioned, each variable is assigned a relative position to a stack pointer for that textual level of the program. The array of these stack pointers, one for each textual level, was called a 'Display' :



In the KDF9 implementations of Atlas Autocode and ASTRA, the display is stored in the modifier parts of the Q-stores. This enables all the variables on the stack to be addressed directly by using the relative address assigned to the variable modified by the contents of the Q-store (modifier part) corresponding to the textual level at which the variable was declared. This holds true even in situations where the variable becomes global i.e. declared at an outer level from the textual level at which the program is currently running. In KDF9 machine code terms :

E(relative address)M(textual level)

For example, the third variable declared at textual level two, could be accessed by :

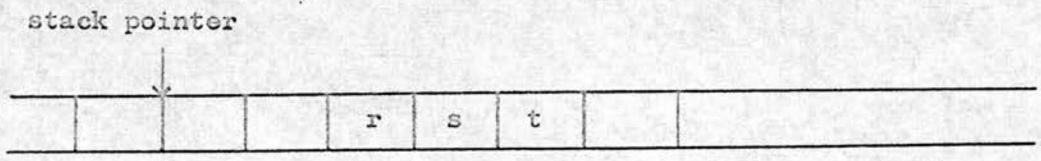
E3M2

(to be strictly accurate E4M2 as is now explained).

At the start of the storage on the stack for each textual level, two locations are set aside for special purposes which are explained subsequently. The relative addresses therefore start at 2 for the first variable declared. The declaration :

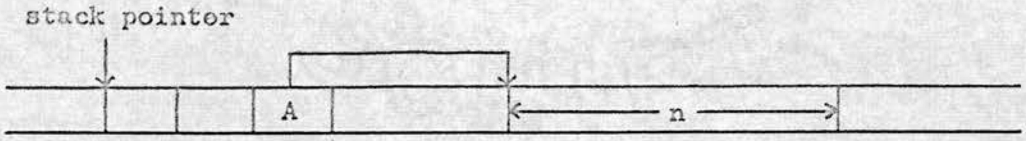
<u>string</u> r, s, t

would set aside storage :

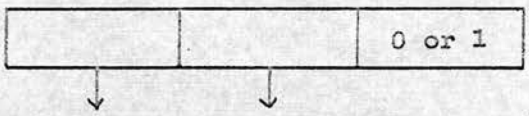


For arrays, one location is assigned and given a relative address. Since the bounds of the array can only be determined dynamically, storage for the array cannot be set aside at compile time. This is therefore done at run-time using positions on the stack beyond those assigned at compile-time and the single location is then set to the address of the array storage position :

<u>stringarray</u> A(1:n)



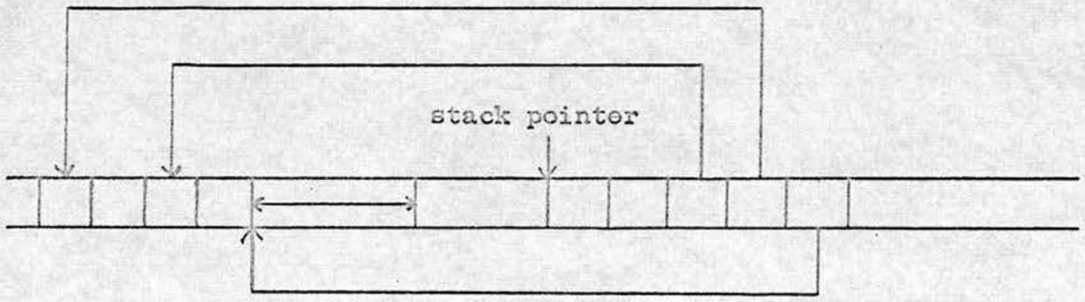
In the case of string variables, these locations contain the pointers to the list structure which represents the value of the string :



For integers, the value itself is of course stored.

The two basic types of parameter in Atlas Autocode and ASTRA are the value type and the name type. Value type parameters are dealt with just as ordinary local variables to the routine or function, the only difference being that they are preassigned with the value of the actual parameter before entry. name type parameters have to be treated as indirect references. Fortunately, the indirect references are defined to remain fixed after entry, unlike ALGOL 60 where the references can change dynamically :

<u>routine</u> RT (<u>stringname</u> u, v, <u>stringarrayname</u> A)



As can be seen, there is no difference between the storage accessing mechanism for arrays and arraynames. Hence there being no distinction in the type numbers given above.

The two locations set aside at the start of each textual level storage area are used during block and routine entry and exit. The first (arbitrarily) is used to hold the previous value of the display pointer for that textual level. It is filled on entry to a block or routine and restored to the display on exit. By this means, the display is maintained in a permanently valid state. The second location is used only for routines, functions and maps and stores the return address. In a stylised form, these entry and exit operations can be described by the following sequences where 'STP' is the pointer to the current top of STACK.

```
comment block entry
STACK(STP) = DISPLAY( textual level of new block )
DISPLAY( textual level of new block ) = STP
STP = STP + ( fixed storage allocation for new block )

comment block exit
STP = DISPLAY( textual level of current block )
DISPLAY( textual level of current block ) = STACK(STP)

comment routine entry
STACK(STP) = DISPLAY( textual level of new routine )
DISPLAY( textual level of new routine ) = STP
STACK(STP+1) = ( return address )
STP = STP + ( fixed storage allocation for new routine )

comment routine exit
STP = DISPLAY( textual level of current routine )
DISPLAY( textual level of current routine ) = STACK(STP)
return to STACK(STP+1)
```

8. COMPILING ALGORITHMS FOR THE ASTRA COMPILER

The phrase structure for the ASTRA compiler is :

```
P[+']=          '+','-',0;
P[OPERAND]=     [NAME][APP][PART],[CONST],'('[+'][OPERAND]
                [RESTOFEXPR]')','!'[+'][OPERAND][RESTOFEXPR]'!',
                ''''[NAME][APP][PART]'''','-'[PART],'_';
P[RESTOFEXPR]=[OP][OPERAND][RESTOFEXPR],0;
P[APP]=         '('[+'][OPERAND][RESTOFEXPR][RESTOFEXPR-LIST]')',0;
P[RESTOFEXPR-LIST]=','[+'][OPERAND][RESTOFEXPR][RESTOFEXPR-LIST],0;
P[OP]=          '.','+','-','**','*','/','_';
P[,']=          ',',0;
P[%IU]=         '%IF','%UNLESS';
P[TYPE]=        '%INTEGER','%STRING';
P[RT]=          '%ROUTINE','%STRINGFN','%INTEGERFN',
                '%STRINGMAP','%INTEGERMAP';
P[FP-DELIM]= [RT],'%INTEGERARRAYNAME','%INTEGERNAME','%INTEGER',
                '%STRINGARRAYNAME','%STRINGNAME','%STRING','%ADDR';
P[FPP]=         '('[FP-DELIM][NAME][RESTOFNAMELIST][RESTOFFP-LIST]')',0;
P[RESTOFFP-LIST]=[,'][FP-DELIM][NAME][RESTOFNAMELIST][RESTOFFP-LIST],0;
P[RESTOFNAMELIST]=','[NAME][RESTOFNAMELIST],0;
P[SC]=          [+'][OPERAND][RESTOFEXPR][COMP][+'][OPERAND][RESTOFEXPR]
                [RESTOFSC],'('[SC][RESTOFCOND]')';
P[RESTOFSC]= [COMP][+'][OPERAND][RESTOFEXPR],0;
P[RESTOFCOND]='%AND'[SC][RESTOFAND-C],'%OR'[SC][RESTOFOR-C],0;
P[RESTOFAND-C]='%AND'[SC][RESTOFAND-C],0;
P[RESTOFOR-C]='%OR'[SC][RESTOFOR-C],0;
P[RESTOFUI]= [ASSOP][+'][OPERAND][RESTOFEXPR],0;
P[%SPEC']=      '%SPEC',0;
P[RESTOFBP-LIST]=','[+'][OPERAND][RESTOFEXPR]':'[+'][OPERAND]
                [RESTOFEXPR][RESTOFBP-LIST],0;
P[RESTOFARRAYLIST]=','[NAME][RESTOFNAMELIST]'('[+'][OPERAND]
                [RESTOFEXPR]':'[+'][OPERAND][RESTOFEXPR][RESTOFBP-LIST]')'
                [RESTOFARRAYLIST],0;
```

```
P[RESTOFSWITCHLIST]=',''[NAME][RESTOFNAMELIST]'('[+'][CONST]':'
            [+'][CONST]')'[RESTOFSWITCHLIST],0;
P[COMP]=    '=','>=','>','#','<=','<','->','¬=';
P[RESTOFSS1]=[S],[%IU][SC][RESTOFCOND][S];
P[PART]=    '['[+'][OPERAND][RESTOFEXPR][RESTOFPART]']',0;
P[RESTOFPART]=':'[+'][OPERAND][RESTOFEXPR],':*',0;
P[ASSOP]=   '=','->','<-';
P[UI]=      [NAME][APP][PART][SETMARKER1][RESTOFUI],
            '->'[N],
            '%CAPTION'[CAPTIONTEXT],
            '%RETURN',
            '%RESULT='[+'][OPERAND][RESTOFEXPR],
            '%STOP',
            '->'[NAME]'('[+'][OPERAND][RESTOFEXPR]')';
P[SS]=      [UI][SETMARKER2][RESTOFSS1],
            '%CYCLE'[NAME][APP]'='[+'][OPERAND][RESTOFEXPR]','[+']
                [OPERAND][RESTOFEXPR]','[+'][OPERAND][RESTOFEXPR][S],
            '%REPEAT'[S],
            [N]':',
            [%IU][SC][RESTOFCOND]'%THEN'[UI][S],
            '!'[TEXT],
            [TYPE][NAME][RESTOFNAMELIST][S],
            '%END'[S],
            [RT][%SPEC'][NAME][FPP][S],
            '%SPEC'[NAME][FPP][S],
            '%COMMENT'[TEXT],
            [TYPE]'%ARRAY'[NAME][RESTOFNAMELIST]'('[+'][OPERAND]
                [RESTOFEXPR]':'[+'][OPERAND][RESTOFEXPR]
                [RESTOFBP-LIST]')'[RESTOFARRAYLIST][S],
            '***A'[S],
            '%BEGIN'[S],
            '%ENDOFPROGRAM',
            '%ASL='[N][S],
            [NAME]'('[+'][CONST]'):',
            '%SWITCH'[NAME][RESTOFNAMELIST]'('[+'][CONST]':'
                [+'][CONST]')'[RESTOFSWITCHLIST][S],
            '%LIST'[S],
            '%ENDOFLIST'[S];
```

It will be noted that where the avoidance of backtracking necessitates the use of [REST OF . .] phrases, instead of defining a phrase such as :

P[EXPR] = [OPD][REST OF EXPR] ;

with only one alternative, the components of that alternative are inserted at all the points where the phrase is required. This has the effect of slightly increasing the size of the syntax tables but makes analysis quicker and avoids an effectively redundant entry in the analysis record. An example of this can be found in the second alternative of SS :

'<u>cycle</u>'[NAME][APP]'='[+'][OPERAND][REST OF EXPR] etc.

The phrase [+'] allows expressions to be prefaced with a sign.

The main changes from the syntax of Atlas Autocode can be briefly summarised.

1. The addition of a [PART] clause after occurrences of [NAME][APP]. Phrase [PART] defines the string indexing facilities e.g.

r[1] , A(2)[3:5] , s[2:*]

2. Three further alternatives to phrase OPERAND :

''''[NAME][APP][PART]'''' ,

'-'[PART] ,

which are used in resolution statements, and

'-' ;

which is the null symbol used in string expressions.

In a number of instances throughout the phrase structure, the syntax as defined allows invalid components to pass through without causing a syntax fault to be monitored. For example :

r = -[1]

would be passed through as valid. This policy was quite deliberate and allows the total phrase structure to be uniform with a minimum of exceptional cases which might increase the time of recognition. Such invalid statements as do pass through are easily detected and monitored by the semantic phase, routine cSS, of the compiler. In fact, the boundary between syntax and semantics is considerably blurred; more or less could be incorporated in the syntax tables to reduce or to increase the amount to be done by the semantic phase. The actual boundary chosen is the one which is most convenient, where there is no easily measurable or distinguishable effect on efficiency.

3. An extra operator, '.' , used in both string expressions and resolutions.

4. Replacement of '<u>real</u>' by '<u>string</u>' in all occurrences.

It was unfortunately a matter of practical necessity to remove some Atlas Autocode facilities in order to reduce the size of the compiler so that room could be found for the new string facilities. The KDF9 in question had only 16384 words of storage, almost all of which was used by the existing Atlas Autocode compiler, leaving little room for further expansion. Short of revising the whole compiling system, say using a two-pass system instead of the one-pass system in use, some major feature or features had to be omitted. The choice fell on real variables and real arithmetic facilities. This had the convenient side-effect of enabling string types to replace real types quite consistently throughout the compiler, thereby minimising the changes to it. There is, of course, no incompatibility between real and string variables coexisting, and future ASTRA implementations should contain integer, real and string types, and perhaps others also such as complex, storage space permitting.

5. The introduction of phrase ASSOP i.e. Assignment Operator, to allow '=', '->', and '<-' any of which can appear where the assignment '=' of Atlas Autocode was allowed.

6. The addition of '->' to the comparators in order to allow resolutions as parts of conditions.

7. A new alternative to SS to allow the length of available space to be set :

'<u>asl</u>' '=' [N] [S] ,

STRING EXPRESSIONS

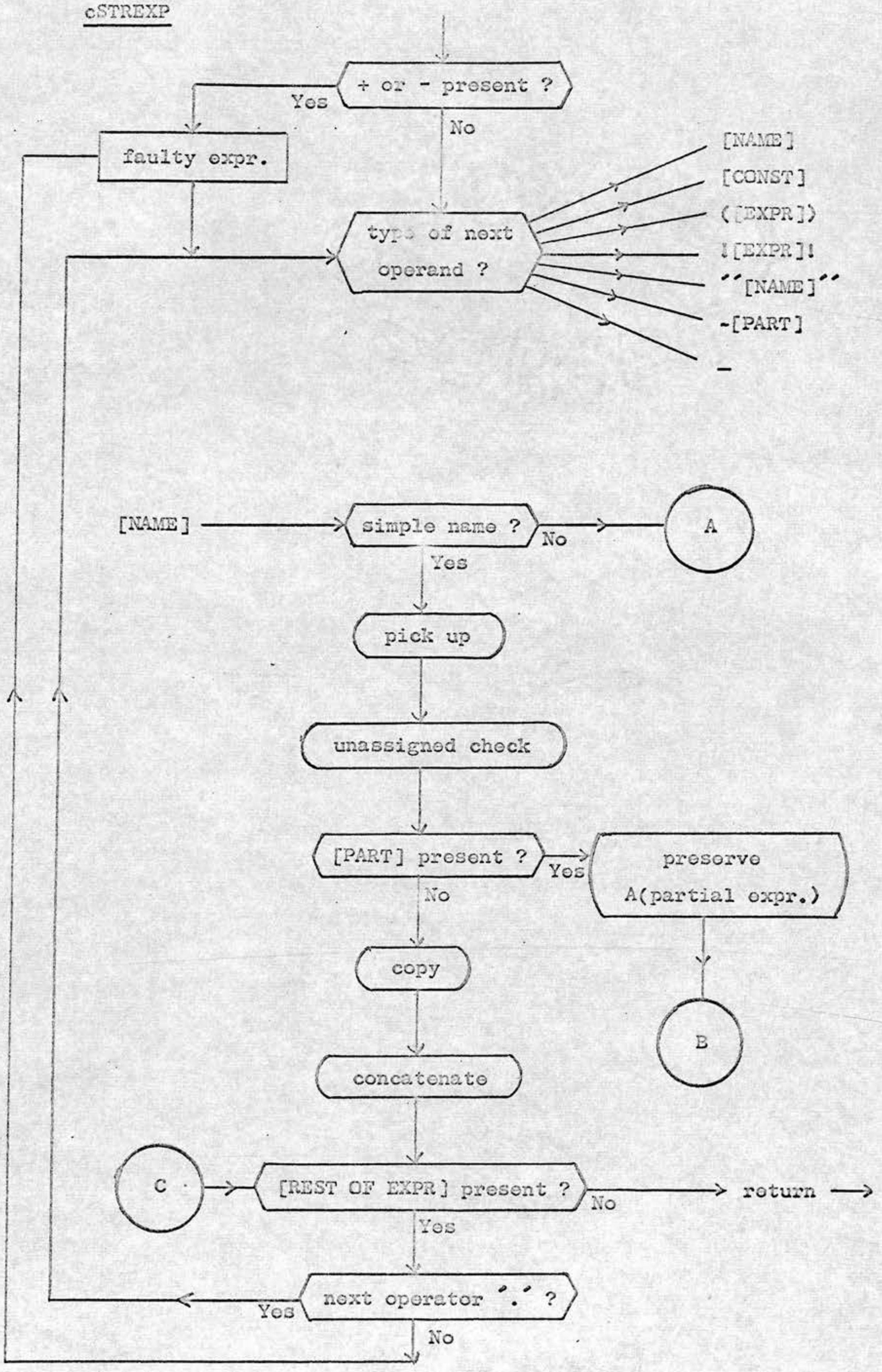
As was described in a previous chapter, the code to be planted for a string expression takes the following general form :

Form representation of operand
Form representation of operand
Concatenate two operands
Form representation of operand
Concatenate two operands
etc.

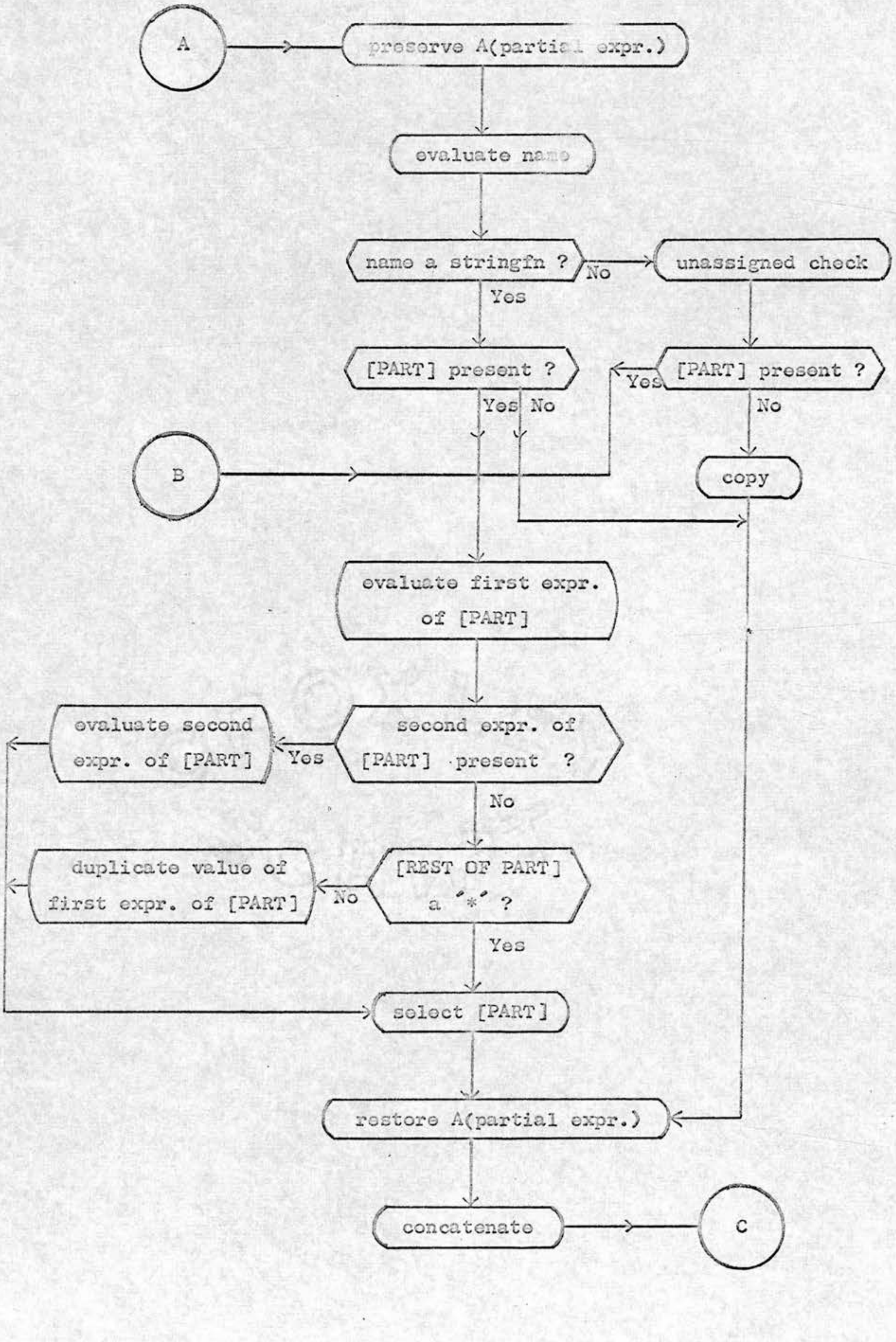
String expressions are processed by a routine named cSTREXP which works on that part of the analysis record corresponding to an expression. On entry, global variable p points to a phrase [+'] which always precedes [OPERAND][REST OF EXPR] (but which is only used for arithmetic expressions). Just as routine cSS consists of sections of coding, branched to on a switch, for each alternative of SS, so routine cSTREXP consists of sections of coding for each type of operand. The major section is that which deals with [NAME][APP][PART], as might be expected. For those forms of operand which are invalid in string expressions but which the syntax allows through the section just consists of a fault monitor. For the type of operand which represents a substring, the section consists of a recursive call on routine cSTREXP.

In the following coding, labels such as 14P refer to Private labels i.e. labels within the 'permanent material' available at run-time with all compiled programs. This contains such things as Input-output routines, basic processes for string expressions and resolutions, and run-time fault mnitors.

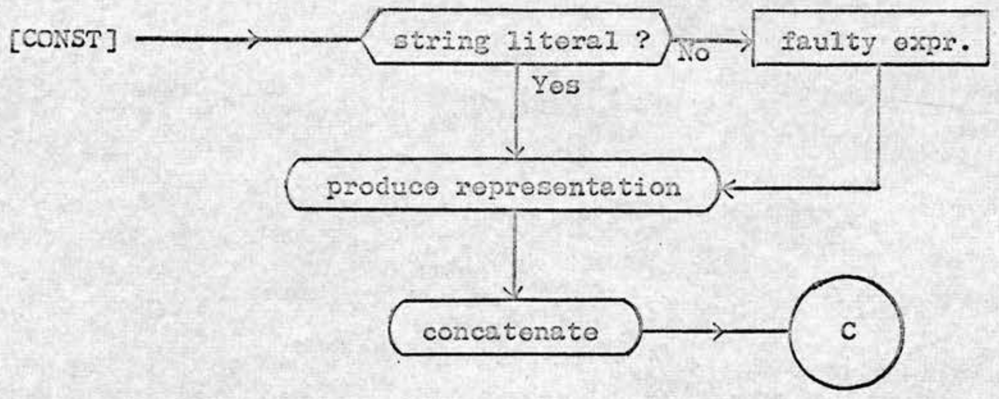
cSTREXP

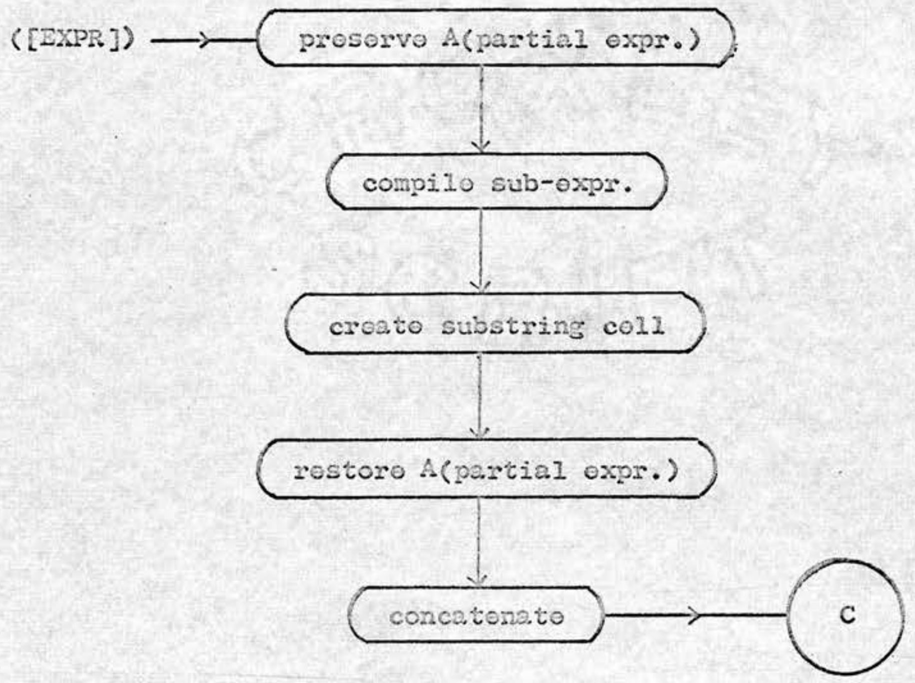


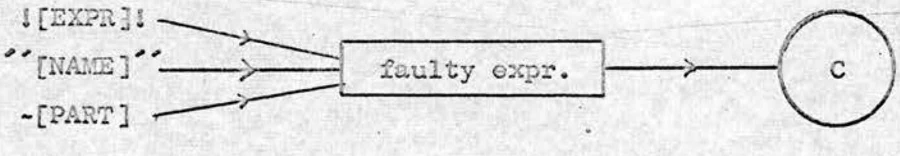
+ or - present ?
Yes
No
faulty expr.
type of next operand ?
[NAME]
[CONST]
([EXPR])
![EXPR]!
''[NAME]''
-[PART]
–
[NAME]
simple name ?
No
A
Yes
pick up
unassigned check
[PART] present ?
Yes
preserve A(partial expr.)
B
No
copy
concatenate
C
[REST OF EXPR] present ?
No
return
Yes
next operator '.' ?
Yes
No

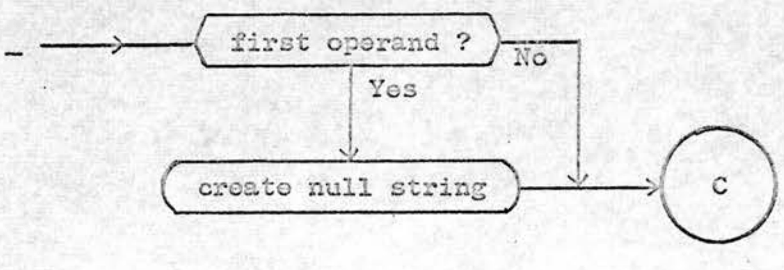


A
preserve A(partial expr.)
evaluate name
name a stringfn ?
No
unassigned check
Yes
[PART] present ?
Yes
[PART] present ?
Yes
No
No
B
copy
evaluate first expr. of [PART]
evaluate second expr. of [PART]
Yes
second expr. of [PART] present ?
No
duplicate value of first expr. of [PART]
No
[REST OF PART] a '*' ?
Yes
select [PART]
restore A(partial expr.)
concatenate
C


[CONST]
string literal ?
No
faulty expr.
Yes
produce representation
concatenate
C


([EXPR])
preserve A(partial expr.)
compile sub-expr.
create substring cell
restore A(partial expr.)
concatenate
C


![EXPR]!
''[NAME]''
-[PART]
faulty expr.
C


_
first operand ?
No
Yes
create null string
C

```
     routine cSTREXP
     integer typep, n, m
     switch S(1:7)              ;! for the 7 alternative forms of OPERAND
     n=0                        ;! operand count
     fault(100) unless A(p)=3   ;! invalid expression if + or - present
12:  n=n+1                      ;! count next operand
     p=p+2                      ;! p on position of OPERAND+1 in anal.rec.
     ->S(A(p-1))                ;! switch on type of operand

S(5):! ''NAME''
     fault(100)                 ;! ''NAME'' invalid in string exprs.

S(1):! NAME
     ->1 if A(p+1)=1            ;! jump if actual parameter part present
     copytag(A(p))              ;! get tags of this name
     ->1 unless type=1 or type=11  ;! jump unless simply string
                                   ! or stringname type
     plant(EkMi)                ;! plant code for pick-up from location
                                ! for this variable
     if type=1 then ->2         ;! jump if string type
     plant(=M10)                ;! indirect pick-up for stringname type
     plant(MOM10)
2:   plant(DUP)
     plant(J14P=Z)              ;! fault monitor if variable unassigned
     p=p+2                      ;! p on PART
     ->3 if A(p)=1              ;! jump if PART present
     plant(JS103P)              ;! basic process to copy string
     p=p+2                      ;! p on REST OF EXPR+1
     ->4
1:   if n>1 then plant(=MOM12Q)  ;! preserve A(partial string expr)
     cNAME(2)                   ;! pick-up for complex name
     fault(100) unless type=1 or type=7  ;! fault monitor if name
                                          ! not string type
     ->13 if type=7             ;! jump for stringfn
     plant(DUP)
     plant(J14P=Z)              ;! fault monitor if variable unassigned
```

```
13:  ->5 if A(p)=1              ;! jump if PART present
     if type=1 then plant(JS103P)  ;! basic process 'copy' - not
                                 ! for functions
     p=p+2                      ;! p on REST OF EXPR+1
     ->9
3:   ->5 unless n>1             ;! jump if first operand
     plant(REV)
     plant(=MOM12Q)             ;! preserve A(partial expr)
5:   p=p+1                      ;! p on +' of expr.
     typep=type                 ;! preserve type
     cSEXP                      ;! evaluate first bound of PART
     m=0                        ;! mark final bound of PART not * type
     ->6 if A(p)=1              ;! jump if final bound present
     ->14 if A(p)=2             ;! jump if final bound type *
     plant(DUP)                 ;! final bound same as first
     p=p+2                      ;! p on REST OF EXPR+1
     ->7
14:  m=2                        ;! mark as * type
     p=p+2                      ;! p on REST OF EXPR+1
     ->7
6:   p=p+1                      ;! p on +' of expr. for final bound
     cSEXP                      ;! evaluate final bound
     p=p+1                      ;! p on REST OF EXPR+1
7:   ->8 if typep=7             ;! jump if finding PART of a fn value
     plant(JS(110+m)P)          ;! basic process for selecting
                                 ! PART of string
     plant(JS103P)              ;! copy selected PART
     ->9
8:   plant(JS(109+m)P)          ;! select part of fn value
9:   ->4 unless n>1             ;! jump if first operand
     plant(M-I12)
     plant(MOM12)
     plant(REV)                 ;! restore A(partial expr)
     ->4
```

```
S(2):! CONST
     fault(100) unless A(p)=2  ;! must be string literal
     plant(SET(A(p+1)))
     plant(JS106P)             ;! produce list rep. for literal
     p=p+3                     ;! p on REST OF EXPR+1
4:   if n>1 then plant(JS107P)  ;! concatenate unless first operand
     ->10

S(3):! ( EXPR )
     if n>1 then plant(=MOM12Q)  ;! preserve A(partial expr)
     cSTREXP                   ;! evaluate substring expr.
     p=p+1                     ;! p on REST OF EXPR+1
     plant(JS108P)             ;! create substring cell from expr.
     ->9

S(4):! ! EXPR !
     fault(100)                ;! invalid operand
     skip exp                  ;! skip past expr. in anal. rec.
     p=p+1                     ;! p on REST OF EXPR+1
     ->10

S(6):! -PART
     fault(100)                ;! invalid operand
     skip exp                  ;! skip expr in anal. rec.
     p=p+1                     ;! p on REXT OF EXPR+1
     ->10

S(7):! _
     if n=1 then plant(JS105P)  ;! create null string if
                                ! first operand
     p=p+1                     ;! p on REST OF EXPR+1

10:  ->11 if A(p-1)=2          ;! jump if REST OF EXPR not present
     fault(101) unless A(p)=1  ;! fault unless operator '.'
     ->12                      ;! continue for next operand
11:  type=1                    ;! string expr. just compiled
     end
```

The code planted makes use of the ´nesting store´ on KDF9, but since this is only 16 cells deep, in situations where deep use of it may occur, cells are preserved on the main store run-time stack and restored when the critical operations are complete. These are the machine code groups :

```
                    =M0M12Q
```

and

```
                    M-I12
                    M0M12
                    REV
```

(M12 contains the current top of run-time stack pointer).

Some examples of the code produced are now presented :

Example 1

```
                    r
```

for which the Analysis record would be :

```
          3  1  id(r)  2  2  2
```

and the code planted would be :

```
                    E2M3            ( pick up r )
                    DUP
                    J 14P =Z        ( check r unassigned )
                    JS 103P         ( make copy of r )
```

supposing that r was the first string variable declared at textual level 3. If r were a string name type parameter the code would be :

```
                    E2M3
                    =M10            ( pick up r indirectly )
                    M0M10
                    DUP
                    J 14P =Z
                    JS 103P
```

Example 2

```
                    r[10] . ´R10´
```

Analysis record :

```
          3  1  id(r)  2  1  3  2  1  10  2  3  1  1  2
                    2  p(´R10´)  2
```

Code produced :

```
                E2M3            ( say )
                DUP
                J 14P =Z        ( check  r  unassigned )
                SET 10          ( produced by cSEXP )
                DUP
                JS 110P         ( evaluate r[10] )
                JS 103P         ( make copy of r[10] )
                SET (p('R10'))  ( position in fixed stack )
                JS 106P         ( produce literal value )
                JS 107P         ( concatenate 'R10' to r[10] )
```

Example 3

```
                r . ( s ) . t
```

Analysis record :

```
        3  1  id(r)  2  2  1  1  3  3  1  id(s)  2  2  2
                1  1  1  id(t)  2  2  2
```

Code produced :

```
                E2M3
                DUP
                J 14P =Z
                JS 103P         ( copy )
                =M0M12Q         ( preserve )
                E3M3
                DUP             ( compiled by recursive
                J 14P =Z           call of cSTREXP )
                J103P
                JS 108P         ( create substring cell )
                M-I12
                M0M12           ( restore )
                REV
                JS 107P         ( concatenate )
                E4M3
                DUP
                J 14P =Z
                JS 103P         ( copy )
                JS 107P         ( concatenate )
```

RESOLUTIONS

Both conditional and unconditional resolutions are compiled by calls of the routine cRES. Since resolution is a three stage process it was found to be convenient to have three subroutines of cRES named cRES1, cRES2, cRES3 to compile the code for each of these stages. Only cRES1 uses the analysis record. The others process a reduced form of record produced as a by-product of cRES1. This simply contains an array of type numbers of the operands. The array ST in the compiler, in which literals and captions are stored is convenient for the purpose. The types formulated are the following :

1. String variable to take a resolved part, no fixed number of elements. E.g. s in :

```
r -> s . ʹJIMʹ . -
```

2. String variable to take a resolved part, fixed number of elements, previous entry in array not type 1. E.g. s in :

```
r -> s[3] . -
```

3. String variable to take a resolved part, fixed number of elements, previous entry type 1. E.g. s in :

```
r -> t . s[2]
```

4. String literal to be scanned for and finally garbage collected. E.g. ʹJIMʹ in :

```
r -> s . ʹJIMʹ . t
```

5. String value to be scanned for and not garbage collected. E.g. s in :

```
r -> t . ʹʹsʹʹ. u
```

6. String to be matched without scanning and finally garbage collected. E.g. ʹJIMʹ in :

```
r -> ʹJIMʹ . s
```

7. String to be matched without scanning and not garbage collected. E.g. s in :

```
r -> ʹʹsʹʹ . t
```

8. Substring to be scanned for. E.g. :

```
r -> - . ( s ) . -
```

9. Substring to be matched without scanning. E.g. :

```
r -> ( s ) . -
```

10. End of substring, previous entry not type 1. E.g. :

```
r -> - . ( s[2] ) . -
```

11. End of substring, previous entry type 1. E.g. :

```
r -> - . ( s ) . -
```

12. End of string, previous entry not type 1. E.g. :

```
r -> s . 'JIM'
```

13. End of string, previous entry type 1. E.g. :

```
r -> s
```

Other formulations could be used. The above was found to fit the requirements of cRES2 conveniently.

There now follows a description of what is intended to be compiled at each stage for each of these types.

Type 1

cRES1: Assign address of variable to stage 1 array :

```
( calculate @variable )
=M0M12Q
M+I12
```

cRES2: Assign address of current cell in string being resolved to stage 2 array :

```
DUP
=M0M10QN
```

cRES3: Assign value to string :

```
JS 125P
```

As described earlier, the stage 1 and stage 2 arrays are interlaced. These are run-time arrays and space for them is allocated at the current end of the run-time stack. M12 indexes this current end and is therefore used by cRES1 operations to advance the pointer and assign space for the array. The original value is, however, preserved in I4 by :

```
M12
=I4
```

before cRES1 is called. Before cRES2, this is transferred to M10 which is used thereafter to index up the array, leaving M12 as a valid end of stack pointer.

Type 2

cRES1: Assign number of elements and address of variable :

```
              ( calculate @variable )
              ( calculate number of elements )
              SHL+16
              OR
              =M0M12Q
              M+I12
```

cRES2: Call basic process to count down components :

```
              JS 114P
              J ( failure )
```

cRES3: Assign value to string :

```
              JS 125P
```

'failure' indicates the position of the failure exit for the current substring level of resolution. Since the basic process which counts down components can indicate a failure condition, the failure route is also compiled.

Type 3

As type 2, except that since the previous type is 1, it is potentially possible to return to the cRES2 point in a backtrack for a further attempt at matching. A backtrack label is therefore set up :

```
        ( backtrack ): JS 114P
                       J ( failure )
```

Type 4

cRES1: Assign A(literal or ''name'')

```
              ( calculate A(literal or ''name'') )
              =M0M12Q
              M+I12
```

cRES2: Call process to scan string for the literal or ''name'' with a backtrack label since backtracking to this point is potentially possible :

```
        ( backtrack ) : JS 115P
                        J ( failure )
```

cRES3: Garbage collect literal or ''name'' value :

```
              JS 102P
```

Type 5

As type 4, except no garbage collection in cRES3 is required.

Type 6

```
cRES1: Assign A(literal or ″name″) :
                ( calculate A(literal or ″name″ ) )
                =M0M12Q
                M+I12
```

cRES2: Call basic process which matches literal or ″name″ without scanning.

```
                JS 116P
                J ( failure )
                J ( success )
```

$\boxed{\text{partial failure}}$

```
        (success):
```

where ′partial failure′ is :

```
    If backtracking possible :
                SET (-2n)
                JS 117P
                J (backtrack)
```

n gives the number of locations in the dynamic control stage 1 and stage 2 array to the correct position for the backtrack. 117P resets such registers as necessary for the backtrack to proceed.

```
    If no possibility of backtracking :
                ERASE
                J (failure)
    cRES3: Garbage collect literal or ″name″
                JS 102P
```

Type 7

As type 6, but without any garbage collection in cRES3.

```
     Type 8
     cRES1:
                      M+I12
                      M+I12
i.e. nothing assigned.
     cRES2:  Call basic process to scan for a substring :
             (backtrack):     JS 118P
                              J (failure)
                             ┌───────────────────────┐
                             │ Resolution of substring │
                             └───────────────────────┘
                              ZERO
                              JS 117P
                              J (backtrack)
             (success):
     cRES3:  No action.
```

'success' indicates the route after successful resolution of the substring, the preceeding three instructions forming the failure route leading to the backtrack label and further scanning for a substring. The code for the resolution of the substring is produced by a recursive call on cRES2.

```
     Type 9
     cRES1:
                      M+I12
                      M+I12
i.e. no assignments.
```

cRES2: Call basic process to match a substring without scanning.

```
                JS 119P
                J (failure)
                J (success 1)
```

$\boxed{\text{partial failure}}$

(success 1):

$\boxed{\text{resolution of substring}}$

$\boxed{\text{partial failure}}$

(success 2):

cRES3: No action.

The ´partial failure´ blocks are the same as those of type 6 and the ´resolution of substring´ is again produced by a recursive call of cRES2. In this case, no backtrack route is automatically available after the substring resolution.

Type 10

cRES1:

```
                M+I12
                M+I12
```

cRES2: Call basic process to check end of substring and exit from substring :

```
                JS 120P
                J (success)
```

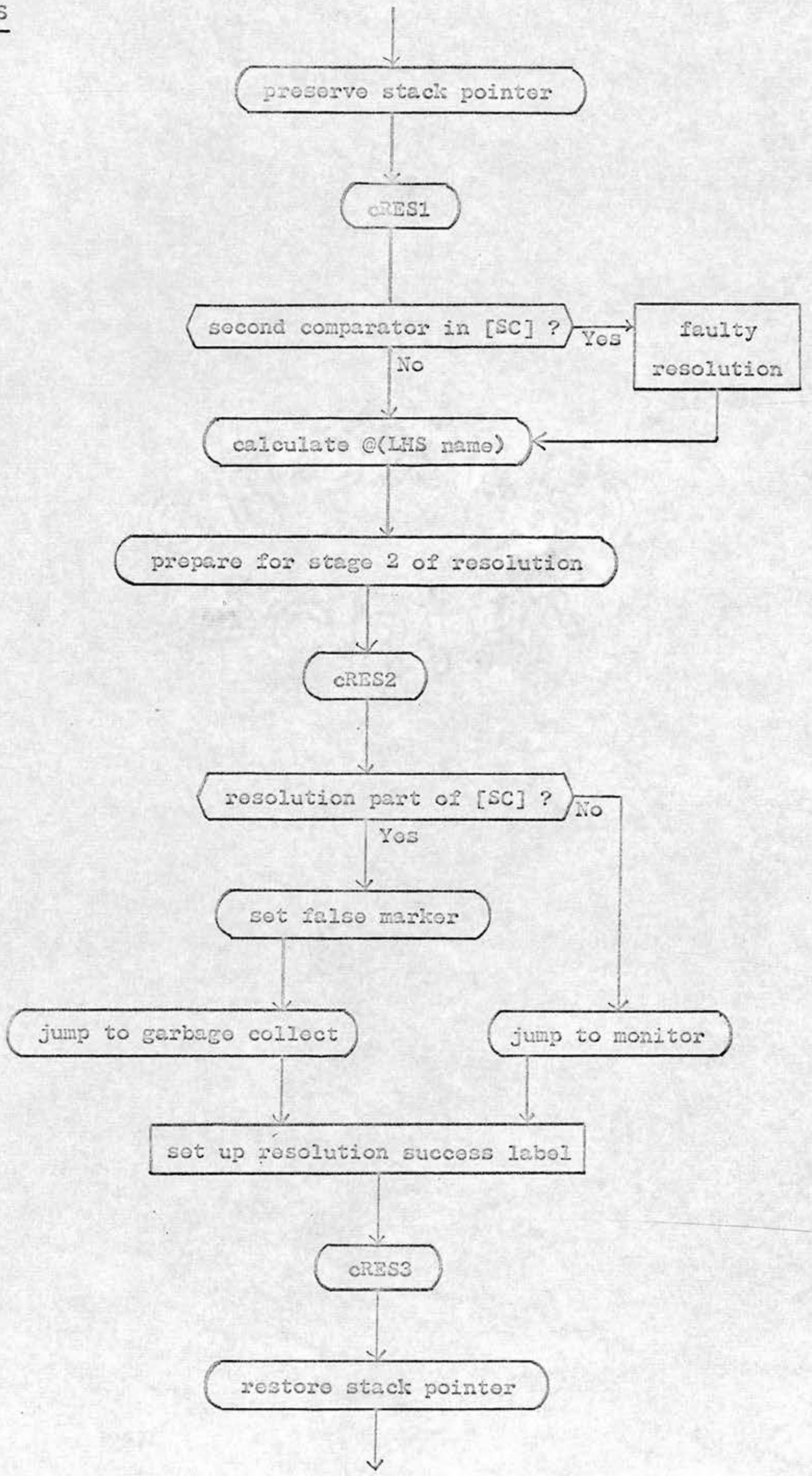
$\boxed{\text{partial failure}}$

```
(failure):      JS 121P
```

cRES3: No action.

```
Type 11
cRES1:
                M+I12
                M+I12
cRES2:  Exit from substring :
                JS 122P
                J (success)
(failure):      J 121P
cRES3:  No action.

Type 12
cRES1:
                M+I12
                M+I12
cRES2:  Call basic process to check end of string :
                JS 123P
                J (success)
               ┌─────────────────┐
               │ partial failure │
               └─────────────────┘
(failure):
cRES3:  No action.

Type 13
cRES1:
                M+I12
                M+I12
cRES2:  Clear up at end of string :
                I4
                =M13
                M0M13N
                =M0M10
                =M0M10QN
                J (success)
(failure):
cRES3:  No action.
```

cRES


preserve stack pointer
cRES1
second comparator in [SC] ?
Yes
faulty resolution
No
calculate @(LHS name)
prepare for stage 2 of resolution
cRES2
resolution part of [SC] ?
No
Yes
set false marker
jump to garbage collect
jump to monitor
set up resolution success label
cRES3
restore stack pointer

```
    routine cRES (integer z,q)
    integer n0p,qq,pp,suc
    routinespec cRES1
    routinespec cRES2
    routinespec cRES3
    n0p=n0                    ;! preserve n0 in n0p
    ST(n0)=0
    n0=n0+1                   ;! set up dummy type 0
    plant(M12)
    plant(=I4)                ;! preserve run-time stack pointer in I4
    cRES1
    ->1 unless z=1            ;! jump for unconditional case
    p=p+1                     ;! p on REST OF SC
    ->1 unless A(p-1)=1
    fault(109)                ;! 'double-sided' resolution
                               ! conditions invalid
    skip exp
1:  qq=p                      ;! preserve p in qq
    p=q
    cNAME(3)                  ;! calculate @(LHS name)
    fault(110) unless type=1 and A(p)=2 and (z=0 or A(p+1)=2)
                               ! resolution invalid unless string
                               ! variable with no PART and res.
                               ! unconditional or REST OF EXPR null.
    plant(JS 124P)            ;! prepare for stage 2
    pp=n0p                    ;! set pp to start of compile-time
                               ! type array
    cRES2
    pp=@(15P)                 ;! failure to resolve monitor address
    if z=0 then ->2           ;! jump if unconditional
    plant(ERASE)
    plant(ZERO)
    plant(NOT)                ;! set false marker
    plabel=plabel-1
    k=plabel
    store jump                ;! set up jump to garbage
                               ! collection section
    pp=0
```

```
2:    plant(J (pp))              ;! jump to monitor or garbage
                                  ! collection section
      pushdown2(label(level),ca,suc)  ;! set up resolution
                                        ! success address
      n0=n0p                     ;! set base of type array
      cRES3
      plant(I4)
      plant(=M12)                ;! restore run-time stack pointer
      p=qq
      return

      routine cRES1
      switch r(1:7),sw(0:2)
      integer l,m
      integerfnspec get type
      l=0                        ;! substring depth counter
1:    fault(107) unless A(p)=3  ;! +' not null
2:    p=p+2                      ;! p on OPERAND+1
      ->r(A(p-1))                ;! switch on type of operand

r(1):! NAME
      cNAME(3)                   ;! calculate @name
      fault(107) unless type=1  ;! fault unless name a string
14:   p=p+1                      ;! p on PART+1
      if A(p-1)=2 then ->11     ;! jump if no PART
      ->sw(get type)

sw(2):cSEXP                      ;! calculate number of components
      plant(SHL+16)
      plant(OR)
      p=p+1                      ;! p on REST OF PART+1
      if A(p-1)=3 then ->12     ;! jump if no second index to PART
      fault(107)                 ;! second index invalid in resolutions
      if A(p-1)=1 then skip exp
12:   ST(n0)=2
      if ST(n0-1)=1 then ST(n0)=3  ;! set type in array
      ->3
11:   m=n0
```

```
13:  m=m-1
     if ST(m)=1 or ST(m)=3 then fault(107)
     if ST(m)=2 then ->13
     ST(n0)=1
     ->3

sw(1):cSTREXP
     plant(OR)
     p=p+1
     if A(p-1)=3 then ->21
     fault(107)
     if A(p-1)=1 then skip exp
     ->21

sw(0):p=p+5
     cSTREXP
     plant(OR)
     plant(SET 1)
     plant(SHC-1)
     plant(OR)
     ->21

r(2):! CONST
     fault(107) unless A(p)=2  ;! must be string constant
     p=p+2
     plant(SET(A(p-1)))       ;! set address of chars in stack
     plant(JS 106P)
21:  ST(n0)=6
     if ST(n0-1)=1 then ST(n0)=4  ;! set type entry
     ->3

r(3):! (EXPR)
     ST(n0)=9
     if ST(n0-1)=1 then ST(n0)=8
     n0=n0+1
     plant(M+I12)
     plant(M+I12)
     l=l+1                    ;! increment substring depth counter
     ->1
```

```
r(4):! !EXPR!
     fault(107)                  ;! no modulus signed exprs. allowed
     skip exp
     ->4

r(5):! ''NAME''
     cNAME(2)                    ;! pick up value of name
     fault(107) unless type=1 or type=7  ;! must be string
                                          ! or stringfn
     ->51 unless type=1          ;! jump for string fn
     plant(DUP)
     plant(J 14P =Z)             ;! check name assigned unless
                                  ! a function
51:  m=0
     if type=7 then m=1          ;! m=1 for garbage collection
     ST(n0)=7-m
     if ST(n0-1)=1 then ST(n0)=5-m
     n0=n0+1
     p=p+1                       ;! p on PART+1
     if A(p-1)=2 then ->7        ;! jump if no PART
     cSEXP                       ;! calculate first index
     if A(p)=3 then plant(DUP)   ;! DUP if no second index
     m=m+2 unless A(p)=2         ;! unless second index '*'
     p=p+1                       ;! p on +' of EXPR or REST OF EXPR+1
     if A(p-1)=1 then cSEXP      ;! calculate second index
     plant(JS (112-m)P)          ;! call on appropriate basic process
     ->7

r(6):! -PART
     plant(ZERO)                 ;! treat as [NAME][PART] with @name=0
     ->14

r(7):! _
     if A(p)=1 then ->6          ;! ignore if not end of expr.
     if ST(n0-1)=0 then fault(107)  ;! fault if e.g. r->_
     ->5
```

```
3:    n0=n0+1
7:    plant(=M0M12Q)
      plant(M+I12)
4:    if A(p)=2 then ->5       ;! end of expr.
6:    p=p+1                    ;! p on OPERATOR
      fault(108) unless A(p)=1  ;! fault unless operator '.'
      ->2                      ;! go for next operand
5:    p=p+1                    ;! p on REST OF EXPR+1
      ST(n0)=10
      if ST(n0-1)=1 then ST(n0)=11
      n0=n0+1
      plant(M+I12)
      plant(M+I12)
      l=l-1                    ;! decrement substring depth counter
      ->4 if l>=0              ;! not end of whole expression
      ST(n0-1)=ST(n0-1)+2
      return

      integerfn get type
      switch sw(1:7)
      integer q
      if A(p)=3 and A(p+1)=6 and A(p+2)=0=A(p+3) and c
              A(p+4)=1 then result=0
      q=p
sw(3):if A(q)<3 then result=2
      q=q+2
      ->sw(A(q-1))
sw(1):copy tag(A(q))
      if parity(type)=1 or type=5 then result=2
sw(5):sw(6):sw(7):result=1
sw(2):if A(q)=2 then result=1
sw(4):result=2
      end

      end
```

```
     routine cRES2
     integer fail,back,backpp,i
     routinespec failj          ;! to set up and plant a jump
     routinespec faill          ;! or label for failure
                                 ! route from a basic process
     routinespec sucj           ;! to set up and plant a jump
     routinespec sucl           ;! or label for success
                                 ! route from a basic process
     routinespec pfail(integer m,n)  ;! to set up a backtrack route
     switch s(1:13)
     fail=0
     back=0                     ;! no failure or backtrack routes
                                 ! yet possible
1:   pp=pp+1                    ;! index to type array set up by cRES1
     ->s(ST(pp))                ;! switch on operand type

s(1):plant(DUP) unless ST(pp+1)>=11  ;! DUP unless end of expr.
     plant(=MOM10QN)
     ->1

s(2):i=114
     ->3

s(3):i=114
     ->2

s(4):s(5):i=115
2:   plabel=plabel-1            ;! set up backtrack label
     back=plabel
     backpp=pp
     pushdown2(label(level),ca,back)
3:   plant(JS (i)P)             ;! jump to basic process  i
     failj                      ;! set up failure exit route
     ->1 unless ST(pp)=8        ;! next operand unless subexpr.
     cRES2                      ;! perform substring resolution
     pfail(0,0)                 ;! substring resolution failure route
     sucl                       ;! substring resolution success route
     ->1
```

```
s(6):s(7):plant(JS 116P)
     failj
     sucj
     pfail(2*(backpp-pp),0)  ;! partial failure route
     sucl                    ;! success route entry point
     ->1
s(8):i=118
     ->2
s(9):plant(JS 119P)
     failj
     sucj
     i=2*(backpp-pp)
     pfail(i,0)
     sucl
     cRES2                   ;! perform substring resolution
     pfail(i,0)
     sucl
     ->1

s(10):plant(JS 120P)
     sucj
     pfail(2*(backpp-pp),1)
     faill                   ;! set up failure to resolve exit
                              ! route from substring resolution
     plant(JS 121P)
     return

s(11):plant(JS 122P)
     sucj
     faill
     plant(JS 121P)
     return

s(12):plant(JS 123P)
     sucj
     pfail(2*(backpp-pp),1)
     faill
     return
```

```
s(13):plant(I4)
     plant(=M13)
     plant(MOM13N)
     plant(=MOM10)
     plant(=MOM10QN)
     sucj
     faill

     routine failj
     ->1 unless fail=0          ;! jump if failure label already set
     plabel=plabel-1
     fail=plabel                ;! set up private label
1:   k=fail
     store jump
     plant( J 0 )               ;! jump address filled in later
     end

     routine faill
     pushdown2(label(level),ca,fail) unless fail=0
     end

     routine sucj
     plabel=plabel-1
     suc=plabel                 ;! set up success label
     k=suc
     store jump
     plant(J 0)
     end

     routine sucl
     pushdown2(label(level),ca,suc)
     end

     routine pfail(integer m,n)
     ->1 unless back=0          ;! jump if backtracking possible
     plant(ERASE)
     failj if n=0
     ->2
```

```
1:   plant(SET(m))
     plant(JS 117P)
     k=back
     store jump
     plant(J 0)                  ;! set up backtrack jump
2:   end

     end  ;! of cRES2

     routine cRES3
     plant(I4)
     plant(=M10)                 ;! set up base of run-time array
     pp=n0
1:   n0p=pp+1
2:   pp=pp+1                     ;! first pass over type array
     ->3 if ST(pp)>=12
     ->2 unless ST(pp)<=4 or ST(pp)=6  ;! no assignments to be made
     plant(SET(2*(pp-n0p)))
     plant(JS 125P)
     ->1
3:   plant(ERASE)
     ->4 if z=0                  ;! unconditional
     plant(ZERO)
     pushdown2(label(level),ca,plabel)
4:   pp=n0
     n0p=pp+1
6:   pp=pp+1                     ;! second pass over type array
     return if ST(pp)>=12        ;! end of expr.
     ->6 unless ST(pp)=4 or ST(pp)=6  ;! loop unless garbage
                                       ! collection types
     plant(SET(2*(pp-n0p)))
     plant(I4)
     plant(+)
     plant(JS 102P)              ;! jump to return string basic process
     ->6
     end

     end  ;! of cRES
```

To illustrate the code generation of cRES , a few examples are given :

Example 1

```
                 r -> s
Analysis record :
          id(r)  2  2  .  .  3  1  id(s)  2  2  2
           q                 p
Compile-time type array :
          0  1  13
Code generated :
          M12
          =I4              (preserve run-time stack pointer)
          (calculate @s) ⎫
          =M0M12Q        ⎪
          M+I12          ⎬(cRES1)
          M+I12          ⎪
          M+I12          ⎭
          (calculate @r)
          JS 124P          (prepare for stage 2)
          =M0M10QN       ⎫
          I4             ⎪
          =M13           ⎪
          M0M13N         ⎬(cRES2)
          =M0M10         ⎪
          =M0M10QN       ⎪
          J (success)    ⎭
          J 15P            (failure monitor - redundant)
(success): I4            ⎫
          =M10           ⎪
          SET 2          ⎬(cRES3)
          JS 125P        ⎪ (make assignment to s)
          ERASE          ⎭
          I4
          =M12
```

The quantity of code generated is substantial for this the simplest case of resolution. It effectively represents the overhead on a resolution, becoming very much less significant for more realistic resolutions.

Simple cases such as this are not in general treated specially for the sake of consistency.

```
     Example 2
               r -> s . ´LIT´ . t
Analysis record :
     id(r) 2 2 . . 3 1 id(s) 2 2 1 1 2 2 p(´LIT´) 1 1 1 id(t) 2 2 2
      q          p
Compile-time type array :
               0  1  4  1  13
Code generated :
               M12
               =I4
               (calculate @s)     ⎫
               =MOM12Q            ⎪
               M+I12              ⎪
               SET p(´LIT´)       ⎪
               JS 106P            ⎪
               =MOM12Q            ⎪
               M+I12              ⎬      (cRES1)
               (calculate @t)     ⎪
               =MOM12Q            ⎪
               M+I12              ⎪
               M+I12              ⎪
               M+I12              ⎭
               (calculate @r)
               JS 124P
               DUP                ⎫
               =MOM10QN           ⎪
(back):        JS 115P            ⎪      (scan for ´LIT´)
               J (fail)           ⎪
               =MOM10QN           ⎪
               I4                 ⎬      (cRES2)
               =M13               ⎪
               MOM13N             ⎪
               =MOM10             ⎪
               =MOM10QN           ⎪
               J (success)        ⎭
```

```
(fail):       J 15P
(success):    I4
              =M10
              SET 0
              JS 125P                    (assign to s)
              SET 2
              JS 125P                    (assign to t)
              ERASE                      (cRES3)
              SET 2
              I4
              +
              JS 102P                    (garbage collect 'LIT')
              I4
              =M12
```

Example 3

r -> s[2].t.'VAL'.u.''w[2:4]''.'WX'.x

Analysis record :

```
     id(r) 2 2 . . 3 1 id(s) 2 1 3 2 1 2 2 3 1 1 1 id(t) 2 2
      1 1 2 2 p('VAL') 1 1 1 id(u) 2 2 1 1 5 id(w) 2 1 3 2 1
     2 2 1 3 2 1 4 2 1 1 2 2 p('WX') 1 1 1 id(x) 2 2 2
```

Compile-time type array :

```
              0  2  1  4  1  5  6  1  13
```

Code generated :

```
              M12
              =I4
              (calculate @s)
              SET 2
              SHL+16
              OR                         (form 2/@s)
              =M0M12Q
              M+I12
              (calculate @t)
              =M0M12Q
              M+I12
```

```
            SET p('VAL')
            JS 106P
            =M0M12Q
            M+I12
            (calculate @u)
            =M0M12Q
            M+I12
            (pick up w)              (cRES1)
            DUP
            J 14P =Z
            SET 2
            SET 4
            JS 110P                  (select w[2:4])
            =M0M12Q
            M+I12
            SET p('WX')
            JS 106P
            =M0M01Q
            M+I12
            (calculate @x)
            =M0M12Q
            M+I12
            M+I12
            M+I12
            (calculate @r)
            JS 124P
            JS 114P                  (match 2 components)
            J (fail)
            DUP
            =M0M10QN
(back1):    JS 115P                  (scan for 'VAL')
            J (fail)
            DUP
            =M0M10QN
```

```
(back2):        JS 115P              (scan for w[2;4])
                J (fail)
                JS 116P              (match 'WX')
                J (fail)             (cRES2)
                J (success 1)
                SET 2
                JS 117P              (backtrack)
                J (back 2)
(success 1):    =M0M10QN
                I4
                =M13
                M0M13N
                =M0M10
                =M0M10QN
                J (success 2)
(fail):         J 15P
(success 2):    I4
                =M10
                SET 0
                JS 125P              (assign to s)
                SET 0
                JS 125P              (assign to t)
                SET 2
                JS 125P              (assign to u)
                SET 4
                JS 125P              (assign to x)
                ERASE                (cRES3)
                SET 4
                I4
                +
                JS 102P              (garbage collect 'VAL')
                SET 10
                I4
                +
                JS 102P              (garbage collect 'WX')
                I4
                =M12
```

Example 4

r -> s . (t . 'TU' . u) . v

Analysis record :

id(r) 2 2 . . 3 1 id(s) 2 2 1 1 3 3 1 id(t) 2 2 1 1 2
2 p('TU') 1 1 1 id(u) 2 2 2 1 1 1 id(t) 2 2 2

Compile-time array :

0 1 8 1 6 1 11 1 13

Code generated :

```
             M12
             =I4
             (calculate @s)
             =M0M12Q
             M+I12
             M+I12
             M+I12
             (calculate @t)
             =M0M12Q
             M+I12
             SET p('TU')
             JS 106P
             =M0M12Q
             M+I12                    (cRES1)
             (calculate @u)
             =M0M12Q
             M+I12
             M+I12
             M+I12
             (calculate @v)
             =M0M12Q
             M+I12
             M+I12
             M+I12
             (calculate @r)
             JS 124P
             DUP
             =M0M10QN
(back):      JS 118P                  (scan for substring)
             J (fail 1)
```

```
               DUP
               =M0M10QN
               JS 116P              (scan for 'TU')
               J (fail2)
               J (success 2)
               ERASE                ((cRES2))
               J (fail 2)           (no backtracking possible)
(success 2):   =M0M10QN
               JS 122P              (exit from substring)
               J (success 1)        (cRES2)
(fail 2):      JS 121P              (exit from substring)
               SET 0
               JS 117P              (backtrack)
               J (back)
(success 1):   =M0M10QN
               I4
               =M13
               M0M13N
               =M0M10
               =M0M10QN
               J (success 3)
(fail 1):      J 15P
(success 3):   I4
               =M10
               SET 0
               JS 125P              (assign to s)
               SET 2
               JS 125P              (assign to t)
               SET 2                (cRES3)
               JS 125P              (assign to u)
               SET 2
               JS 125P              (assign to v)
               ERASE
               SET 6
               I4
               +
               JS 102P              (garbage collect 'TU')
               I4
               =M12
```

REFERENCES

1. Bobrow, D.G. and Raphael, B. A Comparison of List Processing Computer Languages. Comm. ACM 7,4(April 1964).

2. Brooker, R.A. and Rohl, J.S. Atlas Autocode Reference Manual. University of Manchester Computer Science Department.

3. COMIT Programmers Reference Manual. The M.I.T. Press.

4. De Morgan, R.M. and Rutovitz, D. A String Facility for Atlas Autocode. University of Manchester Computer Science Department.

5. Farber, D.J. et al. SNOBOL, A String Manipulation Language. J.ACM 11,2(January 1964).

6. LISP 1.5 Programmer's Manual. The M.I.T. Press.

7. Newell, A. et al. Information Processing Language-V Manual. Prentice-Hall, Englewood Cliffs, N.J.

8. Weizenbaum, J. Symmetric List Processor. Comm. ACM 6,9(September 1963).

APPENDIX A

Some examples of ASTRA programs

```
***A
JOB
CSC004/00000000/ B.S.READ WENT TO MOW A MEADOW
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%INTEGER I,J
%STRING Z
%STRINGARRAY X(1:6)
%STRINGFNSPEC Y(%INTEGER K)
X(1)='ONE'
X(2)='TWO'
X(3)='THREE'
X(4)='FOUR'
X(5)='FIVE'
X(6)='SIX'
Z='WENT_TO_MOW_A_MEADOW'
%CYCLE I=1,1,6
  WRITE STRING('--'.X(I).Y(I).'_WENT_TO_MOW,_'.Z)
  %CYCLE J=I,-1,1
    WRITE STRING('-'.X(J).Y(J))
  %REPEAT
  WRITE STRING('_AND_HIS_DOG-'.Z)
%REPEAT

%STRINGFN Y(%INTEGER K)
%IF K=1 %THEN %RESULT='_MAN'
%RESULT='_MEN'
%END

%ENDOFPROGRAM
```

```
29/04/69   09.49.14
ASTRA   10/06/68


CSC004/00000000/ B.S.READ WENT TO MOW A MEADOW

     0   BEGIN
    19       STRING FN  Y
    22       END OF STRING FN
    23   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2243 WORDS
PROGRAM DUMPED
COMPILING TIME   12 SEC /  3 SEC


ONE MAN WENT TO MOW, WENT TO MOW A MEADOW
ONE MAN AND HIS DOG
WENT TO MOW A MEADOW

TWO MEN WENT TO MOW, WENT TO MOW A MEADOW
TWO MEN
ONE MAN AND HIS DOG
WENT TO MOW A MEADOW

THREE MEN WENT TO MOW, WENT TO MOW A MEADOW
THREE MEN
TWO MEN
ONE MAN AND HIS DOG
WENT TO MOW A MEADOW

FOUR MEN WENT TO MOW, WENT TO MOW A MEADOW
FOUR MEN
THREE MEN
TWO MEN
ONE MAN AND HIS DOG
WENT TO MOW A MEADOW
```

```
FIVE MEN WENT TO MOW, WENT TO MOW A MEADOW
FIVE MEN
FOUR MEN
THREE MEN
TWO MEN
ONE MAN AND HIS DOG
WENT TO MOW A MEADOW

SIX MEN WENT TO MOW, WENT TO MOW A MEADOW
SIX MEN
FIVE MEN
FOUR MEN
THREE MEN
TWO MEN
ONE MAN AND HIS DOG
WENT TO MOW A MEADOW


STOPPED AT LINE  23
CSC004/00000000/ B.S.READ WENT TO MOW A MEADOW
RUNNING TIME   7 SEC /  1 SEC
```

```
***A
JOB
CSC004/00000000/ COUNT WORDS
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGARRAY WORD(1:500)
%INTEGERARRAY NUMBER(1:500)
%STRING S
%INTEGER I,J
%STRINGFNSPEC NEXT WORD
WORD(1)=NEXT WORD
I=1
NUMBER(1)=1
3:S=NEXT WORD
%IF S='FINIS' %THEN ->1
%CYCLE J=1,1,I
%IF S=WORD(J) %THEN ->2
%REPEAT
I=I+1
WORD(I)=S
NUMBER(I)=1
->3
2:NUMBER(J)=NUMBER(J)+1
->3
1:%CYCLE J=1,1,I
WRITE(NUMBER(J),2)
WRITE STRING('____'.WORD(J).'-')
%REPEAT

%STRINGFN NEXT WORD
%STRING LETTER,WORD
WORD=_
1:READ ITEM(LETTER)
->1 %UNLESS 'A'<=LETTER<='Z'
2:WORD=WORD.LETTER
```

```
READ ITEM(LETTER)
->2 %IF 'A'<=LETTER<='Z'
%RESULT=WORD
%END

%ENDOFPROGRAM

FRIENDS, ROMANS, COUNTRYMEN, LEND ME YOUR EARS. I COME TO BURY
CAESAR, NOT TO PRAISE HIM. THE EVIL THAT MEN DO LIVES AFTER THEM.
THE GOOD IS OFT INTERRED WITH THEIR BONES. SO LET IT BE WITH CAESAR.
THE NOBLE BRUTUS HATH TOLD YOU CAESAR WAS AMBITIOUS. IF IT WERE SO
IT WAS A GRIEVOUS FAULT. AND GRIEVOUSLY HATH CAESAR ANSWERD IT.

FINIS
```

```
29/04/69   09.49.38
ASTRA   10/06/68


CSC004/00000000/ COUNT WORDS

     0   BEGIN
    24       STRING FN  NEXTWORD
    33       END OF STRING FN
    34   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2269 WORDS
PROGRAM DUMPED
COMPILING TIME   4 SEC /  2 SEC
  1    FRIENDS
  1    ROMANS
  1    COUNTRYMEN
  1    LEND
  1    ME
  1    YOUR
  1    EARS
  1    I
  1    COME
  2    TO
  1    BURY
  4    CAESAR
  1    NOT
  1    PRAISE
  1    HIM
  3    THE
  1    EVIL
  1    THAT
  1    MEN
  1    DO
  1    LIVES
  1    AFTER
  1    THEM
  1    GOOD
```

```
     1      IS
     1      OFT
     1      INTERRED
     2      WITH
     1      THEIR
     1      BONES
     2      SO
     1      LET
     4      IT
     1      BE
     1      NOBLE
     1      BRUTUS
     2      HATH
     1      TOLD
     1      YOU
     2      WAS
     1      AMBITIOUS
     1      IF
     1      WERE
     1      A
     1      GRIEVOUS
     1      FAULT
     1      AND
     1      GRIEVOUSLY
     1      ANSWERD


STOPPED AT LINE  34
CSC004/00000000/ COUNT WORDS
RUNNING TIME   19 SEC /  14 SEC
```

```
***A
JOB
CSC004/00000000/ SORT WORDS
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGARRAY X(1:20)
%INTEGER I
%STRING D
%ROUTINESPEC STRING QUICKSORT(%INTEGER A,B)
%CYCLE I=1,1,20
READ STRING(X(I))
%IF X(I)='END' %THEN ->1
%REPEAT
1:STRING QUICKSORT(1,I-1)
%CYCLE I=1,1,I-1
WRITE STRING('-'.X(I))
%REPEAT

%ROUTINE STRING QUICKSORT(%INTEGER A,B)
%INTEGER L,U
%RETURN %IF A>=B
L=A
U=B
D=X(U)
->2
1:L=L+1
->4 %IF L=U
2:->1 %UNLESS X(L)>D
X(U)=X(L)
3:U=U-1
->4 %IF L=U
->3 %UNLESS X(U)<D
X(L)=X(U)
->1
4:X(U)=D
```

```
STRING QUICKSORT(A,L-1)
STRING QUICKSORT(U+1,B)
%END

%ENDOFPROGRAM


(ABA)
(A)
(AARDWOLF)
(ABACOT)
(AARDVARK)
(AARONIC)
(ASTRA)
(ABACK)
(AASVOGEL)
(AB)
(END)
```

```
29/04/69   09.50.06
ASTRA   10/06/68


CSC004/00000000/ SORT WORDS

     0   BEGIN
    13       ROUTINE  STRINGQUICKSORT
    32       END OF ROUTINE
    33   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2238 WORDS
PROGRAM DUMPED
COMPILING TIME   4 SEC /  3 SEC

A
AARDVARK
AARDWOLF
AARONIC
AASVOGEL
AB
ABA
ABACK
ABACOT
ASTRA


STOPPED AT LINE  33
CSC004/00000000/ SORT WORDS
RUNNING TIME   1 SEC / 0 SEC
```

```
***A
JOB
CSC004/00000000/ TUTORIAL GROUPS
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRING R,S,SS,T,TT
TT=_
4:READ STRING(R)
%IF R='END' %THEN ->1
R->T.':'.SS
3:->2 %UNLESS SS->S.','.SS
TT=TT.S.'('.T.')-'
->3
2:TT=TT.SS.'('.T.')-'
->4
1:WRITE STRING(TT)
5:READ STRING(R)
%IF R='END' %THEN %STOP
WRITE STRING('-THE_TUTOR_OF_'.R.'_IS_')
T='NOT_KNOWN' %UNLESS TT->-.''R''.'('.T.')'.-
WRITE STRING(T)
->5
%ENDOFPROGRAM

(REES:FIELDING,PAULSEN,YOUNG)
(WHITFIELD:COTTON,FRASER,MCINTOSH)
(FOSTER:YUILLE,STEVENSON)
(END)
(FRASER)
(BLOGGS)
(YOUNG)
(END)
```

```
29/04/69   09.50.16
ASTRA   10/06/68


CSC004/00000000/ TUTORIAL GROUPS

      0   BEGIN
     18   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2281 WORDS
PROGRAM DUMPED
COMPILING TIME   4 SEC /  2 SEC

FIELDING(REES)
PAULSEN(REES)
YOUNG(REES)
COTTON(WHITFIELD)
FRASER(WHITFIELD)
MCINTOSH(WHITFIELD)
YUILLE(FOSTER)
STEVENSON(FOSTER)

THE TUTOR OF FRASER IS WHITFIELD
THE TUTOR OF BLOGGS IS NOT KNOWN
THE TUTOR OF YOUNG IS REES



STOPPED AT LINE  13
CSC004/00000000/ TUTORIAL GROUPS
RUNNING TIME   3 SEC /  1 SEC
```

```
***A
JOB
CSC004/00000000/  DICTIONARY
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%ROUTINESPEC INSERT(%STRINGNAME DICT,WORD)
%STRINGFNSPEC LOOKUP(%STRINGNAME DICT,WORD)
%ROUTINESPEC LIST(%STRINGNAME DICT,%STRING WORD)
%STRING R,D
D=_
2:READSTRING(R)
%IF R='END' %THEN ->1
WRITE STRING('-'.R)
INSERT(D,R)
WRITE STRING ('-'.D)
->2
1:NEWLINES(2)
LIST(D,_)
NEWLINES(3)
3:READ STRING(R)
%IF R='END' %THEN %STOP
WRITE STRING('--WORD_'.R.'_'.LOOKUP(D,R).'_DICTIONARY')
->3

%ROUTINE INSERT(%STRINGNAME DICT,WORD)
%STRING W,X,Y,Z
DICT->X
WORD->W
3:%IF X=_ %THEN ->1
%IF X[1]='.' %THEN ->2
X->Y.(Z).X
%IF W=_ %OR W[1]#Y %THEN ->3
Z->X
W->-[1].W
->3
2:%IF W=_ %THEN %RETURN
```

```
X->-[1].X
->3
1:X<-'.'
4:%IF W=_ %THEN %RETURN
X<-W[1].('.')
X->-.(X)
W->-[1].W
->4
%END

%STRINGFN LOOKUP(%STRINGNAME DICT,WORD)
%STRING W,X,Y,Z
DICT->X
WORD->W
3:%IF X=_ %THEN %RESULT='NOT_IN'
%IF X[1]='.' %THEN ->2
X->Y.(Z).X
%IF W=_ %OR W[1]#Y %THEN ->3
Z->X
W->-[1].W
->3
2:%IF W=_ %THEN %RESULT='IN'
X->-[1].X
->3
%END

%ROUTINE LIST(%STRINGNAME DICT,%STRING WORD)
%STRING X,Y,Z
DICT->X
2:%IF X=_ %THEN %RETURN
%IF X[1]='.' %THEN ->1
X->Y.(Z).X
LIST(Z,WORD.Y)
->2
1:WRITE STRING('-'.WORD)
X->-[1].X
->2
%END
%ENDOFPROGRAM
```

```
(HEBE)
(HECATE)
(HECTOR)
(HELEN)
(HELIOS)
(HERA)
(HERCULES)
(HERMES)
(END)

(FRED)
(HELEN)
(JIM)
(HECTOR)
(HARRY)
(HERA)
(HE)
(END)
```

```
29/04/69   09.50.28
ASTRA   10/06/68


CSC004/00000000/  DICTIONARY

     0   BEGIN
    19      ROUTINE  INSERT
    39      END OF ROUTINE
    40      STRING FN  LOOKUP
    54      END OF STRING FN
    55      ROUTINE  LIST
    66      END OF ROUTINE
    67   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2623 WORDS
PROGRAM DUMPED
COMPILING TIME   7 SEC /  5 SEC

HEBE
H(E(B(E(.))))
HECATE
H(E(B(E(.))C(A(T(E(.))))))
HECTOR
H(E(B(E(.))C(A(T(E(.)))T(O(R(.))))))
HELEN
H(E(B(E(.))C(A(T(E(.)))T(O(R(.))))L(E(N(.)))))
HELIOS
H(E(B(E(.))C(A(T(E(.)))T(O(R(.))))L(E(N(.))I(O(S(.))))))
HERA
H(E(B(E(.))C(A(T(E(.)))T(O(R(.))))L(E(N(.))I(O(S(.))))R(A(.))))
HERCULES
H(E(B(E(.))C(A(T(E(.)))T(O(R(.))))L(E(N(.))I(O(S(.))))R(A(.)C(U
(L(E(S(.))))))))
HERMES
H(E(B(E(.))C(A(T(E(.)))T(O(R(.))))L(E(N(.))I(O(S(.))))R(A(.)C(U
(L(E(S(.)))))M(E(S(.))))))
```

```
HEBE
HECATE
HECTOR
HELEN
HELIOS
HERA
HERCULES
HERMES




WORD FRED NOT IN DICTIONARY

WORD HELEN IN DICTIONARY

WORD JIM NOT IN DICTIONARY

WORD HECTOR IN DICTIONARY

WORD HARRY NOT IN DICTIONARY

WORD HERA IN DICTIONARY

WORD HE NOT IN DICTIONARY


STOPPED AT LINE  16
CSC004/00000000/  DICTIONARY
RUNNING TIME   8 SEC /  6 SEC
```

```
***A
JOB
CSC004/00000000/  CUP AND CAP
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGFNSPEC CUP (%STRING R,S)
%STRINGFNSPEC CAP (%STRING R,S)
%STRING R,S
1:READ STRING(R)
%IF R='END' %THEN %STOP
READ STRING(S)
WRITE STRING('-'.R.'_____'.S.'_____'.CUP(R,S).'_____'.CAP(R,S))
->1

%STRINGFN CUP(%STRING R,S)
%STRING T,U,V
T=S
R->U
1:%IF U=_ %THEN %RESULT=T
U->V[1].U
T=T.V %UNLESS S->-.''V''.-
->1
%END

%STRINGFN CAP(%STRING R,S)
%STRING T,U,V
T=_
R->U
1:%IF U=_ %THEN %RESULT=T
U->V[1].U
T=T.V %IF S->-.''V''.-
->1
%END

%ENDOFPROGRAM
```

```
(AB) (BC)
(ABCD) (CDEF)
(ABCDEFG) (WXYZ)
(A(BC)D(EFG)H) (D(EFGH)H(BC))
(END)
```

```
29/04/69   09.50.49
ASTRA   10/06/68


CSC004/00000000/  CUP AND CAP

     0   BEGIN
     9       STRING FN  CUP
    17       END OF STRING FN
    18       STRING FN  CAP
    26       END OF STRING FN
    27   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2315 WORDS
PROGRAM DUMPED
COMPILING TIME   5 SEC /  2 SEC

AB      BC      BCA      B
ABCD      CDEF      CDEFAB      CD
ABCDEFG      WXYZ      WXYZABCDEFG
A(BC)D(EFG)H      D(EFGH)H(BC)      D(EFGH)H(BC)A(EFG)      (BC)DH


STOPPED AT LINE  5
CSC004/00000000/  CUP AND CAP
RUNNING TIME   4 SEC /  1 SEC
```

```
***A
JOB
CSC004/00000000/ INTO POLISH
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGFNSPEC INTO(%STRING R)
%STRING S
1:READ STRING(S)
%IF S='END' %THEN %STOP
WRITE STRING('--POLISH_FORM_OF-'.S.'-IS-'.INTO(S))
->1

%STRINGFN INTO(%STRING R)
%STRING S
%IF R->(S) %THEN %RESULT= INTO(S[1]).INTO(S[3]).S[2]
%RESULT=R
%END

%ENDOFPROGRAM



(X)
((X+Y))
((X+(Y*Z)))
(((X*Y)+(A*(B-C))))
((X+((A-B)*Z)))
(((X-Y)+Z))
((X+(Y+(Z+W))))
(END)
```

```
29/04/69  09.51.01
ASTRA   10/06/68


CSC004/00000000/ INTO POLISH

     0   BEGIN
     7       STRING FN  INTO
    11       END OF STRING FN
    12   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2183 WORDS
PROGRAM DUMPED
COMPILING TIME   4 SEC /  1 SEC


POLISH FORM OF
X
IS
X

POLISH FORM OF
(X+Y)
IS
XY+

POLISH FORM OF
(X+(Y*Z))
IS
XYZ*+

POLISH FORM OF
((X*Y)+(A*(B-C)))
IS
XY*ABC-*+
POLISH FORM OF
(X+((A-B)*Z))
IS
XAB-Z*+

POLISH FORM OF
((X-Y)+Z)
IS
XY-Z+

POLISH FORM OF
(X+(Y+(Z+W)))
IS
XYZW+++


STOPPED AT LINE  4
CSC004/00000000/ INTO POLISH
RUNNING TIME   4 SEC /  0 SEC
```

```
***A
JOB
CSC004/00000000/  DIFFERENTIATE
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGFNSPEC DIFF(%STRING S)
%STRINGFNSPEC EDIT(%STRING S)
%STRING S
1:READ STRING(S)
%IF S='END' %THEN %STOP
WRITE STRING('--DIFFERENTIAL_OF-'.S.'-WITH_RESPECT_TO_X_IS-')
S=DIFF(S)
WRITE STRING(S.'-I.E.-'.EDIT(S))
->1

%STRINGFN DIFF(%STRING S)
%STRING T
%IF S->(T) %THEN ->1
%IF S='X' %THEN %RESULT='1'
%RESULT='0'
1:%IF T[2]='+' %THEN %RESULT= (DIFF(T[1]).'+'.DIFF(T[3]))
%IF T[2]='-' %THEN %RESULT= (DIFF(T[1]).'-'.DIFF(T[3]))
%IF T[2]='*' %THEN %RESULT=((DIFF(T[1]).'*'.T[3]).'+'. %C
                                      (T[1].'*'.DIFF(T[3])))
%IF T[2]='/' %THEN %RESULT= (((DIFF(T[1]).'*'.T[3]).'-'. %C
                                 (T[1].'*'.DIFF(T[3]))).'/'. %C
                                      (T[3].'*'.T[3]))
%RESULT='FAULT'
%END
```

```
%STRINGFN EDIT(%STRING S)
%STRING T,U,V
%RESULT=S %UNLESS S->(T)
U=EDIT(T[1]).T[2].EDIT(T[3])
%IF U->V.'+0' %OR U->'0+'.V %THEN %RESULT=V
%IF U->V.'-0' %THEN %RESULT=V
%IF U->-.'*0' %OR U->'0*'.- %THEN %RESULT='0'
%IF U->V.'*1' %OR U->'1*'.V %THEN %RESULT=V
%IF U->'0/'.- %THEN %RESULT='0'
%IF U->V.'/1' %THEN %RESULT=V
%RESULT=(U)
%END

%ENDOFPROGRAM



(C)
(X)
((X+Y))
((X*Y))
((X/Y))
((X+(Y*Z)))
(((X*X)-(Y/(X+Z))))
(END)
```

```
29/04/69  09.51.12
ASTRA   10/06/68


CSC004/00000000/  DIFFERENTIATE

    0   BEGIN
   10      STRING FN  DIFF
   20      END OF STRING FN
   21      STRING FN  EDIT
   32      END OF STRING FN
   33   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2599 WORDS
PROGRAM DUMPED
COMPILING TIME   8 SEC /  4 SEC


DIFFERENTIAL OF
C
WITH RESPECT TO X IS
0
I.E.
0

DIFFERENTIAL OF
X
WITH RESPECT TO X IS
1
I.E.
1

DIFFERENTIAL OF
(X+Y)
WITH RESPECT TO X IS
(1+0)
I.E.
1
```

```
DIFFERENTIAL OF
(X*Y)
WITH RESPECT TO X IS
((1*Y)+(X*0))
I.E.
Y

DIFFERENTIAL OF
(X/Y)
WITH RESPECT TO X IS
(((1*Y)-(X*0))/(Y*Y))
I.E.
(Y/(Y*Y))

DIFFERENTIAL OF
(X+(Y*Z))
WITH RESPECT TO X IS
(1+((0*Z)+(Y*0)))
I.E.
1

DIFFERENTIAL OF
((X*X)-(Y/(X+Z)))
WITH RESPECT TO X IS
(((1*X)+(X*1))-(((0*(X+Z))-(Y*(1+0)))/((X+Z)*(X+Z))))
I.E.
((X+X)-((0-Y)/((X+Z)*(X+Z))))


STOPPED AT LINE  5
CSC004/00000000/  DIFFERENTIATE
RUNNING TIME   6 SEC /  1 SEC
```

```
***A
JOB
CSC004/00000000/ WANG ALGORITHM (REF. LISP 1.5 PROGRAMMERS MANUAL)
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGFNSPEC THEOREM(%STRING S)
%STRING S
1:READ STRING(S)
%IF S='END' %THEN %STOP
WRITE STRING('--PROPOSITION_IS-'.S.'-VALUE_IS_'.THEOREM(S))
->1

%STRINGFN THEOREM(%STRING S)
%STRING A1,A2,C1,C2,A,B,C
%STRINGFNSPEC TH(%STRING A1,A2,C1,C2)
    A1=_
    A2=_                                    ;! SET A1 & A2 TO NULL
    S->'->'.(A).(C)                         ;! A=ANTECEDENT, C=CONSEQUENT
3:  %IF A=_ %THEN ->1
    A->B[1].A                               ;! B FIRST FORMULA OF A
    %IF C->-.''B''.- %THEN %RESULT='T'      ;! TRUE IF B MEMBER OF C
    %IF B->(-) %THEN ->2                    ;! JUMP IF B NOT ATOMIC
    A1=B.A1 %UNLESS A1->-.''B''.-           ;! ADD B TO A1 UNLESS B IN A1
    ->3
2:  A2=B.A2 %UNLESS A2->-.''B''.-           ;! ADD B TO A2 UNLESS B IN A2
    ->3

1:  C1=_
    C2=_
5:  %IF C=_ %THEN %RESULT= TH(A1,A2,C1,C2)
    C->B[1].C
    %IF B->(-) %THEN ->4
    C1=B.C1 %UNLESS C1->-.''B''.-
    ->5
4:  C2=B.C2 %UNLESS C2->-.''B''.-
    ->5
```

```
%STRINGFN TH(%STRING A1,A2,C1,C2)
%STRING U,V,A2P,C2P
%STRINGFNSPEC THL1(%STRING V,A1,A2,C1,C2)
%STRINGFNSPEC THR1(%STRING V,A1,A2,C1,C2)
%STRINGFNSPEC THL2(%STRING V,A1,A2,C1,C2)
%STRINGFNSPEC THR2(%STRING V,A1,A2,C1,C2)
%STRINGFNSPEC TH11(%STRING V1,V2,A1,A2,C1,C2)
%STRINGFNSPEC AND(%STRING S,T)

    %IF A2=_ %THEN ->1
    A2->(U).A2P
    %IF U->'NOT'.V %THEN %RESULT= THR1(V,A1,A2P,C1,C2)
    %IF U->'AND'.V %THEN %RESULT= THL2(V,A1,A2P,C1,C2)
    %IF U->'OR'.V %THEN %RESULT= AND(THL1(V[1],A1,A2P,C1,C2),%C
                                        THL1(V[2],A1,A2P,C1,C2))
    %IF U->'IMPLIES'.V %THEN %RESULT= AND(THR1(V[1],A1,A2P,C1,C2),%C
                                        THL1(V[2],A1,A2P,C1,C2))
    %IF U->'EQUIV'.V %THEN %RESULT= AND(THL2(V,A1,A2P,C1,C2),%C
                                        THR2(V,A1,A2P,C1,C2))
    %CAPTION -- FAULT _ 1 --
    %STOP

1:  %IF C2=_ %THEN %RESULT= 'F'
    C2->(U).C2P
    %IF U->'NOT'.V %THEN %RESULT= THL1(V,A1,A2,C1,C2P)
    %IF U->'AND'.V %THEN %RESULT= AND(THR1(V[1],A1,A2,C1,C2P),%C
                                        THR1(V[2],A1,A2,C1,C2P))
    %IF U->'OR'.V %THEN %RESULT= THR2(V,A1,A2,C1,C2P)
    %IF U->'IMPLIES'.V %THEN %RESULT= TH11(V[1],V[2],A1,A2,C1,C2P)
    %IF U->'EQUIV'.V %THEN %RESULT= AND(TH11(V[1],V[2],A1,A2,C1,C2P),%C
                                        TH11(V[2],V[1],A1,A2,C1,C2P))
    %CAPTION -- FAULT _ 2 --
    %STOP
```

```
%STRINGFN THL1(%STRING V,A1,A2,C1,C2)
    %IF V->(-) %THEN ->1
    %IF C1->-.''V''.- %THEN %RESULT= 'T'
    %RESULT= TH(V.A1,A2,C1,C2)
1:  %IF C2->-.''V''.- %THEN %RESULT= 'T'
    %RESULT= TH(A1,V.A2,C1,C2)
%END

%STRINGFN THR1(%STRING V,A1,A2,C1,C2)
    %IF V->(-) %THEN ->1
    %IF A1->-.''V''.- %THEN %RESULT= 'T'
    %RESULT= TH(A1,A2,V.C1,C2)
1:  %IF A2->-.''V''.- %THEN %RESULT= 'T'
    %RESULT= TH(A1,A2,C1,V.C2)
%END

%STRINGFN THL2(%STRING V,A1,A2,C1,C2)
    %IF V->(-).- %THEN ->1
    %IF C1->-.''V[1]''.- %THEN %RESULT= 'T'
    %RESULT= THL1(V[2],V[1].A1,A2,C1,C2)
1:  %IF C2->-.''V[1]''.- %THEN %RESULT= 'T'
    %RESULT= THL1(V[2],A1,V[1].A2,C1,C2)
%END

%STRINGFN THR2(%STRING V,A1,A2,C1,C2)
    %IF V->(-).- %THEN ->1
    %IF A1->-.''V[1]''.- %THEN %RESULT= 'T'
    %RESULT= THR1(V[2],A1,A2,V[1].C1,C2)
1:  %IF A2->-.''V[1]''.- %THEN %RESULT= 'T'
    %RESULT= THR1(V[2],A1,A2,C1,V[1].C2)
%END
```

```
%STRINGFN TH11(%STRING V1,V2,A1,A2,C1,C2)
    %IF V1->(-) %THEN ->1
    %IF C1->-.''V1''.- %THEN %RESULT= 'T'
    %RESULT= THR1(V2,V1.A1,A2,C1,C2)
1:  %IF C2->-.''V1''.- %THEN %RESULT= 'T'
    %RESULT= THR1(V2,A1,V1.A2,C1,C2)
%END

%STRINGFN AND(%STRING S,T)
    %IF S='T' %AND T='T' %THEN %RESULT= 'T'
    %RESULT= 'F'
%END

%END

%END

%ENDOFPROGRAM



(->(P)((ORPQ)))
(->((ORA(NOTB)))((IMPLIES(ANDPQ)(EQUIVPQ))))
(END)
```

```
29/04/69   09.51.31
ASTRA   10/06/68

CSC004/00000000/ WANG ALGORITHM (REF. LISP 1.5 PROGRAMMERS MANUAL)

     0   BEGIN
     7      STRING FN  THEOREM
    30         STRING FN  TH
    56            STRING FN  THL1
    62            END OF STRING FN
    63            STRING FN  THR1
    69            END OF STRING FN
    70            STRING FN  THL2
    76            END OF STRING FN
    77            STRING FN  THR2
    83            END OF STRING FN
    84            STRING FN  TH11
    90            END OF STRING FN
    91            STRING FN  AND
    94            END OF STRING FN
    95         END OF STRING FN
    96      END OF STRING FN
    97   END OF PROGRAM
PROGRAM (+PERM) OCCUPIES 3467 WORDS
PROGRAM DUMPED
COMPILING TIME   19 SEC /  14 SEC

PROPOSITION IS
->(P)((ORPQ))
VALUE IS T

PROPOSITION IS
->((ORA(NOTB)))((IMPLIES(ANDPQ)(EQUIVPQ)))
VALUE IS T

STOPPED AT LINE  4
CSC004/00000000/ WANG ALGORITHM (REF. LISP 1.5 PROGRAMMERS MANUAL)
RUNNING TIME   2 SEC /  1 SEC
```

APPENDIX B

Examples of ASTRA list structures

The list structures shown below are presented in the form produced from a routine built-into the ASTRA permanent material for diagnostic purposes. They are preceded by the programs which generated the strings being displayed. The routine in question is called 'show'.

Four quantities are printed out for each cell in the representation of the string value. They are :

machine address of cell
information field
association list link
list link

A typical cell might be :

```
2221   A    0  2222
```

where the machine address of the cell is 2221, the information is the symbol A, there is no association list attached to the cell, and the machine address of the next cell in the representation is 2222.

A typical substring pointer cell might be :

```
2243   2242  2236  2234
```

where the machine addresses 2242 and 2236 are the first and last cells of the substring being pointed to.

A dummy cell has the information field zero. Links to association lists are negative. The association list itself appears to the right of the cell to which it is attached. For example :

```
2256   C -2234  2258     2234   7226 -2231  0    2231   7225  0  0
```

The addresses 7226 and 7225 are the addresses of the variables on the run-time stack which have been made to refer to this cell in the representation by means of resolution statements.

Cells which form part of substrings are indented.

```
***A
JOB
CSC004/00000000/ SHOW LISTS
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRING R,S,T,U
R='ASTRA'
SHOW(R,'R')
S='NEW'.('INGTON').'1011'
SHOW(S,'S')
S->-.(T).-
SHOW(T,'T')
R=('A*B').'+'.('C*'.('D-E'))
R->(S).-.(T)
T->U[2].-
SHOW(R,'R')
%ENDOFPROGRAM
```

```
29/04/69  09.51.58
ASTRA   10/06/68


CSC004/00000000/ SHOW LISTS

      0   BEGIN
     12   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2243 WORDS
PROGRAM DUMPED
COMPILING TIME   3 SEC / 1 SEC


R    ASTRA

 7223   2221  2226  0

 2221   A     0  2222
 2222   S     0  2223
 2223   T     0  2224
 2224   R     0  2225
 2225   A     0  2226
 2226   0     0     0

S    NEW(INGTON)1011

 7224   2228  2229  0

 2228   N     0  2227
 2227   E     0  2244
 2244   W     0  2243
 2243   2242  2236  2234
    2242   I     0  2241
    2241   N     0  2240
    2240   G     0  2239
    2239   T     0  2238
    2238   O     0  2237
```

```
      2237   N      0  2236
      2236   0      0     0
   2234   1     0  2233
   2233   0     0  2231
   2231   1     0  2230
   2230   1     0  2229
   2229   0     0     0

  T    INGTON

   7225   2242  2236  1

  R    (A*B)+(C*(D-E))

   7223   2254  2267  0

   2254   2269  2253  2255
      2269   A -2229  2270      2229   7224   0  0
      2270   *     0  2252
      2252   B     0  2253
      2253   0 -2230     0      2230   7224   0  0
   2255   +     0  2257
   2257   2256  2266  2267
      2256   C -2234  2258      2234   7226 -2231  0      2231   7225   0  0
      2258   *     0  2259
      2259   0 -2237  2236      2237   7226   0  0
      2236   2260  2263  2266
         2260   D     0  2261
         2261   -     0  2262
         2262   E     0  2263
         2263   0     0     0
     2266   0 -2233     0      2233   7225   0  0
  2267   0     0     0


STOPPED AT LINE  12
CSC004/00000000/ SHOW LISTS
RUNNING TIME   12 SEC /  1 SEC
```

```
***A
JOB
CSC004/00000000/  DIFFERENTIATE
OUTPUT 0 EIGHT-HOLE PUNCH 10 BLOCKS
COMPILER AS

%BEGIN
%STRINGFNSPEC DIFF(%STRING S)
%STRINGFNSPEC EDIT(%STRING S)
%STRING S
1:READ STRING(S)
%IF S='END' %THEN %STOP
WRITE STRING('--DIFFERENTIAL_OF-'.S.'-WITH_RESPECT_TO_X_IS-')
S=DIFF(S)
WRITE STRING(S.'-I.E.-'.EDIT(S))
->1

%STRINGFN DIFF(%STRING S)
%STRING T
%IF S->(T) %THEN ->1
%IF S='X' %THEN %RESULT='1'
%RESULT='0'
1:SHOW(S,'S')
NEWLINES(2)
%IF T[2]='+' %THEN %RESULT= (DIFF(T[1]).'+'.DIFF(T[3]))
%IF T[2]='-' %THEN %RESULT= (DIFF(T[1]).'-'.DIFF(T[3]))
%IF T[2]='*' %THEN %RESULT=((DIFF(T[1]).'*'.T[3]).'+'. %C
                                        (T[1].'*'.DIFF(T[3])))
%IF T[2]='/' %THEN %RESULT= (((DIFF(T[1]).'*'.T[3]).'-'. %C
                                   (T[1].'*'.DIFF(T[3]))).'/'. %C
                                        (T[3].'*'.T[3]))
%RESULT='FAULT'
%END
```

```
%STRINGFN EDIT(%STRING S)
%STRING T,U,V
%RESULT=S %UNLESS S->(T)
U=EDIT(T[1]).T[2].EDIT(T[3])
%IF U->V.'+0' %OR U->'0+'.V %THEN %RESULT=V
%IF U->V.'-0' %THEN %RESULT=V
%IF U->-.'*0' %OR U->'0*'.- %THEN %RESULT='0'
%IF U->V.'*1' %OR U->'1*'.V %THEN %RESULT=V
%IF U->'0/'.- %THEN %RESULT='0'
%IF U->V.'/1' %THEN %RESULT=V
%RESULT=(U)
%END

%ENDOFPROGRAM
```

```
(C)
(X)
((X+Y))
((X*Y))
((X/Y))
((X+(Y*Z)))
(((X*X)-(Y/(X+Z))))
(END)
```

```
30/04/69   23.56.07
ASTRA   10/06/68


CSC004/00000000/  DIFFERENTIATE

    0   BEGIN
   10       STRING FN  DIFF
   21       END OF STRING FN
   22       STRING FN  EDIT
   33       END OF STRING FN
   34   END OF PROGRAM

PROGRAM (+PERM) OCCUPIES 2607 WORDS
PROGRAM DUMPED
COMPILING TIME   7 SEC /  4 SEC


DIFFERENTIAL OF
C
WITH RESPECT TO X IS
0
I.E.
0

DIFFERENTIAL OF
X
WITH RESPECT TO X IS
1
I.E.
1

DIFFERENTIAL OF
(X+Y)
WITH RESPECT TO X IS
```

```
S    (X+Y)

 7577   2624  2619  0

 2624   2623  2620  2619
    2623   X -2618  2622      2618   7578   0  0
    2622   +     0  2621
    2621   Y     0  2620
    2620   0 -2617     0      2617   7578   0  0
 2619   0     0     0

(1+0)
I.E.
1

DIFFERENTIAL OF
(X*Y)
WITH RESPECT TO X IS


S    (X*Y)

 7577   2598  2575  0

 2598   2586  2582  2575
    2586   X -2576  2585      2576   7578   0  0
    2585   *     0  2579
    2579   Y     0  2582
    2582   0 -2581     0      2581   7578   0  0
 2575   0     0     0

((1*Y)+(X*0))
I.E.
Y

DIFFERENTIAL OF
(X/Y)
WITH RESPECT TO X IS
```

```
S    (X/Y)

 7577  2602  2597  0

 2602  2603  2601  2597
    2603   X -2606  2590    2606  7578  0  0
    2590   /     0  2604
    2604   Y     0  2601
    2601   0 -2612     0    2612  7578  0  0
 2597   0     0     0

(((1*Y)-(X*0))/(Y*Y))
I.E.
(Y/(Y*Y))

DIFFERENTIAL OF
(X+(Y*Z))
WITH RESPECT TO X IS


S    (X+(Y*Z))

 7577  2643  2601  0

 2643  2654  2597  2601
    2654   X -2604  2598    2604  7578  0  0
    2598   +     0  2581
    2581  2576  2607  2597
       2576   Y    0  2591
       2591   *    0  2578
       2578   Z    0  2607
       2607   0    0     0
    2597   0 -2590     0    2590  7578  0  0
 2601   0     0     0
```

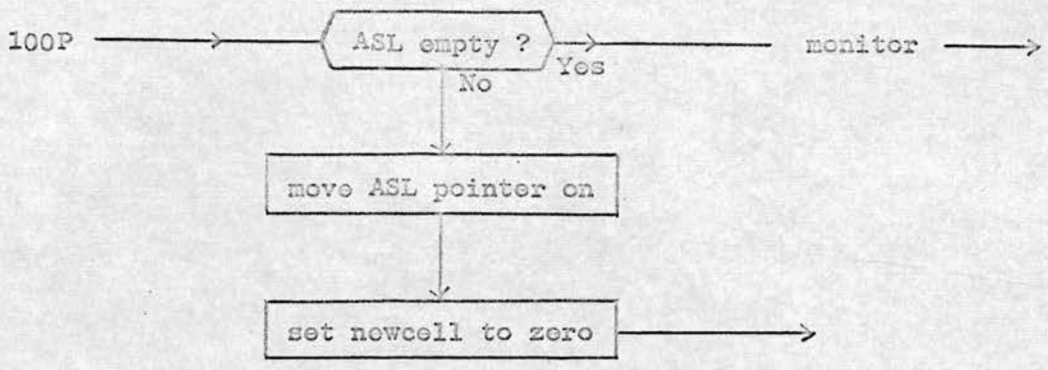
```
S    (Y*Z)

 7582   2615  2575  0

 2615   2602  2582  2575
    2602   Y -2608  2603      2608   7583   0  0
    2603   *     0  2584
    2584   Z     0  2582
    2582   0 -2614     0      2614   7583   0  0
 2575   0      0     0

(1+((0*Z)+(Y*0)))
I.E.
1

DIFFERENTIAL OF
((X*X)-(Y/(X+Z)))
WITH RESPECT TO X IS


S    ((X*X)-(Y/(X+Z)))

 7577   2630  2615  0

 2630   2629  2602  2615
    2629  0 -2608  2614      2608   7578   0  0
    2614   2627  2626  2646
       2627   X     0  2639
       2639   *     0  2625
       2625   X     0  2626
       2626   0     0     0
    2646   -     0  2636
    2636   2633  2654  2602
       2633   Y     0  2635
       2635   /     0  2634
       2634   2641  2642  2654
```

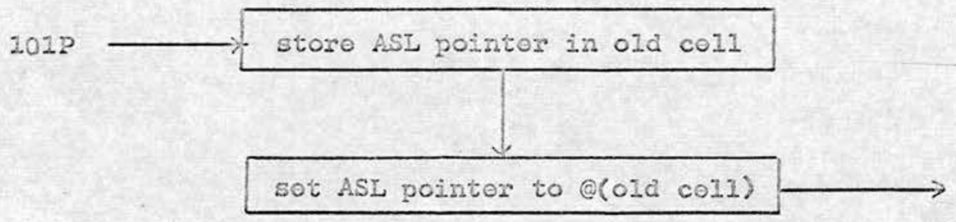
```
            2641   X     0  2618
            2618   +     0  2577
            2577   Z     0  2642
            2642   0     0     0
         2654   0     0     0
      2602   0 -2599     0      2599   7578   0  0
   2615   0     0     0

   S    (X*X)

   7581   2622  2576  0

   2622  0     0  2610
   2610   2603  2581  2576
      2603   X -2591  2643      2591   7582   0  0
      2643   *     0  2598
      2598   X     0  2581
      2581   0 -2578     0      2578   7582   0  0
   2576   0      0     0

   S    (Y/(X+Z))

   7582   2576  2592  0

   2576   2598  2595  2592
      2598   Y -2647  2643      2647   7583   0  0
      2643   /     0  2603
      2603   2610  2591  2595
         2610   X     0  2622
         2622   +     0  2578
         2578   Z     0  2591
         2591   0     0     0
      2595   0 -2655     0      2655   7583   0  0
   2592   0     0     0
```

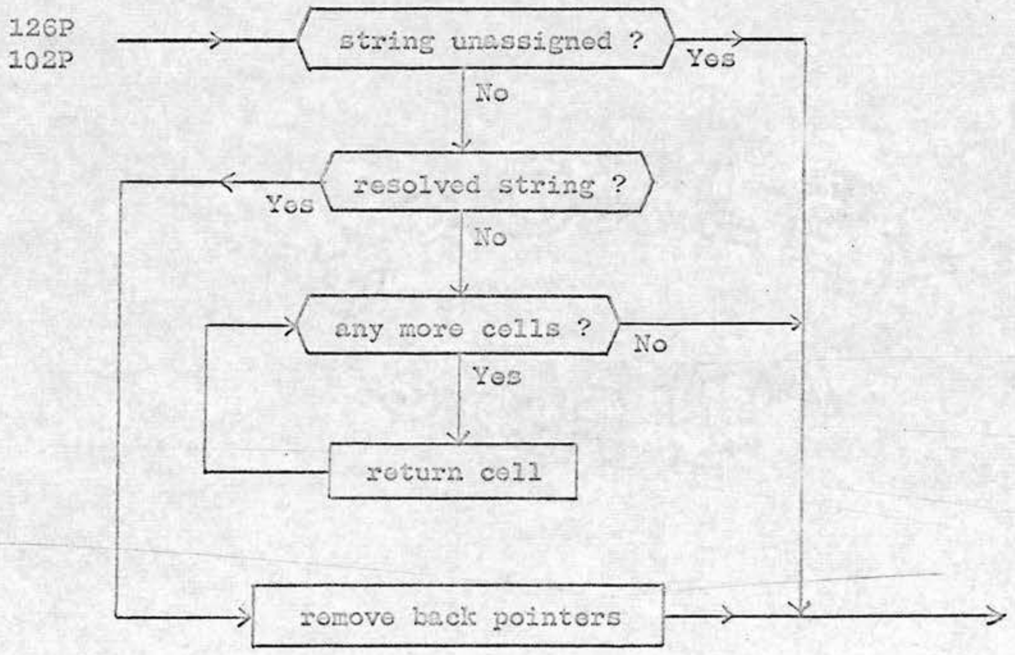
```
S    (X+Z)

 7588   2582  2620  0

 2582   2612  2575  2620
    2612   X -2574  2580    2574   7589   0  0
    2580   +     0  2606
    2606   Z     0  2575
    2575   0 -2616     0    2616   7589   0  0
 2620   0      0     0

(((1*X)+(X*1))-(((0*(X+Z))-(Y*(1+0)))/((X+Z)*(X+Z))))
I.E.
((X+X)-((0-Y)/((X+Z)*(X+Z))))


STOPPED AT LINE  5
CSC004/00000000/  DIFFERENTIATE
RUNNING TIME   32 SEC /  5 SEC
```

APPENDIX C

Basic operations

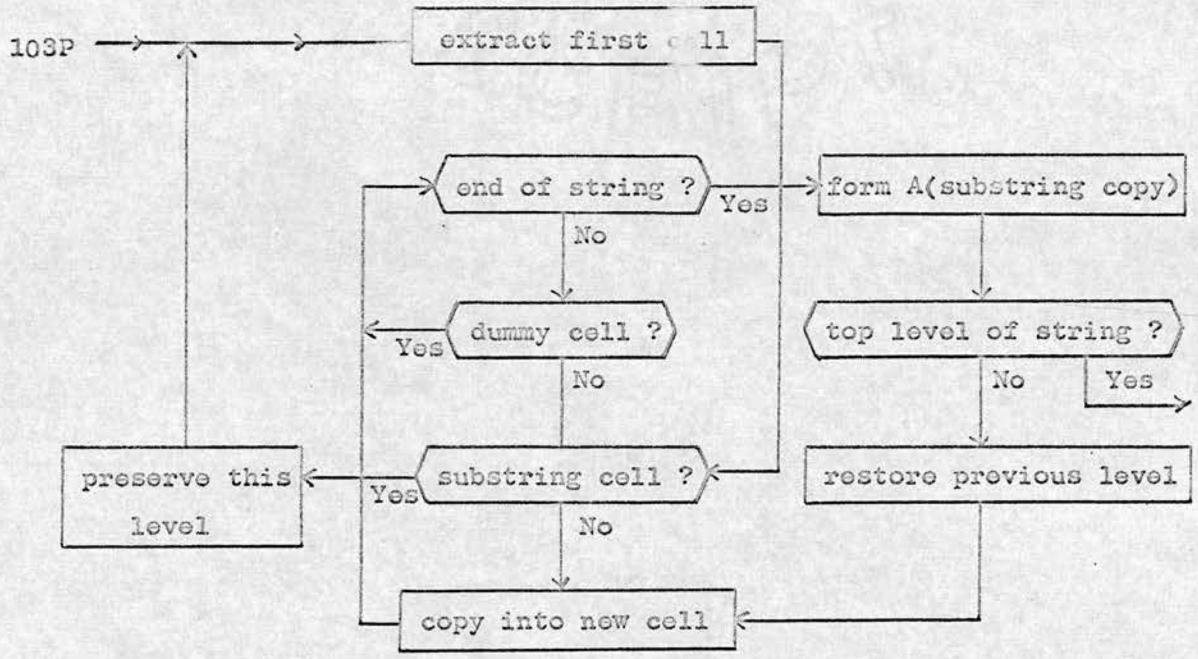

100P
ASL empty ?
Yes
monitor
No
move ASL pointer on
set newcell to zero


101P
store ASL pointer in old cell
set ASL pointer to @(old cell)


126P
102P
string unassigned ?
Yes
No
resolved string ?
Yes
No
any more cells ?
No
Yes
return cell
remove back pointers

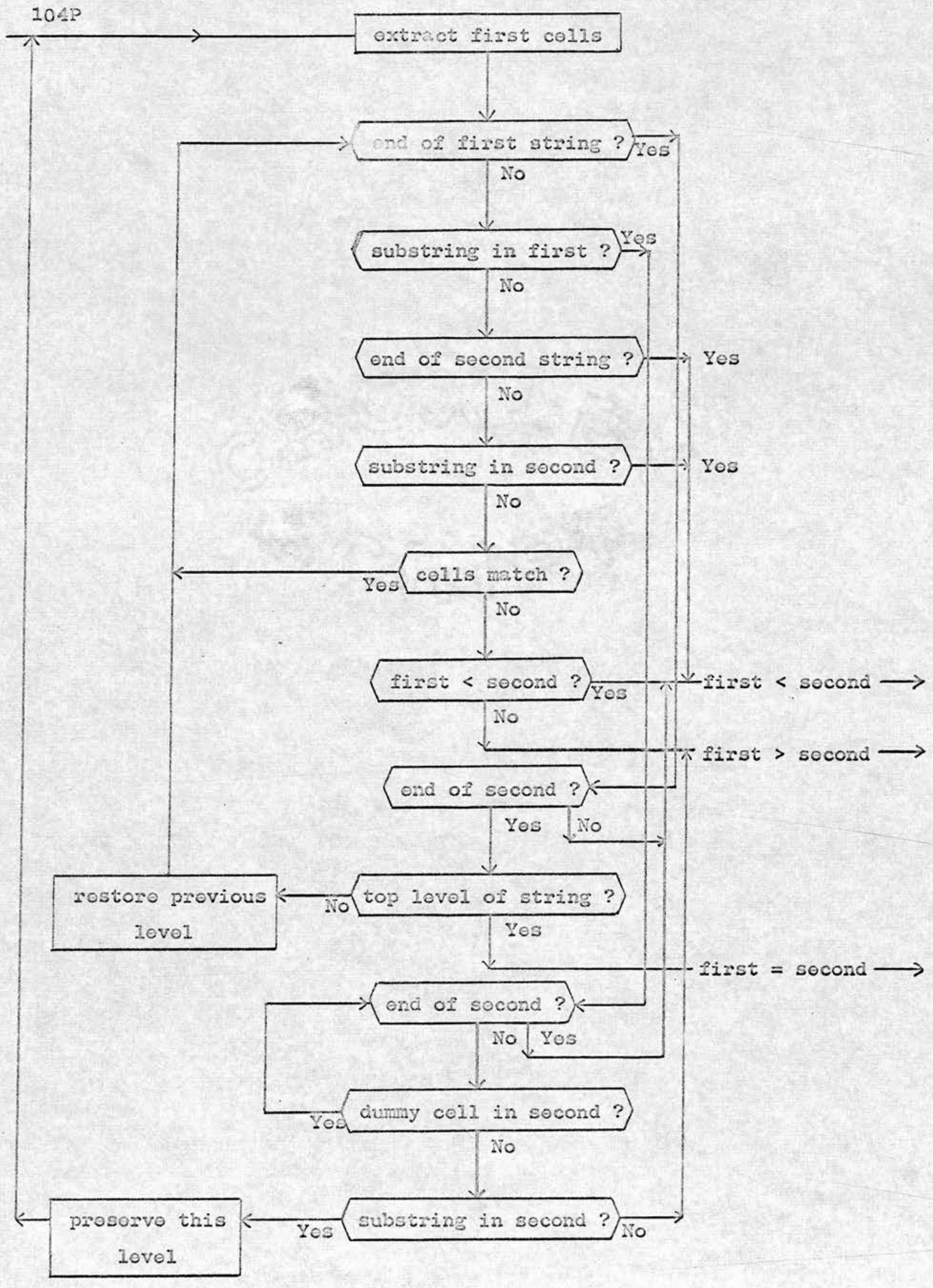
```
100P:   |NEWCELL

        *M13          ;|PRESERVE
        *I3           ;|ASL POINTER
        *DUP
        *J 10000=Z    ;|ASL EMPTY
        *DUP
        *=M13
        *MOM13
        *=I3          ;|NEXT CELL OF ASL
        *ZERO
        *=MOM13       ;|SET NEW CELL TO ZERO
        *REV
        *=M13         ;|RESTORE
        *EXIT 1       ;|N1=POINTER TO NEWCELL


10000:  %CAPTION --ASL_EMPTY
        %STOP

101P:   |RETURN CELL
        |N1=POINTER TO CELL BEING RETURNED

        *M13          ;|PRESERVE
        *REV
        *DUP
        *=M13
        *I3           ;|CURRENT ASL POINTER
        *=MOM13       ;|PUSHDOWN
        *=I3          ;|NEW ASL POINTER
        *=M13         ;|RESTORE
        *EXIT 1


126P:   | RETURN STRING FROM COND
        | N1=A(STRING)
        *=MOM12
        *M12
```

```
102P:   ! RETURN STRING TO ASL
10200:  !N1=@(A(STRING))
        *=M13
        *MOM13        ;! A(STRING)
        *DUP
        *J 10201=Z    ;! STRING UNASSIGNED
        *SHL+1
        *SHL-1
        *DUP
        *SHL+32
        *SHL-32
        *NEG
        *NOT
        *J 10202=Z    ;! RESOLVED STRING
        *SHL-32
        *=M14         ;! ADDR( CURRENT S CELL )

10203:  *M14
        *DUP
        *J 10201=Z    ;! LAST LINK =0
        *MOM14
        *=Q14         ;! NEXT CELL
        *JS 15100     ;! RETURN CELL (COMPLEX)
        ->10203

10202:  *ZERO         ;! REMOVE BACK POINTERS
        *SHLD+16
        *M13
        *JS 15200     ;! REMOVE FIRST
        *SHL-32
        *M13
        *JS 15200     ;! REMOVE SECOND
        *ZERO
10201:  *=MOM13       ;! SET STRING UNASSIGNED
        *EXIT 1
```



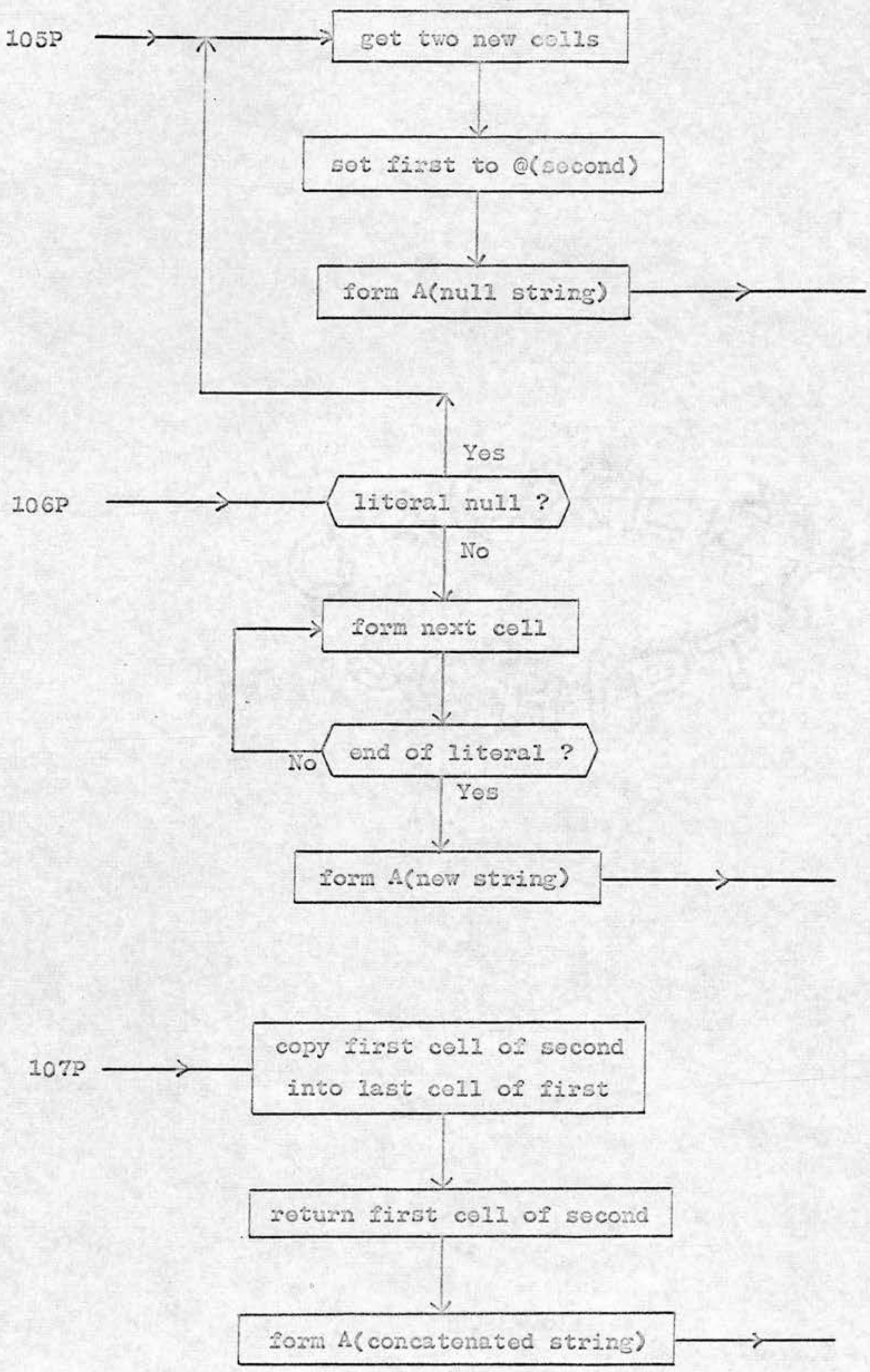
103P
extract first cell
end of string ?
Yes
No
form A(substring copy)
dummy cell ?
Yes
No
top level of string ?
No
Yes
preserve this level
substring cell ?
Yes
No
restore previous level
copy into new cell

```
103P:   !COPY STRING
        !N1=A(STRING TO BE COPIED)

        *C0 TO Q12           ;!DEPTH OF SUBSTRING COUNTER
10306:  *ZERO
        *SHLD+16
        *=M14                ;!BEG. OF STRING
        *MOM14
        *=Q14                ;!FIRST CELL TO BE COPIED
        *=Q13                ;!END OF STRING IN C13
        *JS 100P
        *DUP
        *=I13                ;!PRESERVE START OF COPY
        *=M13                ;!CURRENT END OF COPY
        ->10301              ;!ALWAYS COPY FIRST CELL
10303:  *M14
        *C13
        *-
        *J 10302=Z           ;! END OF STRING
        *MOM14
        *=Q14                ;! NEXT CELL TO BE COPIED
        *J 10303 C14 Z       ;! IGNORE DUMMIES
10301:  *I14
        *J 10304>Z           ;! SUBSTRING
        *C14 TO Q15          ;! COPY INFO.
        *ZERO
        *=I15
10305:  *JS 100P
        *DUP
        *=M15                ;! POINTER TO NEXT CELL
        *Q15
        *=MOM13              ;! DUMP COPIED CELL
        *=M13                ;! CURRENT END OF COPY
        ->10303
10304:  *Q13                 ;! STACK INFO. FOR COPYING SUBSTRING
        *=MOM12Q
        *M14
        *=MOM12Q
        *Q14
        ->10306              ;! GO TO COPY SUBSTRING
10302:  *ZERO
        *=MOM13              ;! LAST CELL OF COPY=0
        *Q13
        *SHL+16              ;! A(COPY OR SUBSTRING)
        *J 10307 C12Z        ;! TOP LEVEL OF SUBSTRINGS
        *=Q15                ;! A(SUBSTRING)
        *M-I12               ;! UNSTACK INFO.
        *MOM12
        *=M14
        *M-I12
        *MOM12
        *=Q13
        *SET 2               ;! SET DEPTH COUNTER BACK
        *=+C12
        ->10305

10307:  *EXIT 1              ;! N1=A(COPY)
```

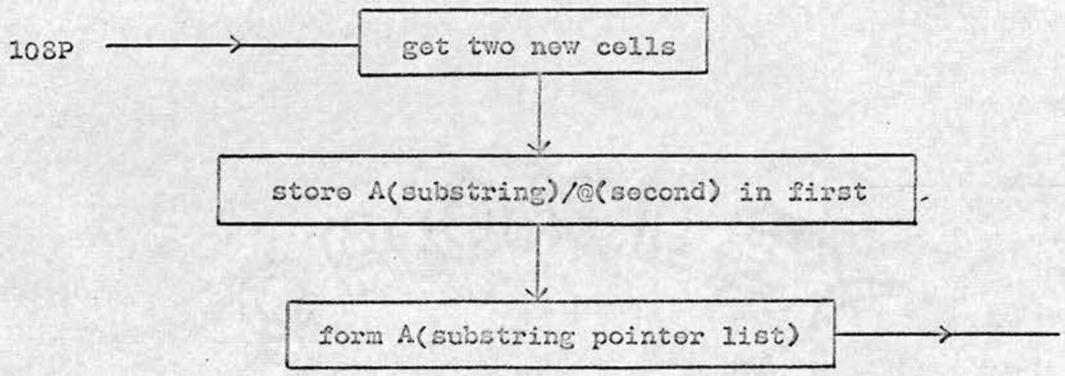

104P
extract first cells
end of first string ?
Yes
No
substring in first ?
Yes
No
end of second string ?
Yes
No
substring in second ?
Yes
No
cells match ?
Yes
No
first < second ?
Yes
No
first < second
first > second
end of second ?
Yes
No
restore previous level
No
top level of string ?
Yes
first = second
end of second ?
No
Yes
dummy cell in second ?
Yes
No
preserve this level
Yes
substring in second ?
No

```
104P:   |COMPARE 2 STRINGS
        |N1=A(FIRST STRING)
        |N2=A(SECOND STRING)

        *C0 TO Q12              ;| SUBSTRING DEPTH COUNTER

10408:  *ZERO
        *SHLD+16
        *=M14                   ;| CURRENT POSN. IN FIRST
        *SHL-32                 ;| LAST CELL IN FIRST
        *REV
        *ZERO
        *SHLD+16
        *=M15                   ;| CURRENT POSN. IN SECOND
        *SHL-32                 ;| LAST CELL IN SECOND

10409:  *REV

10402:  *M14
        *J 10401=               ;| END OF FIRST
        *M0M14
        *=Q14                   ;| NEXT CELL IN FIRST
        *J 10402 C14Z           ;| DUMMY
        *REV
        *I14
        *J 10403>Z              ;| SUBSTRING IN FIRST

10405:  *M15
        *J 10404=               ;| END OF SECOND
        *M0M15
        *=Q15                   ;| NEXT CELL IN SECOND
        *J 10405 C15Z           ;| DUMMY
        *I15
        *J 10404>Z              ;| SUBSTRING IN SECOND
        *C14
        *C15
        *-
        *J 10409=Z              ;| INFO. MATCHES
        *C14
        *C15
        *-
        *J 10412<Z              ;| FIRST<SECOND
10404:  *ERASE
        *ERASE
        *C12
        *=+M12                  ;| RESTORE WORKSPACE PTR
        *ZERO
        *NOT
        *EXIT 1                 ;| SECOND<FIRST, N1=-1
10412:  *ERASE
10410:  *ERASE
        *C12
        *=+M12
        *I12
        *EXIT 1                 ;| SECOND>FIRST, N1=1
```

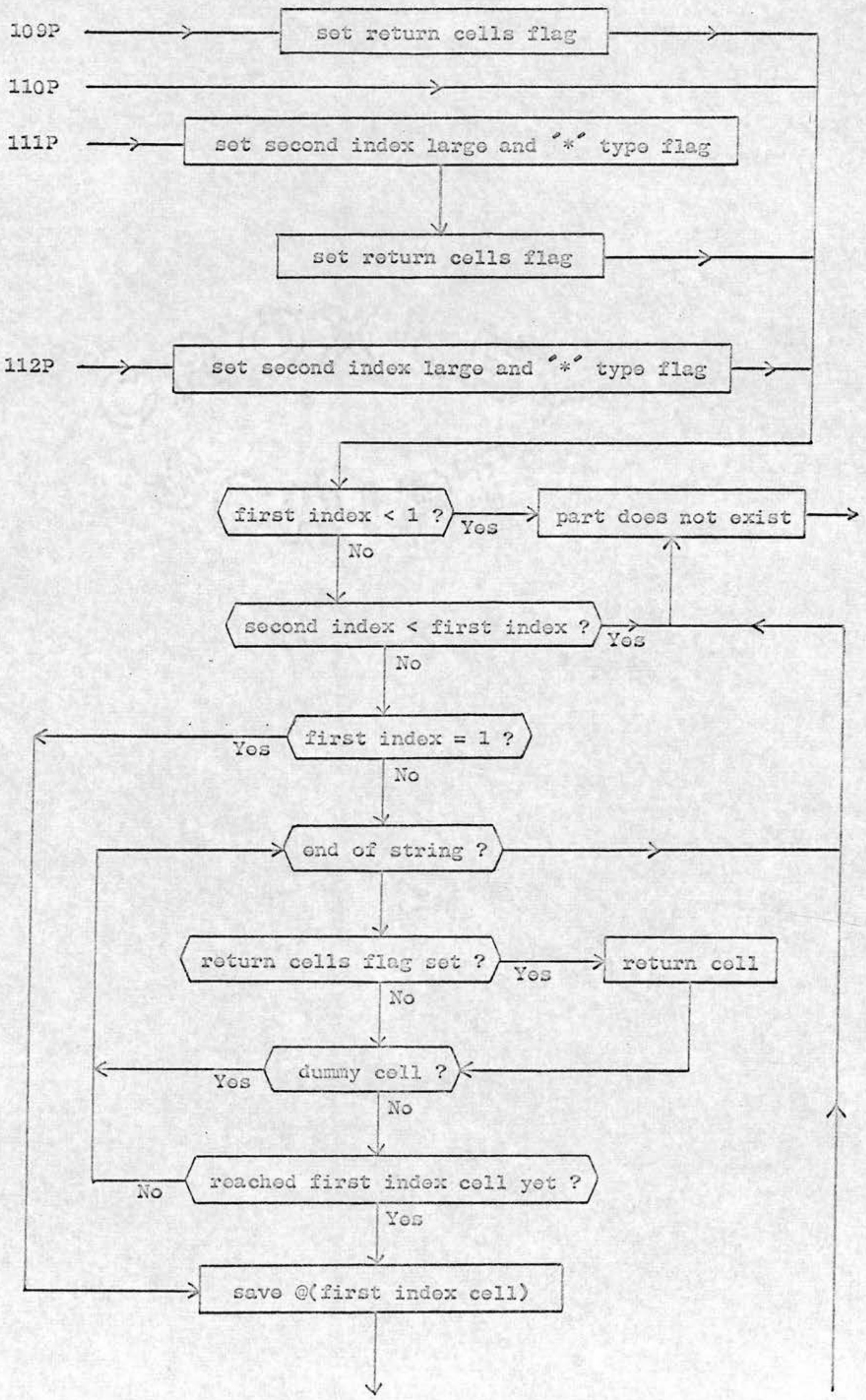
```
10401:  *ERASE

10407:  *M15
        *J 10406=             ;! END OF SECOND
        *MOM15
        *=Q15
        * J 10407 C15Z        ;! DUMMY
        ->10410

10406:  *ERASE
        *J 10411 C12 NZ       ;! SUBSTRING MATCHED
        *ZERO
        *EXIT 1               ;! SECOND=FIRST, N1=0

10411:  *M-I12                ;! RESTORE INFO.
        *MOM12
        *DUP
        *=M15
        *SHL-16
        *M-I12
        *MOM12
        *DUP
        *=M14
        *SHL-16
        *SET 2
        *=+C12
        ->10402

10403:  *M15                  ;! SHOULD BE SUBSTRING
        *J 10404=             ;! END OF SECOND
        *MOM15
        *=Q15                 ;! NEXT CELL IN SECOND
        *J 10403 C15 Z        ;! DUMMY
        *I15
        *J 10412<=Z           ;! NOT SUBSTRING
        *REV
        *SHL+16
        *M14
        *OR
        *=MOM12Q              ;! STORE RETURN INFO.
        *SHL+16
        *M15
        *OR
        *=MOM12Q              ;! STORE RETURN INFO.

        *Q15                  ;! A'(SECOND SUBSTRING)
        *Q14                  ;! A'(FIRST SUBSTRING)
        ->10408               ;! COMPARE
```

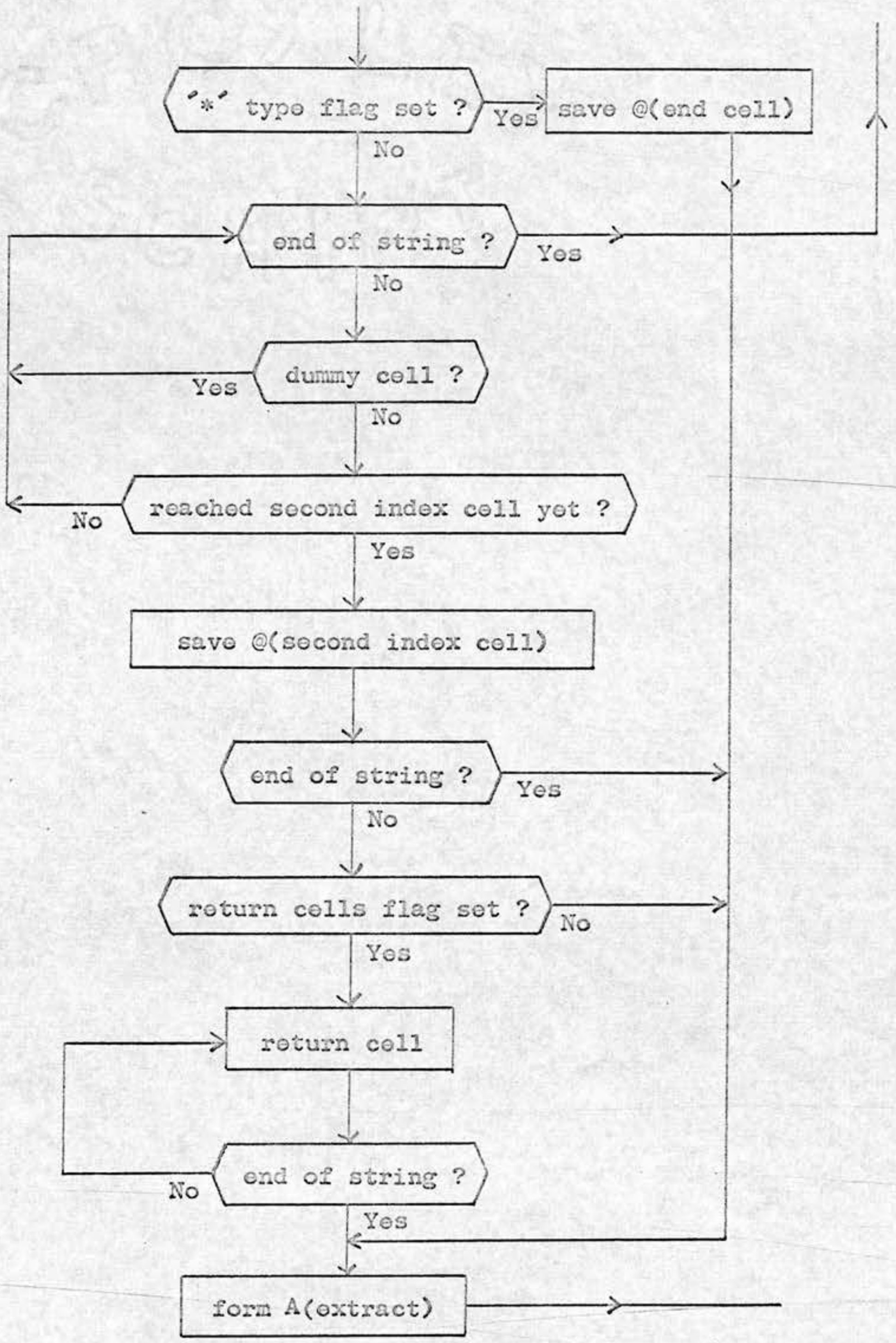

105P
get two new cells
set first to @(second)
form A(null string)
Yes
106P
literal null ?
No
form next cell
No
end of literal ?
Yes
form A(new string)
107P
copy first cell of second into last cell of first
return first cell of second
form A(concatenated string)

```
105P:   ! FORM NULL STRING

        *JS 100P
        *DUP
        *JS 100P
        *DUP
        *=M13
        *REV
        *=MOM13
        *SHL+16
        *OR
        *SHL+16
        *EXIT 1             ;! N1=A(NULL STRING)


106P:   !FORM STRING FROM LITERALS
        !N1=POSN. IN STACK

        *I11                ;! @ST(0)
        *+
        *=RM14
        *MOM14
        *=C14               ;! NO. OF WORDS
        *J 105P C14Z
        *JS 100P
        *DUP
        *=I15               ;!PRESERVE FIRST CELL
        *=M15

10603:  *MOM14QN            ;! NEXT WORD

10602:  *ZERO
        *SHLD+8             ;! NEXT SYMBOL
        *DUP
        *J 10601=Z          ;! END OF WORD
        *SHL+32
        *JS 100P
        *DUP
        *PERM
        *OR
        *=MOM15             ;! DUMP CELL
        *=M15               ;! NEXT CELL
        ->10602

10601:  *ERASE
        *ERASE
        *J 10603 C14 NZ;! MORE WORDS
        *Q15
        *SHL+16
        *EXIT 1             ;! N1=A(STRING)
```


108P
get two new cells
store A(substring)/@(second) in first
form A(substring pointer list)

```
107P:   |CONCATENATE 2 STRINGS
        |N1=A(SECOND STRING)
        |N2=A(FIRST STRING)
        |NEW STRINGS - NO POINTER SUBCHAINS

        *ZERO
        *SHLD+16
        *=M15           ;| FIRST OF SECOND
        *REV
        *SHL-16
        *DUP
        *=M14           ;| LAST OF FIRST
        *SHL-16         ;| FIRST OF FIRST
        *OR
        *SHC-16         ;| A(CONC. STRINGS)
        *MOM15
        *=MOM14         ;| COPY FIRST OF 2ND INTO LAST OF 1ST
        *M15
        *JS 101P        ;| RETURN FIRST OF 2ND
        *EXIT 1         ;| N1=A(CONCATENATED STRINGS)


108P:   | FORM SUBSTRING LIST
        | N1=A(SUBSTRING)
        | NEW STRING

        *JS 100P
        *=M13
        *JS 100P
        *DUP
        *PERM
        *OR
        *=MOM13
        *M13
        *SHL+16
        *OR
        *SHL+16
        *EXIT 1         ;| N1=A(SUBSTRING LIST)
```

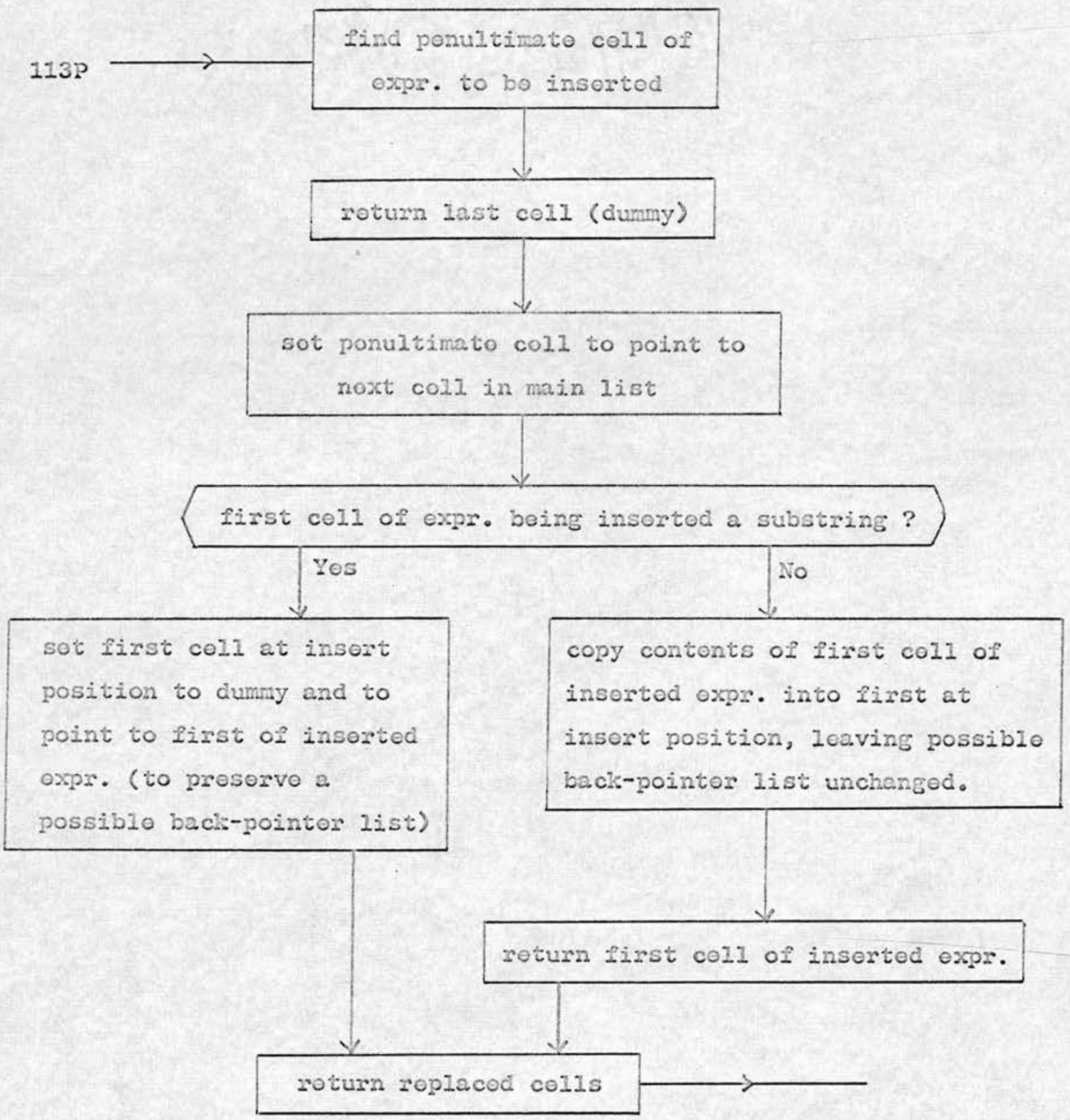

109P
set return cells flag
110P
111P
set second index large and "*" type flag
set return cells flag
112P
set second index large and "*" type flag
first index < 1 ?
Yes
part does not exist
No
second index < first index ?
Yes
No
first index = 1 ?
Yes
No
end of string ?
return cells flag set ?
Yes
return cell
No
dummy cell ?
Yes
No
reached first index cell yet ?
No
Yes
save @(first index cell)



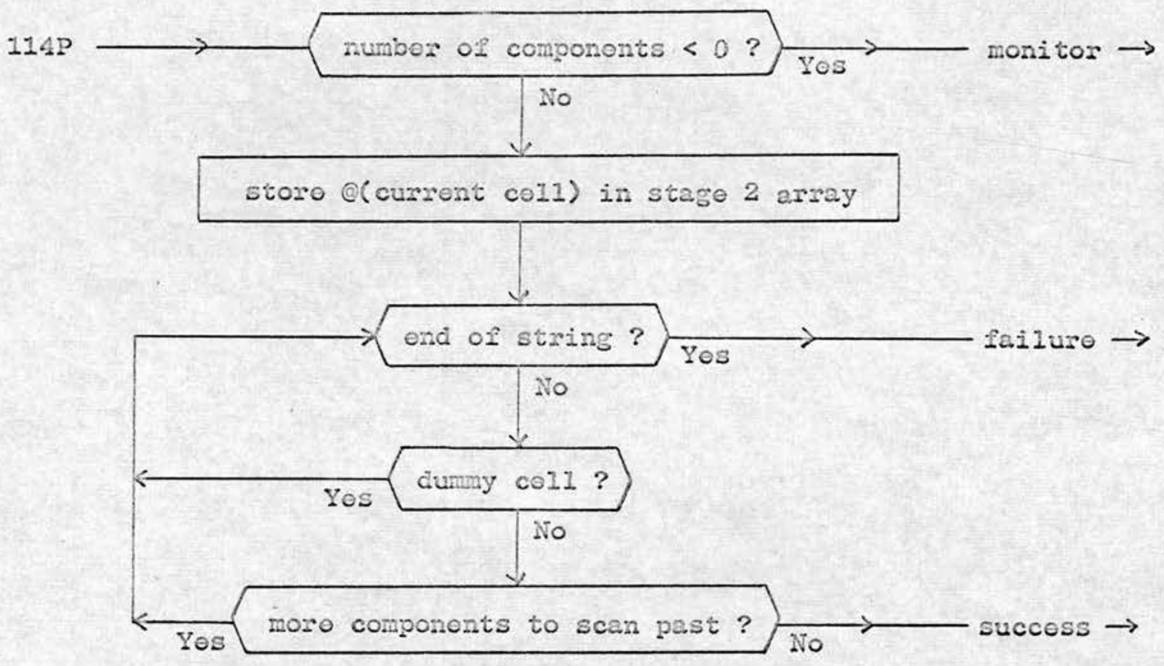
'*' type flag set ?
Yes
save @(end cell)
No
end of string ?
Yes
No
dummy cell ?
Yes
No
reached second index cell yet ?
No
Yes
save @(second index cell)
end of string ?
Yes
No
return cells flag set ?
No
Yes
return cell
end of string ?
No
Yes
form A(extract)

```
109P:   !EXTRACT PART OF STRING-[P:Q]
        !N1=Q
        !N2=P
        !N3=A(STRING)
        !NEW STRING- NO POINTER SUBCHAINS

        *Q0 TO Q15
        *DC15
        *REV
        ->11000


110P:   !GET A(EXTRACT)
        !N1,N2,N3, AS 109P

        *Q0 TO Q15
        *REV
        ->11000

111P:   ! EXTRACT - [P:*]
        ! N1=P
        ! N2=A(STRING)
        ! NEW STRING

        *ZERO
        *=RC15
        *DC15
        *SET 32767
        *REV
        ->11000

112P:   ! GET A(EXTRACT)
        ! N1,N2 AS 111P:

        *ZERO
        *=RC15
        *SET 32767
        *REV

11000:  *NEG
        *NOT                ;! P-1
        *DUP
        *J 11002<Z          ;! P<1
        *DUP
        *=C14
        *-                  ;! Q-P+1
        *DUP
        *J 11002<=Z         ;! Q<P
        *REV
        *ZERO
        *SHLD+16
        *=M13               ;! CURRENT CELL
        *SHL-32             ;! END OF STRING
        *J 11003 C14Z       ;!P=1
```

```
11001:  *M13
        *J 11002=          ;! END OF STRING
        *MOM13
        *J 11004 C15Z
        *M13
        *JS 101P           ;! RETURN CELLS

11004:  *=Q13
        *J 11001 C13Z      ;! DUMMY
        *DC14
        *J 11001 C14NZ     ;! NOT YET REACHED PTH CELL
11003:  *REV
        *=C14              ;! Q-P+1
        *I15
        *M13
        *=I15              ;! SAVE FIRST OF PART
        *J 11005=Z
        *=M15
        ->11006

11005:  *M13
        *J 11002=          ;! END OF STRING
        *MOM13
        *=Q13
        *J 11005 C13Z      ;! DUMMY
        *DC14
        *J 11005 C14 NZ    ;! NOT YET REACHED QTH CELL
        *M13 TO Q15        ;! SAVE LAST OF PART
        *M13
        *-
        *J 11006=Z         ;! END OF STRING ANYWAY
        *J 11006 C15Z
        *MOM13
        *ZERO
        *=MOM13            ;! DUMP CORRECT LAST CELL

11007:  *=Q13
        *MOM13
        *M13
        *JS 101P
        *DUP
        *J 11007 # Z       ;! NOT END OF CLEAN STRING
        *ERASE

11006:  *Q15
        *SHL+16
        *EXIT 1            ;! N1=A(EXTRACT)

11002:  %CAPTION --PART_FAULT
        %STOP
```

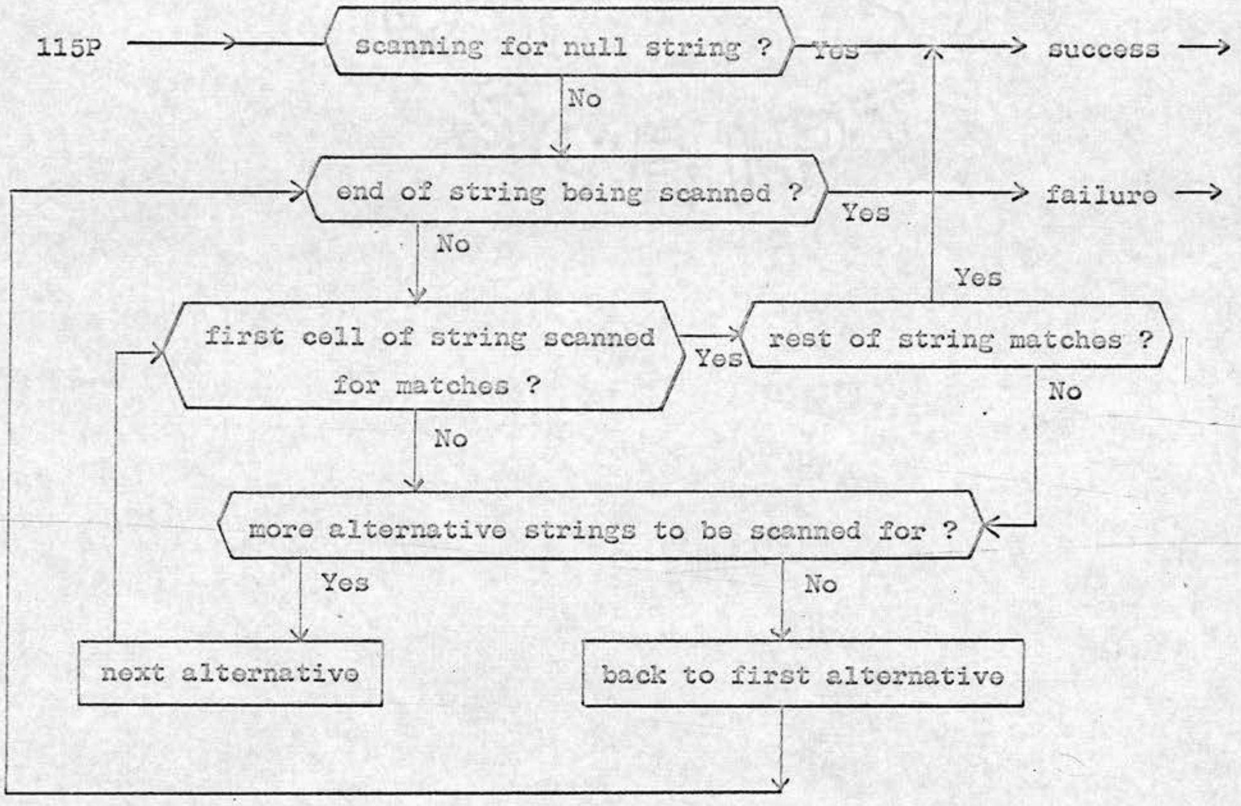

113P
find penultimate cell of expr. to be inserted
return last cell (dummy)
set penultimate cell to point to next cell in main list
first cell of expr. being inserted a substring ?
Yes
No
set first cell at insert position to dummy and to point to first of inserted expr. (to preserve a possible back-pointer list)
copy contents of first cell of inserted expr. into first at insert position, leaving possible back-pointer list unchanged.
return first cell of inserted expr.
return replaced cells

```
113P:   ! INSERT ASSIGN (<-)
        ! N1=A(STRING)
        ! N2=A(EXPRESSION)

        *ZERO
        *SHLD+16          ;! FIRST OF STR.
        *REV
        *SHL-32           ;! LAST OF STR.
        *CAB
        *ZERO
        *SHLD+16          ;! FIRST OF EXPR
        *=M13
        *M13 TO Q15
        *SHL-32           ;! LAST OF EXPR

11302:  *M0M13
        *=Q14             ;! NEXT CELL OF EXPR
        *M14
        *J 11301=         ;! END OF EXPR
        *M14 TO Q13
        ->11302

11301:  *JS 101P          ;! RETURN LAST OF EXPR
        *DUP              ;! LAST OF STR.
        *=M14
        *Q14
        *=M0M13           ;! LAST OF EXPR POINTS TO LAST OF STR
        *M0M15            ;! FIRST OF EXPR
        *=Q13
        *I13
        *J 11303=Z        ;! NOT SUBSTRING
        *M15
        *=Q13             ;! SET UP DUMMY TO COPY INTO FIRST OF STR
        ->11304           ;! SINCE BACK POINTER CHAIN MIGHT BE ERASED

11303:  *M15
        *JS 101P          ;! RETURN FIRST OF EXPR

11304:  *REV
        *=M14             ;! FIRST OF STR
        *M0M14
        *=Q15
        *I15 TO Q13       ;! RETURN BACK CHAIN
        *Q13
        *=M0M14           ;! COPY FIRST OF EXPR INTO STR

11306:  *M15
        *J 11305=         ;! LAST OF STR
        *M0M15
        *M15
        *JS 15100         ;! RETURN CELL OF STR
        *=M15
        ->11306

11305:  *ERASE
        *EXIT 1
```



114P
number of components < 0 ?
Yes
monitor
No
store @(current cell) in stage 2 array
end of string ?
Yes
failure
No
dummy cell ?
Yes
No
more components to scan past ?
Yes
No
success

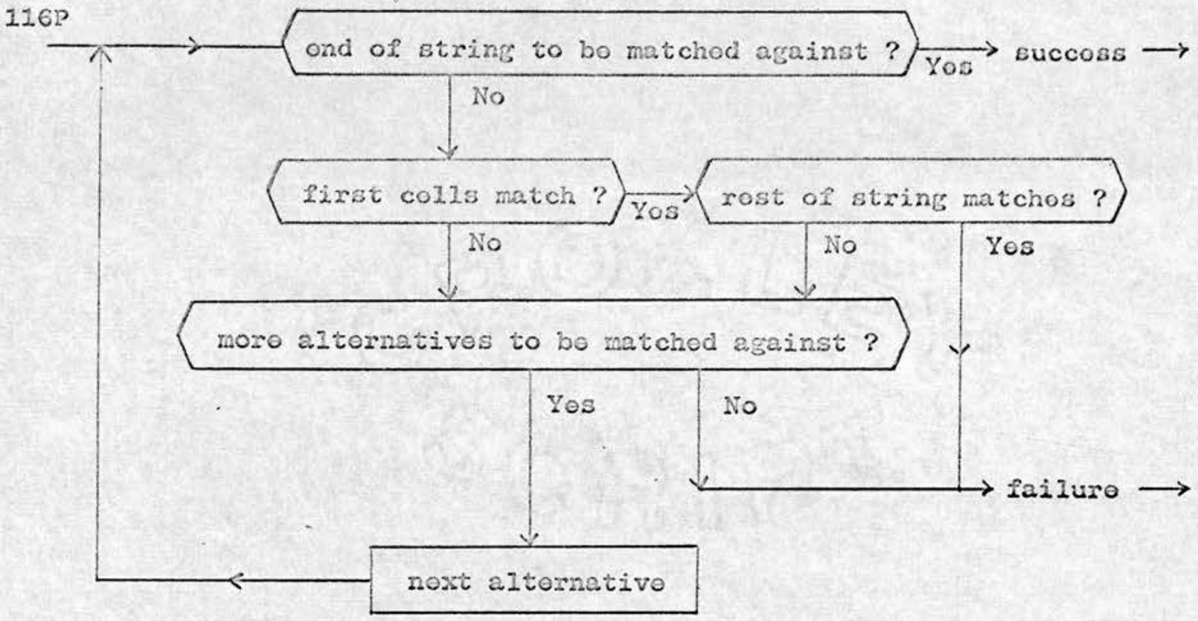
```
114P: ! COUNT DOWN N ITEMS
      ! N1=@(CURRENT)
      ! N2=@(LAST)

      *MOM10              ;! 0/N/@(VARIABLE)
      *SHL+16
      *SHA-32             ;! N
      *DUP
      *J 11401<Z          ;! NONSENSE
      *=C15
      *DUP
      *=MOM10QN           ;! DUMP @(CURRENT)
      *J 11402 C15 Z      ;! N=0
      *=M14
11403:*M14
      *J 11404=           ;! END REACHED
      *MOM14              ;! NEXT ITEM
      *=Q14
      *J 11403 C14 Z      ;! DUMMY
      *DC15
      *J 11403 C15 NZ     ;! MORE ITEMS YET
      *M14                ;! NEW NEXT
11402:*EXIT 2             ;! SUCCESS, N1=@(NEXT), N2=@(LAST)

11404:*ERASE
      *EXIT 1             ;! FAILURE, NEST EMPTY

11401:%CAPTION -- -VE _ NO. _ OF _ ITEMS
      %STOP
```


115P
scanning for null string ?
Yes
success
No
end of string being scanned ?
Yes
failure
No
first cell of string scanned for matches ?
Yes
rest of string matches ?
Yes
No
No
more alternative strings to be scanned for ?
Yes
No
next alternative
back to first alternative

```
115P: ! SEARCH FOR STRING
      ! N1=@(CURRENT ITEM)
      ! N2=@(LAST ITEM)

      *=M15              ;! CURRENT
      *ZERO
      *=MOM12
      *MOM10             ;! A(STRING)
      *DUP
      *J 11513>=Z
      *Q10
      *=MOM12
      *SHL+1
      *SHL-1
11513:*ZERO
      *SHLD+16
      *=M13              ;! FIRST OF STRING
      *SHL-32
11502:*M13
      *J 11501=          ;! SEARCH FOR _
      *MOM13
      *=Q13              ;! FIRST ITEM
      *J 11502 C13 Z     ;! IGNORE INITIAL DUMMIES
      *REV
      *MOM12
      *J 11512=Z
      *Q13 TO Q10
      *REV
      *ERASE
11512:*ZERO
11519:*Q13 TO Q14        ;! PRESERVE FIRST ITEM
      *MOM12
      *J 11505=Z
      *I14
      *J 11611<=Z
      *I14
      *PERM
      *C14
      *=M14
      *MOM14
      *=Q14
      *J 11521 C14 Z
11505:*ERASE
      *M15
      *DUP
      *PERM
      *J 11503=          ;! FAILURE, END OF MAIN STRING
      *REV
      *MOM15             ;! NEXT ITEM OF MAIN CHAIN
      *=Q15
11518:*I14
      *J 11504>Z         ;! SUBSTRING
      *I15
      *J 11516>Z         ;! SUBSTRING
      *C14
```

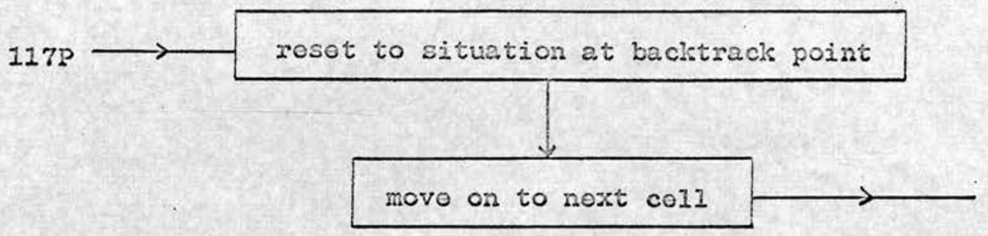
```
       *C15
       *-
       *J 11516#Z          ;! NO MATCH FOR FIRST ITEM YET
11511:*CAB
11507:*M14
       *J 11506=           ;! SUCCESS, END OF STRING
       *M0M14
       *=Q14               ;! NEXT ITEM OF STRING
       *J 11507 C14 Z      ;! DUMMY
       *PERM
       *REV
11508:*M15
       *J 11510=           ;! END OF MAIN STRING
       *M0M15
       *=Q15
       *J 11508 C15 Z      ;! DUMMY
       *REV
       *I14
       *J 11509 > Z        ;! SUBSTRING
       *I15
       *J 11510>Z          ;! SUBSTRING
       *C14
       *C15
       *-
       *J 11511=Z          ;! MATCH FURTHER ITEM
11510:*DUP
       *=M15               ;! ADDR OF FIRST ITEM MATCHED
       *M0M15
       *=Q15               ;! MOVE ON ONE ITEM
       *M0M12
       *J 11520#Z
       ->11519            ;! TRY AGAIN

11504:*I15
       *J 11516<=Z         ;! NO SUBSTRING YET
       *Q13
       *Q14
       *Q15                ;! PRESERVE
       *DUPD
       *M+I12
       *JS 104P            ;! COMPARE SUBSTRINGS
       *M-I12
       *PERM               ;! 0:MATCH, -1:NO MATCH
       *=Q15
       *=Q14
       *REV
       *=Q13
       *J 11516#Z          ;! NO MATCH
       ->11511             ;! MATCH

11516:*M0M12
       *J 11505=Z
11520:*M0M13
       *=Q13
       *CAB
       *ERASE
       *J 11517 C13 Z
```

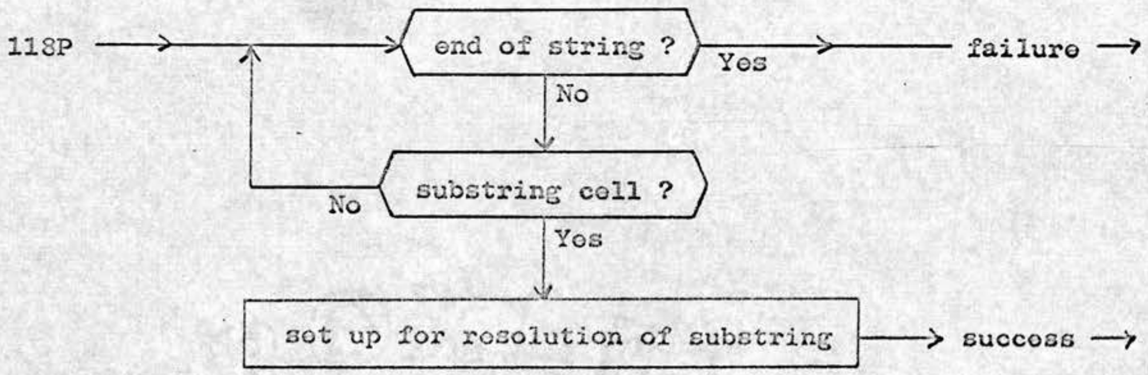
```
      *I13
      *J11611<=Z
      *I13
      *PERM
      *C13
      *=M14
      *MOM14
      *=Q14
      *J 11521 C14 Z
      ->11518
11517:*Q10 TO Q13
      ->11519

11509:*I15
      *J 11510<=Z         ;! NO SUBSTRING
      *Q13
      *Q14
      *Q15
      *DUPD
      *M+I12
      *JS 104P            ;! COMPARE SUBSTRINGS
      *M-I12
      *PERM
      *=Q15
      *=Q14
      *REV
      *=Q13
      *J 11510#Z          ;! NO MATCH
      ->11511             ;! MATCH

11521:*DUP
      *=M15
      *CAB
      ->11506

11501:*M15
      *REV
11506: *ERASE
      *MOM12
      *J 11514=Z
      *MOM12
      *=Q10
11514:*=MOM10QN           ;! ADDR FIRST ITEM MATCHED
      *M15                ;! NEXT ITEM
      *EXIT 2             ;! SUCCESS, N1=NEXT, N2=LAST

11503:*ERASE
      *ERASE
      *ERASE
      *MOM12
      *J 11515=Z
      *MOM12
      *=Q10
11515:*EXIT 1             ;! FAILURE, NEST EMPTY
```


116P
end of string to be matched against ?
Yes
success
No
first cells match ?
Yes
rest of string matches ?
No
No
Yes
more alternatives to be matched against ?
Yes
No
failure
next alternative

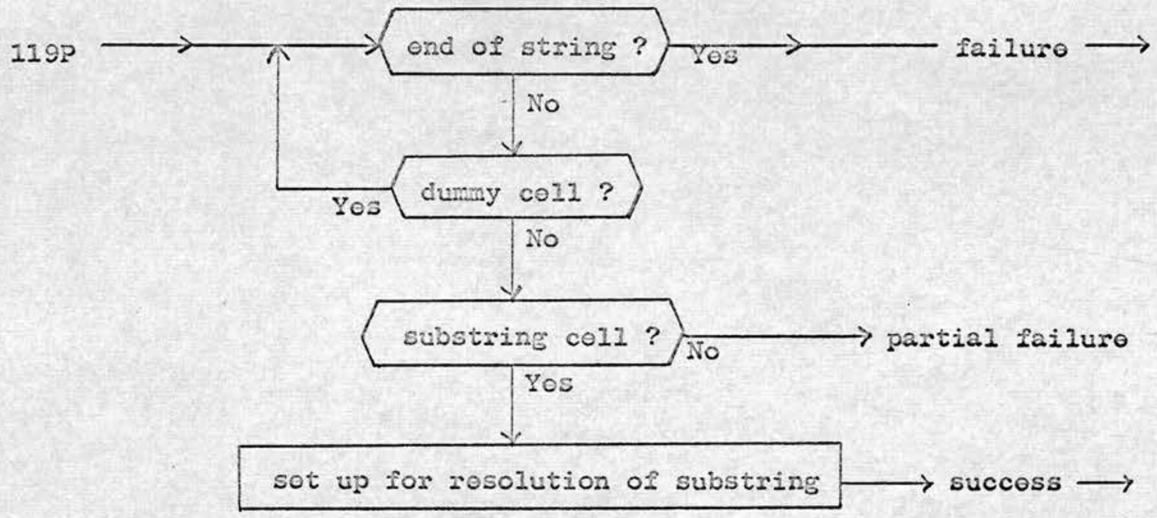
```
116P: ! MATCH STRING
      ! N1=@(CURRENT ITEM)
      ! N2=@(LAST ITEM)

      *DUP
      *PERM
      *=M15
      *ZERO
      *NOT
      *=Q14
      *MOM10               ;! A(STRING)
      *DUP
      *J 11607>=Z
      *SHL+1
      *SHL-33
      *=M14
      *MOM14
      *=Q14
11608:*J 11609 C14 Z
      *I14
      *J 11611<=Z
      *Q14
11607:*ZERO
      *SHLD+16             ;! FIRST OF STRING
      *=M13
      *SHL-32              ;! LAST OF STRING

11602:*M13
      *J 11601=            ;! SUCCESS, END OF STRING
      *MOM13
      *=Q13
      *J 11602 C13 Z       ;! DUMMY
      *REV
11604:*M15
      *J 11606=            ;! ABSOLUTE FAILURE, END OF MAIN STRING
      *MOM15
      *=Q15
      *J 11604 C15 Z       ;! DUMMY
      *REV
      *I13
      *J 11605>Z           ;! SUBSTRING
      *I15
      *J 11606>Z           ;! SUBSTRING, PARTIAL FAILURE
      *C13
      *C15
      *-
      *J 11602=Z           ;! MATCH
```

```
11606:*ERASE
      *Q14
      *J 11609<Z
11610:*MOM14
      *=Q14
      *REV
      *DUP
      *=M15
      *REV
      ->11608
11609:*REV
      *ERASE
      *EXIT 3            ;| PARTIAL FAILURE, N1=@(LAST)
                          | M10 MOVED ON

11605:*I15
      *J 11606<=Z        ;| NO SUBSTRING
      *Q14
      *Q15
      *Q13
      *DUPD
      *JS 104P           ;| COMPARE SUBSTRINGS
      *PERM
      *=Q13
      *=Q15
      *REV
      *=Q14
      *J 11602=Z         ;| MATCH
      ->11606            ;| NO MATCH

11601:*ERASE
      *REV
      *=MOM10QN          ;| DUMP @(FIRST ITEM)
      *M15
      *EXIT 2            ;| SUCCESS, N1=NEXT,N2=LAST

11603:*REV
      *ERASE
      *Q14
      *J 11610>=Z
      *ERASE
      *ERASE
      *EXIT 1            ;| FAILURE, NEST EMPTY

11611:%CAPTION --STRING_INVALID_IN_MULTIPLE_RESOLUTION
      %STOP
```


117P
reset to situation at backtrack point
move on to next cell

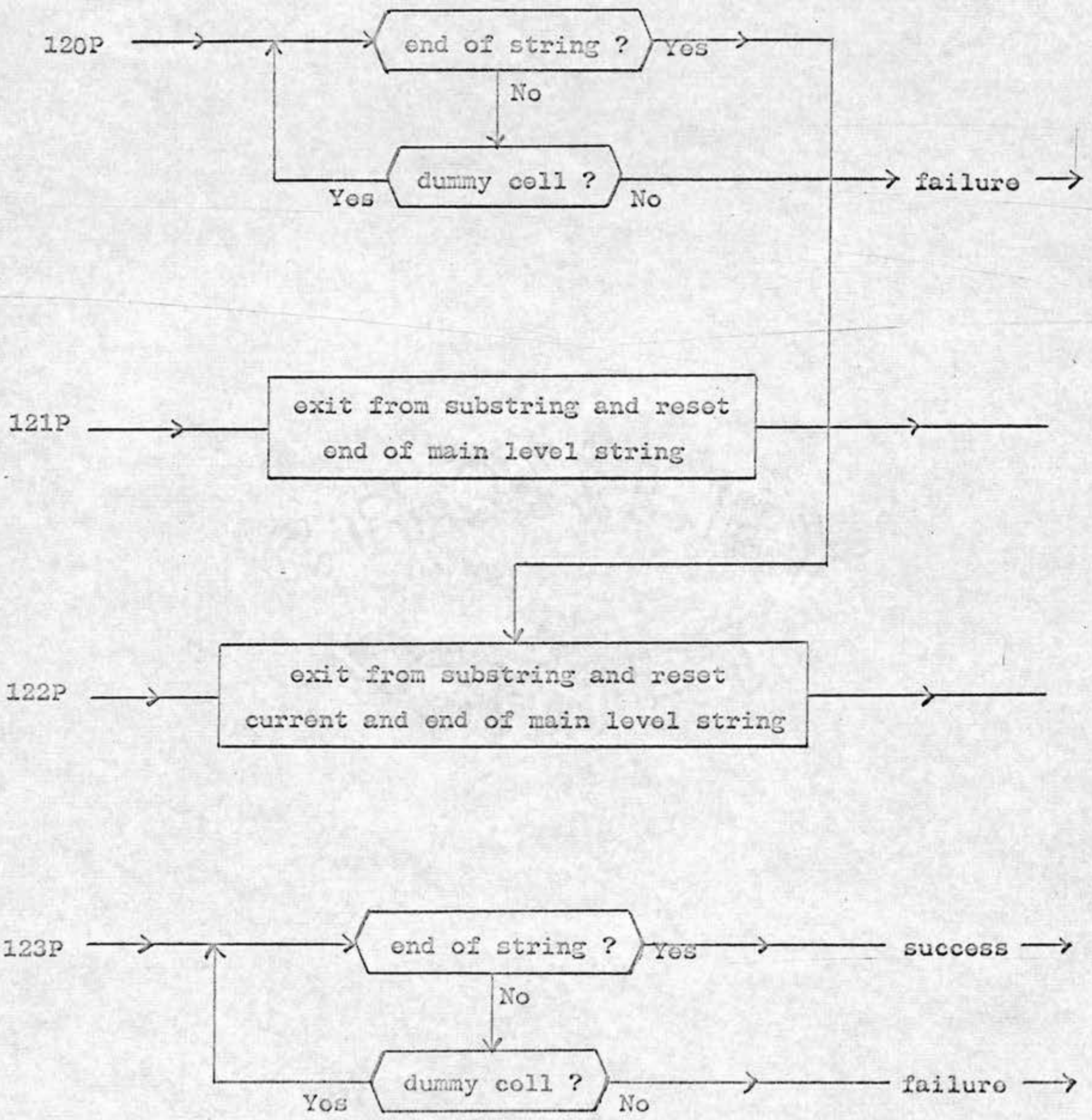

118P
end of string ?
Yes
failure
No
substring cell ?
No
Yes
set up for resolution of substring
success

```
117P: ! BACKTRACK
      ! N1= AMOUNT TO GO BACK
      ! N2=@(LAST ITEM)

      *=+M10
      *MOM10N             ;! PREVIOUS POINTER
      *=M13
      *MOM13              ;! MOVE ON ONE ITEM
      *SHL+32
      *SHL-32
      *EXIT 1
      ! N1=@(ITEM TO START FROM AGAIN)
      ! N2=@(LAST ITEM)



118P: ! SEARCH FOR SUBSTRING
      ! N1=@(CURRENT ITEM)
      ! N2=@(LAST ITEM)

      *=M15
      *ZERO
11801:*ERASE
      *M15
      *DUP
      *PERM
      *J 11802=           ;! END OF STRING
      *REV
      *MOM15              ;! NEXT ITEM
      *=Q15
      *I15
      *J 11801<=Z         ;! NOT SUBSTRING
      *=MOM10N            ;! DUMP ADDR(SUBSTRING ITEM)
      *I4                 ;! ADDR(CURRENT SUBSTRING ON RUNST)
      *SHL+16
      *OR
      *M10
      *=I4                ;! NEW ADDR(CURRENT SUBSTRING)
      *=MOM10Q
      *I15
      *C15
      *EXIT 2             ;! SUCCESS, N1=FIRST, N2= LAST OF SUBSTRING

11802:*ERASE
      *ERASE
      *EXIT 1             ;! FAILURE, NEST EMPTY
```



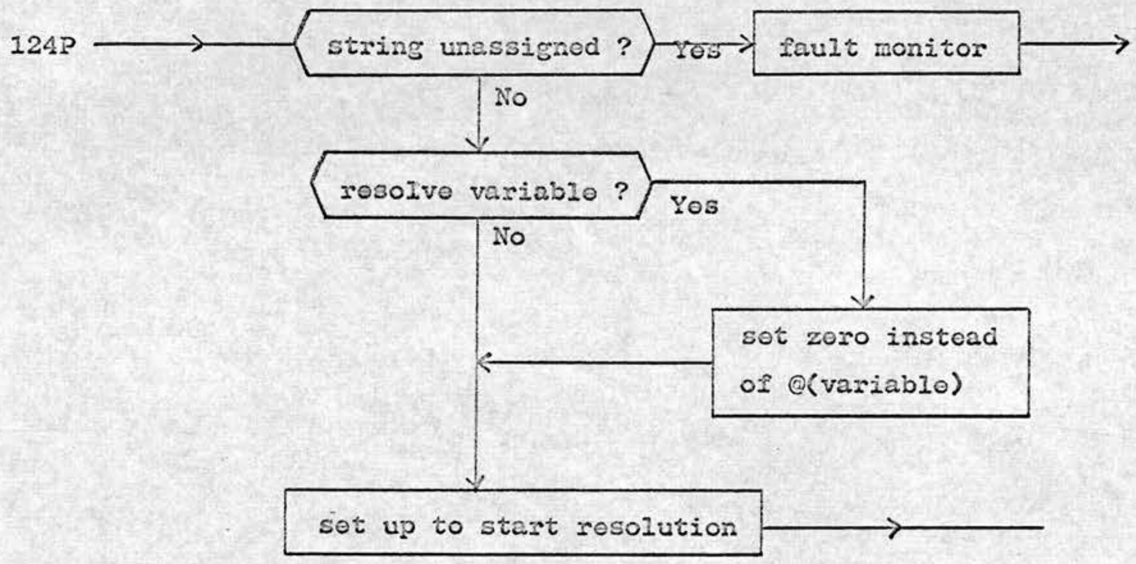
119P
end of string ?
Yes
failure
No
dummy cell ?
Yes
No
substring cell ?
No
partial failure
Yes
set up for resolution of substring
success

```
119P: ! MATCH SUBSTRING
      ! N1=@(CURRENT ITEM)
      ! N2=@(LAST ITEM)

      *=M15
      *ZERO
11901:*ERASE
      *M15
      *J 11902=          ;! END OF STRING
      *M15
      *M0M15             ;! NEXT ITEM
      *=Q15
      *J 11901 C15 Z     ;! DUMMY
      *I15
      *J 11903 <=Z       ;! NOT SUBSTRING
      *=M0M10N
      *I4
      *M10
      *=I4
      *SHL+16
      *OR
      *=M0M10Q
      *I15
      *C15
      *EXIT 2            ;! SUCCESS, N1=FIRST, N2=LAST OF SUBSTRING

11902:*ERASE
      *EXIT 1            ;! FAILURE, NEST EMPTY

11903:*ERASE
      *EXIT 3            ;! PARTIAL FAILURE, N1=@(LAST)
```

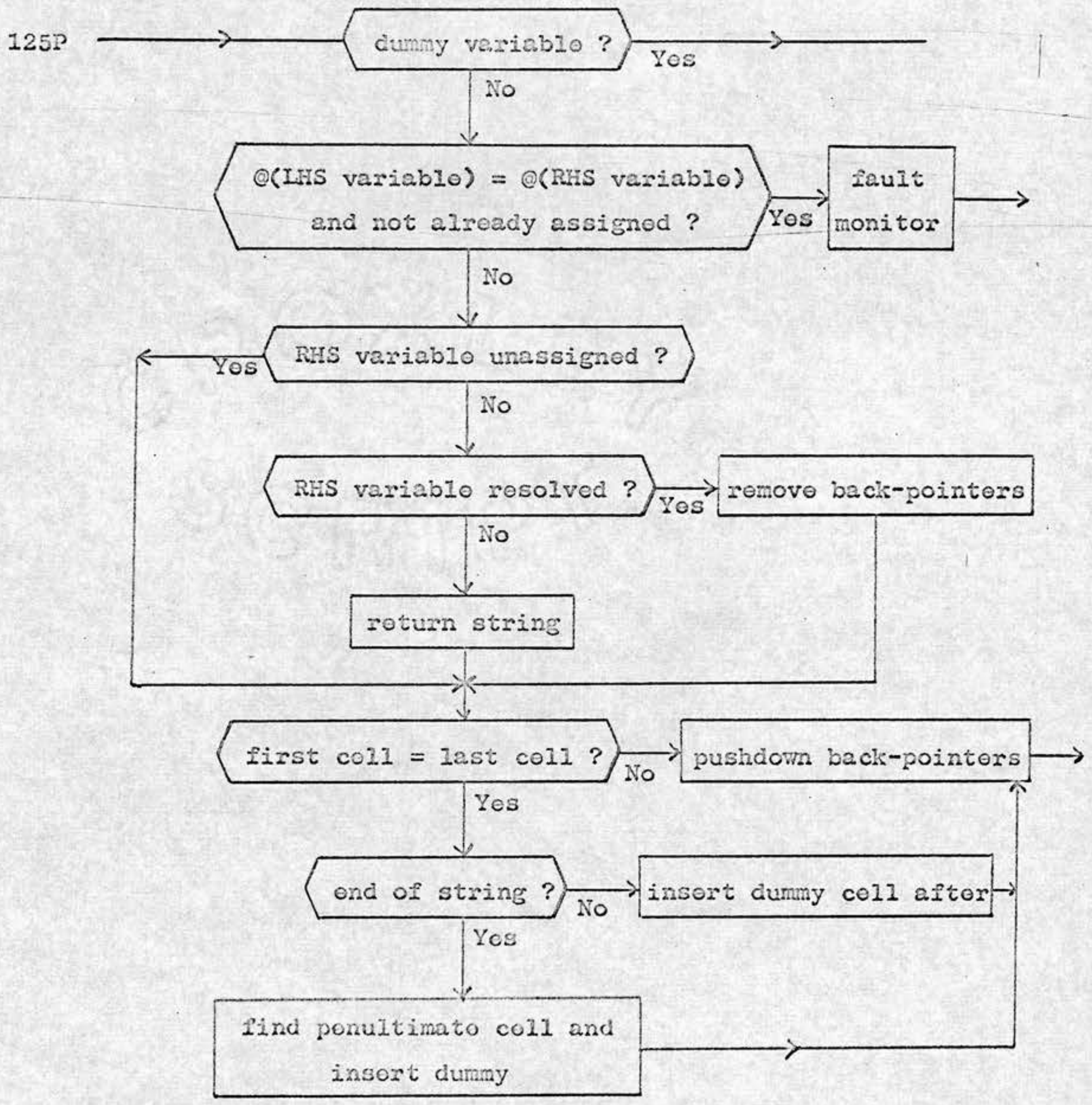


120P
end of string ?
Yes
No
dummy cell ?
Yes
No
failure
121P
exit from substring and reset end of main level string
122P
exit from substring and reset current and end of main level string
123P
end of string ?
Yes
success
No
dummy cell ?
Yes
No
failure

```
120P: ! CHECK FOR END OF SUBSTRING AND EXIT ON SUCCESS
      ! N1=@(CURRENT ITEM)
      ! N2=@(LAST ITEM)

      *=M15
12002:*M15
      *J 12001=            ;! END OF SUBSTRING, SAME AS 122P
      *MOM15
      *=Q15
      *J 12002 C15 Z       ;! DUMMY
      *EXIT 2              ;! FAILURE, N1=@(LAST ITEM)



121P: ! EXIT FROM SUBSTRING AFTER FAILURE
      ! NEST EMPTY

      *I4                  ;! START OF CURRENT SUBSTRING
      *=M10
      *MOM10               ;! 0/OLD I4/OLD @(LAST)
      *=Q13
      *I13 TO Q4           ;! RESET I4
      *M13
      *EXIT 1              ;! N1=@(LAST OF MAIN STRING)



122P: ! EXIT FROM SUBSTRING AFTER SUCCESS
      ! N1=@(LAST ITEM)

12001:*=MOM10N
      *I4                  ;! START OF SUBSTRING
      *=RM13
      *MOM13Q              ;! 0/OLD I4/ OLD END OF STRING
      *=Q14
      *I14 TO Q4           ;! RESET I4
      *M14
      *MOM13Q
      *=M14
      *MOM14               ;! SUBSTRING ITEM
      *SHL+32
      *SHL-32
      *MOM13N
      *=MOM10Q
      *EXIT 1              ;! SUCCESS, N1=NEXT, N2=LAST OF STRING
```


124P
string unassigned ?
Yes
fault monitor
No
resolve variable ?
Yes
No
set zero instead of @(variable)
set up to start resolution

```
123P: ! CHECK FOR END OF STRING
      ! N1=@(CURRENT ITEM)
      ! N2=@(LAST ITEM)

      *=M15
12301:*M15
      *J 12302=          ;! END OF STRING
      *M0M15
      *=Q15
      *J 12301 C15 Z     ;! DUMMY
      *EXIT 2            ;! FAILURE, N1=@(LAST ITEM)

12302:*=M0M10QN
      *EXIT 1            ;! SUCCESS, NEST EMPTY



124P: ! PREPARE FOR CRES 2
      ! N1=@(RESOLVE VARIABLE)

      *DUP
      *=M13
      *M0M13
      *DUP
      *J 14P=Z           ;! UNASSIGNED STRING
      *=Q14
      *M14
      *J 12401=Z         ;! NOT RESOLVED  VARIABLE
      *ERASE
      *ZERO
12401:*I14
      *C14
      *I4
      *=RM10
      *I10=+2
      *EXIT 1
      ! N1=@(FIRST ITEM)
      ! N2=@(LAST ITEM)
      ! N3= 0:RESOLVED VARIABLE
      !     ADDR:OTHERWISE
```

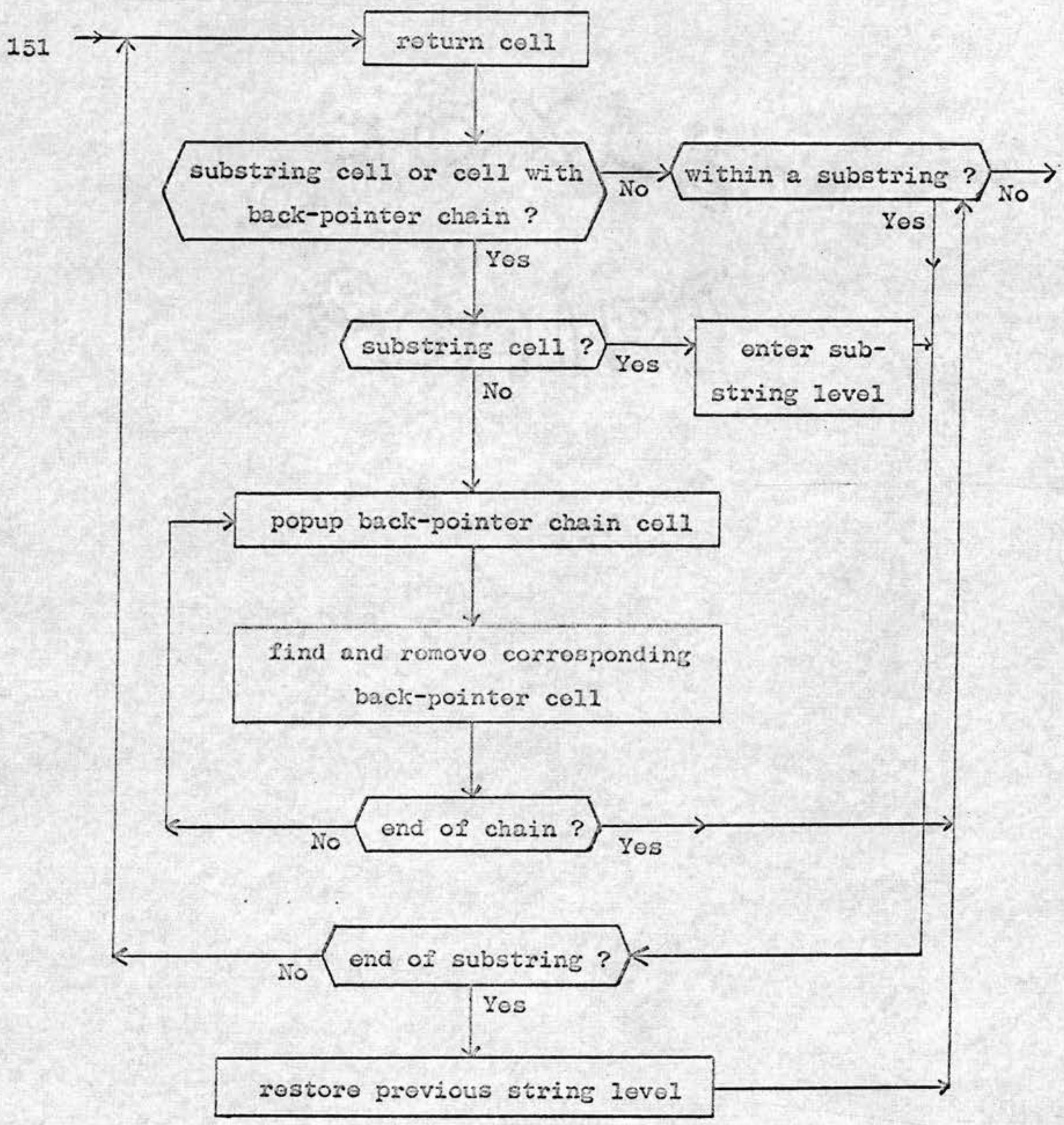

125P
dummy variable ?
Yes
No
@(LHS variable) = @(RHS variable)
and not already assigned ?
Yes
fault
monitor
No
RHS variable unassigned ?
Yes
No
RHS variable resolved ?
Yes
remove back-pointers
No
return string
first cell = last cell ?
No
pushdown back-pointers
Yes
end of string ?
No
insert dummy cell after
Yes
find penultimate cell and
insert dummy

```
125P: ! ASSIGN RESOLVED STRING, SAY  Y->X
      ! N1= M10 INCREMENT
      ! N2=@(Y)

      *=+M10
      *M0M10              ;! @/0 OR N/ @(X)
      *=M13
      *M13
      *DUP
      *J 12501#Z          ;! -[PART] OPERAND
      *ERASE
      *M+I10
      *EXIT 1

12501:*J 16P=             ;! @(Y) =@(X) AND NOT RESOLVED
      *M0M13
      *DUP
      *=Q14
      *J 12502=Z          ;! X UNASSIGNED
      *M14
      *J 12503=Z          ;! X NOT RESOLVED
      *C14                ;! @(FIRST OF X)
      *M13                ;! @(X)
      *JS 15200           ;! REMOVE BACK POINTER
      *I14                ;! @(END OF X)
      *M13
      *JS 15200
      ->12502

12503:*M13
      *DUP                ;! PRESERVE
      *JS 102P            ;! RETURN STRING
      *=M13

12502:*M13                ;! @X
      *M0M10QN            ;! FIRST
      *DUP
      *=RC13
      *M0M10N             ;! LAST
      *DUP
      *=I13
      *J 12504=           ;! FIRST=LAST
      *JS 12510           ;! PUSHDOWN BACK LINK
      *I13
      *JS 12510
      *=M14
      *Q13
      *NOT
      *NEG
      *=M0M14
      *EXIT 1             ;! N1=@(Y)
```

```
12510:*=M14              ;! FIRST OR LAST
      *MOM14
      *=Q15
      *I15
      *J 12511<=Z        ;! NOT SUBSTRING ITEM
      *JS 100P
      *DUP
      *=M15
      *MOM14
      *=MOM15            ;! COPY SUBSTRING ITEM
      *=MOM14
12511:*DUP               ;! @X
      *SHL+32
      *MOM14
      *=Q15
      *I15
      *SHL+32
      *SHL-16
      *OR                ;! @X/-LINK/0
      *JS 100P
      *DUP
      *NEG
      *=I15
      *Q15
      *=MOM14            ;! PUSHDOWN NEW CELL
      *=M15
      *=MOM15            ;! STORE BACK POINTER
      *EXIT 1

12504:*=M14
      *MOM14
      *=Q15
      *M15
      *J 12512=Z         ;! END OF STRING
      *I15
      *J 12505<=Z        ;! NOT SUBSTRING ITEM
      *JS 100P
      *DUP
      *=M15
      *MOM14
      *=MOM15            ;! COPY SUBSTRING ITEM
      *=MOM14
```

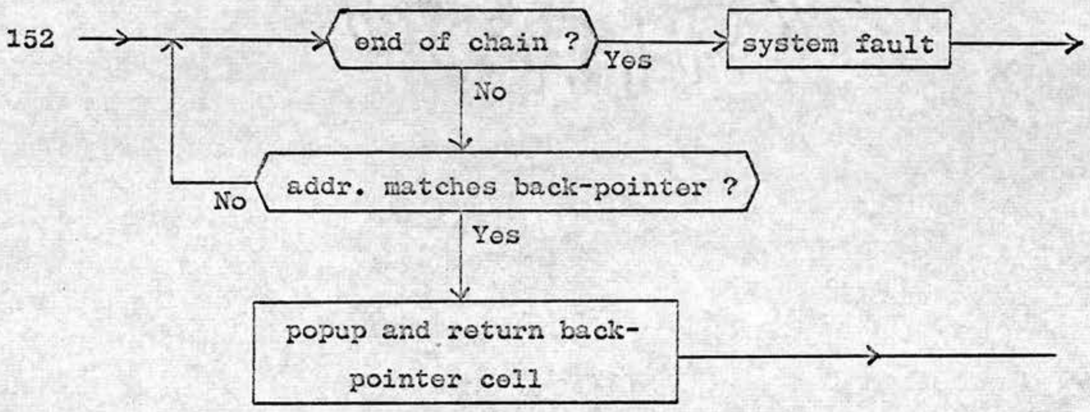
```
12505:*JS 100P
      *=M15
      *MOM14
      *=Q13
      *I13
      *JS 100P
      *DUP
      *NEG
      *=I13
      *Q13
      *=MOM15
      *=M13
      *REV                ;! @X
      *DUP
      *SHL+32
      *=MOM13
      *JS 100P
      *=M13
      *DUP
      *SHL+32
      *CAB
      *SHL+32
      *SHL-16
      *OR
      *=MOM13
      *M13
      *NEG
      *SHL+32
      *SHL-16
      *M15
      *OR
      *=MOM14
      *=M13               ;! @X
      *M14
      *SHL+16
      *M15
      *OR
      *SHL+16
      *NOT
      *NEG
      *=MOM13
      *M15
      *=MOM10N
      *EXIT 1
```

```
12512:*MOM10
12513:*=M13
      *MOM13
      *SHL+32
      *SHL-32
      *M14
      *J 12513#            ;! SKIP DOWN TO END
      *ERASE
      *MOM13
      *SHL-16
      *SHL+16
      *JS 100P
      *OR
      *DUP
      *=MOM13
      *JS 100P
      *=M13
      *REV
      *DUP
      *SHL+32
      *=MOM13
      *M13
      *NEG
      *SHL+32
      *SHL-16
      *M14
      *OR
      *CAB
      *=M13
      *=MOM13
      *DUP
      *SHL+32
      *Q15
      *OR
      *JS 100P
      *=M15
      *=MOM15
      *M15
      *NEG
      *SHL+32
      *SHL-16
      *=MOM14
      *=M15
      *M13
      *SHL+16
      *M14
      *OR
      *SHL+16
      *NOT
      *NEG
      *=MOM15
      *EXIT 1
```


151
return cell
substring cell or cell with back-pointer chain ?
No
within a substring ?
No
Yes
Yes
substring cell ?
Yes
enter sub-string level
No
popup back-pointer chain cell
find and remove corresponding back-pointer cell
end of chain ?
No
Yes
end of substring ?
No
Yes
restore previous string level

```
15100:! RETURN CELL (COMPLEX)
      ! N1=ADDR(CELL)

      *C0 TO Q12          ;! DEPTH COUNTER

15101:*Q13                ;! PRESERVE
      *REV
      *DUP
      *=M13
      *MOM13
      *=Q13               ;! CELL
      *DUP
      *JS 101P            ;! RETURN CELL
      *I13
      *DUP
      *J 15102#Z          ;! NOT SIMPLE
      *ERASE
      *ERASE
      *=Q13               ;! RESTORE
      *J 15103 C12 NZ     ;! BACK TO SUBSTRING
      *EXIT 1

15102:*DUP
      *J 15104>Z          ;! SUBSTRING
      *NEG
      *=M13               ;! ADDR(FIRST BACK)
      *Q15
      *Q14
      *REVD               ;! PRESERVE

15109:*MOM13
      *M13
      *JS 101P
      *ZERO
      *SHLD+16
      *=M14               ;! BACK POINTER
      *SHA-32             ;! -ADDR(NEXT BACK)
      *REV                ;! ADDR(MAIN CHAIN CELL)
      *MOM14              ;! A(RESOLVED STRING)
      *=Q15
      *C15
      *J 15105=
      *I15
      *J 15106#
      *C15
      ->15107
```

```
15105:*I15
15107:*M14
      *JS 15200           ;! REMOVE OTHER LINK
      *ZERO
      *=M0M14             ;! SET RESOLVED STRING UNASSIGNED
      *REV
      *DUP
      *J 15108=Z          ;! END OF CHAIN
      *NEG
      *=M13
      ->15109

15108:*ERASE
      *ERASE
      *=Q13
      *=Q14
      *=Q15               ;! RESTORE
      *J 15103 C12 NZ     ;! BACK TO SUBSTRING
      *EXIT 1

15106:%CAPTION -- NO _ BPC _ MATCH
      %STOP

15104:*ERASE
      *ERASE
      *=M0M12Q            ;! PRESERVE Q13
      *C13
      *=M13

15103:*M13
      *DUP
      *J 15110=Z          ;! END OF SUBSTRING
      *M0M13
      *=Q13
      ->15101             ;! RETURN CELL

15110:*ERASE
      *M-I12
      *M0M12
      *=Q13
      *I12
      *=+C12
      *J 15103 C12 NZ     ;! BACK TO SUBSTRING
      *EXIT 1
```


152
end of chain ?
Yes
system fault
No
addr. matches back-pointer ?
No
Yes
popup and return back-pointer cell

```
15200:! REMOVE BACK POINTER
      ! N1= ADDR TO BE SEARCHED FOR
      ! N2= ADDR OF MAIN CHAIN CELL

      *Q13
      *PERM              ;! PRESERVE
      *REV
      *=M13              ;! CURRENT FIRST
      *Q15
      *Q14
      *REVD              ;! PRESERVE
      *MOM13
      *SHL+16
      *SHA-32            ;! - CHAIN POINTER
15202:*DUP
      *J 15201=Z         ;! FAULT
      *NEG
      *=M14              ;! CURRENT SECOND
      *MOM14
      *=Q15
      *C15
      *J 15203=          ;! FOUND BACK POINTER
      *M14 TO Q13
      *I15               ;! - CHAIN POINTER
      ->15202

15203:*ERASE
      *M14               ;! POPUP
      *JS 101P
      *MOM13
      *=Q14
      *I15 TO Q14
      *Q14
      *=MOM13
      *=Q13              ;! RESTORE
      *=Q14
      *=Q15
      *EXIT 1

15201:%CAPTION -- NO _ BACK _ POINTER
      %STOP
```

APPENDIX D

Permanent routines

```
%ROUTINE READ ITEM(%STRINGNAME S)
%INTEGER I
READ SYMBOL(I)
*JS 100P
*DUP
**I
*SHL+32
*OR
*JS 100P
*=M13
*=MOM13
*M13
*SHL+16
*OR
*SHL+16
**@S
*DUP
*JS 102P
*=M13
*=MOM13
%END


%STRINGFN NEXT ITEM
%INTEGER I
I=NEXT SYMBOL
*JS 100P
*DUP
**I
*SHL+32
*OR
*JS 100P
*=M13
*=MOM13
*M13
*SHL+16
*OR
*SHL+16
%RETURN
%END


%ROUTINE SKIP ITEM
SKIP SYMBOL
%END
```

```
%INTEGERFN STOI(%STRING S)
**S
*SHL-32
*=M15
2:*MOM15
*DUP
*=Q15
*J1=Z
*J2 C15 Z
*C15
*DUP
*J1<Z
4:*MOM15
*DUP
*=Q15
*J3=Z
*J4C15Z
1:%CAPTION -- INVALID _ STOI
%STOP
3:%RETURN
%END


%STRINGFN ITOS(%INTEGER N)
%IF N<=0 %THEN ->1
*JS 100P
*=M13
*JS100P
*DUP
**N
*SHL+32
*OR
*=MOM13
*M13
*SHL+16
*OR
*SHL+16
%RETURN
1:%CAPTION -- INVALID _ ITOS
%STOP
%END
```

```
%ROUTINE READ STRING (%STRINGNAME S)
     %INTEGER I,J,K
     %STRING R
1:   READ SYMBOL(I)
     ->1 %IF I#40              ;! IGNORE SPACES & NEWLINES ETC.

     *JS 100P
     *DUP

2:   I=NEXT SYMBOL
     %IF I=40 %THEN ->3     ;! '('
     SKIP SYMBOL
     %IF I=41 %THEN ->5     ;! ')'
     ->2 %IF I=32 %OR I=10  ;! '_' OR '-'
     *=M13
     **I
     *SHL+32
     ->4

3:   **=J
     **=K
     READ STRING (R)
     **K
     **R
     *JS 103P
     **J
     *=M13

4:   *JS 100P
     *DUP
     *PERM
     *OR
     *=M0M13
     ->2

5:   *DUPD
     *-
     *J6#Z
     *=M13
     *JS 100P
     *DUP
     *=M0M13

6:   *REV
     *SHL+16
     *OR
     *SHL+16
     **@S
     *DUP
     *JS 102P
     *=M13
     *=M0M13
     %END
```

```
%ROUTINE WRITE STRING (%STRING S)
     %INTEGER I
     *E228
     *=Q15
     *CO TO Q12
     **S

4:   *ZERO
     *SHLD+16
     *=M13
     *SHL-32

2:   *M13
     *J1=
     *MOM13
     *=Q13
     *J2 C13 Z
     *I13
     *J3>Z
     *C13
     **=I
     *M13
     **I
     *SET 10
     *J6#
     *SET 13
     *JS 13P
6:   *JS 13P
     *=M13
     ->2

3:   *SHL+16
     *M13
     *OR
     *=MOM12Q
     *Q13
     *SET 40
     *JS 13P
     ->4

1:   *ERASE
     *J5 C12 Z
     *SET 41
     *JS 13P
     *M-I12
     *MOM12
     *DUP
     *=M13
     *SHL-16
     *I12
     *=+C12
     ->2

5:   *Q15
     *=E228
     %END
```

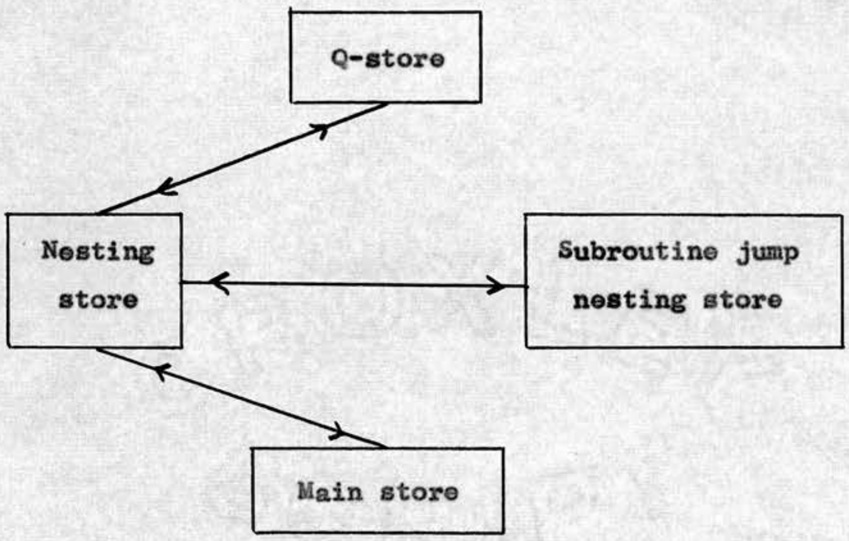
```
%ROUTINE SHOW(%STRINGNAME S,%STRING T)
%INTEGERARRAY ST(1:100)
%INTEGER I,J,K,L,P
WRITE STRING('--'.T.'____'.S.'--')
**@S
**=I
WRITE(I,1)
**S
*=Q13
*C13
**=I
*I13
**=J
*M13
**=K
WRITE(I,6)
WRITE(J,5)
WRITE(K,2)
%RETURN %IF K=1
NEWLINE
P=0

K=I
7:NEWLINE
SPACES(3*P)
WRITE(K,1)
->10 %UNLESS 2000<=K<=15000
**K
*=M13
*MOM13
*=Q13
*C13
**=I
*I13
**=J
*M13
**=K
%IF J>0 %THEN ->1
%IF I=0 %THEN ->2
%IF I=10 %THEN ->3
%IF I=32 %THEN ->4
SPACES(3)
PRINT SYMBOL(I)
->5
10:%CAPTION _PRANG
->6
2:WRITE(0,2)
->5
3:%CAPTION __NL
->5
4:%CAPTION __SP
5:WRITE(J,5)
WRITE(K,5)
%IF J=0=K %THEN ->6
%IF J=0 %THEN ->7
```

```
8:J=-J
WRITE(J,8)
->9 %UNLESS 2000<=J<=15000
**J
*=M13
*MOM13
*=Q13
*C13
**=I
*I13
**=J
*M13
**=L
WRITE(I,6)
WRITE(J,5)
WRITE(L,2)
->8 %UNLESS J=0
->7 %UNLESS K=0
6:%RETURN %IF P=0
K=ST(P)
P=P-1
->7 %UNLESS K=0
->6
9:%CAPTION _CLANG
->7 %UNLESS K=0
->6
1:WRITE(I,6)
WRITE(J,5)
WRITE(K,5)
P=P+1
ST(P)=K
K=I
->7
%END


%INTEGERFN LENGTH(%STRING S)
*ZERO
**S
*SHL-32
*=M13
1:*MOM13
*=Q13
*M13
*J2=Z
*J1C13Z
*NOT
*NEG
*J1
2:%END
```

APPENDIX E

KDF9 Machine Code

The main compnonents of the KDF9 from the programming point of view can be shown diagrammatically :

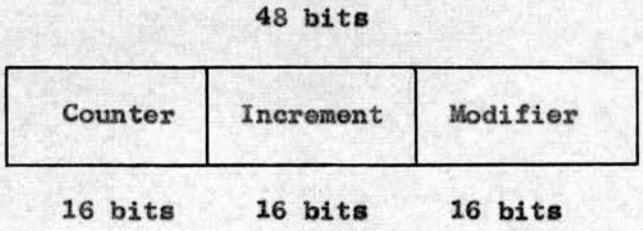


Main store

ASTRA was implemented for a KDF9 with 16384 words each of 48 bits. Word and half-word addressing is available. The Director program occupies the lowest area of main store and is commonly 1216 words in length. The program base is relocated to the end of Director, allowing an effective store from addresses 0 to 15168.

Q-store

There are 16 Q-stores, the name given to the KDF9 index registers, Q0 to Q15. Q0 always contains zero. Each is 48 bits long, but for some purposes can be regarded as three separate 16-bit long fields, named Counter, Increment, and Modifier fields :



Subroutine Jump Nesting Store

The SJNS acts as a pushdown stack with a maximum capacity of 15 cells. Each cell is 16 bits long. It is used to stack subroutine return addresses.

Nesting store

The Nest also acts as a pushdown stack, but with a maximum capacity of 16 cells each of which is 48 bits long. All Q-store, Main store and arithmetic activities make use of the Nest.

Summary of basic instructions

1. En where n is a store word address.

Pushdown the contents of Main store location n into the Nest. E.g.

E382

2. =En

Store top cell of Nest in Main store location n and popup Nest one cell.

The = symbol is consistently used to indicate removal from the Nest and popup one cell and the absence of the = symbol implies the reverse i.e. pushdown a new value into the Nest.

3. EnMm and =EnMm

As 1. and 2. but with the store address n modified by the contents of Modifier cell m. E.g.

E2M4 =E10M12

No half-word addressing is used by the ASTRA system and so it is not discussed here.

4. Qq and =Qq

Fetch (store) the contents of Q-store q to(from) the Nest. E.g.

Q14 =Q10

5. Cq =Cq Iq =Iq Mq =Mq

As 4. on the Counter, Increment and Modifier parts of Q-store q. The 16th-bit of each field is treated as a sign bit and is extended to full word length on fetching to the Nest. E.g.

I4 =M10

6. MpMq and =MpMq

So-called Indirect Addressing takes the sum of the Modifier parts of Q-stores p and q as the Main store address from which to fetch into or store from the Nest. E.g.

M12M13 M0M14

M0M14 would be used in preference to E0M14, which has the same effect, since the instruction is shorter.

7. MpMqQ and =MpMqQ

After performing the actions of 6. the contents of Q-store q are changed as follows : the Counter is decremented by 1, the Increment is unchanged, and the Modifier is incremented by the value of the Increment. E.g.

M13M14Q =M0M12Q

A terminating Q symbol may also be applied to type 3. instructions, with the same effect.

8. MpMqN and =MpMqN

The effective Main store address is taken to be Mp + Mq + 1. Otherwise as 6. QN may be appended in which case the actions of both 7. and 8. are carried out.

9. Qp TO Qq Cp TO Qq Ip TO Qq Mp TO Qq

Transfer either all or the relevant field of Q-store p to the equivalent field of Q-store q.

10. M+Iq and M-Iq

Increment or decrement the contents of Modifier q by the value of Increment q.

11. + - * /

Perform the relevant arithmetic operation on the top two cells of the Nest and leave the result in their place i.e. a popup of one cell : (N2 op N1), N3, . . .

12. ERASE

Erase the top cell of the Nest and popup the remainder.

14. DUP DUPD

Duplicate the top cell (top two cells) of the Nest.

15. REV REVD

Reverse the order of the top two cells (top two pairs of cells) of the Nest i.e. N1,N2 to N2,N1 (N1,N2,N3,N4 to N3,N4,N1,N2).

16. PERM

Reorder the top three cells of the Nest from N1,N2,N3 to N2,N3,N1.

17. CAB

Reorder the top three cells of the Nest from N1,N2,N3 to N3,N1,N2.

18. J a

Jump to address a.

19. J a (comp) Z where (comp) is =, #, >, >=, <, <=.

Jump if N1 compares with zero and popup one cell.

20. J a= J a#

Jump to address if N1=N2 (N1#N2) and in any case popup the Nest by one.

21. JS a

Jump to address a, pushing down the address of the instruction in the SJNS, i.e. the subroutine jump.

22. EXIT n

The subroutine return instruction. Jump to the address given by the contents of the top cell of the SJNS plus n half-words and popup one cell of the SJNS. Since the JS instruction is one half-word in length, the common subroutine exit is :

```
EXIT 1
```

23. SHL n SHLCq

Shift the top cell of the Nest logically n places or Cq places, positive n to the left and negative n to the right.

24. SHA n SHA Cq

Shift as 23. but arithmetically i.e. preserving the sign bit.

25. SHC n SHC Cq

Shift as 23. but cyclically.

26. SHLD n SHLD Cq

Shift logically the double length word formed from the top cells of the Nest, N1 being the more significant half.

These machine instructions and the remaining few not described can be written into any ASTRA program when required, possibly for optimisation, by prefixing them with an asterisk. E.g.

```
                    *ZERO
                    *SHLD+16
                    *=M14
```

Symbolic store locations of ASTRA e.g. i and j declared by :

```
                    integer i, j
```

can be fetched and stored to and from the Nest by pseudo machine code instructions :

```
                    **i
                    **=j
```

Ordinary ASTRA labels (and private compiler labels) can also be incorporated :

```
                    *J 1 = Z
                    *JS 106P
          1:        . . . . .
```