The Automatic Programming Information Centre
Studies in Data Processing No. 5

ALGOL 60 IMPLEMENTATION

B. Randell and L. J. Russell

Academic Press

London New York San Francisco

A Subsidiary of Harcourt Brace Jovanovich, Publishers

ALGOL 60
IMPLEMENTATION

Also in the series

A.P.I.C. Studies in Data Processing
(General Editor: Richard Goodman)

No. 1

Some Commercial Autocodes
A COMPARATIVE STUDY

by

E. L. Willey
Marion Tribe
A. d'Agapeyeff
B. J. Gibbens
Michelle Clark

No. 2

A Primer of ALGOL 60 Programming

by

E. W. Dijkstra

Together with a Report on
the Algorithmic Language ALGOL 60

No. 3

Input Language
for Automatic Programming Systems

by

A. P. Yershov
G. I. Kozhukhin
U. M. Voloshin

No. 4

Introduction to System Programming

Edited by

Peter Wegner

A.P.I.C. Studies in Data Processing
No. 5

ALGOL 60
IMPLEMENTATION

The Translation and Use
of ALGOL 60 Programs on a Computer

by
B. RANDELL *and* L. J. RUSSELL
with a Foreword by
E. W. DIJKSTRA

Published for
THE AUTOMATIC PROGRAMMING INFORMATION CENTRE
Brighton College of Technology, England

by

1964
ACADEMIC PRESS
LONDON AND NEW YORK

ACADEMIC PRESS INC. (LONDON) LTD.
Berkeley Square House
Berkeley Square
London W.1

U.S. Edition published by
ACADEMIC PRESS INC.
111 Fifth Avenue
New York 3, New York

To
E.R. and M.A.R.

Library of Congress Catalog Card No. 63-23326

PRINTED IN GREAT BRITAIN BY
WILLMER BROTHERS AND HARAM LTD.
BIRKENHEAD

Foreword

I am very happy that the authors of this book have been so kind as to ask me to write a foreword to it, because the presence of their manuscript on my desk relieves my somewhat guilty conscience in more than one way.

Accidental circumstances made me play some role in the initial stages of that part of their work to which they refer as 'the Whetstone Compiler', a fact which I should regret forever if their acquaintance with our solutions would have withheld them from looking for better ones themselves. The manuscript of this book, however, has convinced me that there is no reason for such regret. On the contrary: both the structure of the object program and the structure of the translator are of such an illuminating perspicuity that one can only be grateful when one's earlier work has been allowed to act as source of inspiration.

The second reason for my guilty conscience is related to the book itself. When they were urged to write it they were so imprudent as to ask my advice, whether I thought that they could write such a book, whether there was a point in trying it, etc. Impressed by the clarity of their previous writings I answered in a most encouraging and confirmative way, all the time aware of the fact that the preparation of a manuscript like this would be a tremendous task. The production of a piece of mathematical writing is always a considerable task which puts great demands on one's accuracy and ability to be concise but not obscure. Algorithmic translators, however, are hardly a standard subject of scientific publications, and the authors, who had to create a considerable portion of their 'metalanguage' to describe their subject, must have suffered greatly from this lack of a tradition. But the high degree of readability of their product, I am happy to say, removes one more burden from my conscience.

I am very glad that this manuscript has been finished and that it will be published in book form. As far as the latter is concerned I should like to add 'and the sooner the better', because I think its publication to be very important. In the Preface the authors express their hope that their 'attempt at a description of the problems of implementing a language such as ALGOL on present-day computers will be of interest to the designers both of programming languages and computers.' I should like to use my present privileges to point out that I think this hope – all appreciation for understatements included – too modest. Of course I hope that it 'will be of interest' but I hope much more. For most of the difficulties met during translation can be traced down to a single source, *viz.* either an awkward feature of the language or an awkward feature of the machine. Now I hope that we all agree that there is no point in imposing unnecessary burdens on the translator or its compos-

ers. On the contrary they make translators unnecessarily expensive to construct, unnecessarily expensive to run and, worst of all, they tend to affect the reliability and trustworthiness of the whole system. If we agree on this, the art of language designing becomes the art of designing a powerful and systematic language primarily from those elements which can be processed elegantly by a translator – and this book shows that this set is certainly not empty – and the art of machine designing becomes the art of building-in such features that can be used by a translator in a neat and orderly way – and the Control Routine described in this book shows some of these features. Many of the signposts along the road to improvement in the computer field are set by the experiences gained by the implementors. And as this improvement is very dear to my heart I can only end by wishing this book a wide circulation and an intelligent audience that will not fail to hear its message.

Technological University
Eindhoven, Netherlands
October 1963

EDSGER W. DIJKSTRA

Preface

Our main intention in writing this book has been to present a full description of an ALGOL 60 Compiler, originally developed for the English Electric KDF9 Computer. In fact the information contained in this book has already been used in order to produce similar compilers for three other computers, a Ferranti PEGASUS, ACE at the National Physical Laboratory, and an English Electric DEUCE.

We have attempted in the description of the compiler to give the reasons for choosing the particular techniques that we have used, and in certain cases to describe possible alternative techniques; furthermore, in order to place our compiler in a proper perspective, we have included a short survey of the various published descriptions of other ALGOL Compilers and translation techniques. By these means it is hoped that a reader who wishes to implement ALGOL on a computer will be able to evaluate the various possible techniques, and perhaps to take and develop any of the techniques that he might consider to be suited to his own requirements.

It is further hoped that the book will be of use to readers with a more general interest in computers and automatic programming systems – in particular that our attempt at a description of the problems of implementing a language such as ALGOL on present-day computers will be of interest to the designers both of programming languages and computers.

The writing of this book has been a very enjoyable though somewhat exhausting experience. When we originally undertook to write a complete description of our ALGOL Compiler, we had little idea of the magnitude of the task ahead of us. That we have managed to complete this task is largely due to the help and encouragement we have had from many friends and colleagues.

First and foremost it is a pleasure to acknowledge our indebtedness to Professor E. W. Dijkstra, now at Eindhoven University – indeed, without his great assistance in describing to us his pioneering work with Mr J. A. Zonneveld, on the ALGOL Compiler for the X1 Computer at the Mathematical Centre, Amsterdam, our own ALGOL Compiler would have been a very inadequate and inelegant affair, and this book would never have been contemplated. It is also a pleasure to acknowledge the invaluable assistance we have had from Mr M. Woodger of the National Physical Laboratory, from the first days when we were considering whether to attempt this task, right up to the final stages of proof reading.

Our thanks are also due to Mr F. G. Duncan, of the Data Processing and Control Systems Division of the English Electric Company, without whom KDF9 ALGOL would soon have degenerated into a mere dialect of ALGOL, and to Mr M. A. Batty, of the Atomic Power Division, whose painstaking

checking of the drafts, coupled with an unerring eye for a vague or inaccurate description, caused much re-writing. Equally, we are indebted to Mrs M. French, who has typed and re-typed the drafts of this book with patience and accuracy. However, it must be emphasized that all remaining errors and inadequacies in this book are solely the responsibility of the authors.

This book is published by kind permission of the English Electric Company Limited, to whom we wish to express our thanks, in particular for allowing us to include the complete logical flow diagrams of our compiler.

B. RANDELL
L. J. RUSSELL

Leicester, October 1963

Contents

1 ALGOL COMPILERS

1.1 INTRODUCTION

The International Algorithmic Language, 'ALGOL 60', a language for the description of numerical processes, both for purposes of communication and for use on automatic computers, has been the subject of an earlier book in this series—'A Primer of ALGOL 60 Programming', by E. W. Dijkstra [18]. The present book is concerned with the problems of implementing ALGOL on a computer, and contains a detailed description of an ALGOL 60 Compiler, which, by translating ALGOL text into a form suitable for a computer, allows the computer to be programmed in ALGOL. This compiler has been designed by the authors, at the Atomic Power Division of the English Electric Company, Whetstone, Leicester, England, for the KDF9 Computer, and is for convenience referred to as the Whetstone Compiler.

1.1.1 ALGOL 60

The definitive description of ALGOL 60 is contained in the document 'Revised Report on the Algorithmic Language ALGOL 60', issued by the International Federation for Information Processing [55]. The original ALGOL 60 Report [53] was found to contain various errors and ambiguities, and at a meeting of some of the authors of the Report, in Rome, during April 1962, agreement was reached on several amendments, which are incorporated in the Revised Report. The International Federation for Information Processing has assumed responsibility for resolving the remaining ambiguities and for the future development and refinement of the language.

ALGOL 60 is a language, based on normal mathematical notation, which has been widely used for the publication of the details of computational processes, in the form of 'algorithms', and which is sufficiently precise to be used for the programming of automatic digital computers. The basic constituents of ALGOL programs are 'statements' and 'declarations'. Statements are used to indicate the various actions which are to be carried out by a program, and declarations to describe the meaning attached to the various names (or 'identifiers') used in a program.

Example

```
     real a,b; integer i;
L:   a := 3.0;
     i := 1 + a/2;
     b := a - (i + 1) × 4;
     go to L;
```

In this rather trivial example the identifiers *a* and *b* are declared to be

variables of type **real** (i.e. to have normal numerical values, which will be represented to some finite accuracy), and the identifier *i* is declared to be a variable of type **integer** (i.e. to have an integer value, which will be represented exactly).

The identifier *L* is used to label the first statement, which has the effect of assigning the value of the 'expression' given on the right hand side, in this case simply the number *3.0*, to *a*. (The occurrence of an identifier labelling a statement in this way is often described as a 'declaration' of the identifier to be a label.)

The following two statements assign values to *i* and *b*, and the final statement, a 'go to statement', causes a jump to be made to the first assignment statement. Thus the effect of this piece of ALGOL text is to perform a repeated loop assigning values to *a*, *i* and *b*.

The words **'real'**, **'integer'**, etc., given in bold type are considered to be single symbols and have no relation to the individual letters of which they are composed.

The symbols **'begin'** and **'end'** can be used to group statements together to form 'compound statements' and 'blocks' (a block differs from a compound statement in that one or more declarations are given between the **begin** and the first statement). A program is a self-contained block or compound statement.

Example

begin real *a,b*;
a := *1*;
b := *2*
end

A sequence of statements grouped in this way into a block or compound statement can be considered to be a single statement, and thus can be used in the construction of further blocks or compound statements. An important feature of a block is that it 'localises' the validity of a declaration (i.e. the declaration of an identifier is valid only for the block in which it occurs). As a result, the same identifier can be used for different quantities in different blocks; furthermore, storage space need be reserved for variables only for the duration of the block in which they are declared.

Other features of ALGOL include 'conditional' expressions and statements.

Examples

x := **if** *i* > *0* **then** *n* + *1* **else** *n* − *1*;

This statement assigns the value of '*n* + *1*' to *x* if *i* is greater than zero, otherwise the value of '*n* − *1*'.

if *i* = *0* **then** *x* := *y* **else go to** *L*;

This statement assigns the value of *y* to *x* if *i* is zero, otherwise causes a jump to be made to the statement labelled with *L*.

So far identifiers have been used to represent real and integer variables, and labels. By means of a 'procedure declaration' an identifier can be associated with a piece of ALGOL text.

Example

```
begin procedure P; a :=b + c;
      b := 1;  c := 2;
      P;
      b := a;
      P
end;
```

is equivalent to

```
begin b := 1;  c :=2;
      a := b + c;
      b := a;
      a := b + c
end;
```

Here P is used as shorthand for the statement '$a := b + c$', in fact as a 'procedure statement'.

Identifiers can also be used to denote 'Boolean variables', which can have the value **'true'** or **'false'**, 'arrays' of variables, 'switches', which are basically lists of labels, and 'parameters' to procedures.

However these features will not be discussed further, as the aim of this very brief account has been merely to give some idea of the ALGOL language. Readers who are not already familiar with ALGOL are recommended to study the book by Dijkstra mentioned earlier, or any of the various other descriptions and teaching manuals of ALGOL (e.g. Bottenbruch [10], Naur [54], McCracken [51], or the KDF9 ALGOL Manual [72]).

1.1.2 The Implementation of ALGOL

It can be seen that ALGOL is very different from the language of present-day computers. In order to be able to use ALGOL as a programming language it is necessary to overcome the great disparity between ALGOL and normal machine language.

It would of course be possible to program the computer to read in ALGOL text, and to carry out the operations specified by the ALGOL statements (i.e. to obey the ALGOL text 'interpretively'). However it is much more convenient to apply a preliminary transformation to the ALGOL text in order to obtain a program more suitable for use on a computer. (This program may or may not be in machine language.)

The reason for the preliminary transformation is to avoid unnecessary repetition of the work involved in relating an ALGOL program to the equiva-

lent computer operations. For instance the task of examining an arithmetic expression, and converting it into a set of sequential arithmetic operations could well be carried out once, rather than each time the expression is to be evaluated.

Example

$a - (b + c \times d) / e$

Here the order in which the calculation is to be performed, which is given by the normal rules of arithmetic precedence, and by the parentheses, is

1. Multiply c and d
2. Add the result to b
3. Divide the result by e
4. Subtract the result from a

A further example of the usefulness of a preliminary transformation is in connection with the scanning of the ALGOL text which is necessary in order to interpret the meaning attached to an identifier.

Example

```
begin real a;
      ...              (Dots indicate further ALGOL statements)
      begin real b;
            ...
      L:    b := 3;
      M:    b := 2;
            go to N
      end;
      ...
N:    a := 4
end
```

The successor to the statement '**go to** N' is found by first searching the block in which this statement occurs, and then the outer block, until a statement with the label N is found. This scanning need be done only once, and then, each time the statement '**go to** N' is obeyed, it can use the known location of the statement labelled with N.

Such a preliminary transformation is performed by a program known as a 'translator', and the transformed ALGOL program is known as an 'object program'. (The original ALGOL program being known as a 'source program'.) The authors use the term 'compiler' to describe a program which reads ALGOL text, translates this text into an object program and uses this object program to perform the computation specified by the ALGOL text, whilst providing all the necessary facilities for checking and testing the ALGOL text. Thus an ALGOL 60 Compiler allows a computer to be programmed in ALGOL

60, without the need for any knowledge of the normal machine language.

This technique of using a compiler to bridge the gap between normal machine language and a language which a human being finds suitable to use is of course not peculiar to ALGOL. A large number of compilers have been developed over the last ten years or so, particularly 'algebraic compilers' (i.e. compilers which allow the specification of a computation in algebraic statements) for use on scientific problems. However, virtually all of these compilers have been for languages which have been designed with a particular computer in mind. (One of the most widely known and used is the FORTRAN language [6], originally developed for the IBM 704 computer, but since used as a basis for compilers for various other computers.) The importance of ALGOL is that it is completely machine-independent, and has received international recognition as a language for describing numerical processes, for purposes of publication, as well as for use on computers. This is not to say that ALGOL 60 is the final word in the definition of a machine-independent language for numerical processes. On the contrary, just as ALGOL 60 has benefited from experience on earlier algebraic languages, it is to be expected that any future languages will profit from the great amount of work on the design and implementation of languages that has been prompted by the publication of ALGOL 60.

1.2 ALGOL TRANSLATION TECHNIQUES

1.2.1 The Tasks of a Translator

An ALGOL translator is a program which takes as its input the coded representation of an ALGOL source program and produces as its output an object program (which may or may not be in machine code). Whether the object program is in fact in machine code or not, the task of the translator is not defined until it has been decided how to represent the various features of the ALGOL source program in the object program. As a result the boundary between the transformation to and the execution of the object program is somewhat blurred, and may vary considerably in different translators. (For instance storage for scalars and arrays could be allocated during translation or during the running of the object program.) However there are certain aspects of this task of transforming an ALGOL program into a form more suitable for use by a computer which are obviously the responsibility of the translator.

These include:

(*i*) the recognition of the various source language constituents from the coded representation of the source program;

(*ii*) the analysis of the structure of the ALGOL program (e.g. locating corresponding **begin** and **end** symbols, finding the extent of conditional statements, etc.);

(*iii*) the linking up of the use and declaration of identifiers; and

(*iv*) the transformation of arithmetic expressions into a suitable sequence of simple arithmetic operations, by applying the normal precedence rules, and analysing the bracket structure.

The first and last items are to a large extent common to translators for any algebraic language. Item (*i*) is not usually of general interest since it depends largely on the actual method of presenting a source program to a translator. However item (*iv*) has been the subject of much effort ever since the development of the first algebraic compilers and a large number of methods of translating arithmetic expressions have been developed. For this reason the subject of arithmetic expression translation is treated separately in section 1.3.

On the other hand items (*ii*) and (*iii*) are required specifically for the translation of ALGOL, and did not arise in earlier algebraic translators. Item (*ii*), the analysis of the ALGOL statement structure, arises from the recursive definition of statements and hence requires a technique of translation entirely different from those used for earlier 'statement-at-a-time' languages. Item (*iii*) is a task peculiar to ALGOL translators because of the introduction

in ALGOL of the technique of declaring at the head of each block the identifiers used for the variables, arrays, etc., which are local to that block.

1.2.2 One-Pass and Multi-Pass Translation

The conversion of the source program into the object program can be regarded as a series of simple transformations rather than as one complex transformation. A single transformation could be applied to the source program as a whole; alternatively a set of simple transformations could be applied in turn to each small section of the source program. In a 'one-pass' translator the entire sequence of simple transformations is applied in turn to each section of the program, so that the object program is generated during a single scan of the source program. This is in contrast to a 'multi-pass translator' in which a sequence of transformations is applied to the program as a whole.

Stated thus, the differences between a one-pass and a multi-pass translator seem fairly trivial. However this is not usually the case. In a multi-pass translator it is possible to collect much more information about the source program, during one or more preliminary scans, than is available during a single scan translation scheme. This extra information can be used to produce a more efficient object program from the source program. In the more sophisticated multi-pass translators considerable effort is involved in examining the source program, often to the extent of tracing through the flow of the source program, in order to detect situations which can be optimized (e.g. simple use of subscripts within a loop). On the other hand a simple, and fast, translation scheme is very useful where the time taken by such an optimizing translator cannot be justified. (This is often the case during the debugging of a program, when frequent re-translations are required.) In such a situation a strictly one-pass translator can often have the merit that the time taken to read in the source program can largely, or even completely, cover the time taken for translation.

1.2.2.1 Intermediate Languages

The technique of transforming the source program into the object program by means of a sequence of simple transformations leads naturally to the idea of 'intermediate languages', which mark the completion of various important stages in the translation process. For example a simple two-stage translator might involve the translation of a program first to some standard assembly language, and then from the assembly language to machine language. Such a translator can take advantage of the existence of an assembly program which deals with questions of storage allocation, absolute jump addresses, etc., by including this assembly program as the last stage of translation. There has in the past been a tendency for such translators to produce detailed print-outs of the assembly language version of the translated program for use by the programmer for purposes of debugging. However the realization that such a system would require a programmer to know two

languages in order to write and check out a program has spurred the development of translators which allow a programmer to check out his program in the original source language.

In general, however, intermediate languages do not have a separate existence and are entirely internal to the translator, being the concern only of the translator writers. Indeed, users need not even be aware of their existence.

An intermediate language can become of importance in the case of a translator with alternative inputs. Two somewhat different problem oriented languages might be translated into the same intermediate language, and thereafter use the same translation process. Similarly a translator may have alternative outputs. Thus machine coding would be produced for two different computers using a single translator, having two alternative last stages. The whole question of translators with alternative first and last stages has received much attention recently. The need for an intermediate language which could act as a Universal Computer Oriented Language (UNCOL) has been proposed (Strong *et al.* [67], Steel [66]), but the design of such a language gives rise to many problems. Certainly, if an UNCOL could be produced it would greatly decrease the effort necessary for the production of translation systems for new problem-oriented languages or computers. In theory the numbers of translators it is necessary to write for n programming languages and m computers would be reduced from '$m \times n$' to '$m + n$'. However there has been much controversy over the possibility of designing an UNCOL which would be of practical value; comments have ranged from

> '... UNCOL is an exercise in group wishful thinking', (McCarthy [50]),
> to
> 'As each new machine is produced, all that would be necessary in the way of systems programming is a single translator to convert UNCOL to the new machine language.' (Strong *et al.* [67].)

Another use for an intermediate language is with an interpreter. Then the final stage of translation is omitted, and the object program generated by the translator is used by a control routine at run time (this is the case with the Whetstone Compiler). A judicious choice of such an intermediate language can often result in various gains (compact storage of object program, simplicity of translator etc.) which can in certain circumstances offset the decrease in running speed of the object program. This subject has also been dealt with in a paper by Grau [26], who proposes an intermediate language in a form somewhat similar to that used in the Whetstone Compiler. It is pointed out that such an intermediate language could also indicate the way in which machine hardware might develop in the future.

1.2.3 ALGOL from an Implementor's Viewpoint

An important feature of ALGOL is the manner in which it is defined in the Revised ALGOL 60 Report. Earlier languages did not have the benefit of

such a set of well-defined rules of syntax; indeed many early languages were virtually defined by the action (not always predictable) of their compilers. However the Revised ALGOL Report, using a notation that has since become known as 'Backus Normal Form' (see Backus [7]) to describe the syntactical rules governing the formation of ALGOL text, provides a very useful reference manual for translator writers.

Unfortunately the rigour with which the syntax of ALGOL is defined does not extend to the semantics. In fact the informal semantics often serve to qualify the rules of syntax, instead of merely explaining the meaning of syntactically correct ALGOL formations. (An instance of this occurs in section 4.6.6 of the Revised ALGOL Report. Were it not for this section the syntax and semantics of for statements given in the preceding sections would result in the effect of a go to statement leading into a for statement being perfectly well-defined.) An attempt to avoid the shortcomings of the system of using formal syntax and informal semantics to define a language is contained in an interesting paper by Floyd [24]. This paper gives a rigorous and very concise definition of a language similar to ALGOL, but lacking certain important features such as the call by name facility. A paper by Genuys [25] also deals with this subject.

The formal rules of syntax given in the Revised ALGOL Report describe many of the features of ALGOL by means of recursive definitions. This results in a language very different from earlier algebraic languages. Instead of being formed from a simple sequence of separate statements an ALGOL program can be a block (itself a form of statement) which is formed from many constituent blocks and statements. Similarly the recursive definition of arithmetic expressions automatically allows the full variety of arithmetic expressions as subscripts and as parameters to function designators.

This system of recursive definitions has naturally had a considerable effect on the design of translators. Instead of translating each statement separately, using a different routine for each kind of statement, as was possible in translators for earlier algebraic languages, an ALGOL translator must be capable of dealing with a recursive statement and expression structure.

Equally challenging to implementors is the problem of translation of ALGOL programs containing procedures used recursively. Such recursive use of procedures is allowed either directly, by means of a recursive declaration for a procedure, or formally, by means of an actual parameter. In fact the whole subject of procedures and parameters, in particular parameters called by name, which is a development of the earlier concept of subroutines and their arguments, gives rise to many interesting problems in implementation.

One of the most vexed questions of interpretation of the Revised ALGOL Report is in connection with the possibility of a call on a procedure by means of a function designator causing what are sometimes known as 'side effects'. These arise from the ability to make assignments to formal variables called by name or to variables which are non-local to the procedure body, and to leave the procedure body by jumping to a formal or a non-local label. When a procedure which includes such features is called by a function designator it

is possible to write an expression in which the normal commutative and associative laws do not apply. The drawback of interpreting the Revised ALGOL Report as allowing side effects is the lack of strict rules regarding the order of evaluation of primaries in an expression, of parameters called by value, etc. The case against allowing such 'facilities' is made in a letter by Arden, Galler and Graham [4], which has been answered by Dijkstra [17]. The authors' views on this subject are in agreement with those expressed in a paper by Higman [33], in which he states

> '... the plain fact of the matter is (1) that side effects as a principle are necessary, and (2) programmers who are irresponsible enough to introduce side effects unnecessarily will soon lose the confidence of their colleagues, and rightly so.'

Another important feature in ALGOL is that of block structure and the associated concept of the scope of identifiers. Variables and arrays can be declared at the head of the block in which they are needed, and have no existence outside this block. Various storage allocation techniques have been developed to take advantage of this feature in order to make compact use of machine storage. Thus a minimum of working storage is used by a program without the need for any directives extra to the normal program text. Equally important for compact use of machine storage is the ability to use arrays with variable bounds. In most implementations of ALGOL the space required for an array is determined by evaluating such variable array bounds on entry to the block in which the array is declared. By this means the amount of storage allocated to an array can be varied during the running of the translated program.

One of the most immediate consequences of the large number of new and challenging features of ALGOL has been the tendency for implementors to define subsets (or even dialects) of the language. In fact there are several valid reasons for such a course of action.

Even though ALGOL is a great step forward as an algebraic programming language it is by no means perfect. The implementation of certain anomalous features of the Revised ALGOL Report (the consequences of which were probably not even realized by the authors of the report) is of questionable value. (A case in point is that of the use of integer labels in designational expressions—see Appendix 2.)

At many computer installations ALGOL is of interest mainly as a programming language, and the overriding requirement is that programs should be translated and run on the existing equipment as efficiently as possible. In such a situation the importance of some of the more novel aspects of ALGOL (e.g. the ability to call a procedure recursively) tends to be underrated. ALGOL was designed as a machine-independent language whereas earlier languages were usually designed with the features and failings of a particular machine in mind. As a result the efficiency of a full ALGOL translator would often be somewhat lower than that of a translator for a language

designed specifically for a machine, and instead only a subset of ALGOL is implemented.

However in assessing the efficiency of a translation system it is all too easy to concentrate solely on easily measurable quantities such as speed of translation and of translated program, and amount of storage space taken up by the translated program. What really matters in a computer installation is the rate at which useful information can be produced and the cost, both in time and money, of producing this information. Thus the efficiency of a translation system can only be assessed within the context of the installation in which it is used. This means that the ease with which programs are prepared and used must also be taken into account. Program preparation is dependent on such features as the programming language used, the facilities for amending, checking, and testing a program, and also the ease with which a programmer may draw upon published material (in the form of techniques, algorithms, etc.) and library routines. The ease with which a program is used depends very much on the efficacy of the operating system, which can schedule the machine's activities, maintain a logbook, organize the use of peripheral equipment, etc.

However, at establishments where ALGOL is of interest as a theoretical language rather than as a programming language the question of efficiency of translation on present-day computers is of less importance. In fact one of the most important results of the publication of ALGOL has been the stimulating effect it has already had on research into formal languages and their specification, on compiling techniques, and even on machine design.

1.2.4 ALGOL Translators

At the time of writing very few detailed descriptions of any ALGOL translators have been published—in fact the only one known to the authors is that by van der Mey [69] describing the ALGOL 60 translator for the ZEBRA Computer. This is of course not surprising in view of the complexity of the task of translating ALGOL. However many interesting articles have been published which give details of the very varied techniques employed in the various ALGOL translators. This has encouraged the authors to attempt a brief survey of ALGOL translation techniques. In this survey an attempt is made to compare and contrast different ALGOL translation techniques rather than to give a detailed historical survey of the development of ALGOL translators. The authors have attempted to make this survey as complete as possible; apologies are made for any omissions, and for any inadequacies of interpretation of the articles cited. Due to the large number of techniques that have been developed for the translation of arithmetic expressions, not only for ALGOL but for all the earlier algebraic languages, this subject is considered separately, in section 1.3, even though this causes a certain amount of duplication in the description of some of the ALGOL translators.

One of the most important characteristics of an ALGOL translator is the method by which it analyses the structure of the text of an ALGOL program.

Not surprisingly a large number of different techniques have been developed for performing this analysis. One possible classification of these techniques is into two groups, namely direct methods and syntax oriented methods, though the boundary between the two groups is somewhat blurred.

Obviously any method of analysing the structure of an ALGOL program must take into account the syntactic definitions of ALGOL. In many translation schemes the syntactic units of an ALGOL program have a direct counterpart in the translator, either because the translator is composed of routines corresponding to the syntactic units, or because the translator records information about the structure of the text in terms of the syntactic units. This is not the case in what are here described as direct methods, which usually work in terms of the basic symbols of the ALGOL text. The direct methods, though often quite efficient, tend to be rather *ad hoc*; on the other hand, since ALGOL is largely defined using syntactic formulae, there is a certain attractiveness about the idea of a syntax oriented translator. It is likely that theoretical investigations into syntax oriented methods will lead to the development of programming languages which are more suited to this approach than is ALGOL.

1.2.4.1 Direct Methods

The 'direct methods' of translation discussed in this section are for convenience further subdivided into two groups. The first group consists of translators whose course of action is based on pairs of delimiters. In the second group of translators each delimiter has a priority level associated with it, and comparisons of priority levels are used in order to decide the appropriate course of action.

1.2.4.1.1 Delimiter Pair Techniques. The translation technique used by members of the ALCOR Group has been described by Samelson and Bauer [62, 63]. The ALCOR (ALGOL CONVERTOR) Group was formed in 1959 to facilitate the interchange of ALGOL programs between different computers at various scientific institutes. The translators for each computer were prepared using logical plans developed at the Institute for Applied Mathematics at Mainz University. In addition agreements were reached on a common hardware representation (based on 5-hole paper tape) and input-output facilities. The language accepted by translators of the ALCOR Group is a restricted form of ALGOL 60, which has been described by Baumann [9].

The ALCOR translation system is based on the use of a 'push-down store', (i.e. a store with 'last-in-first-out' properties, called a 'cellar' in the Samelson and Bauer paper [62]). During a left-to-right scan through the text of an ALGOL program the cellar is used as a holding store for information obtained from the text that has already been read, until the translation of the information can be completed. The information is kept mainly as representations of various ALGOL basic symbols. The last-in-first-out properties of the cellar are used to permit the analysis of the bracket structure of the program. For

example the delimiter **begin** which starts a block or compound statement is kept in the cellar until the matching **end** is found. During translation of the block or compound statement any further **begin** delimiters will be placed in the cellar so shielding the original **begin**. The last-in-first-out properties of the cellar automatically allow the matching of **end** delimiters with their corresponding **begin** delimiters. These same properties of the cellar are also used to re-order the arithmetic and Boolean operators to allow for the normal rules of algebraic precedence. (This use of the cellar for the translation of arithmetic expressions is discussed in more detail in section 1.3.4.)

The course of translation is controlled by what is called a 'transition matrix'. The rows and columns of this transition matrix correspond to the various ALGOL delimiters. The delimiter at the top of the cellar and the delimiter currently being scanned are used to select an element of the matrix, which then causes the appropriate action to be taken by the translator. A single transition matrix, developed at Mainz, is the basis of the translators for each of the different computers of members of the ALCOR Group.

The final part of the Samelson and Bauer paper [62] describes a technique for efficient handling of subscripted variables used in for statements. The technique is based on the removal from within the for statement of as much as possible of the calculation necessary to determine the particular array element specified by a subscripted variable. This 'optimization' of subscripted variables is applied to those having linear subscript expressions.

The translation techniques used in the NELIAC family of compilers are the subject of a book by Halstead [28]. NELIAC, which is described as 'a dialect of ALGOL', has more in common with ALGOL 58 [56] than ALGOL 60. The most interesting feature of the NELIAC project is that only a single basic NELIAC Compiler was hand coded (for the Univac M-460). This was then used to translate a version of itself written in NELIAC, and the resulting Compiler used to write a full NELIAC Compiler. With only an absolute minimum of further hand coding NELIAC Compilers have been written in NELIAC for various other computers. In fact complete listings, in NELIAC, of compilers for the M-460, the CDC 1604, and the IBM 704 are given by Halstead.

The logical problems of dealing with NELIAC symbols in a NELIAC program are avoided by assigning integer values to each symbol, and then processing these integers rather than the symbols.

Example

if *Delimiter* = *Semi Colon* **then go to** *End Statement*;

Here *Semi Colon* is a simple variable which has been set up with the numerical equivalent of the bit pattern used inside the computer to represent ';'.

The basis of all the NELIAC Compilers is a table of 'next operator' against 'current operator', which is used to select the appropriate generator (i.e. a

subroutine which produces or 'generates' the required coding). Thus this table is similar to the transition matrix of the ALCOR Group.

1.2.4.1.2 Priority Level Techniques. Several ALGOL translators use a technique in which a numerical value (called a 'priority level') is associated with each delimiter. The action of the translator is controlled by comparing the priority levels of delimiters. Naturally the values chosen as a set of priority levels must be dependent on the rules of syntax—an example of this is given in the Revised ALGOL Report itself, in which the rules of precedence of arithmetic operators are given by means of a table (section 3.3.5.1) as well as implicitly by the syntax (section 3.3.1). The relationship between syntax and priority levels is discussed in the paper by Genuys [25] mentioned earlier.

The techniques used in the ALGOL translator developed for the X1 Computer at the Mathematical Centre, Amsterdam, are the subject of a paper by Dijkstra [16]. The translator is again based on the use of a push-down store, here called a 'stack', but the transition matrix of the ALCOR Group is replaced by a 'discrimination vector'. The vector contains one element for each ALGOL delimiter, giving the priority level associated with the delimiter. The translator is made up of a set of separate routines, one for each delimiter. As the ALGOL text is scanned each delimiter invokes the appropriate 'delimiter routine'.

The push-down store, or stack, is again used, mainly as a holding store for delimiters. A comparison of the priority levels of the current delimiter and the delimiter at the top of the stack is used to control the stacking and unstacking of delimiters. This system is used equally for analysing statement structure and for re-ordering arithmetic and Boolean operators. The structure of the program is analysed by matching corresponding delimiters, e.g. **begin** and **end, if then** and **else**, etc., in a similar way to that described for the ALCOR system.

Other papers by Dijkstra [14, 15], describe the techniques of using a run time stack for storage of variables and arrays. This method of storage allocation takes advantage of the block structure of ALGOL to permit a very compact use of storage, and also facilitates the implementation of recursive procedures.

Other translators which are based on the techniques developed by Dijkstra include the Whetstone Translator, a detailed description of which forms the major part of this book, and a translator for a prototype airborne computer (see Higman [32]).

This airborne computer must surely possess the distinction of being the smallest computer on which ALGOL has been implemented—the total storage of the computer comprises 480 words, each of 20 bits. Naturally the very limited amount of machine storage has had its effect on the design of the translator, which in fact uses five passes to convert the ALGOL text into an object program, which is obeyed interpretively. An interesting feature of the paper by Higman is that a detailed description is given of three of the five passes of the translator, and of the interpreter, in ALGOL.

The technique used by Higman to write 'ALGOL in ALGOL' is rather different from the technique used in NELIAC, and is independent of the hardware representation. Symbols given in the program are in general enclosed in string quotes, and transfer functions used to attach a value to each symbol.

Example

if *Next Symbol* = *V*(';') **then go to** *End Statement*;

In order to handle even the string quote symbols, the symbol **c** is introduced. This has the effect of enclosing the symbol following it in implicit 'super-quotes'. Since these super-quotes are implicit, they cause no problems and the symbol **c** itself can be handled by writing **c c.** This provides a means for handling arbitrary sequences of basic symbols, in contrast to that described in section 2.6.3 of the Revised ALGOL Report.

Example

if *Next Symbol* = *V*(**c**') **then go to** *Start String*;

The techniques used in the very successful ALGOL Compiler written at Regnecentralen, Copenhagen, for the GIER computer, have as yet not been published. However the authors have been privileged to see a draft description of parts of the Compiler, in particular a very interesting account of the system for organizing use of the two-level storage of GIER.

The GIER computer has 1024 words of core storage and 12,800 words of drum storage. As a result, the Compiler works in 9 passes (two of which are reverse scans), so that during each pass there is sufficient space in core storage for all the sections of the Compiler which are necessary for the pass. Once again a stack and priority level technique is used to analyse the structure of the ALGOL program and a system of stacked storage is used by the translated program for simple variables and arrays. The translated program is kept on the drum and decisions as to the transfers of program from drum to core store are made dynamically, depending on the amount of storage used for variables and on the relative frequency of use of the various sections of translated program.

A very recent report by van der Mey [69] contains a complete description of the ALGOL Compiler for the ZEBRA Computer. (A brief description of this compiler has also been given by van der Poel [70].) The aim has been to make the compiler as complete as possible; in fact the only restrictions placed on ALGOL are that own arrays must have constant bounds, scalar and array declarations must precede switch and procedure declarations, and the controlled variable of a for statement must be a simple variable.

The report by van der Mey describes the compiler in ALGOL, and as with NELIAC, describes the manipulations carried out on the numerical equivalents of delimeters, etc., rather than using some such technique as given by Higman. The translation process uses stack and priority level techniques but

differs from the methods described earlier in that any identifier or constant following a delimiter is stacked with the delimiter. This has the effect that the evaluation of primaries is done in a different order.

The generated object program is interpreted at run time. Advantage has been taken of the rather unusual order code of the ZEBRA Computer, in which each of the 15 bits of an instruction has a separate significance, in order to carry out the interpretation efficiently. In contrast to the system used by Dijkstra, two stacks are used at run time, one for simple variables and intermediate results, the other for arrays (own arrays are embedded in the object program itself). Arithmetic operations are carried out using a fixed accumulator, and intermediate results are put in the stack only when the accumulator is needed for another operation, instead of stacking all operands, and carrying out all arithmetic operations on the top one or two items in the stack.

A very different approach to the problem of ALGOL translation is contained in a paper by Hawkins and Huxtable [31] which describes the Kidsgrove Optimizing Translator.

During the initial stages of translation, procedures are classified into three groups. The first group consists of type procedures, whose bodies do not contain any procedure calls, and which obey a rigid set of rules, aimed mainly at ensuring that the procedures cannot give rise to any side effects. The remaining procedures form the second and third groups; the third group consisting of those procedures, which considered only in the context of the program being translated, can be shown to be incapable of being called recursively.

The members of the first group of procedures are determined during a backwards scan through the procedure declarations. Then the flow of the ALGOL program is followed through in order to build up a 'correspondence matrix'. This matrix gives details of the calls made on procedures and is processed in order to determine which procedures belong to the third group.

The next stage of translation is the allocation of storage within each procedure. During the running of the translated program the storage needed by a procedure is added to the top of a run time stack when the procedure is entered, and is deleted from the stack on exit from the procedure. The previous classification of procedures is used in order to simplify, where possible, the operations generated to organize procedure entries and the setting up and deleting of stacked storage.

This stage is followed by a process for expanding for statements in each procedure into explicit ALGOL statements, where possible taking advantage of simple cases of subscripted variables in for loops, in order to apply optimization techniques. These techniques are designed to remove all unnecessary calculation of the addresses specified by subscripted variables from within the for loops, and to allocate index registers optimally.

After the for statements have been dealt with the procedures which form the program text are separated, and translated into an intermediate language,

again using a stack and priority level technique. This intermediate language version of the program is then processed in order to remove common sub-expressions before finally generating machine code.

1.2.4.2 Syntax Oriented Methods

It was mentioned earlier that syntax oriented translators can be divided into two groups, namely those which are composed of routines corresponding to syntactic units, and those which relate the text being translated to the syntactic units of which it is composed, and manipulate representations of these syntactic units.

The first group uses what is, in a sense, the most natural method for translating ALGOL, and these translators can be regarded as direct descendants of those used for translation of earlier algebraic languages. In such languages programs consisted of a number of separate statements of various kinds (assignment statements, jump statements, etc.). The translator would be composed of a set of 'statement routines', each capable of translating a particular kind of statement. To use this technique for an ALGOL translator the routines would have to be capable of being used recursively. Furthermore, in ALGOL there is much less distinction between the various kinds of statements—for example all statements (other than dummy statements) can contain expressions. It is therefore logical to base a translator on routines which correspond to each and every syntactic unit, rather than just to statements.

One of the most important features of the second style of syntax oriented translators is that it is possible to make the translator independent of the rules of syntax which it uses in order to translate a program. Indeed such a translator could accept as input data both the program to be translated and the rules for its translation. This has the advantage that it is possible for the rules to be given in a form similar to that used in defining the language, and also that modifications to the language can be accommodated by changing the translation rules rather than the translator. However the efficiency both of the translator and of the object program it produces is dependent on the syntax definitions, and great care must be taken in formulating these definitions if they are to be used to control the working of a translator.

1.2.4.2.1 Syntactic Routine Methods. The ALGOL translator described by Grau [26] is composed of routines which correspond roughly to the various syntactic units, and uses a 'control push-down store'. Items or 'states' stored in the control push-down store indicate the state of the translator at entry to each of the currently activated routines. Thus on exit from a routine the 'state' at the top of the push-down store indicates the reason for entry, and hence determines the course of action to be followed. As the ALGOL program is scanned, the current delimiter and the state at the top of the control push-down store are used to select an element of a 'translation table'. This element determines which routine is to be entered. The main tasks of each routine

are, firstly, the updating of the information given in the control push-down store, and, secondly, the generation of the appropriate object program operations.

A theoretical discussion of 'syntactic routine translation methods' is given in a paper by Lucas [48]. This paper describes the metalanguage used in the Revised ALGOL Report for the syntactic definitions, and then introduces a 'metametalanguage' for describing this metalanguage. The metametalanguage is then used in discussing the various kinds of syntactic definitions used in the Revised ALGOL Report. A description is given of a translator composed of routines for each syntactic unit in ALGOL, and using a push-down store for links and as a holding store for information. Simple flow diagrams are given for each kind of syntactic definition, and a detailed flow diagram given for the routine 'arithmetic expression'. This routine can call itself recursively, and uses the routine 'simple arithmetic expression'.

A third translator which can be regarded as using an approximation to the 'syntactic routine method' of translation is described in a paper by Evans, Perlis, and van Zoeren [21]. This translator does not use a control push-down store but instead is based on the use of threaded lists. A threaded list (see Perlis and Thornton [57]) is an extension of the idea of a simple list structure, i.e. a method of storage of a sequence of items of information, where each item carries with it the address at which the next item is stored. (List structures have the advantage that items can easily be added to, or deleted from, any part of a list, without the need for moving any of the items already in the list; furthermore, since any item can itself indicate a further subsidiary list, they facilitate the manipulation of highly complex data structures). It is stated that an advantage of the threaded list technique is that corrections can be made to an ALGOL text, and the equivalent modified machine code program produced, without the need for complete retranslation. The translator consists of three passes; during the first pass an ALGOL program is converted into a bracket-free form, stored as a threaded list, together with tables of identifiers and constants. The second pass converts this threaded list into relocatable machine code. The third pass takes place during the loading of this relocatable machine code, converting it into an absolute machine code program ready to be obeyed. The conversion of the ALGOL text into threaded list form uses a set of routines called 'recognizers' which in general correspond to syntactic units. The threaded list itself is used to control the translation process. As the ALGOL text is scanned names of the appropriate recognizers are entered into the threaded list. The names are then replaced by the coding obtained from using these recognizers, which might again contain recognizer names. This process continues until the whole of the ALGOL text has been read, and the generation of the threaded list is complete. However, a standard scanning process (in fact the one used in the GAT translator, which is discussed in section 1.3.2) is used to translate arithmetic expressions. It is claimed that this avoids the construction of the needlessly

complicated threaded list structures that would result from following the ALGOL 60 syntax rules governing arithmetic expressions.

1.2.4.2.2 Syntactic Information Methods. Probably the most widely known syntax oriented ALGOL translator is that of Irons. A somewhat expanded version of the original paper by Irons [39] has recently been published [40], giving some detailed worked examples to demonstrate the method of translation. The two inputs to the translator are the program to be translated, and the rules for performing the translation. These rules of translation each consist of a syntax formula, and a description of its semantics (i.e. the action to be taken, machine instructions to be generated, etc.). The syntax formulae are used by the translator in order to analyse (or 'diagram') the ALGOL text. The result of this analysis is a list structure indicating, in sequence, the syntax rules which have been used in order to analyse the ALGOL text. This list structure is then used, together with the semantic information given with each rule of syntax, in order to generate an assembly code version of the ALGOL program. The final task of the translator is to convert this assembly code into machine code.

A variation on the above method of translation has been described by Ledley and Wilson [47]. The syntax formulae, given in the set of translation rules, are tested against the ALGOL program text, which is stored as a set of linked list elements. When a formula is found that applies to a set of symbols in the 'ALGOL list', elements containing these symbols are replaced by a single element which contains a representation of the appropriate syntactic unit. At the same time a small list structure, containing the translated program fragment, is built up, using the semantic information given in the translation rules. This process is repeated until at the end of translation the ALGOL list has been collapsed into a single element, representing the syntactic unit 'program', and the generated lists have merged into a single list structure, which contains the complete translated program.

A third method of translation which can be considered to be a 'syntactic information method' has been briefly described by Floyd [23]. This method uses a push-down store instead of list structures. The translator uses a set of 'productions', which are applied to the program text as it is being scanned, in order to analyse its structure and to generate the appropriate machine coding. The productions are not directly given as syntax formulae, but instead indicate tests and manipulations to be carried out on the items at the top of the push-down store. The items kept in the push-down store in general correspond to syntactic units. The method is thus quite similar to that given by Ledley and Wilson, except that syntactic information which has to await the scanning and analysis of further items of the program text before being used is kept in a push-down store rather than in a list structure.

1.3 TRANSLATION OF ARITHMETIC EXPRESSIONS

Many different techniques have been published for the transformation of an arithmetic expression into a sequence of operations suitable for a computer. Whilst this is only a small part of the total work of an algebraic compiler, it has provided much scope for the development of interesting and varied techniques.

As was mentioned in section 1.2.3, the recursively defined statement and expression structure of ALGOL has led to the development of what might be called 'integrated translation systems', as opposed to 'statement-at-a-time translators' formed of a set of routines for processing expressions and the various kinds of statements. Thus in describing techniques for the translation of ALGOL arithmetic expressions, it must be realized that such techniques are embodied in an integrated translation system, rather than in a separate 'expression routine'.

In this section brief descriptions are given of some of the techniques which have been developed for the translation of arithmetic expressions. Apology is made to the authors of the papers quoted for any inadequacies or inaccuracies in this survey, which does not pretend to be complete.

1.3.1 Early Techniques

Probably the earliest description of a technique for translating arithmetic expressions is given in a paper by Rutishauser [61], published in 1952. (See footnote 1, p. 33.) The method described by Rutishauser uses a technique of repeated scanning of an expression.

During a first left-to-right scan of the expression level numbers are assigned to each operation, operand, and parenthesis. The level starts at zero for the first constituent of the expression, increases by one for each left parenthesis or operand, and decreases by one for each right parenthesis or operator.

The example given by Rutishauser (where the level numbers have been placed under each constituent) is

$$\begin{array}{l} (A_1 \div (A_2 + A_3)) - (A_1 \times A_2 \times A_3) \\ 0\,1\,2\ \ 1\ 2\,3\ \ 2\ \ 3\,2\,1\,0\ \ 1\,2\ \ 1\ \ 2\ \ 1\ \ 2\,1\,0 \end{array}$$

After level numbers have been assigned in this way the expression is scanned repeatedly, in order to generate the equivalent program. The first scan starts with the innermost bracketed sub-expression (found by inspection of the level numbers) and the equivalent program operations are generated. The constituents of the sub-expression are replaced by one operand, with a level equal to that of the operands of the sub-expression. This process is repeated, each time removing the innermost sub-expressions.

By this means the above example would be transformed into a set of operations equivalent to

$$\begin{aligned} R_1 &:= A_2 + A_3; \\ R_2 &:= A_1 \div R_1; \\ R_3 &:= A_1 \times A_2 \times A_3; \\ R &:= R_2 - R_3; \end{aligned}$$

The methods used in the FORTRAN compiler, work on which started in 1954, were not published until some years after completion of the compiler in a paper by Sheridan [65]. This paper contains a description of a very complex system for producing an efficient coding of an arithmetic expression.

Briefly, the translation is performed in a number of steps. (The examples used will be those used in Sheridan's paper.)

The first step is to replace constants and subscripted variables by simple variables. (This can easily be done because of the restrictions on subscript expressions in FORTRAN.) Then extra parentheses are inserted, to make the normal rules of precedence of arithmetic operators explicit.

Thus

$$A + B \uparrow C / D$$

becomes

$$(((A))) + (((B)) \uparrow (C)) / ((D))$$

(The FORTRAN symbols for multiplication and exponentiation are * and **.)

This is done by a set of transformations which operate during a left-to-right scan of the expression.

The third step is to break down the fully parenthesized expression so formed into sub-expressions, or 'segments'; the segments are then broken down further into what are called 'triples'.

The example given is the expression

$$(((A + B) - C) / ((D \times (E + F) / G) - H + J))$$

(The extra parentheses have been omitted.)

The segments formed from this are

1. $(A + B)$
2. $((A + B) - C)$
3. $(E + F)$
4. $(D \times (E + F) / G)$
5. $((D \times (E + F) / G) - H + J)$
6. $(((A + B) - C) / ((D \times (E + F) / G) - H + J))$

A triple is generated from each term in a segment and consists of the segment number, operator, and operand.

The above expression, in triple notation, is

$(1, +, A)$ $(1, +, B)$
$(2, +, 1)$ $(2, -, C)$
$(3, +, E)$ $(3, +, F)$
$(4, \times, D)$ $(4, \times, 3)$ $(4, /, G)$
$(5, +, 4)$ $(5, -, H)$ $(5, +, J)$
$(6, \times, 2)$ $(6, /, 5)$

The set of triples is formed during a left-to-right scan of the fully parenthesized expression. During this scan a counter, which is increased at each left parenthesis, is used to determine segment numbers.

The next step is a right-to-left scan, during which the redundant triples which have been produced from the extra parentheses inserted in step 2 are deleted. The sequence of triples is then repeatedly scanned in order to remove the triples corresponding to common sub-expressions, so that the amount of actual computation performed at run time is minimized.

The triples within each segment are then re-ordered where possible in order to minimize the number of fetch and store operations required, before the triples are finally converted into SAP assembly code. The generation of SAP instructions is performed using a segment of triples at a time, starting with the last segment, and scanning each segment from left-to-right.

1.3.2 Single Scan Techniques

One of the earliest descriptions of a system for breaking an arithmetic expression down into its constituent steps during a single scan is that given by Milnes [52]. The method described by Milnes is, however, not used for the production of machine code by a translator, but as a means of obeying an expression directly, using an interpreter.

During a left-to-right scan of the expression, operations are performed immediately where possible. At the occurrence of a symbol such as a left parenthesis partial results are stored for use after completion of the processing of the sub-expression contained in parentheses.

For example

$$a + (b - c) / (f \times g + e)$$

is performed in the set of steps

$R_1 := a;$
$R_2 := b;$
$R_2 := R_2 - c;$
$R_3 := f;$

$R_3 := R_3 \times g;$
$R_3 := R_3 + e;$
$R_2 := R_2 / R_3;$
$R_1 := R_1 + R_2;$

The actual method of operation is to use one table both for partial results and for subroutine transfer instructions. These instructions which, when obeyed, lead to the appropriate arithmetic operation subroutines, form a record of the operations which are, at any given time, in abeyance awaiting the completion of the calculation of sub-expressions.

The same principles can be used for the generation of machine coding during a single scan of an expression. An early description of such a system is contained in a paper by Wegstein [71]. In this paper a detailed flow diagram is given of a fairly simple algorithm for producing three-address machine code, of a form similar to that shown in the above example.

The translation algorithm scans the expression from left to right, processing one language constituent at a time. The route through the flow diagram is dependent on the type of language constituent encountered (operand, '(', '+', '−', etc.).

Operators are kept in a list until they can be used in three-address instructions. Operands either cause the generating of an instruction for setting up a temporary store

e.g. $R_2 := b;$

or are used in three-address instructions

e.g. $R_3 := R_3 \times g;$

Operations which have had to be held over in the list, until further operations have been dealt with, eventually cause the generation of three-address instructions working entirely on temporary results

e.g. $R_2 := R_2/R_3;$

A paper by Kanner [44] describes a system which is in some ways similar to the above, but which embodies certain improvements, notably concerned with the use of temporary storage. The generation of separate instructions to load a temporary store with an operand is avoided.

Thus

$(a + b) - (c + d) \uparrow e$

instead of being translated into

$$R_1 := a;$$
$$R_1 := R_1 + b;$$
$$R_2 := c;$$
$$R_2 := R_2 + d;$$
$$R_2 := R_2 \uparrow e;$$
$$R_1 := R_1 - R_2;$$

is translated into

$$R_1 := a + b;$$
$$R_2 := c + d;$$
$$R_2 := R_2 \uparrow e;$$
$$R_1 := R_1 - R_2;$$

The system of translation differs from that of Wegstein in that, during the scanning of the expression (again from left to right), both the operands and operators are recorded. The generation of instructions is performed at certain stages by interrupting the recording process, and processing the items that are stored, in the reverse order to the order in which they were recorded. These interruptions occur at right parentheses, and on operators which are lower in priority than the operator currently in abeyance.

A somewhat different technique is described by Arden and Graham [2, 3], in connection with the GAT compiler for the IBM 650 computer. The GAT compiler accepts a much less restricted form of arithmetic expression, allowing mixed mode working (i.e. real and integer variables and constants in an expression) and imposing no limits on subscript expressions. The system of translation differs from those of Wegstein and Kanner, in that arithmetic expressions are scanned from right to left. In addition the algorithm uses a set of 'precedence rankings' to re-order the arithmetic operations in accordance with the normal rules of precedence, instead of branching according to the type of operator.

The algorithm results in the production of a matrix (the 'algorithmic matrix') in which each row is effectively a coded form of three-address instruction.

These three-address instructions use a different temporary storage location for each partial result, and a simple transformation is applied to the matrix to avoid this.

The example given by Arden and Graham

$$(C1 + D1) \times Y2 + 2.768 + (I2 - J2) \times K1 / Z1$$

is thus transformed into the matrix

i	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$
1	*1*	–	*I2*	*J2*
2	*1*	×	*1*	*K1*

3	*1*	/	*1*	*Z1*
4	*2*	+	*C1*	*D1*
5	*2*	×	*2*	*Y2*
6	*2*	+	*2*	*'2.768'*
7	*1*	+	*2*	*1*

This is equivalent to the three-address instructions

$R_1 := I2 - J2;$
$R_1 := R_1 \times K1;$
$R_1 := R_1 / Z1;$
$R_2 := C1 + D1;$
$R_2 := R_2 \times Y2;$
$R_2 := R_2 + 2.768;$
$R_1 := R_2 + R_1;$

The extra information necessary for the production of machine instructions is formed in the algorithmic matrix. Boolean variables giving information as to the type (real and integer) of the operands, whether a partial result is immediately overwritten by the result of the next instruction, etc., are kept in a set of extra columns (six in all) in the matrix. The actual translation into machine instructions is performed using a set of 'translation matrices', one for each operation. The elements of each translation matrix are Boolean functions, to be applied to the last six elements of a row of the algorithmic matrix, in order to decide which machine instructions to generate from a given basic three-address instruction.

One important virtue of this system is that it is, to a large extent, machine-independent, as the actual generation of machine instructions is controlled by the translation matrices alone.

An ingenious system, used in the RUNCIBLE compiler, also on the IBM 650 computer, is described in a paper by Knuth [45]. The translation technique, which is highly machine-dependent, uses a single right-to-left scan.

Because translation is performed during a single pass, the technique used is to generate instructions to meet the most awkward case, and to replace these instructions if later information proves this was unnecessary. Thus, for example, instructions to float a fixed point number are generated in case the number should be used with a floating point operation. If this does not prove to be the case the float instructions are removed. This technique is facilitated by careful choice of internal representations for items manipulated by the compiler (see Lynch [49]).

The paper demonstrates the technique on the example used by Arden and Graham [2], showing that a slightly more efficient object program is generated.

1.3.3 A Double Scan Technique

A variation of the right-to-left scan translation technique has been described by Floyd [22]. In order to translate an arithmetic expression into efficient code for a one-address machine with index registers a bi-directional scan is used. The scanning starts from the left of an expression and continues until the closing bracket of a set of one or more subscript expressions is found. The direction of the scan is then reversed, and the subscript expressions are translated. The left-to-right scan then continues, being interrupted for any further subscript expressions, until the end of the arithmetic expression is reached. Then a right-to-left scan is used to complete the translation of the expression. Thus all translation takes place during scanning from right to left, but precedence is given to subscript expressions.

This can be demonstrated by the transformation of an expression into a set of equivalent simple assignment statements

$$(A\,[i+j\times k]+B)\times(C-D\,[i-j])$$

is transformed into

$$\begin{aligned}
R_1 &:= j \times k;\\
R_1 &:= i + R_1;\\
R_2 &:= i - j;\\
R_2 &:= C - D\,[R_2];\\
R_1 &:= A[R_1] + B;\\
R_1 &:= R_1 \times R_2;
\end{aligned}$$

The algorithm for performing this transformation is given in flow diagram form, the route through the flow diagram being dependent on the type of symbol being processed. The machine storage used for this technique is organized as three 'push-down' stores, (i.e. stores working on a last-in-first-out principle). Push-down stores (which are also known under a variety of other names, e.g. cellars, nesting stores, yo-yo lists), are in fact implicit in all the single scan translations techniques described so far. Thus the 'table' used by Milnes, the 'list' used by Wegstein, though not so described, all work on the last-in-first-out principle. (This subject will be pursued further in section 1.3.4.)

Two of the push-down stores used by Floyd are used for the symbols of the arithmetic expression during the bi-directional scan, the third as a holding store for operations during actual compiling.

However, the major part of the Floyd paper is concerned with improving the efficiency of the resultant coding, and an extended version of the translation algorithm is described. The extensions permit the assessment of the types (real and integer) of partial results, the computation of operations depending solely on constants during translation, the removal of duplicated sub-expressions, etc.

Thus the expression

$$y - z + 1.3 / (z - y)$$

is translated into one-address instructions which are equivalent to the sequence of simple assignment statements

$$R_1 := z - y;$$
$$R_2 := 1.3 / R_1;$$
$$R_1 := R_2 - R_1;$$

1.3.4 Integrated Techniques

It was pointed out in section 1.3 that a technique for translating arithmetic expressions in an ALGOL program would normally be part of an integrated translation system. In this section a few papers describing techniques which are capable of use in such an integrated system are described. (Techniques for the translation of ALGOL are described in section 1.2.4.) An important paper in this field is that by Samelson and Bauer [62], entitled 'Sequential Formula Translation'. (See footnote 2, p. 33.) This contains one of the earliest descriptions of the technique of using a store with 'last-in-first-out' properties, i.e. a push-down store (here called a 'cellar'). Equally important, however, is that it is shown that such a store can be used for the translation of ALGOL statements as well as arithmetic expressions. However, the discussion here will be limited to the translation of arithmetic expressions.

The paper first describes a technique for translating an expression using two cellars, a 'symbol cellar' and a 'number cellar'. The symbol cellar is used as a holding store for operators, whilst parenthesized sub-expressions and further operators with higher precedence are processed. The number cellar contains the stores assigned for the partial results of the expression.

The method of translation is to read one symbol at a time, starting at the left (for the purpose of this description identifiers, numbers, etc., are all treated as single symbols). Identifiers simply cause the generation of an instruction transferring the corresponding variable to the top of the number cellar. The other symbols (parentheses and operators) are compared against the symbol at the top of the symbol cellar, to decide upon the course of action to be taken by the translator. The pair of symbols is in fact used to select an element from a 'transition matrix', which is then used to trigger-off the appropriate action by the translator.

By this means an expression such as

$$(a \times b + c \times d) / (a - d)$$

would be transformed into

$$R_1 := a;$$
$$R_2 := b;$$
$$R_1 := R_1 \times R_2;$$

$R_2 := c;$
$R_3 := d;$
$R_2 := R_2 \times R_3;$
$R_1 := R_1 + R_2;$
$R_2 := a;$
$R_3 := d;$
$R_2 := R_2 - R_3;$
$R_1 := R_1 / R_2;$

This is exactly the same as would be produced by the algorithm given by Wegstein [71]. The symbol and number cellars are essentially the list of operations and the set of locations for partial results used by Wegstein.

By splitting the number cellar into two cellars (a number cellar and an address cellar) Bauer and Samelson show how to avoid operations which simply load a variable into a partial result location. The three-address instructions which would be produced by this modified system would be similar to those produced by the system described by Kanner [44].

A paper by Huskey [35] describes in detail a somewhat similar system of one-pass translation. Successive symbol pairs are used to control the choice of 'generators', which either produce object program operations (in this case in a conventional one-address format) or perform housekeeping tasks necessary for the process of translation. A push-down store (symbol list) is used for the storage of operators until they can be converted into one-address instructions.

An important modification of the Bauer and Samelson method of using a push-down store has been described by Dijkstra [16]. The use of a transition matrix is avoided, and instead a 'discrimination vector' is used. Each symbol is given a priority; the discrimination vector is a table of symbols and their corresponding priorities. The priority of the symbol just read in is compared against the priority of the symbol at the top of the push-down store to control the generation of instructions. This method, and a development of it, using a double priority system, which is used in the Whetstone Compiler, is described fully in section 3.2.1.

An interesting notation for describing the use of a push-down store has been given by Floyd [23]. The notation, which is somewhat reminiscent of that used by Sheridan [65] for describing the method of inserting extra parentheses in an arithmetic expression, is demonstrated on two systems for translating arithmetic expressions. The first system is essentially similar to that of Bauer and Samelson, and the second is the bi-directional scan technique given in an earlier paper by the same author [22].

1.3.5 Reverse Polish

A push-down store can be used for the evaluation of an arithmetic ex-

pression as well as for translating it into its constituent steps. (In fact the system described by Milnes [52] could be said to use one push-down store for simultaneous translation and evaluation.)

This can be demonstrated by expanding the sequences of three-address instructions, produced by the simple left-to-right scan translation techniques, so that all operations work on partial results.

Thus

$$a + (b \times c - d)$$

would be translated into

$$\begin{aligned} R_1 &:= a; \\ R_2 &:= b; \\ R_3 &:= c; \\ R_2 &:= R_2 \times R_3; \\ R_3 &:= d; \\ R_2 &:= R_2 - R_3; \\ R_1 &:= R_1 + R_2; \end{aligned}$$

These instructions fall into two types, namely those which load a partial result location, or which perform an operation on the contents of two consecutive partial result locations. The above set of instructions could be abbreviated by deleting all references to partial result locations and instead writing it as the sequence

$$a, b, c, \times, d, -, +$$

This sequence can be used as instructions to specify the way in which a store with last-in-first-out properties is to be used at run time. A symbol which is an operand causes the sequence

$$R_p := \text{'operand'}; \; p := p + 1;$$

to be obeyed, whereas an operator is equivalent to the operations

$$p := p - 1; \; R_{p-1} := R_{p-1} \text{ 'operator' } R_p;$$

This has been described in a paper by Dijkstra [15] who uses the term 'stack' for the set of partial result locations, and 'stack pointer', for the index p. The paper goes on to develop the system of using a stack for the whole of the storage of variables in an ALGOL program, and demonstrates how this facilitates the implementation of recursive procedures.

The sequence of symbols used as instructions controlling the stack are in fact obtained by rewriting the original expression in 'Reverse Polish' notation. This parenthesis-free notation is a variant of that introduced by the Polish logician J. Łukasiewicz, and has the property that any operation is preceded by its operands.

Thus

$$a + b$$

is written as

$$a, b, +$$

and

$$(a + b) \times c$$

as

$$a, b, +, c, \times$$

A recent paper by Hamblin [30] describes the various forms of Polish notation, and simple techniques for translating between these notations and normal algebraic notation. In a much earlier paper [29] the same author described an interpretive program, for the English Electric DEUCE computer, which allowed users to program in Reverse Polish notation. This interpretive program was written as part of an investigation into the possibility of designing a computer with a Reverse Polish form of instruction code. In fact several recent computers have been of this form; see, for example, papers by Allmark and Lucking [1] describing the English Electric KDF9, by Barton [8] describing the Burroughs B5000, and by Takahashi, *et al.* [68] describing the E.T.L. Mk. 6 Computer.

Several extensions to the normal technique of one-pass translation of an arithmetic expression into Reverse Polish have been described in a paper [60] by the authors. Two markers are introduced, one to indicate type of operands, the other to indicate whether an operand is a constant or a variable. When an operator is placed in the stack, copies of the current values of the two markers are stored with it. During a single left-to-right scan of the arithmetic expression, it is translated into a form of Reverse Polish in which the types of all the operands, and the instructions to convert operands from real to integer, are given explicitly. At the same time the constant marker is used to evaluate all operators whose operands are constants. This is done by using the store in which the Reverse Polish is being generated as a second push-down store.

The techniques used in the Kidsgrove multi-pass optimizing ALGOL 60 compiler, for producing efficient coding of arithmetic expressions, are described in a paper by Hawkins and Huxtable [31]. The expression is reordered into a Reverse Polish form, which is then scanned repeatedly to remove common sub-expressions, and to minimize the number of fetch operations. This Reverse Polish then forms the basis of the generated machine code object program, in which the computation is reorganized where possible to take advantage of the nesting store accumulator of the KDF9, and to

minimize the use of the various manipulative instructions such as PERM, REV, DUP, etc. (see Appendix 3).

1.3.6 Summary

It will be seen that the original methods of translating arithmetic expressions were quickly replaced by simpler techniques. These techniques were then extended in order to produce more efficient object programs. The next stage was the development of integrated systems, and a coming into prominence of Reverse Polish notation. This last trend has had its influence on machine design.

Footnotes added in proof

1. It has been brought to our notice that in this description of early techniques for the translation of arithmetic expressions we have omitted to mention

Böhm, C. (1952.) "Calculatrices Digitales. Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme". (Dissertation, Zurich 1952). *Ann. Mat. pura appl.* (1954), **4**, 37, pp. 5–47.

This paper, considered by many to have merited more attention than it originally received, describes a single scan translation technique, using a rudimentary 'transition matrix' (see section 1.3.4).

2. We are indebted to Professor F. L. Bauer for the following details of further papers dealing with the 'cellar' technique.

Samelson, K. (1955). "Probleme der Programmierungstechnik. Aktuelle Probleme der Rechentechnik". Ber. Int. Mathematiker-Kolloquium, Dresden, 22–27 Nov., 1955. VEB Deutscher Verlag der Wissenschaften, Berlin (1957), pp. 61–68.

Bauer, F. L. and Samelson, K. (1960). "Verfahren zur Automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens". *Deutsche Patentauslegeschrift*, 1 094 019.

Bauer, F. L. and Samelson, K. (1962). Maschinelle Verarbeitung von Programmsprachen. "Digitale Informationswandler" (ed. by W. Hoffmann), pp. 227–268. Vieweg, Brunswick.

1.4 THE WHETSTONE COMPILER

1.4.1 ALGOL on KDF9

The Whetstone Compiler is one of the two ALGOL 60 Compilers written for the English Electric KDF9 Computer, as part of the 'KDF9 ALGOL System'; see Duncan [19]. During the planning of the software for the KDF9 it was realized that it would be undesirable to attempt to satisfy the diverse requirements of all users of an algebraic programming scheme with a single compiler. This is because a compiler, like any other program, takes time and space to perform its function. During development and testing of an ALGOL program, it is undesirable to use much time in compiling, since the compiled program will have a very short-lived existence. On the other hand a tested program which is used extensively should make economic use of the computer, and it is worth spending more time and effort on the compilation of such a program in order to produce an efficient object program. This has prompted the development of two distinct, but compatible, ALGOL 60 compilers, one a fast single-pass compiler, written at the Atomic Power Division of the English Electric Company, Whetstone, Leicester, England, and the other a multi-pass optimizing compiler, written at the Data Processing and Control Systems Division, Kidsgrove, Stoke-on-Trent, England. For the convenience of operators and users these two compilers are embedded in a single operating system, the 'KDF9 ALGOL System', which also includes facilities for amending ALGOL programs, and for extracting routines from an 'ALGOL library', stored on magnetic tape.

The Whetstone Compiler produces an object program in an intermediate language, which is interpreted at run time. During translation, which takes place as the ALGOL program is being read in, an exhaustive series of checks is performed, in order to ensure the validity of the ALGOL text. At run time an extensive range of program testing facilities is available, enabling a programmer to test his program in terms of the original ALGOL text. After an ALGOL program has been tested, it is possible to recompile it, using the Kidsgrove Optimizing Compiler, in order to produce an efficient machine code program.

Since the two compilers are designed to be used interchangeably within the KDF9 ALGOL System, it is essential that they should be completely compatible as regards the specification of the ALGOL source language which they will accept. As a result the logical sum of the restrictions on ALGOL inherent in either translation system forms the complete set of restrictions used to define the subset of ALGOL 60 which is called KDF9 ALGOL. This compatibility extends even to code procedures—both compilers will accept procedure declarations whose bodies are written in KDF9 User Code. (User Code, the standard assembly language for KDF9, is briefly described in Appendix 3.)

Since programs containing code procedures may be compiled by either compiler, the rules for writing code procedures are designed to ensure that no use is made of the internal properties of a compiler, or of the object program it generates, other than through the parameter list.

The main restrictions which have been placed on ALGOL 60, as defined by the Revised ALGOL Report, are:

(*i*) all formal parameters must have specifications;
(*ii*) labels must be identifiers, and not unsigned integers; and
(*iii*) dynamic own arrays are not allowed.

A full list of the restrictions is given in Appendix 2.

1.4.2 The Design of the Whetstone Compiler

The translation techniques used in the Whetstone Compiler are a development of those used in the ALGOL Translator written for the X1 Computer at the Mathematical Centre, Amsterdam, by E. W. Dijkstra and J. A. Zonneveld. A brief description of the Whetstone Compiler has been given in an earlier book in this series [58]. Because of the need for a fast translation system the Whetstone Compiler is designed to translate an ALGOL program during a single scan through the text. In particular, when a program is presented directly on paper tape, instead of magnetic tape, translation takes place as the tape is being read, and the generated program is complete in the core storage, ready to be obeyed, virtually as soon as the last item of ALGOL has been read.

The existence of the Kidsgrove Optimizing Compiler has allowed the authors of the Whetstone Compiler a considerable amount of freedom in the design of the object program produced by their compiler. Thus the object program is designed in order to facilitate the task of translation and to provide easy means of communication between program and programmer, in terms of the original ALGOL text. Rather than expand the compactly coded object program into a sequence of machine code instructions, it is obeyed interpretively using a Control Routine. In fact the object program language can be thought of as approximating to the authors' ideas for an order code of an ideal computer (ideal that is for running ALGOL programs!).

Because the object program has been designed in this way, and because the KDF9 is a new computer with, as yet, no 'traditional programming conventions', there is little in the fundamental design of the Whetstone Compiler which is in any way machine-dependent. In fact the Whetstone Compiler has been implemented to date on three other computers, of widely differing characteristics, namely a Ferranti PEGASUS, ACE at the National Physical Laboratory, and an English Electric DEUCE.

The Whetstone Compiler is divided into two sections, called the 'Translator', and the 'Control Routine'. The Translator uses the stack and priority level techniques which have been briefly described in section 1.2.4.1.2. The resultant object program, which consists of a sequence of 'object program

operations' together with their parameters, is obeyed interpretively by the Control Routine, using a stack for the storage of simple variables and arrays. Sections 2 and 3 of this book consist of a detailed description of the design and use of the Object Program, and of the Translator respectively.

Since the Whetstone Compiler is intended mainly for use during the development of ALGOL programs, considerable attention has been paid to providing facilities for checking and testing programs. A further facility available with both the Whetstone and the Kidsgrove Compilers is that of program segmentation.

1.4.2.1 Program Checking

During translation an exhaustive series of checks is made on the ALGOL text, and any errors found cause the printing of an 'error message' which gives details of the type of error and the position in the text where the error has become evident. Details of the various checks are given in the description of the Translator in section 3, and in the Translator flow diagrams in Appendix 11. The method of translation makes it difficult to check out certain errors during translation, such as compatibility of actual and formal parameters of a procedure. In such cases the checks are performed at run time by the Control Routine.

The basis of the system for checking errors is the assumption that the statement bracket structure of a program is correct. This assumption cannot be verified until the end of the text is reached, by finding an equal number of **begin** and **end** symbols (ignoring those occurring within comments and strings). Thus if the final error message produced in fact states that the statement bracket structure is incorrect, any previous error messages should be regarded with suspicion. However, if the bracket structure is correct it is possible to produce a list of errors which, though not exhaustive, will not contain any spurious errors (i.e. apparent inconsistencies arising from earlier errors). This is done by treating each block separately. When an error is found in a block, the rest of the block is skipped. Any internal blocks are checked, but it is assumed that valid declarations for any non-local variables have been missed during the skipping of the block which contained an error.

Example

```
begin real a;
      a = 3;
      begin real b;
            b := 2 - c
      end;
      a := 2
end
```

The error in the outer block ('$a = 3$' instead of '$a := 3$') causes the rest of that block to be skipped. The inner block is checked but the use of an undeclared variable c will not be found.

Since the outer blocks are not affected by the errors in an inner block, it is possible to resume normal checking after the end of a block containing an error has been reached. This system has the advantage that a fair proportion of the errors in a program can be found during a single scan of the text, without any complicated system of analysing the probable cause of each error in an attempt to get the Translator 'back on the rails'. The implementation of this system is described in section 3.4.7.

The position at which an error has been found is indicated by a message containing:

(*i*) line number (counting only lines which contain printed characters);

(*ii*) position identifier (the last declaration of a procedure identifier or a label prior to the error);

(*iii*) relative line number (number of lines counted from the position identifier);

(*iv*) last delimiter; and

(*v*) last identifier or constant.

As a further aid to the checking of ALGOL programs various warning messages can be produced during the course of translation. These messages are designed to draw attention to parts of the ALGOL text which, though valid, are probably in error.

One of the most frequent causes of errors in statement bracket structure is the omission of a semi-colon after an **end**, which turns the text following the **end** into a comment.

Example

```
x := 3;
begin real a;
      a := 2
end
x := 1;
```

This is a perfectly valid, though somewhat unlikely, piece of ALGOL text, in which '*x* := *1*' is a comment.

For this reason a warning message is printed whenever a comment after the symbol **end** contains a delimiter. Thus the normal uses of the **end** comment convention, giving an identification of the block or procedure, or the controlled variable of the for statement it finishes, will not give rise to any warning messages.

Two other warning messages are concerned with the declaration of identifiers. If an identifier (other than a label) is declared but not used, or if a formal parameter of a procedure is rendered inaccessible by a declaration at the head of the procedure body, then warning messages are printed.

Example

```
begin real procedure F (a); real a;
      begin real a;
            F := 2
      end;
      . . .
```

Here the parameter *a* is inaccessible in the procedure *F*, and the local variable *a* is declared but not used, and hence both types of warning message will be printed.

Since the Whetstone Compiler uses only the first eight characters of an identifier a message is printed when an identifier of nine or more characters is encountered. This message gives the full sequence of characters forming the identifier, as an aid to avoiding mistakes due to the use by the Compiler of an abbreviated form of such identifiers.

1.4.2.2 Program Testing

The essential feature of the program testing facilities is that their use requires no knowledge of the object program produced by the Translator, or of KDF9 machine code.

The testing facilities fall into two categories—those which can be used during the running of a program, and those which can be used when a program has reached a failure. The various program testing facilities are controlled by means of directives given to the Compiler at the head of the ALGOL program, and by ALGOL statements within the program.

Brief details of the implementation of the various program testing facilities are given in Appendix 5.

1.4.2.2.1 Facilities Used During the Running of a Program. The testing of an ALGOL program is accomplished by examining the results it produces, including any information produced as a result of a failure. Thus the normal program testing strategy is to organize the printing out of extra information, such as partial results, to indicate the course taken by the program, and to help validate the results obtained from the program. A very powerful system for controlling the output of test information can be built up using normal ALGOL procedures.

Example

```
procedure TEST SHOT (a,b); value b; string a; real b;
begin
        print text (a);
        print text (' ␣ = ␣ ');
        print (b)
end;
```

If this procedure were called by the statement

TEST SHOT ('*alpha*', *alpha*);

where *alpha* has the value *1.2345*, the result would be to print

alpha = *1.2345*

In this, and succeeding examples, some suitable declarations for simple output procedures are assumed.

The power of such a simple procedure can be greatly increased if its action can be controlled during the running of a program.

Example

```
procedure TEST SHOT (a,b,c); value b,c; string a;
real b; integer c;
if TEST CONTROL ≥ c then begin print text (a);
                               print text (' ␣ = ␣ ');
                               print (b)
                         end;
```

The sequence of statements

```
read (TEST CONTROL);
TEST SHOT ('a',a,1);
TEST SHOT ('b',b,2);
TEST SHOT ('c',c,1);
```

would produce no output, the values of *a* and *c*, or the values of *a*, *b* and *c*, depending on whether *TEST CONTROL* was read in as zero, one or two, respectively.

Two program testing facilities are provided in the Whetstone Compiler to supplement the facilities that can be provided by ALGOL procedures such as those given above. The first of these is the automatic deletion, on request, of all procedure declarations and procedure statements concerning procedure identifiers starting with the letters *TEST*.

The second facility, a form of trace, can be harnessed to a system of levels of testing (such as that demonstrated in the above example). The trace, when operative, causes the printing of each position identifier and of the value of each assignment statement that is passed. A 'position identifier' is defined as an identifier used as a label attached to a statement, or as a procedure identifier at the declaration of the procedure.

Example

```
...
integer procedure CALC; CALC := 4;
 L: n  := 3;
    m := 2 × CALC;
M: m := 0;
...
```

would produce a trace consisting of

L, 3, CALC, 4, 8, M, 0

The trace facility provides a means of performing a very detailed check on the action of an ALGOL program. However, it would not be practicable to trace a complete program, and therefore a means has been devised for switching the trace facility on and off by means of ALGOL statements. This is done by replacing the variable *TEST CONTROL* used in earlier examples by an integer procedure *TEST*. Although this procedure has a body written in code, its declaration is equivalent to

```
integer procedure TEST (E); value E; integer E;
begin own integer i;
      if E < 0 then TEST := i else i := E
end;
```

Thus the statement

TEST (3);

will assign the value *3* to *i*, whilst *TEST* can be used by a function designator with a negative parameter, in order to obtain the current value of *i*.

However, since the body is in fact given in code (written by the authors of the Compiler) the integer *i* does not have to be stored in the stack in the normal way. Instead a special store is used, and the action of the Control Routine at assignment statements and at position identifiers is dependent on the value given in this special store, so that procedure statements of the form *TEST (E)* can be used to switch the trace on and off.

Thus a very powerful system for obtaining partial results and trace information can be built up using ALGOL procedures whose action is dependent on the integer *i*, the value of which is controlled by a set of procedure statements of the form *TEST (E)*. When required these procedure declarations and statements can be deleted automatically.

The trace facility requires that the characters forming the various position identifiers be held in the computer during the running of the object program. Since this is necessary only when the trace facility is to be used the position identifiers are stored only when a directive is given to the Compiler before reading in the ALGOL program.

1.4.2.2.2 Facilities Used at the Failure of a Program. No matter how much care is taken in the incorporation of testing facilities of the kind described above in a program, all too often a failure occurs in a section of the program within which few, if any, partial results have been requested. For this reason, several further program testing facilities in the Whetstone Compiler are concerned with the output of information at a failure, for use in attempting to analyse the cause of the failure.

The normal failure information consists of a message detailing the type of

failure (e.g. integer overflow) and the position in the ALGOL text at which the failure occurred, given as an absolute line number, the last position identifier, and the line number relative to this position identifier.

Often such failure information is of little value in itself; for instance a failure might occur within a procedure, and what is really needed is information concerning the particular call of the procedure which lead to the failure. This sort of information is given by what is here called a 'retroactive trace' which is provided automatically at a failure if the trace facility has been requested (whether or not it is actually switched on at the time of failure).This retroactive trace consists of the last sixteen position identifiers passed, during the running of a program, prior to the failure. By this means the route taken to the failure (i.e. the sequence of labelled statements obeyed, and procedures entered) is known.

The final program testing facility is known as the 'post-mortem'. This consists of a print-out of the identifiers and current values of the simple variables in the current block, and in the surrounding blocks. The number of block levels which are 'post-mortemed' in this way is governed by a directive given to the Compiler before the ALGOL text is read in.

1.4.2.3 The Segmentation of Programs

In both the Whetstone and the Kidsgrove Compilers it is possible to divide an ALGOL program into segments and to translate these segments separately. During the running of the program, the translated segments are kept on magnetic tape and occupy space in the core store only during the activation of the segment.

One of the main complaints against ALGOL has been that an ALGOL program is not suitable for segmentation. In fact the structure of ALGOL is such that a 'complete procedure' (i.e. a procedure with declarations or specifications for all identifiers used within its body) lends itself very well to such treatment. Some details of the techniques used in the Whetstone Compiler to translate and run segmented programs are given in Appendix 6.

1.4.2.4 Input/Output

The input/output facilities in KDF9 ALGOL are based on the use of code procedures. Thus the input/output facilities are completely independent of the compilers. A standard set of procedures is provided, but any user can modify, replace, or extend this standard set to suit his own requirements.

A general discussion of the philosophy behind the method used for input/output in KDF9 ALGOL, and brief details of the standard set of procedures, are given in a paper by Duncan [20].

1.4.3 The Description of the Whetstone Compiler

It was stated earlier that the basic design of the Whetstone Compiler is, to a large degree, independent of the computer for which it was developed

(the English Electric KDF9). In detail, of course, many features of the KDF9 have influenced the Compiler. However, in the description of the Compiler an attempt has been made to prevent such details from obscuring the main issues. In many cases, whilst describing various internal features of the Compiler, the authors have attempted to describe alternative techniques, and to indicate the reasons (sometimes rather arbitrary) behind the choice of the particular technique used.

The environment within which the Whetsone Compiler is used has naturally had a large effect on the design of the Compiler. For instance, the fact that it is meant for use mainly during program development has prompted the authors to use a strictly one-pass translation system, together with a Control Routine which obeys the object program interpretively at run time. However, neither of these features can be regarded as really fundamental to the basic logical design of the Compiler.

The one-pass techniques used in the Compiler are made feasible by the fact that there is sufficient high speed storage on the KDF9 to allow the object program to be built up within the computer. If this were not the case it would be perfectly possible, say, to use a preliminary pass to collect details of the identifiers used in a program, and a final pass after translation to insert addresses in forward jump instructions. If this were done the object program could be output as it was generated, and the third pass, dealing with forward jump instructions, could operate as the object program was being read back in.

Similarly, the division of responsibility between the Translator and the Control Routine could easily be altered. In the Whetstone Compiler the object program is obeyed interpretively and the addresses and types of intermediate results are determined at run time. All these features help to simplify the Translator but are not essential to the basic techniques described in section 2, on the Object Program.

However, sections 2 and 3 give a detailed description of what has actually been implemented on KDF9. It has not been possible to keep the description completely machine-independent, mainly because it is necessary to describe the way in which various items of information are stored inside the machine. It would have been rather unrealistic to ignore the fact that in many cases several items have been packed into one word (48 binary digits on KDF9); the way in which this is done on a given computer depends on the number of digits necessary for a store address, the size of words, the facilities for packing and unpacking information, etc.

It has been very difficult to decide how to present the detailed logic of the Compiler. The idea of describing—indeed even programming—a compiler in its own language has a certain appeal, but it is by no means certain that ALGOL, a language designed primarily for describing computational processes, is ideal for describing a compiler, particularly to a human reader. Naturally if an ALGOL compiler for some machine had been readily available it would have been used, but as this was not the case the programming was done in an assembly language. Before starting this detailed programming the authors drew up a set of logical flow diagrams originally meant only for

their own use. However, with the aid of sections 2 and 3 of this book, several copies of the compiler have now been made from these logical flow diagrams. In view of this the detailed logic of the compiler is given by means of these logical flow diagrams (Appendices 10 and 11) which are to be regarded, not so much as a rigid plan for programming, but as a basis to work from, having regard for the order code and structure of a given computer. The authors' only defence against criticisms of this decision to use the logical flow diagrams, and of the notation used in the flow diagrams, is that they have already been used successfully to implement ALGOL Compilers on several widely differing computers.

2 THE OBJECT PROGRAM

2 THE OBJECT PROGRAM

The object program, generated from the ALGOL by the Translator, consists of a set of operations, with parameters where necessary, which are obeyed interpretively by the Control Routine, using the remaining core storage as a stack. Each operation, which for purposes of description is given a mnemonic abbreviation, is represented by an 8-bit code. The inner loop of the Control Routine is used to fetch an operation code and then to select the routine appropriate to the operation. Thus the Control Routine consists mainly of a set of separate routines, one for each type of operation. However, there is also a small set of sub-routines which is common to several routines.

Operations and their parameters are packed into words by syllables of eight binary digits and are addressed using a 'program counter' working in units of one syllable. Progress through the object program is normally made by increasing the program counter, where necessary allowing for the syllables used by parameters. For reasons of efficiency, the program counter is split into two parts—the first indicating the word number, the second giving the position of the syllable within this word, but this fact will be disregarded in what follows.

2.1 ASSIGNMENT STATEMENTS

2.1.1 Arithmetic Expressions

It has been mentioned earlier that the object program corresponding to an arithmetic expression is essentially in Reverse Polish notation. The evaluation of such an expression is accomplished by using the stack as a simple push-down store.

Example

$$e + (a \times b + c) / b - f \times g$$

in Reverse Polish becomes

$$e,a,b,\times,c,+,b,/,+,f,g,\times,-$$

This could easily be expanded into a set of commands, using explicit stack addresses, as follows

$S\ [0] := e;$
$S\ [1] := a;$
$S\ [2] := b;$
$S\ [1] := S\ [1] \times S\ [2];$
$S\ [2] := c;$
$S\ [1] := S\ [1] + S\ [2];$
$S\ [2] := b;$
$S\ [1] := S\ [1] / S\ [2];$
$S\ [0] := S\ [0] + S\ [1];$
$S\ [1] := f;$
$S\ [2] := g;$
$S\ [1] := S\ [1] \times S[2];$
$S\ [0] := S\ [0] - S\ [1];$

Here the vector S has been used as a stack, and the actual identifiers are used to indicate the addresses of the variables.

However, it is obviously not necessary to give actual addresses to all these operations, since all can be made to work under control of a single stack pointer p, which is always set to indicate the first free store after the end of the stack. Thus one would treat the Reverse Polish as a set of commands, modifying the stack pointer appropriately.

Each identifier causes the value of the simple variable to be placed in the stack at the position given by the stack pointer, which is then increased by one. Each operation works on the contents of the top two stack positions

(i.e. $S[p-2]$ and $S[p-1]$) placing its result in $S[p-2]$, and then subtracting one from the stack pointer.

A slight complication arises from the possibility of using '+' and '−' as unary operators. Unary '+' can be ignored, and unary '−' can be changed into the operation *NEG*, which negates the value at the top of the stack, ($S[p-1]$), and does not alter p.

Example

$$-b \times (+a - b) \uparrow (-c)$$

becomes

$$b,\ NEG,\ a,b,-,c,NEG,\uparrow,\times$$

The operation of this is then equivalent to the explicit commands

$S[0] := b;$
$S[0] := -S[0];$
$S[1] := a;$
$S[2] := b;$
$S[1] := S[1] - S[2];$
$S[2] := c;$
$S[2] := -S[2];$
$S[1] := S[1] \uparrow S[2];$
$S[0] := S[0] \times S[1];$

So far no consideration has been given to the question of real and integer working. The KDF9 computer has full facilities for arithmetic in both floating point and fixed point modes and these, naturally, are made to correspond with the ALGOL types **real** and **integer**. The Revised ALGOL Report gives explicit rules in section 3.3.4 concerning operators and types and the object program must be made to reflect this.

The method used is to expand the above system of a 'Reverse Polish' notation for the object program, by the inclusion of type information in the operations which fetch the various operands into the stack. This information is obtained by the Translator from the relevant type declarations. When an operand is put into the stack it is accompanied by a second word, containing an identifying bit pattern. Thus the arithmetic operators now work on one or two word pairs, each word pair containing an operand and the information necessary for decisions about real and integer working. Such word pairs are called 'accumulators', and are addressed by means of a special stack pointer which is called *AP*, the accumulator pointer, to avoid confusion with various other stack pointers which will soon be introduced. *AP* works on exactly the same principle as the stack pointer used earlier, and thus is always set to indicate the first free accumulator after the end of the stack. However, *AP* changes by one for each word, and hence by two for each accumulator.

It is now worth writing out a portion of object program in the form which

was described earlier, i.e. as a set of operations, with parameters where necessary. However, until the method of storage of variables has been explained the original identifiers will be used as parameters.

Example

The expression

$$i - (x + j) \times y$$

where x and y are of type **real**, i and j of type **integer**, which in Reverse Polish is

$$i,x,j,+,y,\times,-$$

is represented in the object program by

TIR i, TRR x, TIR j, +, TRR y, ×, −

Here two new operations *TIR* (Take Integer Result) and *TRR* (Take Real Result) are introduced. These 'Take Result' operations each have one parameter which is the address of a simple variable. The operation occupies 8 bits (one syllable) and the parameter occupies two syllables. For convenience the object program is written out in tabular form, with a syllable count appended.

Syll	*Op*	*Par*
0	*TIR*	*i*
3	*TRR*	*x*
6	*TIR*	*j*
9	+	
10	*TRR*	*y*
13	×	
14	−	
15	...	

Examining the action of these operations in some detail, assuming that *AP* has an initial value of, say, 10

TIR	*i*	Sets value of *i* in *S*[*AP*], i.e. *S*[*10*], and a bit pattern meaning 'integer result' in *S*[*AP*+*1*], i.e. *S*[*11*]. Two added to *AP*, which thus becomes *12*.
TRR	*x*	*S*[*12*] := *x*; *S*[*13*] := 'real result'; *AP* := *AP*+*2*;
TIR	*j*	*S*[*14*] := *j*; *S*[*15*] := 'integer result'; *AP* := *AP*+*2*;
+		Examines *S*[*AP*−*3*] and *S*[*AP*−*1*], i.e. *S*[*13*] and *S*[*15*], converts '*j*' in *S*[*14*] to real, sets the real result '*x*+*j*' in *S*[*12*], decreases *AP* by two.

TRR	*y*	$S[14] := y$; $S[15]$:='real result'; $AP := AP+2$;
×		No conversions necessary. $S[12] := S[12] \times S[14]$; $AP := AP-2$;
−		Convert *i* in $S[10]$ to real. $S[10] := S[10]-S[12]$; $S[11]$:='real result'; $AP := AP-2$;

The system is extended to deal with simple assignment statements by means of 'Take Address', and 'Store' operations.

Example

$x := y := i + j;$ (where x and y are real variables, i and j are integer variables)

is translated into

Syll	*Op*	*Par*
0	*TRA*	*x*
3	*TRA*	*y*
6	*TIR*	*i*
9	*TIR*	*j*
12	+	
13	*STA*	
14	*ST*	
15	...	

TRA (Take Real Address), stacks the address of an operand, and a bit pattern meaning 'real address'. *ST* (Store), working on the top two accumulators, (an address and a result), does any necessary conversion between real and integer, stores the result at the given address, and decreases *AP* by four. *STA* (Store Also), used only in multi-assignment statements, performs conversion and storage as in the operation *ST*, but then replaces the contents of the 'address accumulator' by the contents of the 'result accumulator' and only decreases *AP* by two. Thus *STA* performs storage, but leaves the result in the top accumulator, for use by succeeding *STA* or *ST* operations.

Thus the action of an assignment statement is to stack one or more addresses, before calculating the value of the expression, which eventually is placed in the top accumulator, immediately above the addresses. Any *STA* operations successively move this result down one place, until a final *ST* operation deletes the result and the final address.

It should thus be noted that after completion of an assignment statement the accumulator pointer returns to the value it had originally. Furthermore, the correct working of an assignment statement takes place independently of the starting value of *AP*. The importance of this fact will become obvious when the method of storage allocation of variables is described.

2.1.1.1 Exponentiation

The implementation of the exponentiation operator '↑' is in accordance with the definition given in section 3.3.4.3 of the Revised ALGOL Report.

This can be done because all anonymous intermediate results carry an indication of their type. Thus the case of forming '$a \uparrow i$', where i is of type **integer**, can produce a result either of the same type as a, or of type **real**, depending on the value of i.

In the case of an integer exponent the rule for giving the result of the exponentiation operation involves the repeated multiplication

$$a \times a \times a \times \ldots \times a \ (i \text{ times})$$

This can be done efficiently by repeated squaring, and is easily implemented on a binary machine such as KDF9. The result is formed as the continued product of various factors formed by repeated squaring of the base. The digits of the binary representation of the exponent are used to decide which factors are in fact necessary.

Thus

$$a^5 = a^4 \times a^1 = (a^2)^2 \times a^1$$
$$a^{13} = a^8 \times a^4 \times a^1 = ((a^2)^2)^2 \times (a^2)^2 \times a^1,$$
etc.

The calculation is performed by means of a small loop of instructions which shift down the exponent and repeatedly square the base. Each time round the loop that the least significant digit of the exponent is non-zero the partial result is multiplied by the current value obtained from the repeated squaring of the base.

2.1.2 Constants

Any constant appearing in an arithmetic expression is translated into a 'Take Constant' operation with, in general, the value of the constant as a 6-syllable parameter. The effect of such an operation is to stack its parameter, together with the appropriate bit pattern specifying type.

In a somewhat arbitrary fashion a special case has been made for two commonly used integer constants, namely 'zero' and 'one'. These each have a special parameterless operation (*TIC0* — Take Integer Constant Zero, and *TIC1* — Take Integer Constant One, respectively).

Example

$$(a + 2) \times 3.0 - 1/N$$

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*TRR*	*a*	
3	*TIC*	*'2'*	(Take Integer Constant)
10	+		
11	*TRC*	*'3.0'*	(Take Real Constant)
18	×		
19	*TIC1*		
20	*TIR*	*N*	
23	/		
24	−		
25	...		

Note that in this representation of the object program, quotes are used to show that a parameter is a constant. This also serves as a reminder that the parameter takes up 6 syllables.

An alternative, and perhaps better method of dealing with constants would be for the Translator to set up a list of all the different constants used in a program, and then to generate operations whose parameters refer to this list, rather than include each constant in the program each time it occurs. Such a system would be slightly more complicated for the Translator as it would involve scanning a list each time a constant was used.

Methods of avoiding run time real-integer conversions on constants, and of performing any operations which depend solely on constants during translation are discussed briefly in section 1.3.

2.1.3 Subscripted Variables

The method of representation of subscripted variables in the object program is quite straightforward. Without going into details regarding the method of storage of arrays, it is sufficient to assume the existence, for each array, of an 'array word', which is used to address the 'storage mapping function' of the array. A storage mapping function is basically a set of coefficients calculated from the subscript bounds given in the relevant array declaration.

These coefficients can be used, together with the values of the actual subscripts, by an 'indexing routine', to find the element of the array to which the subscripted variable refers.

Thus, to obtain the address of an array element, the object program must assemble, as data for the indexing routine, the address of the array word, and the value of each subscript expression. This is done by using a 'Take Address' operation to stack the address, not of a variable this time, but of the array word, and the type of the elements of the array, followed by the normal object program representation of each subscript expression. The operation *INDA* (Index Address) follows, and this uses the accumulators that have been set up by the previous operations.

Example

$A[i, j + 2, i - j \times k]$

is translated into

Syll	*Op*	*Par*
0	*TRA*	*A*
3	*TIR*	*i*
6	*TIR*	*j*
9	*TIC*	*'2'*
16	+	
17	*TIR*	*i*
20	*TIR*	*j*
23	*TIR*	*k*
26	×	
27	−	
28	*INDA*	
29	...	

In this, and following examples, suitable declarations are assumed for the various identifiers.

Assuming that *AP* has the value (say) *7* before this set of object program operations is obeyed, then when *INDA* is reached the following accumulators will have been set up:

S[*13*] and *S*[*14*]	(Top Accumulator) Value of '$i - j \times k$'
S[*11*] and *S*[*12*]	Value of '$j + 2$'
S[*9*] and *S*[*10*]	Value of *i*
S[*7*] and *S*[*8*]	Address of array word *A*

The operation *INDA* works down through the stack, starting with the top accumulator, examining the identifying bit pattern of each accumulator in turn. Any real numbers are rounded off and converted to type **integer**, as required by section 3.1.4.2 of the Revised ALGOL Report. This process terminates when an accumulator containing an address is reached. This address enables the array word and hence the storage mapping function to be obtained. The indexing routine is then used to process the storage mapping function and the stacked subscripts, to obtain the address of the array element. This address is placed in the accumulator which contained the address of the array word, and the accumulator pointer is decreased so as to make this the top accumulator. Thus the total effect of the set of operations is just the same as a 'Take Address' operation for a simple variable.

In an arithmetic expression the value, rather than the address, of an array element is required, and in such a case the operation *INDR* (Index Result) is used instead of *INDA*. This works in the same way as *INDA* but finishes by fetching the actual element.

This system of using *INDR* to replace the address of the array word and the values of the subscripts by the value of the subscripted variable is used to implement multiple subscripting.

Example

$A[A[2]] := A[3+A[1]];$

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*TRA*	*A*	
3	*TRA*	*A*	
6	*TIC*	*'2'*	
13	*INDR*		Finds value of $A[2]$.
14	*INDA*		Finds address of $A[A[2]]$.
15	*TRA*	*A*	
18	*TIC*	*'3'*	
25	*TRA*	*A*	
28	*TIC1*		
29	*INDR*		Finds value of $A[1]$.
30	+		
31	*INDR*		Finds value of $A[3+A[1]]$.
32	*ST*		
33	...		

This system of taking the addresses of the variables (simple or subscripted) in the left part list of an assignment statement before evaluating the right hand side allows multi-assignments to subscripted variables to be carried out in accordance with the rules given in section 4.2.3 of the Revised ALGOL Report.

Example

$n[i] := i := 6;$

which is translated into

Syll	*Op*	*Par*
0	*TIA*	*n*
3	*TIR*	*i*
6	*INDA*	
7	*TIA*	*i*
10	*TIC*	*'6'*
17	*STA*	
18	*ST*	
19	...	

If i is initially (say) *10*, the value *6* is assigned to i, and to $n[10]$, rather than to $n[6]$.

This treatment of subscripted variables is very inefficient for simple uses of subscripted variables, although sufficiently general to deal with all the possibilities inherent in the recursive definition of subscript expressions given in the Revised ALGOL Report. Optimization of subscripted variables, and indeed of arithmetic expressions in general, will normally involve some changing of the order of operations. This can only be done in ALGOL when it has been proved that none of the operands involved is a function designator whose evaluation induces side-effects. A discussion of function designators must be deferred until the object program representation of procedures has been described.

However, bearing in mind this limitation, optimized treatment of subscripted variables, particularly in for statements is certainly possible. Papers dealing with this subject include those of Hawkins and Huxtable [31], Samelson and Bauer [62], Hill, Langmaack, Schwarz and Seegmüller [34].

2.1.4 Simple Boolean Expressions

Simple variables of type **Boolean** are represented in core storage by a word consisting of binary zeroes for the value **false** or binary ones for the value **true**. The object program representation for simple Boolean expressions is again essentially derived from Reverse Polish notation, and uses the operation *TBR* (Take Boolean Result) to stack an accumulator with the value of a simple Boolean variable, together with an identifying bit pattern. The Boolean operators $\wedge$, $\vee$, $\supset$, $\equiv$, work on the top two accumulators, replacing them by the resulting Boolean value. The unary operation $\neg$ merely changes the top accumulator from **true** to **false**, or vice-versa.

Example

$$\neg\, b \wedge (c \equiv d) \vee e$$

is translated into

Syll	*Op*	*Par*
0	*TBR*	*b*
3	$\neg$	
4	*TBR*	*c*
7	*TBR*	*d*
10	$\equiv$	
11	$\wedge$	
12	*TBR*	*e*
15	$\vee$	
16	...	

Relational operators work on the top two accumulators, which must be real or integer, do any necessary conversion to type **real**, and if the relation is satisfied, replace the accumulators by a Boolean accumulator with the value **true**, otherwise with the value **false**.

Example

$x + y > z \wedge z \leq 0$

is translated into

Syll	*Op*	*Par*
0	*TRR*	*x*
3	*TRR*	*y*
6	+	
7	*TRR*	*z*
10	>	
11	*TRR*	*z*
14	*TIC0*	
15	$\leq$	
16	$\wedge$	
17	...	

The re-ordering of the operations which is performed by the Translator ensures that the arithmetic expressions involved in relations are evaluated before the Boolean value of the relation is assessed.

The Boolean constants **true** and **false** are set up by the parameterless operations *TBCT* (Take Boolean Constant True) and *TBCF* (Take Boolean Constant False), which are analogous to the special operations *TIC0* and *TIC1* used to set up the integers 'zero' and 'one'.

Simple Boolean assignment statements use the operation *TBA* (Take Boolean Address) and the storage operations *ST* and *STA* which have been described earlier.

Example

$b := c := x > 0 \wedge$ **true**;

is translated into

Syll	*Op*	*Par*
0	*TBA*	*b*
3	*TBA*	*c*
6	*TRR*	*x*
9	*TIC0*	
10	>	
11	*TBCT*	
12	$\wedge$	
13	*STA*	
14	*ST*	
15	...	

Boolean arrays are dealt with in a similar way to real or integer arrays, but use the operation *TBA* to fetch the array word.

More efficient methods of dealing with Boolean expressions involve use of an object program representation which avoids performing unnecessary operations.

For example

$$a \wedge b \wedge c \wedge d$$

has the value **false** if any of the operands is **false.**

Optimization based on these principles is again somewhat restricted by the problems of function designators and side-effects but can, for involved Boolean expressions, prove most effective. Bottenbruch and Grau [11] give several 'optimized' representations of Boolean expressions, and papers by Huskey and Wattenburg [36], and Arden, Galler, and Graham [5], describe methods of compiling optimized Boolean expressions.

2.1.5 Conditional Expressions

The description of arithmetic and Boolean expressions can now be extended by a discussion of conditional expressions.

Essentially a conditional expression uses the truth values of Boolean expressions to choose between various simple expressions. The object program representation of simple expressions consists of sets of operations which are executed strictly in sequence. Two operations *IFJ* (If False Jump) and *UJ* (Unconditional Jump) are used to break the normal sequencing of operations in order to choose the set of operations corresponding to the required simple expression.

IFJ is used after the program generated from the Boolean expression in an if clause. It allows a jump to be performed to the operations corresponding to the expression following the **else** of the conditional expression, if the Boolean expression has the value **false.**

IFJ does this by inspecting, and deleting, the top accumulator, which must be Boolean. If the accumulator had the value **false** then the program counter is replaced by the parameter to the *IFJ* operation.

UJ is used after the unconditional expression between **then** and **else,** to avoid the expression following the **else.** This is done by a simple replacement of the program counter. The parameter to both *IFJ* and *UJ* is a *16*-bit counter.

Example

if $x > 0$ **then** $y + z$ **else** $y \times z$

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*TRR*	*x*	
3	*TIC0*		
4	>		
5	*IFJ*	(*18*)	$x > 0$?
8	*TRR*	*y*	
11	*TRR*	*z*	
14	+		
15	*UJ*	(*25*)	
18	*TRR*	*y*	
21	*TRR*	*z*	
24	×		
25	...		

This mechanism can of course be used inside subscripts, within the Boolean expression of an if clause, etc.

2.1.6 Conditional and Compound Statements

A technique similar to that used for conditional expressions, is used for conditional statements.

Example

if $x = 0$ **then** $x := 1$ **else** $x := 2$;

is translated into

Syll	*Op*	*Par*
0	*TIR*	*x*
3	*TIC0*	
4	=	
5	*IFJ*	(*16*)
8	*TIA*	*x*
11	*TIC1*	
12	*ST*	
13	*UJ*	(*27*)
16	*TIA*	*x*
19	*TIC*	'*2*'
26	*ST*	
27	...	

On the other hand an if statement can be used to decide whether a given statement is to be obeyed, rather than to choose between two statements.

Example

if $x = y$ **then** $x := x - 1$;

is translated into

Syll	*Op*	*Par*
0	*TIR*	*x*
3	*TIR*	*y*
6	=	
7	*IFJ*	(*19*)
10	*TIA*	*x*
13	*TIR*	*x*
16	*TIC1*	
17	−	
18	*ST*	
19	...	

The statement brackets **begin** and **end**, when used to group statements into compound statements, do not have any direct equivalent in the object program.

Example

if *b* **then begin** $x := 1$; $y := 2$ **end**;

is translated into

Syll	*Op*	*Par*
0	*TBR*	*b*
3	*IFJ*	(*22*)
6	*TIA*	*x*
9	*TIC1*	
10	*ST*	
11	*TIA*	*y*
14	*TIC*	'*2*'
21	*ST*	
22	...	

2.1.7 Summary

Expressions are represented in the object program by re-ordering the operators, to generate a Reverse Polish form of the expression. 'Take Result' operations are used to give information regarding the type of variables. The evaluation of expressions is performed using a stack. Indexing operations, working on the values of subscript expressions, are used to find the particular elements of arrays specified by subscripted variables. Conditional expressions are implemented using two implicit jump operations, *IFJ* (If False Jump), and *UJ* (Unconditional Jump).

Assignment statements are translated using 'Take Address' operations and the operations *ST* (Store) and *STA* (Store Also) to assign values to the variables given in the left part list.

Conditional and compound statements, for the moment demonstrated simply using assignment statements, are built up without needing to introduce any further operations.

Taking a final example of an assignment statement

A[**if** *b* **then** *1* **else** *2*] := *X* := (**if if** *b* **then** *X* > *0* **else** *X* ≥ *0* **then** *X* + *Y* **else** *X* − *Y*) ↑ *2*;

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*TRA*	*A*	
3	*TBR*	*b*	
6	*IFJ*	(*13*)	*b*?
9	*TIC1*		
10	*UJ*	(*20*)	
13	*TIC*	'*2*'	
20	*INDA*		Address of *A*[**if** *b* **then** *1* **else** *2*]
21	*TRA*	*X*	
24	*TBR*	*b*	
27	*IFJ*	(*38*)	*b*?
30	*TRR*	*X*	
33	*TIC0*		
34	>		
35	*UJ*	(*43*)	
38	*TRR*	*X*	
41	*TIC0*		
42	≥		
43	*IFJ*	(*56*)	*X* > *0* ? or *X* ≥ *0* ?
46	*TRR*	*X*	
49	*TRR*	*Y*	
52	+		*X* + *Y*
53	*UJ*	(*63*)	
56	*TRR*	*X*	
59	*TRR*	*Y*	
62	−		*X* − *Y*
63	*TIC*	'*2*'	
70	↑		
71	*STA*		
72	*ST*		
73	...		

2.2 BLOCKS AND PROCEDURES

2.2.1 Block Structure

One of the most important concepts in ALGOL is that of block structure. Each block introduces a new level of nomenclature; identifiers declared in a block are local to that block and have no existence outside that block. This fact can be used so as to make efficient use of core storage.

The most obvious saving would be to use the same storage spaces for variables in separate blocks.

Example

```
B: begin real a;
        . . .
        begin real b;
              . . .
        end;
        . . .
        begin real c;
              . . .
        end;
        . . .
   end
```

Here *b* and *c* are in separate blocks, each contained in the block labelled *B*, and hence could use the same storage space.

The situation becomes more complicated when procedures are considered, if a compact use of storage is required. This is because a procedure can be called into action at various points in a program, which need not have the same set of currently valid declarations.

Example

```
begin real a;
      procedure Q;
      begin real b;
            . . .
      end;
      begin real c;
            Q;
            . . .
      end;
      Q;
      . . .
end
```

At the first call of Q (in the innermost block) there are valid declarations for a and c, whereas at the second call of Q only a has a valid declaration.

The most compact use of space is to have dynamic allocation of storage during the running of the object program. In the Whetstone Compiler this is done using a system whereby space for variables is called into existence only for the duration of the block in which they are declared. The implementation of this uses a stack.

2.2.1.1 The Stack

As each block in the object program is entered, its working storage, i.e. the storage space required for its local variables, is added to the top of the stack. When a block is left this working storage is deleted from the stack. Thus, in the above example, within the procedure body of Q during its first call, the stack will contain the variables a,c and b, and later, during the second call, only the variables a and b will be in the stack.

If a procedure body is re-entered by means of a recursive procedure call before it has been left, then a further set of working storage is created on top of the stack, rendering the original set inaccessible for the duration of the recursive call. Thus the use of a stack allows the same system to be used for the allocation of space for working storage of both recursive and non-recursive procedures.

Dynamic arrays (arrays whose size vary during the execution of a program) are also implemented using the stack. On entry to a block, the expressions in an array declaration are evaluated and the amount of storage space necessary for the array elements is reserved on top of the stack. Thus only the exact amount of space required by an array for the current activation of the block in which it is declared is set aside in the stack.

The technique of dynamic storage allocation using a stack, though fairly simple to implement, and very efficient as regards use of storage, involves a certain amount of extra work at run time. An alternative system (Irons and Feurzeig [41]) involves detecting the recursive use of procedures at run time. In this system the working storage of a given block or procedure is copied into a push-down store if a further recursive activation occurs. In a paper by Jensen, Mondrup and Naur [42], a description is given of a storage allocation scheme, for all but recursive procedures, which allows the programmer to control the use of backing storage.

2.2.1.1.1 Implementation of a Stack. With the stack system the Translator, instead of allocating a separate fixed address to each variable in a program, treats each block separately. Each variable declared in a block is allocated an address relative to the start of the working storage of the block. At run time, when a block is entered, its working storage space is set up in the stack; during the activation of the block the relative addresses of its variables

can be used to calculate the actual addresses. During different activations of a block its variables may well have different actual addresses. (This is shown in the case of the variable *b* in the above example.)

At any point during the running of the object program the stack contains storage space for the variables of each block which has been activated and not yet left. However only those sets of variables which have currently valid declarations will be accessible. It should be noted that in ALGOL the scope of a declaration is determined from the written structure of the program, i.e. lexicographically, rather than from the dynamic flow of the program.

Thus in the above example, although both *a* and *c* are accessible at the first call of *Q*, within *Q c* is shielded, and only *a* and *b* are accessible. On exit from *Q c* will once more become accessible.

In the examples of ALGOL text the indentation which has been used to show the block structure can be used to illustrate the concept of block levels. During translation the lexicographic structure of the program can be used to assign a level to each block, starting at one for the main program. Then the body of procedure *Q* and the inner block, of the above example, both have the main program as their surrounding block, and thus are both said to have a level or 'block number' (*BN*) of two. Since the system of assigning a block number to each block is related to the lexicographical structure, it can be used to determine the scope of a declaration.

Relating this to the stack system, the block numbers corresponding to each set of working storage can be used to determine whether the working storage is currently accessible. One or more sets of variables will be left in the stack, but rendered inaccessible, when a procedure call involves a jump to a parallel, or lower block level. Thus, during the activation of a block at level *n* when the current local variables are at the top of the stack, the only other sets of accessible variables will be in the lexicographically containing blocks, i.e. with levels $n-1$, $n-2$, etc. The sets of variables for any inaccessible blocks will be in the stack, between the various accessible sets. Thus, between the sets corresponding to accessible blocks on levels $i+1$ and i, say, (where $i = 1, 2, \ldots n-1$) any sets with levels greater than i will be inaccessible. As a result, wherever a procedure is activated from within its body only a fixed number of sets of stacked variables (given by the level of the procedure declaration) are accessible, though many more sets may be stacked and inaccessible.

Example

Using the ALGOL text of the above example, during the first call of *Q*, the stack will contain two sets of accessible variables— *a* on level *1*, *b* on level *2*, and between *a* and *b* there will also be the inaccessible *c* on level *2* (a more detailed example is given in section 2.2.1.1.3).

2.2.1.1.2 The Procedure Pointer. During the running of the object program it is necessary to keep a record of the starting position of each set of stacked working storage. For the current block this position is kept in a

counter called *PP*. (This in fact stands for Procedure Pointer – the relationship of blocks and procedures is described later, in section 2.2.1.2.) In order to delete a set of working storage from the stack on exit from a block, the record is consulted to find the value of *PP* that the containing block had used, and *PP* is reset to contain this value.

The record of previous values of *PP* is also used to find the actual address, from its relative address, of a variable in the stack. However, within a block the only variables that can be used are those which are currently accessible. Thus only the values of *PP* for blocks which are currently accessible will be needed to find the addresses of variables. It is therefore preferable to keep two separate records of values of *PP*. The first record gives the start of the stacked working storage of each block that has been activated and not yet left. It is this record that would be used to reset *PP* each time a block is left. The second record contains only the values of *PP* which can be used, in the current block, to find the actual address of a variable.

These records are in fact kept in the stack rather than in two sets of consecutive stores. At the start of each set of working storage in the stack a space is reserved for 'link' data. The link data for a block contains two values of *PP*, which comprise one element from each list. It has previously been stated that *PP* contains the address of the first position of the working storage of the current block. To be exact it contains the address of the link data of the block, which precedes the working storage proper.

Thus each value of *PP* in a record is kept in the word whose address is given by the value of *PP* which forms the next item in the record. The elements of the list of currently accessible blocks form what is called the static chain, whilst the elements of the list of all sets of stacked block information (i.e. link data and working storage) form the dynamic chain. Thus the record of all accessible blocks can be found by starting at the set of link data for the current block, and by working down the static chain (each element giving the address of the next element).

*2.2.1.1.3 **Display.*** For reasons of efficiency, since it must be referred to at each use of a declared variable, the static chain is duplicated in a set of fixed stores (effectively index registers) called *DISPLAY*. Thus the set of values of *PP* for currently accessible blocks, though primarily given as a 'chain', is also given in a straightforward list.

In the Whetstone Compiler, 64 stores have been allocated to *DISPLAY*. This means that the number of block levels which can be declared within each other has been limited to 64. However it is considered extremely unlikely that this limit will be reached in any practical program.

An object program operation referring to a declared variable does so by using a 'dynamic' address, of the form (n,p). Here n is the level, or block number of the relevant declaration, and p is the address of the variable within the working storage of the block. However, for own variables, which are stored as though they are declared at the head of a program, n is zero.

The actual or 'static' address of a variable is then given by

$$DISPLAY[n] + p$$

If *DISPLAY* were not used a declared variable could only be found by working back down the static chain until the correct set of working storage was located. The disadvantage of duplicating the static chain in this way is that care must be taken, during manipulation of the stack, to ensure that *DISPLAY* does in fact continue to mirror the static chain.

```
begin real a;
      procedure Q1;
      begin real b,c;
            . . .
      Q2:   begin real e;
                  procedure R3;
                  begin real f,g;
                        . . .
                  L :   g:=0
                  end R3;
            Q3:   begin real h;
                        . . .
                  M:    R3;
                        . . .
                  end;
                  . . .
            end
      end Q1;
P1:   begin real i,j;
            . . .
      P2:   begin real l;
            N:    Q1;
                  . . .
            end
      end
end
```

FIG. 1(a). Example of block structure.

Example

In the example given in Fig. 1(a), which again uses indentation to show block structure, labels and procedure identifiers have been chosen so that the label of each block (or, if the block is a procedure body, the procedure identifier) shows the level of the block and whether it is part of the main program or of one of the procedures. These labels and procedure identifiers are used to represent the values of *PP* marking the starting positions of the stacked information for their blocks. The main program is assumed to have the label *P*.

(*i*). At label *N* only the blocks *P*, *P1* and *P2* have been activated. These blocks are on successive levels, and have been activated in the order that they occur in the text, so the static and dynamic chains coincide. Thus the link data stacked at the position *P2* contains two elements, both specifying *P1* and the link data at *P1* contains two elements specifying *P*.
The copy of the static chain in the vector *DISPLAY* is as follows

DISPLAY [*1*] = *P*
DISPLAY [*2*] = *P1*
DISPLAY [*3*] = *P2*

(*ii*). At label *M* in procedure *Q1*, which has been called from block *P2*, the situation has become more complicated. On entry to *Q1* the blocks *P1* and *P2*, though still stacked, become inaccessible. Therefore the static chain links together

P, *Q1*, *Q2*, *Q3*

whereas the dynamic chain links together

P, *P1*, *P2*, *Q1*, *Q2*, *Q3*

DISPLAY contains

DISPLAY [*1*] = *P*
DISPLAY [*2*] = *Q1*
DISPLAY [*3*] = *Q2*
DISPLAY [*4*] = *Q3*

(*iii*). At label *L* in procedure *R3* the block *Q3* has become inaccessible. The static chain now links

P, *Q1*, *Q2*, *R3*

whilst the dynamic chain links

P, *P1*, *P2*, *Q1*, *Q2*, *Q3*, *R3*

DISPLAY now contains

DISPLAY [*1*] = *P*
DISPLAY [*2*] = *Q1*
DISPLAY [*3*] = *Q2*
DISPLAY [*4*] = *R3*

Figure 1(b) shows the contents of the stack at this point. In this stylized representation the chain elements have been given in square brackets; the dynamic chain is illustrated on the right of the stack, the static chain on the left.
The sets of link data not currently joined by the static chain still contain a static chain element. This is the element that had been set up when the block was first activated, and will again be used when (and if) the block once again becomes accessible.

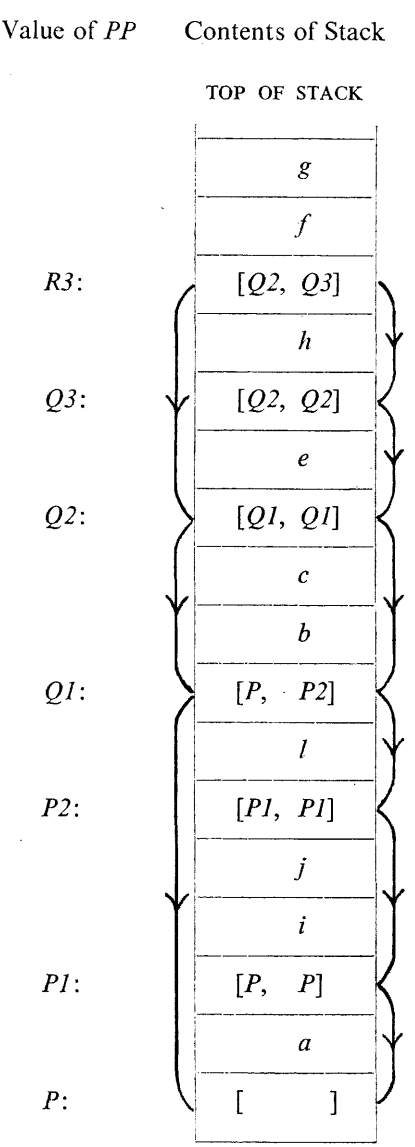


FIG. 1(b). Stylised Representation of contents of stack, as at label *L* in procedure *R*3.

2.2.1.2 Procedures

In the Whetstone Compiler little distinction is made between blocks and procedures. The concept of block levels described above is extended to include procedures, and the activation of both procedures and blocks involves forming a new set of link data at the top of the stack. In the case of a procedure an accumulator space is left below the link data for a possible procedure value or 'result', and space is allocated in the stack above the link data for any parameters. In a block, which naturally does not need a result accumulator, the working storage (for local variables) is placed above the link data. However, in the case of a procedure whose body is an unlabelled block the block level is only increased by one. (Thus the levels given to the blocks in the previous examples are still correct.) The stack space required for the procedure and block is combined, and consists of

(*i*) A Result Accumulator,
(*ii*) Link Data,
(*iii*) Parameter Space, and
(*iv*) Working Storage.

Thus a block could be thought of as a parameterless and nameless procedure which is called at the same point as its 'declaration'.

As was mentioned earlier, recursive activation of a procedure is easily implemented using the stack system. When a procedure is called from within itself a further set of link data, space for parameters, and working storage for the procedure body is stacked, causing the original set of stacked information to become inaccessible. When the recursive activation is finished its stacked information is deleted from the stack, uncovering the original set. Within such a procedure, blocks entered recursively will similarly cause further sets of link data and working storage to be created on the stack, shielding the original set.

A variation on this technique of using a stack for dynamic storage allocation has been described by Hawkins and Huxtable [31]. The method of storage allocation is based on procedure levels rather than block levels; use is made of the fact that a block can be entered recursively only by means of a recursive call on the procedure in which it is embedded. Hence it is possible to assign storage locations to the simple variables declared in each block, which are fixed relative to the start of the procedure (where possible using the same locations for variables in parallel blocks). During translation the structure of the ALGOL program is examined to determine which procedures are capable of being called recursively. Non-recursive procedures use a *DISPLAY* for access to non-local variables; entry and exit to such procedures does not involve any complicated systems for keeping *DISPLAY* up to date. Recursive procedures use the sets of stacked link data in order to gain access to non-local variables.

*2.2.1.2.1 **Function Designators.*** During the description of assignment statements it was mentioned that the evaluation of an expression (using a stack as

a simple push-down store) is not dependent on the starting value of the accumulator pointer (*AP*). As a result it can be arranged that the top of the stack above the working storage for the current block is used for the evaluation of expressions. Thus *AP* has as its starting value (to which it returns after each assignment statement) the address of the first free store above the stacked working storage. This address is called *WP*, the working storage pointer. Then if evaluation of the expression involves activation of a procedure (by means of a function designator) this can set up its link data and working storage on top of the stack, above any accumulators containing anonymous intermediate results of evaluation of the expression. Eventually the value of the function designator is placed in the result accumulator space which has been left under its link data, and the link data and the working storage are deleted from the stack. Thus the total effect of the function designator is to set up one accumulator on top of the stack, and hence evaluation of the expression can proceed normally.

2.2.1.3 Re-declaration of an Identifier

In the above no mention has been made of the possibility of re-declaring an identifier and so rendering a previous declaration of the identifier inaccessible. This situation is dealt with by the Translator, rather than at run time, by ensuring that any use of an identifier is translated into object program operations referring to the currently valid declaration.

The implementation of calls of a procedure occurring outside the scope of any non-local quantities of the procedure body is in accordance with the Revised ALGOL 60 Report.

Example

```
begin real a;
      procedure P (b); integer b; b := a;
      a := 5;
      begin integer a, i;
            P(i)
      end
end
```

In this example the call of procedure *P* in the inner block is outside the scope of the non-local variable *a* used in the procedure body, because of the re-declaration of the identifier *a*. This would have been undefined, according to section 4.7.6 of the original ALGOL 60 Report. However, in the Revised Report section 4.7.6 has been deleted and section 4.7.3.3 amended, so that the above example is valid. In fact it is stated that in such a case conflicts between identifiers are avoided by suitable systematic changes of the identifiers whose declarations are valid at the place of the procedure call. The method described earlier of using the static chain (by means of *DISPLAY*) to address currently accessible variables

automatically ensures that within a procedure body the non-local variables which have declarations valid at the procedure declaration are used. This is obviously equivalent to the systematic changes described in the Revised Report.

2.2.2 Link Data

It is now necessary to consider the subject of link data further. Whenever a block is activated a set of stores is set aside at the top of the stack for link data and working storage. For convenience, within this section on Link Data, the term 'block' is being used synonymously with the term 'procedure'. The link data that has been considered so far consists of the dynamic and static chain elements. These elements are in fact the values of *PP* (the procedure pointer) for previous blocks. The dynamic chain element is the value of *PP* for the block in which the activation takes place, whilst the static chain element is the value of *PP* for the lexicographically containing block of the block being activated, i.e. the block in which the 'declaration' of the block in question occurs.

Take the example of a block *a* in which there is an activation of block *b*, where the declaration of block *b* is in block *c*. The dynamic and static chain elements in the link data for block *b* are the values of *PP* that were current during the most recent activation of blocks *a* and *c*, i.e. *PPa* and *PPc*, respectively.

PPa can be considered as the part of the link data which allows the program to resume block *a* after completion of block *b*. *PPc* can be considered as part of the information that block *b* needs during its activation, in order to perform its task properly. Thus the link data of a block can be regarded as having two distinct tasks – firstly to enable the program to resume correctly after completion of the block, secondly to contain information currently needed by the block. The further items of link data which must now be described can also be categorized according to which of the above tasks they perform.

Two extra items are needed for the correct resumption of working after a block has been completed. A block (and certainly a procedure) can be regarded as a subroutine, and needs the normal form of subroutine link mechanism. Thus the first item is a conventional link, i.e. a 'return address', which is a value of the program counter indicating the position reached in the program of the outer block, when the current block was activated. Stacking this return address in the link data enables the program counter to be reset for the Control Routine to start obeying the object program operations of the new block. At the end of this block the return address is used to restore the program counter to its original value. The second item is a machine code link, needed because a certain form of block is sometimes called from within the Control Routine. The reason for this is explained in section 2.5.5.1.1.

The three remaining items of the link data contain information currently needed by the block. They are *BN* – the block number, or level, of the current

block, *WP*—the current value of the working storage pointer, and *FP*—the formal pointer. *BN* and *WP* have already been described: the formal pointer will be described in section 2.5.6.

The method of storage of the various items of link data depends largely on particular machine characteristics. In the Whetstone Compiler for the KDF9 Computer the seven items have been, for convenience, packed into three words. (see Fig. 2).

Word \ Syllable	0	1	2	3	4	5
2	*RAa*			Machine code link		
1	*BNb*	*WPb*		*FP*		
0	*PPc*			*PPa*		

FIG. 2. Packing of Link Data.

This has been done by packing *BN* and *WP* together, and then allocating a half word for each item. *BN*, though given eight bits in the link data, only needs six since the vector *DISPLAY*, which it addresses, has been arbitrarily limited to 64 elements.

2.2.3 Activation of Blocks and Procedures

Within the object program the main distinctions between a block and a procedure are that a block can never have parameters, and can never define the value of a function designator.

For the present it will be sufficient to state that at the call of a procedure the object program representation of its actual parameters is used to set up a group of 'formal accumulators' in the stack immediately above the link data. A discussion of formal accumulators will be deferred until section 2.5. Within the body of a procedure, references to formal parameters are represented by calls on these formal accumulators. When a procedure (as distinct from a block) is called, a double-word space (called the result accumulator) is reserved at the top of the stack, before stacking the link data. Any value assigned to the procedure identifier is stored in this result accumulator. When the activation of the procedure is finished only the result accumulator is left on the stack. For convenience any activation of a procedure, either by a procedure statement or by a function designator, sets up a result accumulator.

Immediately after entry to a procedure has been completed (i.e. after the

operation *PE*) the stack has been set up with a result accumulator, link data, formal accumulators, and 'first order' working storage (see Fig. 3). First order working storage consists of scalars (real, integer, and Boolean variables), and one word for each array. The full amount of working storage will not have been set up until all the array declarations have been processed so as to set up mapping functions, and space for array elements. (Mapping functions and array elements are said to use 'second order' working storage.)

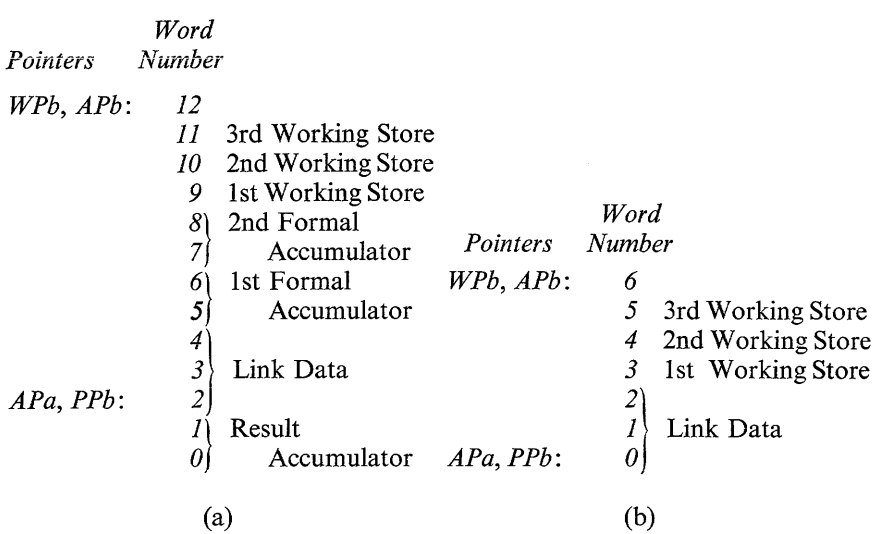


FIG. 3. Stack Reservations on Entries to Procedures and Blocks. (a) Procedure with two formal parameters, three working stores. (b) Block with three working stores.

PP is set to indicate the first word of link data and has in fact been obtained from the value contained in *AP* (the accumulator pointer) before the block or procedure was activated (a result accumulator is reserved simply by adding two to *AP* before using it to form the new *PP*). *AP* is then set to indicate the first free store above the working storage. This position is also given in *WP*, which is kept in the link data. Thus at any time within the activation of a block or procedure *AP* gives the extent of the stack and *PP* can be used to find such quantities as *WP* and *BN*.

The actual process of activating a procedure is shared between two operations. The first operation *CF* (Call Function) or *CFZ* (Call Function Zero) does some preliminary setting up of the link data, sets up any formal accumulators, and then jumps to the object program operation at the start of the procedure, which is *PE* (Procedure Entry). *PE* completes the setting up of the link data, calculates the new values of *AP* and *WP*, and copies the new static chain element into the appropriate element of *DISPLAY*.

CF has two parameters – *a* (a two-syllable program address) and *m* (one syllable, giving the number of parameters). *CF* increases *AP* by two, and uses the stack address thus formed as the position of the start of the link data. The dynamic chain element *PPa* is the current value of *PP*. The return address (the current value of the program counter), and a machine code link, are then placed in the link data. A new value of *PP* is formed from *AP*, the formal accumulators are set up, and finally the program counter is replaced by the value *a*. The control routine as a result jumps to the operation *PE*. An *UJ* operation is placed in front of the operation *PE*, and is used to avoid the procedure on entry to the block in which it is declared.

The system of translation used to generate the object program makes it necessary to make a special case of a procedure with no parameters. The operation *CFZ*, which has a single parameter *a* (a two-syllable program address), is used to call such procedures.

The operation *PE* has three parameters, *n* and *L*, which are packed into two syllables, and *m*, which occupies one syllable. *n* gives the block number, or level, of the procedure; as *DISPLAY* has been limited to 64 elements *n* can be fitted into six bits. The remaining ten bits have been thought sufficient for *L*, which gives the number of words that will be needed in the stack for the formal accumulators and first order working storage. The main task of *L* is to enable *PE* to set aside the working storage space. It is only for convenience that *L* includes the formal accumulator space, which could have been worked out from the parameter *m*, which gives the number of formal parameters belonging to the procedure. This is checked at run time against the number of actual parameters as given with the operation which called the procedure. In the case of a call using *CF* or *CFZ* this is just duplicating a check made at translation time. The check is really needed at run time for calls on formal procedures (see section 2.5.5.6). The parameter *n* is used to obtain the static chain element *PPc*. *PPc* is the value of *PP* for the last activation of a block with a level of '*n* – 1'. This could be found by working back down the dynamic chain, but is in fact obtained from *DISPLAY* [*n* – 1]. The current value of *PP* is placed in *DISPLAY* [*n*]. *BN* (given by *n*), *WP* (calculated using *L*), and *FP* (set up as the stack address of the first formal accumulator) are placed in the link data, and finally *AP* is set equal to *WP*.

The activation of a block uses the operations *CBL* (Call Block) and *BE* (Block Entry). *CBL* has no parameters – *m* is obviously not needed since it must be always zero. Furthermore, because a block is activated at the same point as it is declared, it is unnecessary to give *CBL* the address of the *BE* operation as a parameter, since the operation *BE* is fixed relative to *CBL* (see section 2.2.7). *BE* is a simplified version of *PE* (Procedure Entry) without the parameter *m*, since no checking is necessary.

2.2.4 Completion of Blocks and Procedures

When the action of a block or procedure has been completed its link data and working storage must be deleted from the stack, so that the action of the

containing block (or procedure) can be resumed. This is done by the operation *RETURN*. In the case of completion of a procedure *RETURN* has the extra task of ensuring that *DISPLAY* is reset to the state it was in before the activation of the procedure.

Example

Using the example given in Fig. 1, after entry to procedure *Q1* the dynamic chain is

P, P1, P2, Q1

and *DISPLAY* is

DISPLAY [*1*] = *P*
DISPLAY [*2*] = *Q1*

When procedure *Q1* is finished a return must be made to the block in which *Q1* was activated (namely *P2*), and *DISPLAY* updated such that

DISPLAY [*1*] = *P*
DISPLAY [*2*] = *P1*
DISPLAY [*3*] = *P2*

However, leaving block *P2* to return to block *P1* just causes the third item in *DISPLAY* to become redundant.
Thus *DISPLAY* is effectively

DISPLAY [*1*] = *P*
DISPLAY [*2*] = *P1*

The task of updating *DISPLAY* to duplicate the static chain is performed by a subroutine UDD (Update Display) which is called into action by the *RETURN* operation. Since UDD, by inspection of *DISPLAY*, can determine whether any resetting is in fact necessary, it is convenient to use exactly the same *RETURN* operation for blocks as for procedures.

2.2.4.1 Update Display

The method of updating *DISPLAY* is to work back down through the static chain given in the various sets of stacked link data, using each static chain element to correct the corresponding element of the vector *DISPLAY*. In theory it is sufficient to terminate this process as soon as the level at which the static chain and *DISPLAY* coincide has been reached. However, in practice extra care must be taken when exit is being made from one level to a higher level (e.g. in the above example, of return from procedure *Q*). This is because the elements of *DISPLAY* for the higher levels may contain obsolete values of *PP*. Thus it is necessary to check *DISPLAY* down to the level of the procedure being left, and thereafter until the static chain and *DISPLAY* coincide. Alternatively, if so desired, it is possible to avoid this complication by zeroing elements of *DISPLAY* when they become redundant.

2.2.5 Assignment to Procedure Identifier

An assignment to a procedure identifier is needed in a procedure which is to be called by a function designator. In addition, the declaration of such a procedure must start with a type declarator (**real**, **integer**, or **Boolean**). (See section 5.4.4 of the Revised ALGOL Report). For convenience of checking, the Whetstone Compiler insists that type procedures include an assignment to the procedure identifier, even if they are not to be called by a function designator. In the object program this is represented by an assignment to the result accumulator at the head of the stacked information for the procedure. This is done using the normal 'Take Address' and 'Store' operations.

The stack address of any variable is given in the object program by a dynamic address of the form (n,p); n enables the value of *PP* for the block or procedure containing the variable to be found from *DISPLAY*, p is the position of the variable in the stack relative to this value of *PP*. *PP* is set to indicate the stack address of the link data, and thus the stack address of the first word of the result accumulator is '*PP*-2'. For convenience this is represented by a dynamic address $(n,0)$, rather than $(n,-2)$, which would involve using one of the ten bits allotted to p as a sign digit.

The 'Take Address' operation, when it has a parameter of the form $(n,0)$, stacks the value of '*DISPLAY* $[n]-2$', and an identifying bit pattern. The bit pattern is used to identify the address as being that of a result accumulator. As a result the 'Store' operation, in addition to its normal task of real-integer conversion, and storage of the value given in the top accumulator, sets up a bit pattern in the second word of the result accumulator.

Example

integer procedure *P*; *P* := *3*;

is translated into

Syll	*Op*	*Par*
0	*PE*	*(2,0), 0*
4	*TIA*	*(2,0)*
7	*TIC*	*'3'*
14	*ST*	
15	*RETURN*	
16	...	

It is assumed here that the procedure is given a block number, or level, of two.

2.2.6 The Operation *REJECT*

It is necessary to allow for the possibility of a type procedure being called by means of a procedure statement instead of a function designator. In such a case an unwanted result accumulator will have been left at the top of the stack after return from the activation of the procedure. This accumulator is

deleted by the operation *REJECT*, which simply decreases *AP* by two. For convenience all procedure statements use a 'Call Function' operation followed by *REJECT*.

2.2.7 The 'Return Mechanism' for Blocks

Since a block is activated at the point where it is 'declared', it is necessary to allow for the action of the operation *RETURN*, which causes a jump to the operation following the *CBL* operation.

In fact the layout of the object program corresponding to a block is

```
        CBL
        UJ      (a)
        BE      (n,L)
        ...
        ...
        RETURN
a:      ...
```

Thus the *CBL* operation jumps forward four syllables to the *BE* operation. Eventually *RETURN* jumps to the *UJ* operation which jumps around the 'declaration' of the block. (An alternative system would be to introduce a special operation, *BLOCK RETURN*.)

2.2.8 Summary

By way of summary to the description of blocks and procedures, an example of a small 'program' is given in Fig. 4, in which a type procedure is called both by a function designator and by a procedure statement.

```
begin integer i, j;
      real procedure P;
      begin real x;
            x := 0;
            P := x-1
      end;
      j := 1;
      begin integer j, k, l;
            l := P
      end;
      P
end
```

FIG. 4(a). Algol Text.

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		Call Program Block
1	*UJ*	(*54*)	
4	*BE*	(*1,2*)	Program Block
7	*UJ*	(*29*)	
10	*PE*	(*2,1*), *0*	Procedure *P*
14	*TRA*	(*2,3*)	*x*
17	*TIC0*		
18	*ST*		*x* := *0*
19	*TRA*	(*2,0*)	*P*
22	*TRR*	(*2,3*)	*x*
25	*TIC1*		
26	–		
27	*ST*		*P* := *x*–*1*
28	*RETURN*		
29	*TIA*	(*1,4*)	*j*
32	*TIC1*		
33	*ST*		*j* := *1*
34	*CBL*		Call Inner Block
35	*UJ*	(*49*)	
38	*BE*	(*2,3*)	Inner Block
41	*TIA*	(*2,5*)	*l*
44	*CFZ*	(*10*)	Call *P*
47	*ST*		*l* := *P*
48	*RETURN*		
49	*CFZ*	(*10*)	Call *P*
52	*REJECT*		
53	*RETURN*		
54	*FINISH*		

FIG. 4(b). Object Program.

The first operation, *CBL*, starts to set up the first set of link data, and calls the *BE* operation at syllable *4*, which marks the beginning of the program block. The *BE* operation specifies the block number to be one, and the number of local variables to be two. The first action of the program block is to jump around the declaration of procedure *P*, i.e. to syllable *29*. The next three operations perform '*j* := *1*'. The dynamic address of *j*, since it is the second local variable in a block of level one is given as (*1,4*)–as always, allowing three words for link data.

The *CBL* operation at syllable *34* causes a jump to the *BE* operation which starts the inner block. In this block the address of *l* is stacked, and then the operation *CFZ* is used to call procedure *P*. The *PE* operation which starts the procedure, at syllable *10*, gives its level as two, and the amount of working space as one. The first three operations of the procedure perform '*x* := *0*'. The *TRA* operation at syllable *19* then stacks the address of the result

accumulator of *P*. After computing the right hand side using the next three operations the *ST* operation at syllable *27* stores '$x-1$' in the result accumulator of *P*. The operation *RETURN* deletes the link data and working storage of procedure *P* from the stack, leaving its result ('$x-1$') as the top entry, before returning to syllable *47*. (This syllable number was stacked when the procedure was called by the operation *CFZ* at syllable *44*.) Thus the value of the function designator *P* is stored in *l*. The next operation, *RETURN*, deletes the stacked information corresponding to the inner block, and returns to syllable *35*, which is a jump around the declaration of the inner block, to syllable *49*. This is another call on procedure *P*. Eventually a return is made from *P*, with its result in the result accumulator on top of the stack, to syllable *52*. The operation *REJECT* deletes the result of *P*. The next operation *RETURN* completes the main program block; control is passed to syllable *1* and hence to syllable *54*. This is an operation *FINISH*, which has not previously been mentioned. Its task is to mark the end of a translated program.

2.3 ARRAYS

The first order working storage, which is set aside in the stack when a block or procedure is activated, contains scalars (real, integer, and Boolean) and array words. In the object program all references to an array refer to the appropriate array word.

Example

$B[1, 10] := 0;$

is translated into

Syll	*Op*	*Par*	
0	*TRA*	*(3,7)*	
3	*TIC1*		The dynamic address of the
4	*TIC*	*'10'*	array word of *B* is *(3,7)*, say.
11	*INDA*		
12	*TIC0*		
13	*ST*		
14	...		

The array elements and their storage mapping function occupy second order working storage. A storage mapping function is a linear expression (described in section 2.3.1) used by the operation *INDA* or *INDR* to locate a particular array element. An array declaration is translated into object program operations, which at run time calculate the storage mapping function. A storage mapping function is set up for each array segment. The number of words required for a storage mapping function is equal to the number of dimensions of the arrays.

Example

integer array $A, B, C, D[1:4, 0:5, -2:0], E, F[0:7];$

This declaration will cause space for six array words to be allocated in first order working storage. The number of elements in each array of the first array segment is $4 \times 6 \times 3 = 72$. The arrays in the second array segment have 8 elements each. Therefore the amount of second order working storage needed is $3+4 \times 72+1+2 \times 8 = 308$ words.

Each array segment is translated into a set of object program representations of each arithmetic expression in the bound pair list, followed by the operation *MSF* (Make Storage Function). *MSF* has as its parameters the dynamic address allocated to the last array word, and the number of array

identifiers in the segment. Array words are allocated dynamic addresses in sequence.

Example

Boolean array *B* [*i*: *j* + *3*, *0*: *1*], *C*, *D* [− *2*: *j*];

is translated into

Syll	*Op*	*Par*
0	*TIR*	(*2,4*)
3	*TIR*	(*2,5*)
6	*TIC*	'*3*'
13	+	
14	*TIC0*	
15	*TIC1*	
16	*MSF*	(*3,6*), *1*
20	*TIC*	'*2*'
27	*NEG*	
28	*TIR*	(*2,5*)
31	*MSF*	(*3,8*), *2*
35	...	

Here *i* and *j*, declared in an outer block, are assumed to have addresses (*2,4*) and (*2,5*). The array words for arrays *B*, *C*, and *D*, have been allocated addresses (*3,6*), (*3,7*) and (*3,8*) respectively. When the operation *MSF*, at syllable *16*, is reached, the top of the stack will contain the current values of the lower and upper bounds of each subscript of the array. *MSF* uses these values as data in order to calculate the storage mapping function.

2.3.1 The Storage Mapping Function

The elements of an array are stored sequentially, each element taking one word of storage.

Example

array *A* [−*1*: *1*, *0*: *2*];

has its elements stored in the order

A[−*1,0*], *A*[*0,0*], *A*[*1,0*], *A*[−*1,1*], *A*[*0,1*], *A*[*1,1*], *A*[−*1,2*], *A*[*0,2*], *A*[*1,2*]

The storage mapping function of an array consists of a set of integer coefficients which are used by the operations *INDA* and *INDR* to find the location of a particular element of an array specified by means of a subscripted variable.

The location of the element

$$A[s_0, s_1, \ldots, s_{n-1}]$$

of the n-dimensional array can be given as

$$b + s_0 \times c_0 + s_1 \times c_1 + \ldots + s_{n-1} \times c_{n-1},$$

where b is the base address of the array, i.e. the address of $A[0,0,\ldots,0]$ (this need not be an element of the array). The integers c_i $(i = 0, \ldots, n-1)$ are calculated from the subscript bounds of the array; if the declaration of array A is

array $A\ [l_0: u_0, l_1: u_1, \ldots, l_{n-1}: u_{n-1}]$;

then the integers c_i are given by the recurrence relation

$$c_i = c_{i-1} \times (u_{i-1} - l_{i-1} + 1),\ i = 1, \ldots, n-1,$$

where $c_0 = 1$.

Here '$u_i - l_i + 1$' gives the 'range' of the $(i+1)^{\text{th}}$ subscript, and hence the integers c_i give the amount by which the address of an element changes corresponding to a change of one in the $(i+1)^{\text{th}}$ subscript. c_0 has the value one, because each element occupies one word.

In fact when space is to be allocated in second order working storage for a set of array elements it is the address, w, say, of the first element $(A[l_0, l_1, l_2, \ldots, l_{n-1}])$, rather than the base address b, which is known.

However

$$w = b + l_0 \times c_0 + \ldots + l_{n-1} \times c_{n-1}$$

Therefore

$$b = w - (l_0 \times c_0 + \ldots + l_{n-1} \times c_{n-1})$$

Example

array $B\ [-1: 2, 0: 4, 1: 4]$;

w is given as, say, 96.

Here

$$c_0 = 1$$
$$c_1 = 1 \times (2 - (-1) + 1) = 4$$
$$c_2 = 4 \times (4 - 0 + 1) = 20$$
$$b = 96 - ((-1) \times 1 + 0 \times 4 + 1 \times 20) = 77$$

The number of elements in an array is obtained using the recurrence relation for c_i, and is

$$c_n = c_{n-1} \times (u_{n-1} - l_{n-1} + 1)$$

The storage mapping function for an n-dimensional array consists of n integers

$$c_n, c_1, c_2, \ldots c_{n-1}$$

(Advantage has been taken of the fact that $c_0=1$ and therefore need not be stored, to replace it by c_n – the size of the array.) The storage mapping function, and the array word, are set up by the operation *MSF*. The array word consists of three 16-bit integers – *w,b* and the address of the first word of the storage mapping function.

By this system of storing certain items of the information required to locate an array element in the array word, rather than with the storage mapping function, it is possible to use the same storage mapping function for each array in an array segment.

A paper by Sattley [64] describes a somewhat similar system of storage allocation for arrays, using, for each array, a form of storage mapping function ('dope vector'!) which contains the values of the subscript bounds.

2.3.1.1 The Operation MSF

The task of the operation *MSF* is to set up a storage mapping function, and an array word for each array in an array segment, and to allocate space in the stack for the elements of each array.

An array segment is translated into a set of object program representations of the subscript bound expressions, followed by *MSF*. Thus when the operation *MSF* is obeyed the top of the stack will contain '$2\times n$' accumulators, each giving the current value of a subscript bound expression. It is not necessary to give n (the number of dimensions of the arrays) as a parameter to *MSF*, since it can be found from *WP* (the working storage pointer, which is kept in the stacked link data), and *AP*. This is because it is known that before the evaluation of the subscript bound expressions was started, *AP* and *WP* were equal.

Therefore

$$n = (AP - WP)/4$$

The division by four is because each accumulator takes two words, and there are '$2 \times n$' accumulators.

MSF uses the accumulators (if necessary performing conversions from real to integer) and replaces them with n words containing the storage mapping function. Then space is allocated in the stack above the storage mapping function, for each array. As this is done the corresponding array word in first order working storage is set up. Finally *AP* and *WP* are increased to indicate the first free store in the stack above those allocated for the arrays.

Example

integer array *A, B, C* [*0*: *1*, *−1*: *1*];

This is translated into

Syll	*Op*	*Par*
0	*TIC0*	
1	*TIC1*	
2	*TIC1*	
3	*NEG*	
4	*TIC1*	
5	*MSF*	*(1,7), 3*
9	...	

A representation of the stack, after completing the evaluation of the subscript bound expression is given in Fig. 5(a). Stack addresses are given relative to the current position indicated by *PP*.

For this array segment

$$c_0 = 1$$
$$c_1 = 1 \times (1 - 0 + 1) = 2$$

The number of elements,

$$c_2 = 2 \times (1 - (-1) + 1) = 6$$

Therefore the storage mapping function is

c_2, c_1, i.e. *6,2*.

Fig. 5(b) shows the stack after completion of the operation *MSF*. It has been assumed that the block has seven first-order working stores, and that the array words *A*, *B* and *C* have been allocated dynamic addresses *(1,5)*, *(1,6)* and *(1,7)* respectively. Because *MSF* has the *(n,p)* address of the last array word as a parameter, it is convenient to allocate space for the sets of array elements in inverse order to that of the corresponding array identifiers.

During translation, various checks are made on the use of subscripted variables to reference elements of arrays, in particular that only identifiers declared, or specified, to be array identifiers are used for subscripted variables. Thus it is sufficient for the address of an array word to be stacked (for use by *INDA* or *INDR*) with a bit pattern indicating, say, 'real address', rather than 'real array address'. Similarly it is in general possible for the Translator to ensure that the correct number of subscript expressions is given in each subscripted variable. However, although a check is made that each use of a subscripted variable referring to a formal array has the same number of subscript expressions, the Translator does not check that corresponding actual and formal arrays have the same number of dimensions. This particular check is therefore carried out at run time by the operations *INDR* and *INDA*, which in fact always check that the correct number of subscript expression

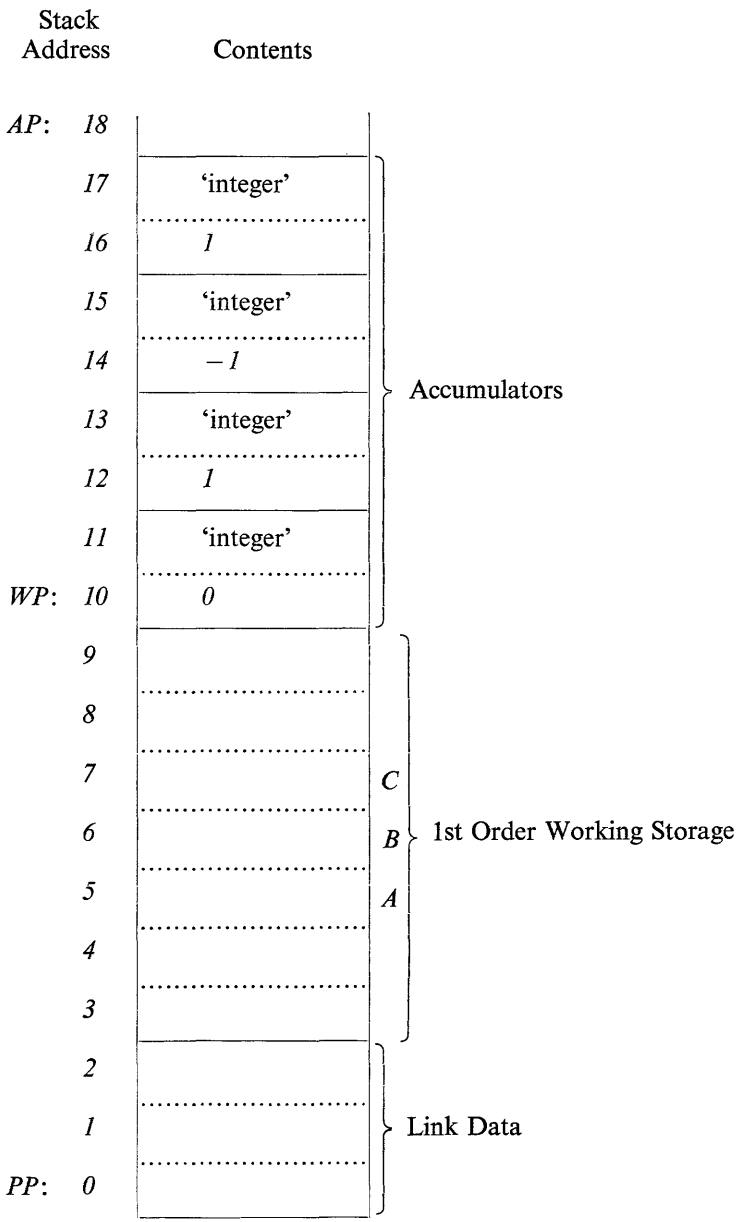


FIG. 5(a). After evaluation of subscript bound expressions.

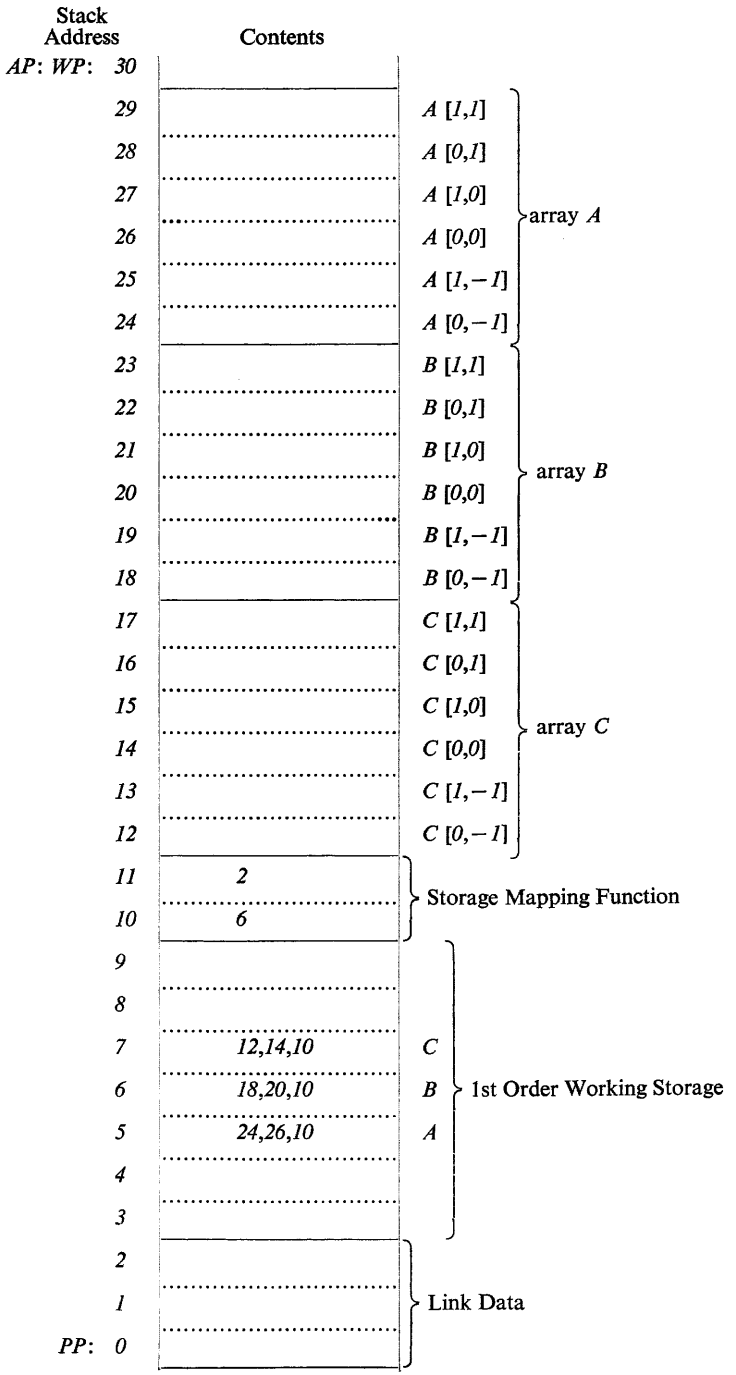


FIG. 5(b). After operation *MSF*.

values is given at the top of the stack above the address of the array word. In order that this check can be carried out, the number of dimensions of the arrays in a given array segment, which is calculated by the operations *MSF* and *MOSF*, is for convenience placed in the most significant half of the first word of the storage mapping function.

2.3.2 The Operations *INDA* and *INDR*

These operations work on the top accumulators, which contain the values of the subscript expressions, and the address of an array word. The array word is used to locate the storage mapping function. The array element specified by the subscripted variable can then be found, as described in the section on the storage mapping function.

In the Whetstone Compiler it has been decided that it will be sufficient to check that a subscripted variable does in fact specify an element within the confines of the array. A complete check that each subscript is within the subscript bounds could easily be performed, but would increase the amount of information needed in the storage mapping function.

The type (real, integer or Boolean) of an array element is given in the operation which stacks the address of the array word, and hence need not be given with an indexing operation, or with the operation *MSF*.

2.3.3 Own Arrays

It has already been mentioned that own variables are treated as though they were declared outside the program block. In fact space is set aside for 'own' working storage before stacking the link data for the program block. This working storage is again divided into first order working storage for scalars and array words, and second order working storage for array elements and storage mapping functions. One of the main restrictions to ALGOL imposed in the Whetstone Compiler is with regard to 'dynamic own arrays'. Only own arrays with subscript bounds which are integers are accepted. Because of this the Translator can calculate the number of elements contained in each own array. As a result the Translator can produce, for use by the object program, two counters L_p and L_0, which give the extent of first and second order own working storage. The value of *PP* for the program block (which gives the location of its link data) is taken to be '$L_p + L_0$'. Own variables and own array words are allocated dynamic addresses of the form $(0,p)$.

Own arrays are set up by the operation *MOSF* (Make Own Storage Function). This is very similar to *MSF* but uses L_p, instead of *WP*, to give the position at which the storage mapping function is to be set up. Then L_p, which originally indicated the extent of first order own working storage, is increased to allow space for the storage mapping function and the own array elements.

Because *MOSF* increases L_p each time it is obeyed, it is necessary to ensure that *MOSF* is only obeyed the first time that the block containing the array

declaration is entered. This is done using the operation *AOA* (Avoid Own Array). The first time *AOA* is used it acts as a dummy operation, but on any subsequent uses it acts as an unconditional jump around the operations which set up the own array.

Example

own real array *A*, *B* [*1*: *4*], *C* [−*1*: *2*];

is translated into

Syll	*Op*	*Par*
58	*AOA*	(*86*)
61	*TIC1*	
62	*TIC*	'*4*'
69	*MOSF*	(*0*,*7*), *2*
73	*TIC1*	
74	*NEG*	
75	*TIC*	'*2*'
82	*MOSF*	(*0*,*8*), *1*
86	...	

Here the array words *A*, *B* and *C* have been given the addresses (*0*,*6*), (*0*,*7*) and (*0*,*8*).

This system has the disadvantage that space is used for own arrays even before their declaration is met for the first time, but it permits use of the normal stack mechanism.

The implementation of own arrays in ALGOL is a somewhat vexed question and the above does not pretend to be any other than a very inadequate treatment. A further discussion of the subject of own arrays is contained in Appendix 2.

2.4 LABELS AND SWITCHES

2.4.1 Simple go to Statements

A go to statement can be used to jump to a statement in the current block or in an outer block. In the simple case of a jump within the current block all that is necessary is to alter the program counter appropriately; the working of this go to statement need be no more complicated than that of the implicit jump operation *UJ*. However, in the more complicated case of a jump out of a block, it is necessary to ensure that when the outer block is reached it will be able to resume its working correctly. Furthermore it is necessary to jump to the last activation of this outer block.

In order that a block can resume correct working it is necessary to update *DISPLAY*, and the counters *AP* and *PP*. It is known that at the object program operation equivalent to the start of the labelled statement, *AP* will have the value to which it returns between each statement, i.e. *WP*, the working space pointer. The value of *WP* is kept in the link data of a block, and can be obtained once the *PP* of the block is known. Similarly, once this value of *PP* is known, *DISPLAY* can be updated, using the routine *UDD*, which works down the static chain. For these reasons the object program representation of a label used in a go to statement consists of a value of *PP*, and of the program counter. However, in different activations of a block the value of *PP* will vary, and hence the label can be said to have as many different 'values' as the block in which it is 'declared' has different activations.

It can be ensured that a jump is made to the last activation of a block by using *DISPLAY* to find the appropriate value of *PP* to use in a label. This is done by the operation *TL* (Take Label), which is a form of 'Take Result' operation. *TL* has two parameters—a value of the program counter (two syllables), and a one-syllable parameter which gives the block number, or level, in which the label is 'declared' (i.e. attached to a statement).

The action of *TL* is to obtain a value of *PP* from the element of *DISPLAY* corresponding to its block number parameter, and to stack an accumulator containing this value of *PP*, and the value of the program counter. This accumulator is used (and deleted) by the parameter operation *GTA* (Go To Accumulator), which effects the required jump.

Example

go to *L1*;

is translated into

Syll	*Op*	*Par*
19	*TL*	*(59), 2*
23	*GTA*	
24	...	

This will cause a jump to the object program at syllable *59* in a block on level two.

However, the basic symbol **go to** is in general followed by a designational expression, of which a label is but a special case. The method of implementing conditional designational expressions is similar to the method used for conditional arithmetic expressions (see section 2.1.5).

Example

go to if *b* **then** (**if** *c* **then** *L1* **else** *L2*) **else** *L3*;

is translated into

Syll	*Op*	*Par*	
0	*TBR*	*b*	
3	*IFJ*	(*26*)	
6	*TBR*	*c*	The original identifiers have
9	*IFJ*	(*19*)	been used as parameters to the
12	*TL*	*L1*	'Take' operations
16	*UJ*	(*23*)	
19	*TL*	*L2*	
23	*UJ*	(*30*)	
26	*TL*	*L3*	
30	*GTA*		
31	...		

It has not been thought worthwhile to make a special case of a jump to a label which is in the current block. Thus the operation *GTA* needs to test whether a jump is within a block or to an outer block. This is done by comparing the current value of *PP* with that given in the top accumulator. Only if these two values are different is it necessary to reset *PP*, *AP* and *DISPLAY*.

2.4.1.1 Labelled Blocks

It has already been stated that, in the case of a procedure body which is an unlabelled block, the stack space required for the procedure and its body is combined. Thus such a procedure causes an increase in block level of only one. However, if the body is a labelled block, it must be possible to leave the block, by a go to statement, without leaving the procedure body.

Example

```
    procedure PR;
L: begin array B [1:n];
        . . .
        go to (L);
        . . .
   end;
```

The statement '**go to** (*L*)' could be used to re-enter the block, and re-declare the array *B*, using a different value of the non-local variable *n*, without leaving the procedure.

A similar situation arises when a program is a labelled block. It must be possible to restart the program by means of a go to statement in the block, and hence a 'double block' is generated.

Example

```
PROG: begin real i;
            go to PROG
      end
```

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		Call Program
1	*UJ*	(*21*)	
4	*BE*	(*1,0*)	Entry to Program
7	*CBL*		Call Block
8	*UJ*	(*20*)	
11	*BE*	(*2,1*)	Entry to Block
14	*TL*	(*7*), *1*	*PROG*
18	*GTA*		**go to** *PROG*
19	*RETURN*		
20	*RETURN*		
21	*FINISH*		

The reasons for generating a double block in the above situation are rather trivial. Firstly, it should be possible to restart the program, having caused the variables declared at the head of the labelled block to become undefined, by jumping out of the block to its label. Secondly, the alternative method of effectively moving the label to within the block would cause problems at translation time in the case where there is a declaration of the same identifier within the block.

For convenience a program is always treated as a block, whether in fact it consists of a block or a compound statement. Jumping out of a compound statement does not have the same significance as jumping out of a block; as a result it is not necessary to make a double block out of a program which is a labelled compound statement.

2.4.2 Switch Declarations

A switch declaration allows a switch designator to be used to choose one of several designational expressions. The definition of a switch declaration is recursive. Thus the designational expression appearing in a switch declaration may include calls on the switch either explicitly or inside function designators.

Example

```
switch S := L1, if b then L2 else S [3], L3;
Boolean procedure b;
        begin b := true ;
              if fail = 0 then go to S [1]
        end;
```

For this reason the object program representation of a switch must be capable of being used recursively, and as a result a switch is treated as if it were a procedure. In fact a switch could be regarded as a 'label procedure', which like any 'type procedure', has the net effect of leaving its 'result' (a label) in the top accumulator on the stack.

The actual choice of a designational expression from the switch list is governed by the value of the subscript expression, or 'switch index', in the switch designator. In the object program this choice is effected using the operation *DSI* (Decrement Switch Index), which precedes the set of operations corresponding to each designational expression. *DSI* decreases the switch index by one, and unless the index is then zero, jumps to the next designational expression. At the end of each expression an *UJ* operation is used to jump to the end of the switch declaration.

Example

L1, **if** *b* **then** *L2* **else** *L3*, *L4*

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*DSI*	(*10*)	
3	*TL*	*L1*	*L1*
7	*UJ*	(*44*)	
10	*DSI*	(*33*)	
13	*TBR*	*b*	
16	*IFJ*	(*26*)	
19	*TL*	*L2*	**if** *b* **then** *L2* **else** *L3*
23	*UJ*	(*30*)	
26	*TL*	*L3*	
30	*UJ*	(*44*)	
33	*DSI*	(*43*)	
36	*TL*	*L4*	*L4*
40	*UJ*	(*44*)	
43	*ESL*		
44	...		

In the above example the parameterless operation *ESL* (End Switch List) is introduced. This is used to check out the use of a switch designator with an out of range switch index. It will be seen that each *DSI* operation has as

a parameter the syllable number of the next *DSI* operation, or finally, of the *ESL* operation. Each *UJ* operation which follows a designational expression has the syllable number of the operation following *ESL* as its parameter.

Thus if the switch index has an initial value of one, the first *DSI* operation will zero it, and hence the *TL* operation at syllable *3* will be used. This sets up a label accumulator with the object program representation of *L1*, before a jump is made to syllable *44*. If the switch index does not have the value 1, 2 or 3 then the operation *ESL* is reached, and a failure indication is given.

The operation *ESL* could be used instead to permit the implementation of a go to statement using a switch designator which is undefined because the switch index is out of range, acting as a dummy statement (see section 4.3.5 of Revised ALGOL Report). This is not done in the Whetstone Compiler for reasons of compatibility with the Kidsgrove Optimizing Compiler. In this case the *ESL* operation instead of leading to a failure would set up a 'dummy label'. The operation *GTA* (Go To Accumulator) would be modified such that when it found a label accumulator containing a dummy label no jump would be performed.

A switch declaration is made into a 'switch block' which, however, has a 'result accumulator' and stacks link data in the normal way. The set of operations corresponding to the designational expressions are preceded by a *BE* (Block Entry) operation, and are followed by the operation *EIS* (End Implicit Subroutine). The *EIS* operation, which is also used in the object program representation of certain actual parameters, stores the result of the chosen designational expression in the result accumulator, before performing the action of the operation *RETURN*. The switch index is kept in the one and only working store of the block, Hence the *BE* operation specifies the amount of working storage to be one. The switch block is preceded by an *UJ* operation, used to avoid the switch block on entry to the block in which it is declared.

Example

switch *S* := *L1*, *L2*;

is translated into

Syll	*Op*	*Par*	
0	*UJ*	(*28*)	
3	*BE*	(*4,1*)	
6	*DSI*	(*16*)	
9	*TL*	*L1*	The level of the switch block
13	*UJ*	(*27*)	has been chosen arbitrarily
16	*DSI*	(*26*)	
19	*TL*	*L2*	
23	*UJ*	(*27*)	
26	*ESL*		
27	*EIS*		
28	...		

2.4.3 Switch Designators

Because of the system of translation used in the Whetsone Compiler, it is necessary that the object program representation of a switch designator be similar to that of a subscripted variable. In fact the *INDR* operation, which is primarily used as an indexing operation, is also used to activate the block which has been generated from a switch declaration. In place of the 'Take Address' operation which stacks the address of an array word the operation *TSA* (Take Switch Address) is used.

Example

go to *S* [*i* − *1*];

is translated into

Syll	*Op*	*Par*
0	*TSA*	*S*
3	*TIR*	*i*
6	*TIC1*	
7	−	
8	*INDR*	
9	*GTA*	
10	...	

The operation *TSA* has a two-syllable parameter which gives the address of the *BE* operation of the relevant switch declaration. Its action is simply to stack an accumulator containing this address. *INDR* works back down the stack, performing conversions to integer if necessary, until it finds an accumulator containing an address. When *INDR* is being used for a switch designator the bit pattern in this accumulator identifies it as a switch address. (Checks incorporated in the translation process ensure that a switch designator will have one and only one subscript expression.) *INDR* then positions the switch index, given in the top accumulator, in the stack so that it will be in the first working storage of the 'switch block'. Finally *INDR*, acting rather like the operation *CFZ*, calls the switch block, positioning its link data so that its 'result accumulator' is at the stack address previously occupied by the accumulator containing the switch address. As a result, when the switch block is left, using the operation *EIS*, its label result is in an accumulator which has replaced the accumulators containing the switch address and the switch index.

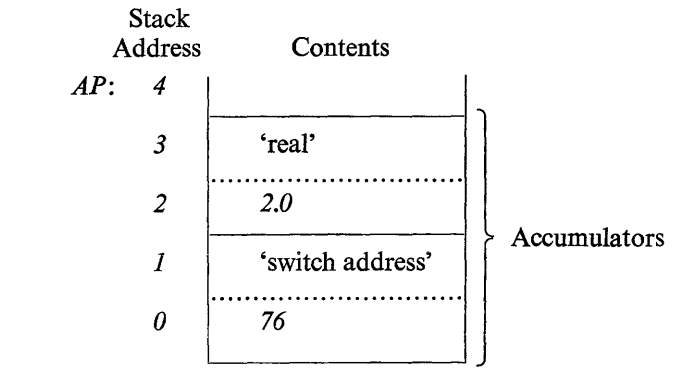
Example

go to *S* [*2.0*];

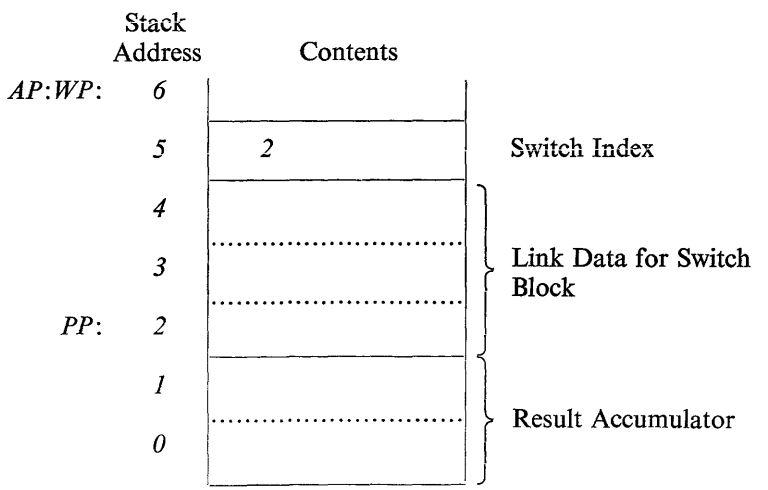
is translated into

Syll	*Op*	*Par*
0	*TSA*	*(76)*
3	*TRC*	*'2.0'*
10	*INDR*	
11	*GTA*	
12	...	

In Fig. 6(a) is shown the top of the stack, containing two accumulators. The first contains the address of the *BE* operation of the switch block generated from the declaration of switch *S* (at syllable *76*, say). The top accumulator has been set up by the operation *TRC* (Take Real Constant).



(a). Before operation *INDR*.



(b). After entry to switch block.

FIG. 6. Use of the stack by switch designators.

In Fig. 6(b) the same portion of the stack is shown immediately after entry to the switch block. *INDR* has converted the real number *2.0* to integer, and has moved it to stack address *5* (stack addresses have been given relative to the switch address accumulator). The switch block has been entered at syllable *76*, its link data starting at stack address *2*. As a result the value *2* (the switch index) is in the single working store of the switch block. *PP* is set at *2*, and *AP* and *WP* indicate the first free store at the top of the stack.

When the switch block is left, using the operation *EIS*, the result (a label) is placed in the result accumulator which becomes the top accumulator. The total 'effect' of the operations generated from the switch designator is thus to stack a label, for use by the operation *GTA*.

It has been mentioned that a switch designator is translated as if it were a subscripted variable. As a result it is convenient in certain circumstances to generate the operation *INDA* rather than *INDR* (see section 3.4.2). Therefore both *INDA* and *INDR* can be used to set up a switch block in the stack, and are in this case equivalent.

The implementation of a switch designator occurring outside the scope of a quantity entering into a designational expression in the switch list is in accordance with the amended version of section 5.3.5 of the original ALGOL Report given in the Revised ALGOL Report. (An essentially similar situation has been described in section 2.2.1.3 in connection with procedure calls.)

Example

```
      begin switch S := L, M;
M:    begin real L;
            go to S[I];
            L := 0
      end;
L: end
```

In the inner block a switch designator is used, and selects the label *L* from the switch list. However, since the switch designator caused a temporary return to the position of the switch declaration, a conflict with the declared *L* in the inner block will be avoided, and a jump will be made to label *L* at the end of the outer block.

2.5 PARAMETERS

The parameters of a procedure provide a channel of communication between the body of the procedure, and the external conditions appertaining to its call. When a procedure is activated, any actual parameters will be written subject to the declarations valid at the position of its call. In the procedure body the corresponding formal parameters will give access to these actual parameters, even if the procedure body is at a level from which these declarations are inaccessible.

Example

```
begin real procedure P(a); integer a;
          P := a × (a − 1);
      real b;
      b := 1;
      begin real c;
            c := 2;
            begin real d;
                  d := 3;
                  d := P(b × c × d)
            end
      end
end
```

In this 'program' there are two declarations in the program block, i.e. on level *1*. These are for the procedure *P* and the real variable *b*. Therefore, within the body of procedure *P*, i.e. on level *2*, the only non-local variable that can be used is *b*. However, there is an activation of *P* in the innermost block of the program, i.e. on level *3*, with the expression '$b \times c \times d$' as an actual parameter. Within the procedure body the corresponding formal parameter (called by name) is *a*. Therefore this parameter is being used as a means of access to the variables *b*, *c*, and *d*. Of these variables only *b* would normally be accessible as *c* is declared in a parallel block, i.e. also on level *2*, and *d* is declared in a block on level *3*.

Thus the use of a formal parameter can be treated as a temporary exit from the body of a procedure back to the position of a procedure call. In fact this is only necessary for certain types of parameters, and a simpler treatment of procedure parameters is often sufficient.

In the Whetstone Compiler procedure calls are translated without reference to the corresponding procedure declarations for details about the parameters. At run time the object program representations of the procedure call

and of the procedure declaration communicate via a set of formal accumulators. The formal accumulators of a procedure occupy space in the stack immediately above the link data for the procedure. At the call of a procedure the operations generated from the actual parameters ('actual operations') are used to set up these formal accumulators. Within the procedure body uses of formal parameters are represented by object program operations referring to the formal accumulators. Thus a single translation of a procedure body can be used by various different procedure calls.

Because no use is made of the information about parameters given in a procedure heading when translating a procedure call, checks cannot be made at translation time on the correspondence of formal and actual parameters. These checks have to be deferred until run time, and are carried out by special object program operations. These checking operations, which are used immediately after a procedure has been entered, i.e. after the operation *PE* (Procedure Entry), check the contents of the stacked formal accumulators. The formal accumulators, which have been set up by 'actual operations', are checked against the needs of a procedure declaration as given by its specification part.

It is at this point that the second important restriction which is imposed on ALGOL 60 by the Whetstone Compiler becomes apparent. This is the not uncommon restriction that the specification part of a procedure heading must give details of all the formal parameters. The subject of optional specifications is discussed further in Appendix 2.

2.5.1 Parameters Called by Name

It has already been shown that in certain circumstances a formal parameter implies the ability to return temporarily from within a procedure body to the conditions appertaining to the relevant call of the procedure. The full consequences of the 'call by name' concept of ALGOL 60 are brought into play by the use of an expression as an actual parameter corresponding to a formal parameter called by name.

Example

```
procedure CALC (a, b, c, i); real a, b, c; integer i;
begin i := 1; a := 0; b := 1;
loop: a := a + c;
      b := b × c;
      if i = 10 then go to finish;
      i := i + 1; go to loop;
finish: end;
```

This procedure can be used to form the continued sum and product of a sequence of ten expressions, using the procedure statement

CALC (sum, product, b×(b−j), j);

This is shown by application of the rules given in section 4.7.3 of the Revised ALGOL Report. This states that the effect of the procedure statement is equivalent to that of a suitably modified copy of the ALGOL statement which forms the procedure body. The modifications necessary in this case, (where all the parameters are called by name), are the replacement of the formal parameters, at each occurrence in the procedure body, by the corresponding actual parameters, having enclosed these actual parameters in parentheses wherever possible. In general (but this is not in fact necessary in this case) conflicts between identifiers inserted through this process, and other identifiers already present are to be avoided by systematic changes of the latter identifiers. Thus the action of the procedure statement is equivalent to that of the statement

```
begin j := 1; sum := 0; product := 1;
loop: sum := sum + (b × (b − j));
      product := product × (b × (b − j));
      if j = 10 then go to finish;
      j := j + 1; go to loop;
finish: end;
```

The result of this statement and hence of the procedure statement is to form

$$sum = \sum_{j=1}^{j=10} b \times (b - j)$$

and

$$product = \prod_{j=1}^{j=10} b \times (b - j)$$

Instead of implementing the actual process of copying out and modifying the procedure body, a single copy is made of the object program representation of the procedure body, in which the use of a formal parameter is, in general, represented by a call on a subroutine generated from an actual parameter. By this means an actual parameter expression will be evaluated each time its corresponding formal parameter is used (this is in fact necessary for the correct working of the procedure in the above example). This will automatically have the same effect as enclosing the expression in parentheses, since the evaluation of the expression will be completed before the subroutine is left, and will avoid all problems of clashes of identifiers. In fact in simpler cases of use of the 'call by name' facility a simpler technique can be used. Thus in the above example all that is necessary for the use of the actual parameters *sum*, *product* and *j*, is to give their corresponding formal parameters access to the stack location of these three variables.

This method of implementing parameters called by name must be designed to allow for recursive activation of a procedure, by means of an actual parameter.

Example

```
real procedure SUM (a, i); real a; integer i;
    begin real partial sum;
          partial sum := 0; i := 1;
    loop: partial sum := partial sum + a;
          if i = 10 then go to finish;
          i := i + 1; go to loop;
    finish: SUM := partial sum
    end;
```

This declaration defines a procedure *SUM* which can be activated by a function designator, for use in an expression. Such an expression could be used as an actual parameter to a procedure. For example, the statement

$$c := SUM\,(SUM\,(b\,[j, k], j), k);$$

will cause a double summation to be performed, setting

$$c = \sum_{k=1}^{k=10} \sum_{j=1}^{j=10} b\,[j, k]$$

Thus during the action of a procedure an evaluation of an actual parameter may cause a further activation of the procedure to take place.

In the object program this is accomplished using the normal stack mechanism. A use of a formal parameter is represented by a reference to a formal accumulator, and in such a case will cause a subroutine of object program operations to be obeyed. This subroutine will have been generated during the translation of the procedure call, and will be written in terms of the declarations valid at that point. Within the subroutine will be operations which cause the procedure to be activated once more, allocating space for its link data and working storage at the top of the stack. This will shield the information contained in the stack for the original call of the procedure.

At the end of the second activation of the procedure its link data and working storage will be removed from the stack, leaving the subroutine free to return to the original activation of the procedure.

2.5.2 Parameters Called by Value

The calling of parameters by value is a very much simpler concept than call by name, and essentially simply involves evaluating an expression and assigning the result to the formal parameter, which is thereafter treated as if it were a local variable of the procedure body. The only exception to this is a formal parameter with a specifier **label** – see section 2.5.6.3.

However, during the translation of a procedure call, it is not known which, if any, of its parameters are called by value. Thus all actual parameters are treated as if they correspond to formal parameters not appearing in the value

list of the procedure declaration. Then, at run time, immediately after a procedure has been activated, each parameter which is called by value is evaluated and the result placed in the appropriate formal accumulator. Thereafter the formal accumulator is treated as if it were part of first order working storage, and its second word, which contained an identifying bit pattern, is ignored.

This task of evaluating any parameters called by value is combined with the task of checking formal-actual correspondence. A set of 'parameter list operations' is generated immediately after the *PE* operation which starts a procedure. The set contains one operation for each parameter. Each operation checks the corresponding formal accumulator and, where appropriate, replaces the accumulator with the value of the corresponding actual parameter expression.

Some details of a similar method of implementing calls by name and by value are contained in a paper by Ingerman [37], and a description of a system capable of handling all non-recursive uses of procedures has been given by Jensen and Naur [43].

2.5.3 Procedure Calls

The operation *CF* (Call Function) is used by both procedure statements and function designators to start the activation of a procedure. (The simple case of a procedure with no parameters, which uses the operation *CFZ*, need not be considered further.) *CF* forms the first part of the link data of the procedure, and then sets up the formal accumulators in the stack, before jumping to the operation *PE*. At the end of the activation of the procedure a return is made to the operation following *CF*.

The formal accumulators are set up using the 'actual operations'. These 'actual operations' are thus not obeyed directly by the Control Routine but are processed by the *CF* operation. The 'actual operations' are positioned in the object program immediately preceding the *CF* operation, and are given in reverse order to that of the list of actual parameters. The number of 'actual operations' is given by the parameter *m* of the *CF* operation. *CF* works backwards through the object program, processing each 'actual operation' in turn.

All 'actual operations' occupy *4* syllables. In the case of an actual parameter which is just an identifier an 'actual operation' is complete in itself – otherwise it gives the address of the object program representation of the actual parameter. The operations generated from the actual parameter list are preceded by an *UJ* operation. This operation causes a jump to be made around the 'actual operations' to the *CF* operation.

Thus a procedure call is translated into

```
      UJ      (a)
      'Actual Operations'
a:    CF      (b), m
```

Example

$P(x, y, i, j)$

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	*(19)*	Jump to Call Function
3	*PI*	*j*	'Actual operations'
7	*PI*	*i*	
11	*PR*	*y*	
15	*PR*	*x*	
19	*CF*	*P, 4*	
23	...		

In this example two 'actual operations' are introduced – *PR* (Parameter Real), and *PI* (Parameter Integer), which will be described in detail later. The *UJ* operation at syllable *0* causes a jump to *CF* at syllable *19*. This processes the four 'actual operations', at syllables *15*, *11*, *7* and *3* in that order, and then jumps to the procedure *P*, having stacked *23* as a return address.

2.5.4 Actual Parameters

The implementation of the various kinds of actual parameter is now described in detail. In each case a description is given of

(*i*) The object program representation of the actual parameter,

(*ii*) The setting-up of the corresponding formal accumulator.

Certain formal parameters can have various corresponding types of actual parameters. For instance, a formal parameter specified to be real could correspond to an actual parameter which is a simple or subscripted variable, a constant, a function designator, or, in the general case, an arithmetic expression. A discussion of the method by which calls on formal parameters are implemented will be deferred until the various kinds of actual parameters have been described.

2.5.4.1 Simple Variables

Within a procedure body a formal parameter having a simple variable as its corresponding actual parameter can be used on the left hand side of an assignment statement, as well as within an expression. Thus the address, rather than just the value, of the simple variable is needed within the procedure body.

The only complication is that a simple variable, which by virtue of the block structure would otherwise be inaccessible to the procedure body, can be used by means of a formal parameter.

Such an occurrence is reflected in the object program by the fact that the

static chain, and hence *DISPLAY*, are no longer set up in the appropriate fashion to enable the address of the variable to be found.

Example

```
begin procedure P (a); real a;
                    a := a-1;
      begin real b;
                    P (b)
      end
end
```

In this example both *a* and *b* are at level *2*. At the call of procedure *P*, *DISPLAY* [*2*] will be set so as to enable the location of *b* to be found, but inside the procedure it will enable the formal accumulator corresponding to *a* to be found.

The method chosen to solve this problem is straightforward. The formal accumulator corresponding to such an actual parameter is set up with the stack address of the simple variable, evaluated from the dynamic address given with the 'actual operation' before entering the procedure. Then, within the procedure, *DISPLAY* can be changed without affecting the accessibility of the actual parameters. (An obviously inefficient alternative system is to keep the dynamic address in the formal accumulator and to reset *DISPLAY* temporarily each time the accumulator is used.)

The 'actual operations' corresponding to the various types of simple variables are

PR (Parameter Real)
PI (Parameter Integer)
PB (Parameter Boolean)

Each of these has a two-syllable parameter (*n*,*p*) – the dynamic address of the simple variable. *CF* processes such operations, and sets up a formal accumulator with '*DISPLAY* [*n*] + *p*' and an identifying bit pattern. The bit pattern indicates the type of the variable, and the fact that the accumulator contains an address.

All 'actual operations' occupy four syllables (the reason for this is described in section 3.4.3.2.1), and in the above cases the fourth syllable is unused and is left blank.

Example

P (*a*, *i*, *b*)

where the address of *a* (real) is (*1*,*4*), *i* (integer) is (*2*,*6*), *b* (Boolean) is (*1*,*6*), and the address of *PE* operation of procedure *P* is *54*. This is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*15*)	
3	*PB*	(*1*,*6*), –	*b*
7	*PI*	(*2*,*6*), –	*i*
11	*PR*	(*1*,*4*), –	*a*
15	*CF*	(*54*), *3*	
19	...		

2.5.4.2 Constants

An actual parameter which is a constant can only be used in an expression, so the corresponding formal accumulator is simply set up with the value and type of the constant.

The 48-bit constant is stored in the object program and the 'actual operation' gives its syllable address. Such constants are placed in the program between the operation *UJ* (which causes a jump to the operation *CF*) and the set of 'actual operations'.

Example

P (*5.0*, *a*, **true**, *1*)

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*37*)	
3	'*5.0*'		
9	'**true**'		
15	'*1*'		
21	*PIC*	(*15*), –	(Parameter Integer Constant)
25	*PBC*	(*9*), –	(Parameter Boolean Constant)
29	*PR*	(*1*,*4*), –	*a*
33	*PRC*	(*3*), –	(Parameter Real Constant)
37	*CF*	(*54*), *4*	*P*
41	...		

P and *a* have been given addresses as in the previous example. The *UJ* operation causes a jump to be made to the *CF* operation at syllable *37*, thus avoiding the constants and the 'actual operations'. *CF* works back through these 'actual operations', where necessary using their parameters to find the 48-bit representations of the constants.

It has not been thought worthwhile to make special provision for the constants *0*, *1*, **true**, and **false**, when used as actual parameters.

2.5.4.3 Expressions

It has already been stated that in the general case of an actual parameter which is an expression, a subroutine of object program operations is generated for evaluating the expression. The corresponding 'actual operation' is

PSR (Parameter Sub-Routine), which gives the address of the start of the subroutine. Such subroutines are positioned, with any constants, between the operation *UJ* and the first 'actual operation'.

However such a subroutine can itself be used recursively.

Example

```
real procedure Q (n); integer n;
    begin . . .
          P(A + B × Q (n − 1));
          . . .
    end;
```

In this example a call on procedure *Q* will cause a call on procedure *P*, which has an expression as actual parameter. In this expression is a further call on *Q*. This again calls *P*, which again starts to evaluate the expression, hence using the expression recursively. It is assumed that the other statements, indicated by dots, give some means of ending this continued nesting of calls on procedure *Q*.

Because of this possibility the subroutine of operations generated from an actual parameter expression is made into a block. This 'subroutine block' stacks link data in the usual way, and hence can use the normal mechanism for recursive blocks and procedures. A result accumulator is reserved in the stack for the result of the subroutine. The subroutine starts with a *BE* (Block Entry) operation. This has as parameters (*n*) the block level and *0* – as there is no first order working storage. The subroutine ends with the operation *EIS* (End Implicit Subroutine). This operation has already been mentioned in connection with switch declarations. Its action is to take the top accumulator (the value of the expression) and store it in the result accumulator of the subroutine block, before performing the action of the operation *RETURN*.

On calling a procedure with such a parameter the *CF* operation processes the 'actual operation' and sets up a formal accumulator with the address of the subroutine, the value of *PP* ruling at the time of the procedure call, and an identifying bit pattern. The value of *PP* can be used, when this formal accumulator is called from within the procedure body, in order to reset conditions temporarily to what they were at the procedure call. This allows the subroutine, which was generated from an expression written subject to declarations valid at the procedure call, to perform its work correctly.

Example

$$P(-A, 5, B + C[I])$$

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	*(44)*	Jump to Call Function
3	*BE*	*(3,0)*	Subroutine to evaluate '$-A$'
6	*TRR*	*A*	
9	*NEG*		
10	*EIS*		
11	'*5*'		
17	*BE*	*(3,0)*	Subroutine to evaluate '$B+C[I]$'
20	*TRR*	*B*	
23	*TRA*	*C*	
26	*TIR*	*I*	
29	*INDR*		
30	+		
31	*EIS*		
32	*PSR*	*(17), −*	
36	*PIC*	*(11), −*	
40	*PSR*	*(3), −*	
44	*CF*	*P, 3*	
48	...		

The level of the block in which this procedure call occurs has been taken to be two, and hence each implicit subroutine starts with '*BE (3,0)*'.

In certain rather infrequent circumstances during the generation of the object program it is not possible for the Translator to detect whether an actual parameter expression is a designational expression or not. Unless this is known, it cannot be decided whether operations generated from the identifiers in the expression will be 'Take Label' operations whose parameters occupy three syllables or 'Take Result' operations whose parameters occupy two syllables. In cases of doubt a four-syllable space is left free, to be filled in later either with a 'Take Label' operation or with a 'Take Result' operation followed by an operation *DUMMY*. At run time these *DUMMY* operations have no effect, other than to increase the program counter by one syllable. (A more detailed account of the problem of translating actual parameter expressions is given in section 3.4.3.2.1.3.4.)

2.5.4.4 Arrays

Actual parameters which are arrays use the 'actual operations'

PRA (Parameter Real Array)
PIA (Parameter Integer Array)
PBA (Parameter Boolean Array)

These are very similar to the operations *PR*, *PI*, and *PB*, and have the dynamic address of an array word as their parameter. The corresponding formal accumulators are set up with the evaluated address of the array word, and a bit pattern, indicating 'real array address', say. Thus the array word,

and hence the array elements, can be used inside the procedure body by means of a formal parameter, even if the array declaration becomes inaccessible.

2.5.4.5 Subscripted Variables

Actual parameters that are subscripted variables can correspond to formal parameters that are used in an expression or on the left hand side of an assignment statement. Just as an actual parameter which was an expression had to be evaluated at each use of the corresponding formal parameter, so the address of a subscripted variable must be repeatedly evaluated.

Example

```
procedure S (a, i); real a; integer i;
begin  i := 0;
       a := 0;
       i := 1;
       a := 0
end;
S (A [j], j);
```

The consequences of the call on this apparently trivial procedure are shown by writing out the procedure body with the actual parameters substituted for the formal parameters. The formal parameter corresponding to '*A* [*j*]' is used twice. At the first use '*A* [*0*]' is zeroed, at the second use '*A* [*1*]'.

Thus an actual parameter which is a subscripted variable is translated into an implicit subroutine for evaluating the address of the subscripted variable, and an 'actual operation' *PSR*. The parameter of the *PSR* operation gives the program address of the start of the implicit subroutine.

Example

S (*A* [*j*], *j*)

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*22*)	
3	*BE*	(*n*,*0*)	
6	*TRA*	*A*	
9	*TIR*	*j*	Implicit Subroutine for *A* [*j*]
12	*INDA*		
13	*EIS*		
14	*PI*	*j*, –	
18	*PSR*	(*3*), –	
22	*CF*	*S*, *2*	
26	...		

The corresponding formal accumulator is set up in the same way as for an actual parameter which is an expression.

2.5.4.6 Labels

An actual parameter which is a label is translated into an 'actual operation' *PL* (Parameter Label). The parameters of the operation *PL* are similar to those of a *TL* (Take Label) operation – i.e. *a* (a two-syllable program address) and *n* (a one-syllable block number).

The *PL* operation and its parameters thus occupy all four syllables allotted to 'actual operations'. (It is in fact the only 'actual operation' which needs more than three syllables, but it is convenient for the Translator to standardize on four-syllable 'actual operations'.)

Example

Q (*1.0*, *L*)

is translated into

Syll	*Op*	*Par*
0	*UJ*	(*17*)
3	'*1.0*'	
9	*PL*	(*a*), *n*
13	*PRC*	(*3*), –
17	*CF*	*Q*, *2*
21	...	

The corresponding formal accumulator is set up by the *CF* operation with an identifying bit pattern and the evaluated label, i.e.

DISPLAY [*n*], *a*

2.5.4.7 Switches

An actual parameter which is a switch is simply translated into an 'actual operation' *PSW* (Parameter Switch), whose parameter gives the program address of the *BE* operation of the switch declaration. The corresponding formal parameter is set up with this program address, the value of *PP* ruling at the time of the procedure call, and an identifying bit pattern. The value of *PP* is needed to reset conditions temporarily to what they were at the procedure call whilst the switch is being used from inside the procedure body.

2.5.4.8 Procedures

Actual parameters which are procedures use the appropriate 'actual operation' from the list

PPR	(Parameter Procedure)
PFR	(Parameter Function Real)
PFI	(Parameter Function Integer)

PFB (Parameter Function Boolean)

These are very similar to *PSW* (Parameter Switch), having a program address as a parameter, and setting up a formal accumulator with this program address, *PP*, and a bit pattern.

2.5.4.9 Strings

In the case of an actual parameter which is a string, a coded form of the sequence of ALGOL basic symbols that constitute the string is stored in the object program. (The code, which uses 8 bits for each symbol, is the same as that used in the Kidsgrove Compiler.) The corresponding 'actual operation' is *PST* (Parameter String) which has a parameter giving the address of the first symbol of the string (i.e. the opening string quote). This 'actual operation' causes the corresponding formal accumulator to be set up with this address.

For the convenience of writers of machine code procedures (an ordinary procedure cannot use a string, other than as an actual parameter to an inner procedure) the first opening string quote is stored at the start of a word. Thus there may be a gap between the last object program operation and the syllable containing the opening string quote. Similarly there may be a gap after the closing string quote, so that the next object program operation can be placed at the start of the next word.

Example

P ('*rope*')

is translated into

Syll	*Op*	*Par*
0	*UJ*	(*16*)
6	‘	
7	*r*	
8	*o*	
9	*p*	
10	*e*	
11	’	
12	*PST*	(*6*), –
16	*CF*	*P*, *1*
20	...	

2.5.4.10 Formal Parameters

It is possible to use a formal parameter as an actual parameter in a further procedure call.

Example

```
real procedure P (a); real a;
        begin real procedure Q (b); real b; Q := b - 3;
              ...
              P := Q (a);
              ...
        end;
c := P (c + 3);
```

Thus in this example a use of *b* in the body of procedure *Q* involves a use of the actual parameter *a*, which was written in terms of conditions holding at the position of the call of *Q*. However, this actual parameter *a* is itself a formal parameter of *P*, and thus is being used as a means of using the corresponding actual parameter '*c* + *3*'. Thus a use of *a* is essentially a call of '*c* + *3*', which is written in terms of conditions holding at the position of the call of *P*.

It can be seen that handing on a formal parameter as an actual parameter essentially needs some means of handing on the ability to reach the original actual parameters. This is done simply by handing on the contents of the formal accumulator.

An actual parameter which is itself a formal parameter uses the 'actual operation' *PF* (Parameter Formal). This has as its parameter the dynamic address (*n*,*p*), of the formal accumulator to be handed on. In such a case *CF* sets up the new formal accumulator by evaluating the dynamic address, and copying the formal accumulator given at this address into the new formal accumulator space.

Example

SUM (*m*)

where *m* is the second formal parameter of a procedure which has been given the level two, is translated into

Syll	*Op*	*Par*
0	*UJ*	*(7)*
3	*PF*	*(2,5), –*
7	*CF*	*SUM, 1*
11	...	

2.5.5 Formal Parameters

In general, formal parameters which are called by value are evaluated immediately after entry to a procedure, and are thereafter treated as local variables. This will be described in section 2.5.6. The various ways in which formal accumulators can be set up have been given in the preceding sections, and it is now necessary to describe how these formal accumulators are used.

2.5.5.1 Real and Integer

A formal parameter specified to be arithmetic (real or integer) can have various corresponding actual parameters. If it is used on the left hand side of an assignment statement it must correspond to a simple or subscripted variable. However, if it is only used in expressions the corresponding actual parameter could also be a constant, an expression, or the name of a type procedure (if this type procedure has no parameters – see section 4.7.5.4 of the Revised ALGOL Report).

2.5.5.1.1 Assignments to Arithmetic Formal Parameters. In the Whetstone Compiler, if an arithmetic formal parameter is used on the left hand side of an assignment statement its corresponding actual parameter must be of the type given in the specification. Thus real-integer conversions are not allowed in assignments to a formal parameter.

Thus the action of the operation generated for an assignment to a formal parameter specified to be real or integer is to use the formal accumulator to obtain the address of the corresponding actual parameter and to check its type. The formal accumulator will contain the type and evaluated address of the actual parameter already, or will have been set up from the operation *PSR* (Parameter Sub-Routine). In the second case the formal accumulator will contain the address of the start of the subroutine, and the value of *PP* ruling at the call of the procedure.

The object program operations generated for taking the address of an arithmetic formal variable are

TFAR	(Take Formal Address Real)
TFAI	(Take Formal Address Integer)

These operations each have the dynamic address of the appropriate formal accumulator as a parameter.

Example

$x := y := z + 3;$

where y is a formal parameter, is translated into

Syll	*Op*	*Par*
0	*TRA*	*x*
3	*TFAR*	*y*
6	*TRR*	*z*
9	*TIC*	*'3'*
16	+	
17	*STA*	
18	*ST*	
19	...	

It has already been mentioned that an implicit subroutine must be made into a block so that it can be used recursively if necessary. Therefore to evaluate an implicit subroutine, the value of *PP* contained in the formal accumulator is used to update *DISPLAY* so that the declarations valid at the call of the procedure can be used by the subroutine. This is done after various items of link data have been placed at the top of the stack, above a space left for the accumulator which will be set up by the subroutine. A jump is then made to the *BE* operation at the start of the subroutine, which completes the link data. The effect of the operations forming the subroutine is to produce an address in the top accumulator, above the link data of the subroutine. The final operation *EIS* (End Implicit Subroutine) moves this accumulator to the space below the link data, and then performs the work of *RETURN*. This uses the link data of the subroutine to reset *PP*, *AP* and *DISPLAY* to the values they had before entry to the subroutine. Thus the net effect of the subroutine, even if it involved further calls on blocks or procedures, is just to stack one accumulator.

Example

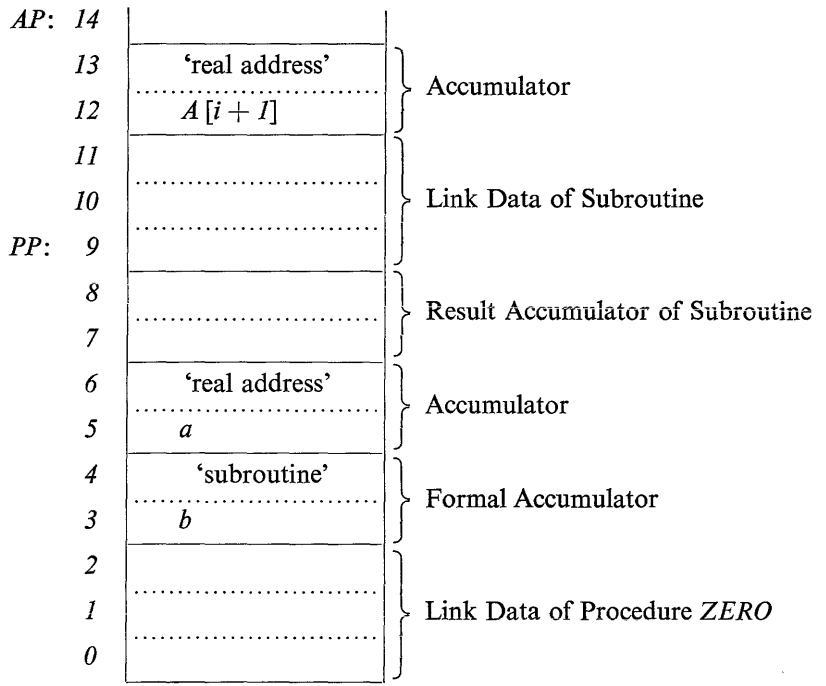
procedure *ZERO* (*b*); **real** *b*; *a*:= *b*:= *0*;
ZERO (*A* [i + *1*]);

The assignment statement in the body of procedure *ZERO* is translated into

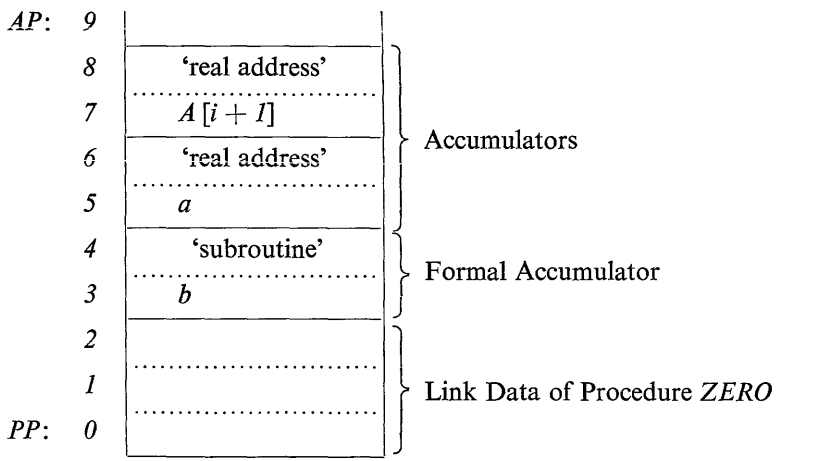
Syll	*Op*	*Par*
0	*TRA*	*a*
3	*TFAR*	*b*
6	*TIC0*	
7	*STA*	
8	*ST*	
9	...	

and the call on procedure *ZERO* is translated into

Syll	*Op*	*Par*
0	*UJ*	(*20*)
3	*BE*	(*n*,*0*)
6	*TRA*	*A*
9	*TIR*	*i*
12	*TIC1*	
13	+	
14	*INDA*	
15	*EIS*	
16	*PSR*	(*3*), −
20	*CF*	*ZERO*, *1*
24	...	



(a). After operation *INDA* in Subroutine.



(b). After operation *TFAR*.

FIG. 7. The Use of the Stack for Implicit Subroutines.

In Fig. 7(a) is shown the stack after the operation *INDA* in the implicit subroutine, which has been activated by the operation *TFAR*. An accumulator space has been left above the accumulator containing the address of *a*. *TFAR*, on examining the formal accumulator *b*, has found a value of *PP* (that which was current before the entry to procedure *ZERO*), and the address of the operation *BE*. The subroutine has been entered, and has evaluated the address of '*A* [*i* + *1*]' which is in the top accumulator.

In Fig. 7(b) the stack is shown after *EIS* has caused the accumulator containing the address of '*A* [*i* + *1*]' to be moved down into the result accumulator, and has deleted the link data of the subroutine. *PP* has been reset to indicate the start of the link data of procedure *ZERO*.

A call of an implicit subroutine differs from a call of a procedure mainly in that *DISPLAY* must be updated before entering the subroutine, as well as on leaving the subroutine. This is because calls of a subroutine, as distinct from a procedure, can occur at a position where the declarations subject to which it was written are no longer valid. A description of the technique of updating *DISPLAY* has been given in section 2.2.4.1.

After the call of the subroutine by the operation *TFAR* or *TFAI* has been completed it is necessary to check that the resulting accumulator contains an address, and that the address is of the specified type. Thus the subroutine of object program operations is being called from, and is returning to, the machine coding of the operation *TFAR* or *TFAI*. Since such a subroutine could be called from various other 'Take Formal' operations (which have yet to be described) it is necessary to have a double link system. The first link has already been met in the description of blocks and procedures, and is a value of the program counter. This gives the syllable address of the object program operation after the one that called the block or procedure. The second link is needed when return is not to be made directly to object program level. This is therefore the normal kind of machine code link giving the address of the machine code instruction within the operation *TFAR* or *TFAI* to which a return must be made.

In certain cases a procedure can be called from inside an operation (see section 2.5.5.1.2) and thus both *RETURN* and *EIS* use the double link system. Thus each resets the program counter from the 'object program link', and then obeys the machine code instruction at the address given in the 'machine code link'. In simple cases, such as calls on blocks and procedures by the operations *CBL* and *CF* the machine code link will be the address of the start of the main control loop. Thus the control loop will immediately obey the next object program operation as given by the reset program counter. In other cases this system allows the work of an operation, which called a subroutine or a procedure, to be completed before the next operation is obeyed.

This double link technique could be avoided if operations such as *TFAR*

were split into two operations, say *TFA* (Take Formal Address), and *CAR* Check Address Real). However this is not convenient for the system of translation employed in the Whetstone Compiler.

2.5.5.1.2 Use of Arithmetic Formal Parameters in Expressions. The operations generated from the use of an arithmetic formal parameter in an expression are

TFR (Take Formal Real)
TFI (Take Formal Integer)

These operations have the dynamic address of the appropriate formal accumulator as a parameter.

Example

$x := y + A[i];$

where y and i are formal, is translated into

Syll	*Op*	*Par*
0	*TRA*	*x*
3	*TFR*	*y*
6	*TRA*	*A*
9	*TFI*	*i*
12	*INDR*	
13	+	
14	*ST*	
15	...	

The action of *TFR* or *TFI* is dependent on the bit pattern found in the appropriate formal accumulator, which can have been set up by the 'actual operations'

PR, PI, PRC, PIC, PFR, PFI, PSR

(*i*) *Formal accumulator set up by PR or PI.* This happens when the corresponding actual parameter is a simple variable. The value of the simple variable, taken from the evaluated address in the formal accumulator, is stacked, having performed any necessary real-integer conversion to the type specified in the operation *TFR* or *TFI.*

(*ii*) *Formal accumulator set up by PRC or PIC.* This case is even simpler, as the value of the actual parameter (a constant) has been placed in the formal accumulator. Once again this value is stacked, having performed any necessary real-integer conversions.

(*iii*) *Formal accumulator set up by PFR or PFI.* In this case the formal accumulator will correspond to an actual parameter which is the identifier of a type procedure with no parameters. The formal accumulator will contain the address of the *PE* operation of the procedure and the value of *PP* ruling at the point where the actual parameter was given.

DISPLAY is reset, using this value of *PP*, so that the procedure will be

called in conditions which allow it to use any non-local variables that were valid at the point of its use as actual parameter. Then the procedure is activated in the normal way, stacking a result accumulator, link data, etc. At the end of the procedure activation, i.e. at the operation *RETURN*, the link data and working storage are deleted, leaving the result accumulator as the top accumulator. The double link system described in section 2.5.5.1.1 allows the operation *TFR* or *TFI* to resume its working. In fact all that is necessary is to check that the top accumulator does contain an arithmetic value, and to do any real-integer conversions.

(*iv*) *Formal accumulator set up by PSR.* *PSR* will have been used where the actual parameter is a subscripted variable or the more general form of an expression. The way in which such an implicit subroutine is activated has been described in section 2.5.5.1.1, in connection with the operation *TFAR* and *TFAI*. Thus a return will eventually be made to the operation *TFR* or *TFI*, with the top accumulator containing the result of the subroutine. If the subroutine has been generated from a subscripted variable then the accumulator will contain an address, otherwise it will contain a value. Therefore the final actions of *TFR* and *TFI* are to check the top accumulator, fetch the appropriate value if the accumulator contains an address, and finally to perform any necessary real-integer conversions.

2.5.5.2 Boolean

The implementation of calls on formal parameters specified to be Boolean is exactly the same as for real or integer formal parameters, except that no type conversions are needed.

If a Boolean formal parameter is used on the left hand side of an assignment statement the operation

TFA (Take Formal Address)

is generated.

Otherwise the operation

TFB (Take Formal Boolean)

is used. Each of these operations has the dynamic address of the appropriate formal accumulator as a parameter.

Example

$b := c \wedge d;$

where b and d are formal, is translated into

Syll	*Op*	*Par*
0	*TFA*	*b*
3	*TBR*	*c*
6	*TFB*	*d*
9	$\wedge$	
10	*ST*	
11	...	

It will be seen that assignments to Boolean formal parameters use the general operation *TFA*, rather than a special operation 'Take Formal Address Boolean'. *TFAR* and *TFAI* were needed so as to check that assignments to arithmetic formal parameters do not call for real-integer conversions; in the case of a formal Boolean, the address produced by the operation *TFA* will be adequately checked at the operation *ST*.

2.5.5.3 Array

A formal parameter with an array specification must have a corresponding actual parameter whose declaration agrees exactly with this specification. Thus no real-integer conversions are allowed in the case of arithmetic arrays. This correspondence will have been checked by a special operation generated at the start of the procedure, from the parameter list (see section 2.5.6).

All uses of formal arrays use the operation *TFA* (Take Formal Address), which in this case just stacks the address of the array word given in the appropriate formal accumulator.

Example

$A[i] := A[j] + 1;$
$b[i] := b[j] \wedge b[i];$

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*TFA*	*A*	
3	*TIR*	*i*	
6	*INDA*		$A[i]$
7	*TFA*	*A*	
10	*TIR*	*j*	
13	*INDR*		$A[j]$
14	*TIC1*		
15	+		
16	*ST*		$A[i] := A[j] + 1$
17	*TFA*	*b*	
20	*TIR*	*i*	
23	*INDA*		$b[i]$
24	*TFA*	*b*	
27	*TIR*	*j*	
30	*INDR*		$b[j]$
31	*TFA*	*b*	
34	*TIR*	*i*	
37	*INDR*		$b[i]$
38	$\wedge$		
39	*ST*		$b[i] := b[j] \wedge b[i]$
40	...		

2.5.5.4 Label

The use of a formal parameter specified to be a label is represented by the operation

TFL (Take Formal Label)

The action of this operation is dependent on the contents of the formal accumulator given by its dynamic address parameter, which can have been set up by *PL* or *PSR*.

In the first case the next accumulator is set up with the contents of the formal accumulator, (a value of *PP*, and a program address, which is the usual object program representation of a stacked label).

In the second case the implicit subroutine is activated, as described in section 2.5.5.1.1. At the end of the activation of the subroutine a return is made to the operation *TFL* which checks that the resulting top accumulator does contain a label.

Example

go to if *b* **then** *L1* **else** *L2*;

where *L1* and *L2* are formal parameters, is translated into

Syll	*Op*	*Par*
0	*TBR*	*b*
3	*IFJ*	(*12*)
6	*TFL*	*L1*
9	*UJ*	(*15*)
12	*TFL*	*L2*
15	*GTA*	
16	...	

2.5.5.5 Switch

The use of a formal switch is represented by the operation *TFA*, and is treated like a use of a formal array by a subscripted variable.

Example

go to *S* [*I*];

is translated into

Syll	*Op*	*Par*
0	*TFA*	*S*
3	*TIC1*	
4	*INDR*	
5	*GTA*	
6	...	

The action of *TFA* is to stack the formal parameter, which in this case contains a value of *PP* and a program address. The indexing operation works back down the stack until it finds an address, in this case identified as being that of a 'formal switch'. The action of *INDR* or *INDA* is then as described for ordinary switch designators, except that *DISPLAY* must be updated before the 'switch block' is activated. This resetting of *DISPLAY* is done using the value of *PP* stacked by the operation *TFA*. By this means conditions are temporarily reset to what they were at the use of the switch as an actual parameter.

2.5.5.6 Procedure

All calls on a formal parameter specified to be a procedure or a type procedure, by procedure statements or function designators, use the operations

CFF (Call Formal Function)
CFFZ (Call Formal Function Zero)

Both operations have the dynamic address of the appropriate formal accumulator as a parameter. In addition *CFF* has a one-syllable parameter giving the number of parameters of the procedure statement or function designator that it represents. (*CFFZ* is used for a procedure call with no parameters.)

The operations *CFF* and *CFFZ* are very similar to the operations *CF* and *CFZ* and perform preliminary setting up of the link data of a procedure before a jump is made to the appropriate *PE* operation. However, the address of this *PE* operation is given in the formal accumulator, rather than as a parameter, and before the jump is made *DISPLAY* is reset, using the value of *PP* also given in the formal accumulator. In the case of a procedure being called by means of a procedure statement the operation *REJECT* is used to delete an unwanted result accumulator.

It will be seen that the action of *CFFZ* is exactly the same as that of *TFR*, *TFI* or *TFB* in the case of the corresponding formal parameter being set up by *PFR*, *PFI* or *PFB* respectively. Thus an actual parameter, which is the identifier of a type procedure with no parameters, can be translated without inspecting the specification of the corresponding formal parameter to find whether it is used as a procedure or as an expression.

2.5.6 Parameter List Operations

A set of 'parameter list operations' is placed immediately after the *PE* operation which starts a procedure body. These operations check that the setting of the formal accumulators is in accordance with the specification of each formal parameter. In addition, if a parameter is called by value, then its corresponding parameter list operation evaluates the actual parameter, and replaces the formal accumulator by this value.

The parameter list operations are given in the order of the formal parameters, and use *FP*, the formal pointer (kept in the stacked link data) to address the appropriate formal accumulator. Each parameter list operation

checks the formal accumulator given by *FP* and then increases *FP* so that it indicates the next formal accumulator.

In the case of formal parameters specified to be real, integer, Boolean or label, the checking cannot be complete, since the actual parameter might be an expression, which will have a simple *PSR* (Parameter Sub-Routine) as its 'actual operation'. In such cases it is the checking built into operations like *TFAR* (Take Formal Address Real), *TFB* (Take Formal Boolean) or *ST* (Store), which will detect errors. However, it has been thought worthwhile to perform as much checking as possible on entry to a procedure, rather than leave everything until a formal parameter is used.

The operations which are used for evaluating parameters called by value are essentially similar to the various 'Take Formal Result' operations (e.g. *TFR*, *TFL*), but use *FP* rather than an explicit (*n*,*p*) parameter, and place their result in a formal accumulator rather than on top of the stack. In the case of real, integer or Boolean parameters the single word occupied by this result is thereafter treated as an ordinary working store, and the second word of the formal accumulator (containing a bit pattern) is ignored. Label parameters called by value are treated somewhat differently (see section 2.5.6.3).

The parameter list operations for formal parameters specified to be arrays (called by name), switches, procedures and strings are quite straight forward, and perform a complete check on the setting of the formal accumulators.

The operations are

CAR	(Check Array Real)
CAI	(Check Array Integer)
CAB	(Check Array Boolean)
CSW	(Check Switch)
CPR	(Check Procedure)
CFR	(Check Function Real)
CFI	(Check Function Integer)
CFB	(Check Function Boolean)
CST	(Check String)

Example

procedure *P* (*a*, *b*, *c*, *d*); **array** *c*; **switch** *a*; **procedure** *b*;
real procedure *d*; ;

This procedure, with a body which is a dummy statement, is translated into

Syll	*Op*	*Par*
0	*PE*	(*3*,*8*), *4*
4	*CSW*	
5	*CPR*	
6	*CAR*	
7	*CFR*	
8	*RETURN*	
9	...	

In this, and succeeding examples, the level of the procedure is taken to be *3*. The parameter *L* of *PE* is *8* − made up of *4* double-word formal accumulators.

2.5.6.1 Real or Integer Formal Parameters

The check operation for real or integer formal parameters is *CA* (Check Arithmetic). This checks that the formal accumulator has been set up by an implicit subroutine, an arithmetic constant or simple variable or an arithmetic procedure with no parameters. It is not possible to do a complete check regarding real and integer, since type conversions are allowed for calls by value, and on use of formal parameters in expressions. The checking-out of type conversions in assignments to formal parameters called by name is done by the operations *TFAR* and *TFAI*.

In the case of calls by value, the operations *CSR* (Check and Store Real) and *CSI* (Check and Store Integer) are used. These operations correspond to the operations *TFR* and *TFI*.

Example

procedure *SUM* (*a, i, j*); **value** *i*; **real** *a*; **integer** *i, j*; *a* := *i*;

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*PE*	(*3,6*), *3*	
4	*CA*		*a*
5	*CSI*		*i*
6	*CA*		*j*
7	*TFAR*	(*3,3*)	
10	*TIR*	(*3,5*)	
13	*ST*		*a* := *i*
14	*RETURN*		
15	...		

It is because an operation such as *CSI* can cause an implicit subroutine to be activated that *FP* (the Formal Pointer) is kept in the stack. This is because within such a subroutine there might be a call on a function designator, for instance, which would need its own *FP*.

2.5.6.2 Boolean Formal Parameters

The check operation for Boolean formal parameters is *CB* (Check Boolean). This checks that the formal accumulator has been set up by an implicit subroutine, a Boolean constant or simple variable or a Boolean procedure with no parameters. The final checking of an implicit subroutine which delivers the address of a subscripted Boolean variable (having been called by *TFA*), is done at the operation *ST* or *STA* when the address is used. If the implicit

subroutine is called in an expression it is checked by the operation *TFB* (Take Formal Boolean).

Calls by value on Boolean parameters use the operation *CSB* (Check and Store Boolean), which corresponds to *TFB*.

Example

```
procedure B (b1, b2); value b1; Boolean b1, b2;
        b2 := b1 ∧ b2;
```

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*PE*	*(3,4), 2*	
4	*CSB*		*b1*
5	*CB*		*b2*
6	*TFA*	*(3,5)*	
9	*TBR*	*(3,3)*	
12	*TFB*	*(3,5)*	
15	∧		
16	*ST*		*b2 := b1 ∧ b2*
17	*RETURN*		
18	...		

2.5.6.3 Label Formal Parameters

The check operation generated in the case of a label called by name is *CL* (Check Label). This operation checks that the formal accumulator has been set up by an implicit subroutine or a label. If it has been set up by an implicit subroutine then the final checking will be carried out by the operation *TFL* (Take Formal Label).

Labels called by value are not amenable to the treatment given to, say, real parameters called by value. This is because an ordinary label need not occupy space in the working storage of a block, as assignments cannot be made to labels (see section 4.2.1 of the Revised ALGOL Report). The information generated from a label is given with the operation *TL* (Take Label) and only appears in the stack in an accumulator. Thus *TL* is like *TRC* (Take Real Constant), rather than *TRR* (Take Real Result).

The solution is to use *TFL* (Take Formal Label) in a procedure body, even for labels called by value, but to evaluate a designational expression actual parameter on entry to the procedure. This is done by the operation *CSL* (Check and Store Label). The result of *CSL* is that the formal accumulator contains a label (as if set up by *PL* (Parameter Label)). Since the accumulator is not to be used as a simple working store, the bit pattern must be retained.

Example

```
procedure JUMP (b, L1, L2); value b, L1; Boolean b;
        label L1, L2;
        go to (if b then L1 else L2);
```

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*PE*	*(3,6), 3*	
4	*CSB*		
5	*CSL*		
6	*CL*		
7	*TBR*	*(3,3)*	
10	*IFJ*	*(19)*	
13	*TFL*	*(3,5)*	*L1*
16	*UJ*	*(22)*	
19	*TFL*	*(3,7)*	*L2*
22	*GTA*		
23	*RETURN*		
24	...		

2.5.6.4 Arrays called by Value

If a formal parameter specified to be an array is called by value then the elements of the array given as an actual parameter are copied into the second order working storage of the procedure. The contents of the formal accumulator are replaced by a suitably modified array word, which can be used to address the storage mapping function and the new set of array elements. Within the procedure the array so formed is treated as if local to the procedure body.

This copying is performed by one of the parameter list operations

CRFA (Copy Real Formal Array)
CIFA (Copy Integer Formal Array)
CBFA (Copy Boolean Formal Array)

Each operation checks the setting of the formal accumulator, which should give the stack address of the array word of the actual parameter array. This array word is used to locate the storage mapping function and elements of the array. The storage mapping function gives the number of elements of the array. These elements are copied to the top of the stack, *WP* (the Working Storage Pointer) increased so that the elements become part of second order working storage and then *AP* is set equal to *WP*. During this copying by *CRFA* and *CIFA* real-integer conversions can be performed. A copy of the array word, modified to point at the new set of elements (but still indicating the original storage mapping function) is stored in the formal accumulator. Within the procedure body, calls on the formal array by means of subscripted variables use a 'Take Address' operation, giving the *(n,p)* address of this array word.

Example

procedure *ARRAY* (*a*, *b*, *c*); **value** *b*, *c*; **real array** *a*, *c*;
Boolean array *b*;
begin *a* [*1*] := *c* [*1*] := *2* × *c* [*1*];
...

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*PE*	*(3,6), 3*	
4	*CAR*		*a*
5	*CBFA*		*b*
6	*CRFA*		*c*
7	*TFA*	*(3,3)*	
10	*TIC1*		
11	*INDA*		*a [1]*
12	*TRA*	*(3,7)*	
15	*TIC1*		
16	*INDA*		*c [1]*
17	*TIC*	*'2'*	
24	*TRA*	*(3,7)*	
27	*TIC1*		
28	*INDR*		*c [1]*
29	$\times$		
30	*STA*		*c [1]* := 2 $\times$ *c [1]*
31	*ST*		*a [1]* := 2 $\times$ *c [1]*
32	...		

This procedure could be called by the statement

ARRAY (A,B,C);

where *A* is a real array, *B* a Boolean array and *C* is either a real or integer array.

2.5.7 Summary

Procedure declarations and calls are translated independently, the linking up (and checking of) actual and formal parameters being performed at run time, through a set of formal accumulators.

The operations *CF* and *CFF* set up the formal accumulators, using 'actual operations', which correspond to the set of actual parameters. 'Actual operations' are in general complete in themselves, but in certain cases (expressions, constants, and strings) just give the address of the object program representation of the actual parameter.

After entry to a procedure a set of parameter list operations is used to check the setting of the formal accumulators, and to evaluate any parameters called by value. Uses of parameters called by name are represented by various 'Take Formal' or 'Call Formal' operations. Parameters called by value are in general used as if local to the procedure body.

A 'Call Formal' operation, or a 'Take Formal' operation on any but the simplest forms of actual parameters causes a temporary exit from the procedure body. During this exit it is quite possible to make a recursive activation of the procedure. (This is 'formal recursion', as distinct from the recursion caused by a recursive declaration of a procedure or a switch.)

2.6 FOR STATEMENTS

A statement can be executed repeatedly, for various values of a controlled variable, by preceding the statement with a for clause, and so making it into a for statement. The for clause contains a set of for list elements, which control the way in which values are assigned to the controlled variable. In the Revised ALGOL Report the action of the step-until element and the while element are described in terms of the controlled statement and additional ALGOL statements.

In the Whetstone Compiler a for statement is implemented by using the controlled statement as a 'subroutine' to a section of the Control Routine called the For Routine. Since a for statement can involve further for statements the For Routine must be capable of being used recursively.

Example

for $V[i] := A$ **step** i **until** B **do**
for $j := 1$ **step** 1 **until** 10 **do** $C[V[i], j] := 0$;

Here a for statement has a further for statement as its controlled statement. However, the For Routine must also be capable of dealing with a further for statement whilst organizing the setting up and testing of the controlled variable. Thus in the above example i could be a function designator. This could call a procedure in which a for statement was used.

Because of these possibilities a for statement is made into a block, and the various quantities needed by the For Routine are kept in the stacked information of the block. By this means the working of the For Routine can safely be interrupted so as to deal with a further for statement, during evaluation of an expression in the for list, or of the address of the controlled variable, or during the execution of the controlled statement.

This method of implementing for statements, though capable of dealing with all the possibilities inherent in the definitions of the Revised ALGOL Report, is very inefficient when used for simple for statements. More efficient techniques implementing for statements, involving the detection of simple forms of for statements at translation time, have been described by Hawkins and Huxtable [31], and Hill, Langmaack, Schwarz and Seegmüller [34].

Section 4.6.5 of the Revised ALGOL Report contains the rule that after exit from a for statement due to exhaustion of the for list, the value of the controlled variable is undefined. However, when a subscripted variable is given as the controlled variable of a for statement, it is quite possible for the values of the subscript expressions to be altered during the action of the for statement.

Example

```
i := V[i] := 0;
for V[i + 1] := V[i] + 1 while V[i + 1] < 4 do
    i := i + 1;
```

In such a situation the phrase 'the value of the controlled variable' is meaningless, as various different elements of an array have been used as a controlled variable. As a result, in KDF9 ALGOL this section of the report is interpreted to mean 'the value of the controlled variable that caused exit from the for statement'. Advantage of this rule is taken only in the Kidsgrove Optimizing Compiler, and not in the Whetstone compiler.

2.6.1 For Blocks

Due to the method of translation that is used, the controlled variable, the expressions in the for list elements and the controlled statement must be translated into object program operations in the order of their occurrence in the ALGOL text.

The operation *CFZ* (Call Function Zero) is used to call the block which has been formed from the for statement. This 'for block' starts with the operation *FBE* (For Block Entry) and ends with the operation *FR* (For Return). By positioning the *CFZ* operation in front of the object program representation of the first for list element the address of this element is automatically stacked as the return address by *CFZ*. The *FBE* operation, in addition to the normal (*n*,*L*) parameters of an ordinary *BE* (Block Entry) operation, has a two-syllable parameter indicating the start of the object program representation of the controlled variable.

If the controlled statement is itself an unlabelled block this block is combined with the for block, and the parameter *L* given with *FBE* will include the first order working storage of the unlabelled block.

If the controlled statement is a labelled block it is not possible to combine the two blocks in this way, because it must be possible to leave the block, without leaving the for statement (a similar situation was described in section 2.4.1.1 in connection with a procedure body consisting of a labelled block).

Various links needed by the For Routine are kept in the result accumulator set up by the *CFZ* operation which called the for block. In addition, four working stores are needed for the operation of any step-until elements. These working stores are set aside in the stack by the operation *FBE*. Thus the parameter *L* of *FBE* covers these four working stores, plus any needed by a controlled statement which is a block.

The execution of the for statement is actually controlled by sets of operations generated from the for list elements. Each set of operations assigns a value to the controlled variable, and causes the controlled statement to be obeyed zero or more times, before control is handed on to the next set of operations. The set of operations corresponding to the last for list element is

followed by the operation *FSE* (For Statement End). This operation deletes the link data and working storage of the for block from the stack, and causes a jump to be made to the statement following the for statement.

Thus the general form of the object program representation of a for statement is

```
          Op          Par
          UJ          (a)
d:  (Controlled Variable)
a:        CFZ         (b)
    (For List Elements)
          FSE         (c)
b:        FBE         (n,L), (d)
    (Controlled Statement)
          FR
c:        ...
```

(A detailed example is given in section 2.6.2.1)

The first operation is an Unconditional Jump, which is used to avoid the operations generated to evaluate the address of the controlled variable. This causes a jump to the *CFZ* operation, which in turn calls the for block which starts with the operation *FBE*. *FBE* has the address of the operations generated from the controlled variable as a parameter. This address is kept in the stack for use by the sets of operations generated from the for list elements. The *FBE* operation then causes a jump to the operation whose address has been stacked as a return address by *CFZ*, i.e. the address of the first for list element. This for list element then assigns a value to the controlled variable and either hands control to the next for list element, or causes the controlled statement to be obeyed. At the end of the controlled statement the operation *FR* returns control to the current for list element. Eventually the operation *FSE* is reached. This deletes the stacked information of the for block, and causes a jump to the statement following the for statement, whose address it has as a parameter.

The 'return address', originally set up in the link data of the for block by *CFZ*, is used to indicate the for list element currently being used. The object program representation of each for list element includes the sets of operations generated from the expressions contained in the element. These sets of operations, and the set of operations for evaluating the address of the controlled variable, are used as subroutines by the For Routine. During the activation of these subroutines a machine code link is kept in the stack. At the end of each of these subroutines the special object program operation *LINK* is used to return control to the For Routine, at the address given as a machine code link.

The subroutine generated from an expression in a for list element can be used recursively.

Example

```
real procedure F;
begin . . .
      for A := A + F step 1 until 10 do S;
      . . .
end;
```

This situation is similar to that described in section 2.5.4.3, with regard to the implicit subroutine generated from an actual parameter expression. However, it is not necessary to make a for list element expression into a block, as was necessary with an actual parameter expression. This is because the for statement has already been made into a block, and all the link data required to allow recursive activation of the for list element expression is kept in the stack.

In fact the double-word result accumulator set up by *CFZ* is used to hold various quantities needed by the For Routine (e.g. the addresses of the first object program operation of the controlled variable and the controlled statement). By this means all the 'working storage' of the For Routine is kept in the stack, either in this result accumulator or as link data, during the action of any of the object program subroutines, or of the controlled statement. Thus the For Routine, even though written in machine code, can be used recursively, as is needed in the example given in section 2.6.

2.6.2 For List Elements

Each for list element is represented by a set of object program operations, consisting of a subroutine for each expression in the for list element, and one or more 'for operations'. The various for operations call the appropriate sections of the For Routine, and hence control the action of the for statement.

2.6.2.1 Arithmetic Element

An arithmetic for list element uses a single for operation,

FORA (For Arithmetic)

This operation is followed by a set of operations for evaluating the arithmetic expression, which is in turn followed by the operation *LINK*.

The action of *FORA* is to call firstly the subroutine which evaluates the address of the controlled variable, and secondly the subroutine which evaluates the arithmetic expression, and then to assign the resulting value to the controlled variable. After setting the return address given in the link data to indicate the next for list element, a jump is made to the start of the controlled statement.

Example

```
for V[i] := A + 1, A + 2 do C := C + V[i];
```

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*11*)	
3	*TRA*	*V*	Controlled Variable
6	*TIR*	*i*	
9	*INDA*		
10	*LINK*		
11	*CFZ*	(*37*)	
14	*FORA*		First For List Element
15	*TRR*	*A*	
18	*TIC1*		
19	+		
20	*LINK*		
21	*FORA*		Second For List Element
22	*TRR*	*A*	
25	*TIC*	'*2*'	
32	+		
33	*LINK*		
34	*FSE*	(*58*)	
37	*FBE*	(*2,4*), (*3*)	
42	*TRA*	*C*	Controlled Statement
45	*TRR*	*C*	
48	*TRA*	*V*	
51	*TIR*	*i*	
54	*INDR*		
55	+		
56	*ST*		
57	*FR*		
58	...		

The operation *CFZ* at syllable *11* will set up the stacked return address with *14*, the address of the first for operation, before jumping to the operation *FBE*. The *FBE* operation records the addresses of the first of the sets of operations generated from the controlled variable, and the controlled statement, (i.e. *3* and *42*) in one of the words set aside as a result accumulator by *CFZ*. These addresses, (*3* and *42*), are given as a parameter to *FBE*, and by the current value of the program counter, respectively. (*FBE* has been given an arbitrary block level of *2*.) *FBE* then causes a jump to be made to the address given in the stacked return address, i.e. the operation *FORA*.

FORA records the current value of the program counter in the second word of the result accumulator, and then sets up the program counter with the address *3* given in part of the first word of the result accumulator. A machine code link is placed in the link data, and by this means *FORA* uses the operations for evaluating the address of the controlled variable as a subroutine. These operations stack this address, and then

the operation *LINK*, using the machine code link, causes a return to be made to the appropriate position in the routine corresponding to *FORA*. The information in the result accumulator is then used to reset the program counter, and the operations for evaluating the arithmetic expression, starting at syllable *15*, are used as a subroutine. The top two accumulators are then used to perform a 'Store' operation, and the return address is set up with *21*, the current value of the program counter, which is the address of the next for list element. Finally the program counter is set up with the address of the controlled statement.

The controlled statement is then obeyed, and eventually the operation *FR*, at syllable *57*, uses the return address to cause a jump to be made to the second for list element.

This second element works in exactly the same way, and after the controlled statement has been obeyed using the value of this for list element the operation *FSE* at syllable *34* is reached. This operation deletes the stacked information of the block, and causes a jump to be made to syllable *58*, the first operation after the end of the for statement.

2.6.2.2 While Element

A while element uses the for operation

FORW (For While)

followed by subroutines corresponding to the arithmetic and Boolean expressions contained in the element. *FORW* causes the arithmetic expression to be evaluated and assigned to the controlled variable in a similar way to that described for *FORA*. Then the Boolean expression is evaluated, and depending on its truth value, either the controlled statement or the next for list element is obeyed.

Example

for *V* := *V* + *1* **while** *V* < *10* **do** *A* [*V*] := *0*;

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	*(7)*	
3	*TIA*	*V*	Controlled Variable
6	*LINK*		Controlled Variable
7	*CFZ*	*(32)*	
10	*FORW*		For List Element
11	*TIR*	*V*	For List Element
14	*TIC1*		For List Element
15	+		For List Element
16	*LINK*		For List Element
17	*TIR*	*V*	For List Element
20	*TIC*	*'10'*	For List Element
27	<		For List Element
28	*LINK*		For List Element

Syll	*Op*	*Par*	*Remarks*
29	*FSE*	*(47)*	
32	*FBE*	*(3,4), (3)*	
37	*TRA*	*A*	Controlled Statement
40	*TIR*	*V*	
43	*INDA*		
44	*TIC0*		
45	*ST*		
46	*FR*		
47	...		

After the operation *FORW* has performed the assignment

$V := V + 1;$

the program counter will be set at *17* and the return address at *10*. The second subroutine is then evaluated, leaving its result, a Boolean value, as the top accumulator. At this point the program counter will be set at *29*. The top accumulator is inspected, and if it contains the value **true** the program counter is set to indicate the first operation of the controlled statement (at syllable *37*). Otherwise the return address is set up with the value *29* (which is the program address of the next for list element, or in this case the operation *FSE*). In either case the final action is to delete the top accumulator. By this means the controlled statement will be obeyed repeatedly, each time returning to *FORW*, whilst $V < 10$. However, as soon as this condition no longer holds the operation *FSE* will be used to finish the activation of the for block, and to jump to syllable 47.

2.6.2.3 Step-Until Element

Since the publication of the original ALGOL Report there has been considerable discussion on section 4.6.4.2 which defines the action of the step-until element. This matter has not been clarified in the Revised ALGOL Report and, therefore, of the various suggested reformulations of section 4.6.4.2, a somewhat arbitrary choice of the one given in ALGOL Bulletin 14 has been made for KDF9 ALGOL.

for $V := A$ **step** B **until** C **do** $S;$

is interpreted as

$S1 := V := A;$
$S2 := B;$
$L1$: **if** *sign* $(S2) \times (S1 - C) > 0$ **then go to** *Element exhausted*;
Statement $S;$
$S2 := B;$
$S1 := V := V + S2;$
go to $L1;$

Thus the values of *V*, the controlled variable, and *B*, the step, are calculated only once each time around the loop. The re-calculation needed for the original definition is avoided by the introduction of *S1* and *S2*.

In fact the basic system of implementation is not greatly affected by this reformulation, except that a for block is given four working stores for the storage of the accumulators for *S1* and *S2*. The step-until element uses two for operations

FORS1 (For Step – 1st Entry)
FORS2 (For Step – 2nd Entry)

The for list element

A **step** *B* **until** *C*

is represented in the object program by

FORS1
Take *A*
LINK
FORS2
Take *B*
LINK
Take *C*
LINK

The operation *FORS1* is used for the first call of the for list element, and performs the initial setting of the controlled variable. *FORS2* is used on subsequent calls for the for list element, and increases the controlled variable by the appropriate step. Both *FORS1* and *FORS2* end by testing the value of the controlled variable to find whether the action of the for list element is complete, or whether the controlled statement is to be obeyed. The four working stores set aside in the stack by the operation *FBE* are used as accumulators for the values of *S1* and *S2*.

FORS1 avoids the *FORS2* operation, after having used the subroutine for evaluating the first arithmetic expression, by simply increasing the program counter by one. However, the return address is set up to point at *FORS2*, and therefore on any subsequent uses the controlled statement returns to the operation *FORS2*.

Example

for *i* := *j* **step** – *1* **until** *k* **do**;

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*7*)	
3	*TIA*	*i*	Controlled Variable
6	*LINK*		Controlled Variable
7	*CFZ*	(*26*)	
10	*FORS1*		Step-Until Element
11	*TIR*	*j*	Step-Until Element
14	*LINK*		Step-Until Element
15	*FORS2*		Step-Until Element
16	*TIC1*		Step-Until Element
17	*NEG*		Step-Until Element
18	*LINK*		Step-Until Element
19	*TIR*	*k*	Step-Until Element
22	*LINK*		Step-Until Element
23	*FSE*	(*32*)	
26	*FBE*	(*3,4*), (*3*)	
31	*FR*		
32	...		

The operation *FORS1* uses the controlled variable subroutine, and the subroutine for the first arithmetic expression of the for list element, in order to perform the assignment

$$S1 := i := j;$$

(*S1* uses the first two words of the working storage of the for block as an accumulator.) After this has been done the program counter will have the value *15*, indicating the operation *FORS2*. This value is then placed in the stacked return address. The program counter is then increased by one, to avoid the operation *FORS2*, and to allow the subroutine for the second arithmetic expression of the for list element to be used, in order to perform the assignment

$$S2 := -1;$$

(*S2* uses the third and fourth words of the working storage of the for block.)
FORS1 then uses the third subroutine in the for list element, to set the value of *k*, and thus find the value of

$$sign\ (S2) \times (S1 - k) > 0$$

If the value is **false** the controlled statement is obeyed (the program counter is set to indicate the first operation of the controlled statement using information in one word of the result accumulator of the for block). At the end of the controlled statement the operation *FR* (For Return) sets the program counter to the value contained in the return address, and hence causes a jump to be made to *FORS2*.
On the other hand, if the value is **true** the return address is set up, using

the program counter, to indicate the next for list element (in this case the operation *FSE* at syllable *23*).
The operation *FORS2*, using the various object program subroutines, performs the assignments

$$S2 := -1;$$
$$S1 := i := i + S2;$$

(The subroutine for evaluating the address of i, the controlled variable, is used twice, the second time in order to get the value of i.)
Then the operation *FORS2* joins the machine code of the *FORS1* operation and evaluates the third arithmetic expression and the test of *S1* against k. The operation *FORS2* is used repeatedly, each time causing an assignment to be made to the controlled variable, and the controlled statement to be obeyed, until $S1 < k$, when control is handed to the next for list element.

2.7 CODE PROCEDURES

In section 5.4.6 of the Revised ALGOL Report it is stated that the body of a procedure may be expressed in non-ALGOL language. In fact, other than by extending ALGOL, it is only by the use of code procedures that such essential facilities as the input of data and output of results can be provided.

It has been described in section 1.4 how the Whetstone Compiler is just part of a twin-compiler system, and is designed to be compatible with a multi-pass optimizing ALGOL Compiler being developed at Kidsgrove by the Data Processing and Control Systems Division of the English Electric Company. Naturally this compatibility must extend even to code procedures. Therefore agreement has been reached on a set of rules for the incorporation of code procedures into KDF9 ALGOL programs. These rules (see Randell, Duncan and Huxtable [59]), form the required expansion of section 5.4.6 of the Revised ALGOL Report.

The design of these rules has been guided by the fact that a KDF9 ALGOL program containing code procedures must be acceptable to either of two fundamentally different compilers. (In contrast to the Whetstone Compiler, which produces an object program to be interpreted at run time, the Kidsgrove Compiler produces a normal machine code object program.) The most obvious consequence of this is that writers of code procedures must not make use of any internal features of a particular compiler or of the object program it produces.

2.7.1 User Code Procedures in KDF9 ALGOL

A sequence of User Code instructions can be incorporated in a KDF9 ALGOL program by enclosing it by the symbols **KDF9** and **ALGOL**, and prefacing it by a normal procedure heading. (A brief description of KDF9 User Code is given in Appendix 3.) The main channel of communication between the User Code and the surrounding ALGOL is via the parameters given in the procedure heading. A User Code procedure body can include 'pseudo-instructions', which use a formal parameter enclosed in string quotes.

Formal parameters can be used this way in fetch, store and jump instructions. The dynamic end of a User Code procedure is indicated by the symbol **EXIT.**

Example

```
procedure ADD (a, b, c); real a, b, c;
        KDF9 'a';
             'b';
             +F;
             ='c';
             EXIT
        ALGOL;
```

This rather trivial example performs

$c := a + b;$

The other method of communication is by means of the 'value' of a code procedure, when called by a function designator. This value must be left in the top cell of the nesting store, before reaching the symbol **EXIT**.

Example

```
real procedure ADD (a, b); real a, b;
        KDF9 'a';
             'b';
             +F;
             EXIT
        ALGOL;
```

The pseudo-instructions using the formal parameters are converted into normal User Code, by either the Whetstone or the Kidsgrove Compiler, which allow a programmer to use the formal parameters as normal 'fetch', 'store', and 'jump' User Code instructions, without regard for the complexity of the corresponding actual parameters.

Example

$c := 2 \times ADD\ (a + b\,[i - 3],\ (c + d) \uparrow e);$

Apart from the fact that a code procedure cannot be used recursively, the full possibilities of use of parameters called by name or by value are available. For example it is possible to rewrite the Innerproduct procedure, given in section 5.4.2 of the Revised ALGOL Report, in User Code so as to accumulate the products in a double-length accumulator (see a paper by Duncan [19]).

One detail which has been omitted in this brief description of User Code procedures is that the first User Code instruction should be preceded by a short description of the User Code. The description, composed of a set of four integers, gives such details as number of Q-stores used, etc.

2.7.1.1 Types of Parameters

The full range of parameters, apart from switches and procedures, can be used in a procedure with a body in User Code.

2.7.1.1.1 Real, Integer and Boolean Parameters. The pseudo-instructions '*a*' and ='*a*' have the effect of fetching or storing the corresponding actual parameter, and can be used as if for a normal fetch or store of one number in core storage. If the formal parameter is called by value then the corresponding actual parameter will be evaluated before entry to the code procedure. In

such a case the replacement of the instruction is fairly straightforward. When the parameter is called by name, the instructions '*a*' and ='*a*' must be replaced by instructions which cause the evaluation of the object program representations of the corresponding actual parameter. This again is made to appear as a simple fetch or store instruction.

2.7.1.1.2 Label Parameters. The pseudo-instruction *J* '*a*' can be used to leave the User Code procedure and jump to the label given by the corresponding actual parameter. If the formal parameter is called by value then the actual parameter is evaluated before entry to the code procedure.

2.7.1.1.3 Array Parameters. The pseudo-instruction '*a*' can be used to fetch the array word of the actual parameter array to the top of the nesting store. The writer of the User Code procedure can then use the array word to locate and use the storage mapping function of the array as described in section 2.3.1. If the array is called by value a copy is made of the array (with real-integer conversion, if necessary) before entry to the code procedure.

2.7.1.1.4 Strings. In section 2.5.4.9 it was described how a string is stored as a set of basic symbols beginning with the opening string quote, and ending with the closing string quote. In this case the pseudo-instruction '*a*' will fetch the address of the word containing the opening string quote to the top of the nesting store.

2.7.2 The Implementation of Code Procedures

In the Whetstone Compiler the implementation of code procedures needs a system of communication between sequences of interpreted object program operations, and User Code instructions. A somewhat similar situation was described in section 2.6.1 in connection with the use of the For Routine to control the operation of a for statement.

The heading of a code procedure is translated normally, and is followed by the operation

DOWN

which has a one-syllable parameter *m*. This operation causes a descent to the level of machine code, having preserved in the stack the contents of any stores which are available to the writers of the code procedure. The parameter *m* indicates which sequence of machine code is to be called.

At translation time the code body itself is copied into a backing store, converting any pseudo-instructions into one or more normal User Code instructions. At the end of translation a User Code version of the Control Routine and the code procedures are processed by the User Code Compiler. This is a standard assembly program, used to produce the actual machine code from User Code. By this means the Control Routine and the code procedures are formed into an integrated program.

The sequence of User Code instructions which replace the pseudo-instructions are in general complete in themselves. However, in the case of scalar parameters called by name these User Code instructions use sequences of object program operations as subroutines.

In the object program the code body is replaced by any such sequences of operations as are necessary. These sequences end with one of the operations

UP1,
UP2,
or *GTA*

UP1 is used for returning to the code procedure with the value of an actual parameter, which is placed at the top of the nesting store; *UP2* is used when it is required to store a value in the top cell of the nesting store at the address given as an actual parameter. The operation *GTA* (Go To Accumulator) has been described in section 2.4.1 and is used in the case of jumping to the label given by an actual parameter.

This technique of using subroutines of object program operations is of course only one of the various possible ways in which the User Code procedure body could use parameters called by name. Its main merit is that it involves the creation of only two extra object program operations (*UP1* and *UP2*) and needs a minimum of expansion of the pseudo-instructions.

2.7.2.1 Real, Integer and Boolean Parameters

If the code procedure contains a pseudo-instruction for fetching the value of a real parameter, called by name, then the sequence of instructions

TFR (*n*,*p*)
UP1

is generated.

If there is a pseudo-instruction for assigning to such a parameter, then the sequence

TFAR (*n*,*p*)
UP2

is generated. (The dynamic address (*n*,*p*) is that of the corresponding formal accumulator.)

Example

```
procedure CALC (a, b); value b; real a, b;
                       KDF9 'a'; 'b'; -F;
                            'a'; ×F; ='a'; EXIT
                       ALGOL;
```

This procedure performs

$a := (a - b) \times a$

and is translated into

Syll	*Op*	*Par*	*Remarks*
0	*PE*	(*n*,*4*), *2*	
4	*CA*		
5	*CSR*		
6	*DOWN*	*m*	
8	*TFR*	(*n*,*3*)	Subroutine for '*a*'
11	*UP1*		
12	*TFAR*	(*n*,*3*)	Subroutine for ='*a*'
15	*UP2*		
16	...		

After the procedure has been entered and the parameter list operations have checked the actual parameter corresponding to *a* and evaluated the actual parameter corresponding to *b*, the operation *DOWN* is reached. This operation transfers control to the sequence of User Code instructions which formed the code body. The instructions which replace the pseudo-operations '*a*' and ='*a*' use the sets of object program operations starting at syllables *8* and *12* as subroutines. The instructions which replace '*b*' just fetch the contents of the formal accumulator to the top of the nesting store.

The corresponding sets of object program operations for integer and Boolean parameters, called by name, and used by pseudo-instructions '*a*' and ='*a*', are:

TFI (*n*,*p*) *UP1*	Integer '*a*'
TFAI (*n*,*p*) *UP2*	Integer ='*a*'
TFB (*n*,*p*) *UP1*	Boolean '*b*'
TFA (*n*,*p*) *UP2*	Boolean ='*b*'

2.7.2.2 Label Parameters

The pseudo-operation using a label parameter is *J* '*a*'. When the parameter is called by name the set of operations

TFL (*n*,*p*)
GTA

is generated. Thus, when this set is used by a subroutine, it is equivalent to a go to statement which leaves the procedure.

3 THE TRANSLATOR

3.1 INTRODUCTION

The task of the Translator is to convert the ALGOL text into object program operations, which will then be obeyed interpretively by the Control Routine. A description of the object program has already been given, and a list of operations is given in Appendix 7.

Various basic methods of translation have been briefly described in section 1.2.4. In the Whetstone Compiler a one-pass translation technique is used, and translation takes place whilst the ALGOL text is being read in to the computer. The object program is generated in the core storage and it is complete and ready to be obeyed almost as soon as the reading of the ALGOL is finished. The ALGOL program itself need not be stored in the computer although in fact, for reasons of efficiency, a small amount of storage is used for buffers associated with the input.

At the heart of the Translator is a routine called the 'basic cycle routine'. This routine reads in the hardware representation of the ALGOL text and converts it into ALGOL basic symbols.

The object program is essentially a form of 'Reverse Polish' notation, and the Translator uses a stack (or push-down store) to perform the necessary re-ordering of the ALGOL symbols.

Details concerning the declaration and use of the various identifiers are kept in a 'name list', for use by the Translator during the generation of the object program operations.

During translation an exhaustive series of checks is performed on the legality of the ALGOL text. However, some checks which are difficult to carry out at translation time, because of the nature of the one-pass techniques employed, are deferred until run time.

3.1.1 One-Pass Translation

It is possible in ALGOL for identifiers to appear in expressions or statements before the occurrence of their declaration.

Example

begin real procedure P; $P := x + y$;
real x, y;
. . .

The procedure P uses the two variables x and y. They are declared in the same block head as P, but their declaration occurs after that of P in the ALGOL text.

Thus, if the translator is to generate the object program during a single scan through the text of the ALGOL program, such possibilities must be allowed for.

A given identifier may be used in different ways in an ALGOL program (i.e. in the left part list of an assignment statement, in an expression, as an actual parameter, etc.) and the appropriate object program operation must be generated for each use of the identifier. For this reason it is not sufficient to leave a space in the object program for an identifier which has appeared in the ALGOL text before its declaration; it would then be very difficult to decide which type of operation to put in the space when the declaration details are available.

Example

$x := y + x - P(x);$

(Assuming that y is known to be of type **integer**, P to be a real procedure, and that no declaration details are available for x.) The object program which would result when spaces are left for the operations generated from each use of the identifier x would be

Syll	*Op*	*Par*	*Remarks*
0			*x*
3	*TIR*	*y*	
6			*x*
9	+		
10	*UJ*	*(17)*	
13			*x*
17	*CF*	*P, 1*	
21	−		
22	*ST*		
23	...		

When the declaration details for x become available, the spaces left for the three uses of x must be filled in. However, it is impossible to know, without working through the object program, that these three spaces have to be filled in with the operations *TRA*, *TRR* and *PR*, respectively, when x is found to be of type **real**.

Thus, it is necessary to place a bit pattern, giving details as to the type of use of the identifier, in the space set aside in the object program. These bit patterns are known as 'skeleton operations'.

3.1.1.1 Skeleton Operations

Four types of skeleton operations are sufficient to allow the correct operation to be filled in when the declaration details become available. These are listed below.

(*i*) AO (*Address Operation*)
This skeleton operation indicates that an operation of the form 'Take Address' is required.

AO could be replaced by any of the following operations

TRA	*TFA*
TIA	*TFAR*
TBA	*TFAI*

(*ii*) *RO* (*Result Operation*)
The skeleton operation *RO* is used when an identifier appears in an expression and indicates that the value of the quantity associated with the identifier, rather than its address, is required. The identifier might prove to be a scalar, when this skeleton operation would be replaced by an operation of the form 'Take Result', or a function designator with no parameters, when it would be replaced by *CFZ* or *CFFZ*.
RO could be replaced by any of the following operations

TRR	*TFR*	*CFZ*
TIR	*TFI*	*CFFZ*
TBR	*TFB*	
TL	*TFL*	

(*iii*) *PO* (*Parameter Operation*)
This skeleton operation is used to indicate that the identifier has been used as an actual parameter and that it must be replaced by the appropriate 'actual operation'. Thus *PO* could be replaced by any of the following 'actual operations'

PR	*PRA*	*PFR*	*PPR*	*PF*
PI	*PIA*	*PFI*	*PSW*	
PB	*PBA*	*PFB*	*PL*	

(*iv*) *FO* (*Function Operation*)
The skeleton operation *FO* is used for a procedure call when insufficient details are available for the procedure identifier. This skeleton operation will be replaced by either *CF* or *CFF*.

In some cases more information could be given in a skeleton operation than is necessary for the four types given above.

Example

if *B* **then**

This use of *B* shows that it must be a Boolean scalar or function designator.

However, for convenience, the number of different types of skeleton operations has been kept to a minimum. Any additional information concerning the use of an identifier is preserved in the name list (see section 3.3).

3.1.1.1.1 Chaining of Skeleton Operations. An identifier may appear many times in the ALGOL text before the occurrence of its declaration, resulting in the generation of many skeleton operations. When the declaration details for the identifier are available all of the skeleton operations must be replaced by the appropriate operations. In order to be able to locate the positions of all of the skeleton operations that have been generated for an identifier they are linked together into a 'chain'. The part of the operation that is reserved for its parameter is used for the chain link in the following way.

The first skeleton operation that is generated for an identifier is given an empty parameter part. The object program address of the parameter of this operation is preserved in the name list entry for this identifier (see section 3.3.2). At the next use of the identifier another skeleton operation is generated, and its parameter space filled with the address of the parameter of the first skeleton operation (taken from the name list). The address of the parameter of this skeleton operation replaces that of the previous skeleton operation in the name list. This process is repeated for each skeleton operation generated for an identifier. In this way the skeleton operations which are created for each identifier that has no declaration details available are linked together into a chain, each operation enabling the previous one in the chain to be found. The address of the last skeleton operation generated (i.e. the start of the chain) is kept in the name list.

When the declaration details are available for an identifier the address of the operation at the start of the chain is taken from the name list and the skeleton operation at this address replaced by the appropriate object program operation. The parameter space of the skeleton operation will give the address of the next skeleton operation to be replaced; this is noted and the parameter of the operation is filled in. This process is repeated for the next skeleton operation in the chain and so on through the chain until the last skeleton operation in the chain (i.e. at the finish of the chain), which contains no address in its parameter position, is reached.

Example

$x := y + x;$
$y := y + 1/y;$

The following object program will be generated (assuming that no declaration details are available for x and y).

Syll	*Op*	*Par*	*Remarks*
0	*AO*	( )	x
3	*RO*	( )	y
6	*RO*	(*1*)	x
9	+		
10	*ST*		$x := y + x$
11	*AO*	(*4*)	y
14	*RO*	(*12*)	y

Syll	*Op*	*Par*	*Remarks*
17	*TIC1*		
18	*RO*	*(15)*	*y*
21	/		
22	+		
23	*ST*		$y := y + 1 / y$
24	...		

The chain of skeleton operations for x starts at syllable seven and finishes at syllable one and the chain for y starts at syllable nineteen and finishes at syllable four.

The addresses of the starting points of these chains (i.e. seven for x and nineteen for y) are kept in the name list with the other information about the types of uses of x and y respectively.

3.1.2 The Method of Translation

In order to find the extent of an identifier or a number the basic cycle routine reads in the ALGOL text until a delimiter is reached. The basic cycle routine then calls the appropriate 'delimiter routine' into operation. The Translator consists mainly of these delimiter routines, together with a set of 'subroutines' which are used by the delimiter routines to do some of the more common tasks.

Thus the basic cycle routine can be said to deliver the ALGOL a 'section' at a time where a section will consist of either

(*i*) a delimiter;
(*ii*) an identifier and a delimiter; or
(*iii*) a constant (i.e. number or logical value) and a delimiter.

Example

if | **true then** | $x :=$ | $-$ | *3* **else** | **go to** | L;

Vertical lines have been used to show the segmentation of the above statement into sections.

The basic cycle routine processes the characters forming an identifier or a number into a coded form. For convenience the delimiters '+', '−', '$_{10}$' and '.' which occur within a number are dealt with by the basic cycle routine.

Example

$y :=$ | $-$ | $\cdot 083_{10} - 02 +$ | $_{10} + 5 /$ | z;

During translation it is necessary to keep a record of certain features of the ALGOL that has passed, in order to be able to translate the current section of the ALGOL text correctly. Certain delimiters may be used for several different purposes, and in such cases it is necessary to be able to decide how the

current delimiter is being used. For this reason a set of 'state variables' is used by the delimiter routines. For example the state variable *E* (set to one for statements, zero for expressions) will indicate whether the delimiter **if** is being used for a conditional statement or a conditional expression.

The structure of ALGOL, which allows statements to be nested within statements, for instance, makes it necessary to preserve values of the state variables corresponding to the various statements. Therefore it is not sufficient to keep simply a single set of state variables, and so values of certain of the state variables are preserved from time to time in the translator stack. More details of the state variables and their uses are given in the description of the translation stack and of the delimiter routines and a list is given in Appendix 8.

During the translation of expressions a certain amount of checking is performed on the types of identifiers and on the operators contained in the expressions.

Certain features of ALGOL (e.g. the operator '↑') make it impossible to carry out a full check on the types of identifiers used in expressions during translation. As a result, a full set of checks has been incorporated in the run time Control Routine and only such checks as can easily be performed during one-pass translation are incorporated into the Translator.

Three state variables are used by the Translator to enable this checking to be carried out. These are the state variables *E*, *ARITH* and *TYPE*. The first of these has been mentioned earlier and is used to decide if an expression is being translated; it is set to zero for the translation of an expression, otherwise to one.

The state variable *ARITH* is needed only for checking out the use of relational and logical operators in arithmetic expressions. Since arithmetic expressions can be conditional, the value of *ARITH* is stacked whilst the if clause (which can contain relational and logical operators) is being translated. To avoid undue complication, *ARITH* is ignored in this and succeeding sections, but is dealt with in full in Appendix 11.

The state variable *TYPE* is used to perform a partial check on the types of identifiers used in expressions. At the start of an expression *TYPE* is set up with a sequence of binary digits, representing the range of acceptable types of identifiers. During the translation of the expression any additional information which enables this range of acceptable types to be further restricted is incorporated into *TYPE*. The term 'arithmetic' is used to represent **real** or **integer**. In order to reserve the term **Boolean** for the case when only Boolean variables are allowed, the term 'algebraic' is used to indicate the range of types allowable in a Boolean expression. Thus algebraic includes **real, integer** and **Boolean**. The other possible setting of *TYPE* is 'designational', used for the expression following **go to**, for example.

3.2 TRANSLATOR STACK

The translator stack is used as a holding store to enable expressions to be converted into the Reverse Polish form required by the object program, and to deal with the nested statement structure.

In the Reverse Polish form of an expression the variables and constants appear in the same order as in the original expression but the delimiters have been re-ordered to allow for the precedence of operators.

Example

$$a + b \times c \div 3$$

is converted into

$$a, b, c, \times, 3, \div, +$$

To re-order the delimiters they are stored temporarily in the translator stack until they can be transferred to the object program. By allotting a priority to each of the delimiters and stacking the priority along with the delimiter the re-ordering into Reverse Polish can take place automatically.

In the examples given in this section, the translation process is demonstrated for the production of Reverse Polish and details of the corresponding object program operations are ignored.

When an identifier or a constant is reached in the expression it is transferred to the Reverse Polish. However, when a delimiter is reached it is placed in the stack, after first unstacking into the Reverse Polish expression any delimiters whose priorities are not less than that of the current delimiter.

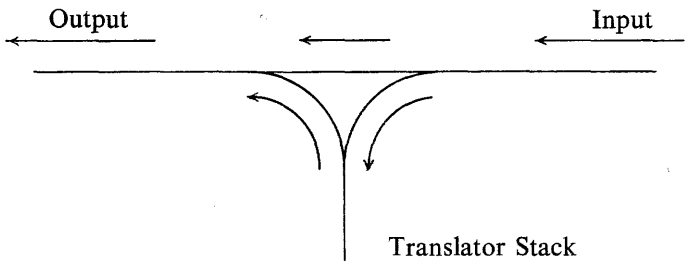
3.2.1 Translation of Expressions

3.2.1.1 Simple Arithmetic Expressions

Dijkstra [16] uses the analogy of a railway shunting yard to explain this technique of stacking delimiters with a priority to convert expressions into Reverse Polish.

A set of priorities which would allow arithmetic expressions to be converted would be

Delimiter	*Priority*
$+ \; -$	*0*
$\times \; / \; \div$	*1*
$\uparrow$	*2*

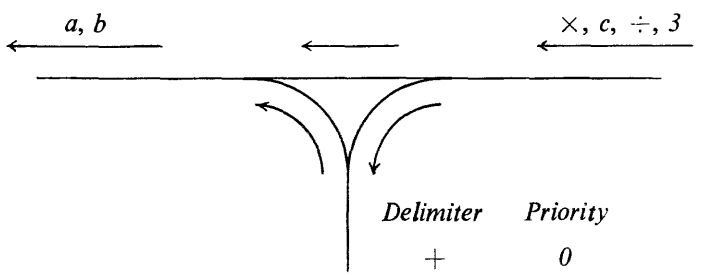


The shunting yard has the form of a 'T' junction with the translator stack as the branch line to perform the 'shunting' or re-ordering. Identifiers or constants pass straight from input to output but in general the delimiters reach the output via the translator stack. Before the delimiters are shunted into the translator stack any delimiters at the top of the stack which have a priority equal to or greater than that of the current delimiter, are allowed to pass to the output.

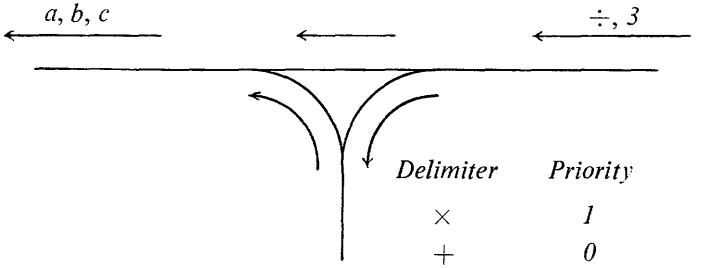
Example

$a + b \times c \div 3$

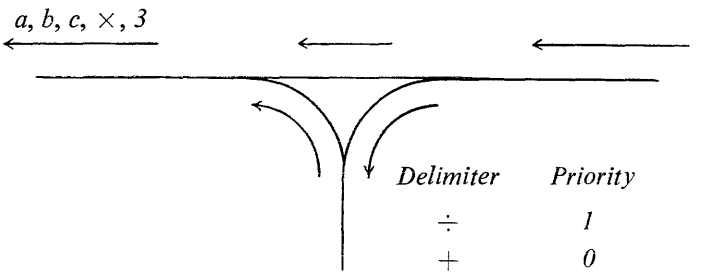
The identifier *a* passes straight to the output and, as the stack is empty, the delimiter '+' cannot cause any unstacking before being stacked. Similarly the identifier *b* passes straight to the output and the diagram of the shunting yard is then



The delimiter '×' with its priority of one cannot cause the '+' to be unstacked before it, too, is stacked. The identifier *c* transfers straight to the output and the picture is as follows



The current delimiter '÷', having a priority equal to that of '×' at the top of the stack, causes the delimiter '×' to be unstacked into the output. The current delimiter '÷' is then stacked. After the constant *3* has passed to the output the picture shows the following



Since there is no more input left the remaining items in the stack are emptied into the output with the result that the expression has been re-ordered into Reverse Polish as follows

a, *b*, *c*, ×, *3*, ÷, +

The operation *NEG* replaces the delimiter '−' when this delimiter is used as a unary operation. (e.g. when it appears immediately after a left round bracket). By giving *NEG* the same priority as '×', '/' or '÷' rather than that of '+' or '−' the unary operation will be carried out as soon as possible at run time. This allows the expression to be evaluated from left to right as far as possible.

Example

(− *a* × *b*)

will be re-ordered into '*a*, *NEG*, *b*, ×' rather than '*a*, *b*, ×, *NEG*'.

The delimiter '+' used as a unary operation is ignored by the Translator after checking the validity of its use and after checking whether it starts an actual parameter expression.

However, to allow for brackets in an arithmetic expression the set of priorities given above must all be increased by one so that the delimiter '(' can be given a lower priority, i.e. zero. In fact this delimiter must be stacked without first doing any unstacking. The delimiter ')' is given a priority one higher than that of '(' so that the stack will be emptied down to but not including the opening bracket when the closing bracket is being translated. Thus the top of the stack may be inspected after the automatic unstacking to check that the opening bracket is indeed there, whereupon it is unstacked and discarded. The delimiter ')' is never stacked. The set of priorities then becomes

Delimiter	*Priority*
(	*0*
$+ \; -$)	*1*
$\times \; / \; \div$ *NEG*	*2*
$\uparrow$	*3*

Example

$$- a + b \times c \uparrow (d \div e) / f$$

This will be re-ordered, using the above set of priorities, into

$$a, NEG, b, c, d, e, \div, \uparrow, \times, f, /, +$$

At the delimiter '(' the contents of the generated Reverse Polish expression consists of

$$a, NEG, b, c,$$

and the contents of the stack are

Delimiter	*Stack Priority*
$\uparrow$	*3*
$\times$	*2*
$+$	*1*

The left round bracket is then stacked.
The part of the expression between the brackets causes d and e to be added to the Reverse Polish and '$\div$' to be stacked.
The closing bracket ')' with its priority of one unstacks into the Reverse Polish the remaining delimiters belonging to the part of the expression between the brackets. The delimiter '(', now at the top of the stack, is unstacked and discarded. At this point the stack contains

Delimiter	*Stack Priority*
$\uparrow$	*3*
$\times$	*2*
$+$	*1*

and the Reverse Polish consists of

$$a, NEG, b, c, d, e, \div$$

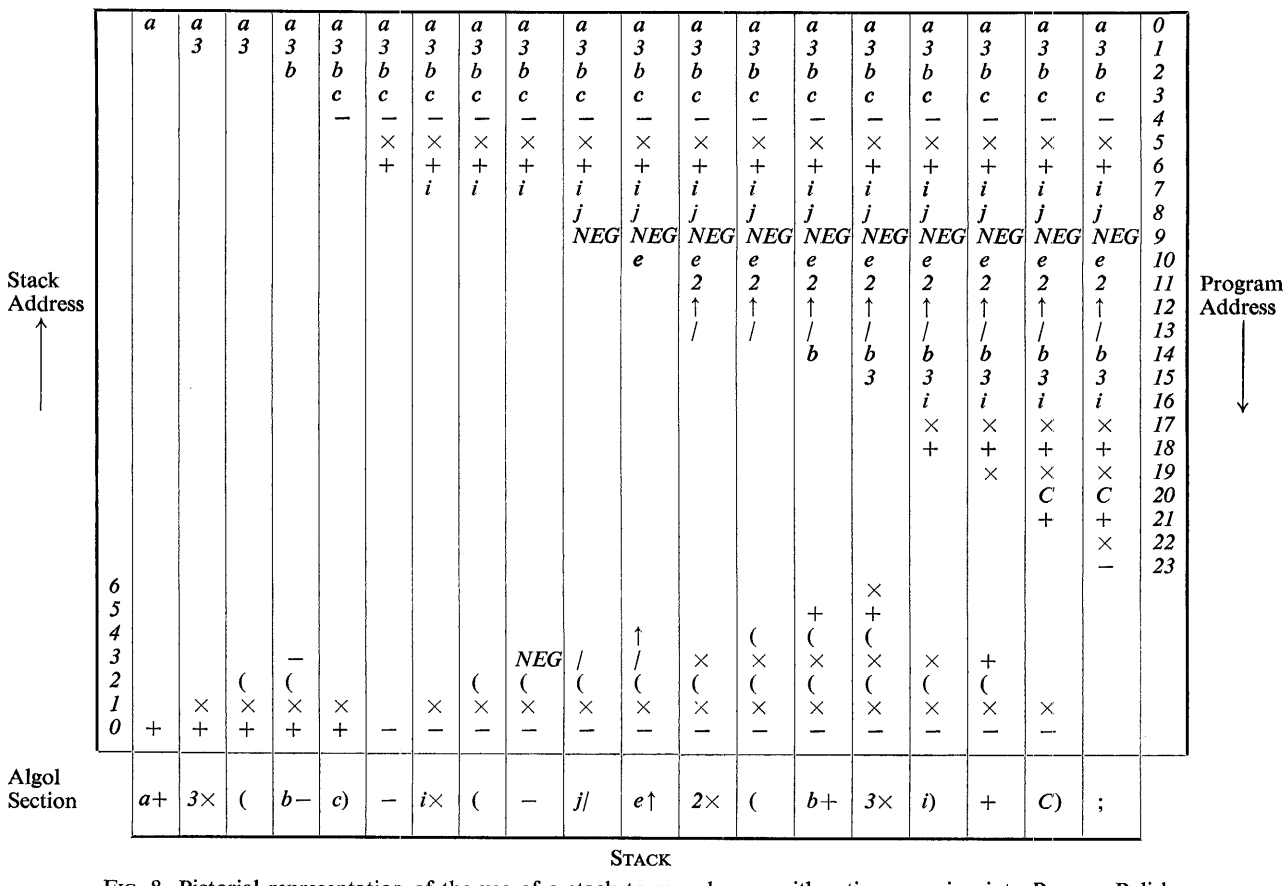


FIG. 8. Pictorial representation of the use of a stack to re-order an arithmetic expression into Reverse Polish.

The current delimiter '/' with its priority of two will now cause the delimiters '↑' and '×' to be unstacked into the Reverse Polish. When the identifier *f* has been transferred to the Reverse Polish expression the delimiters '/' and '+' still remain in the stack. Thus a 'closing bracket' for the expression is needed to clear the remains of the expression from the stack into the Reverse Polish.

The delimiter ';' is used for this task and, as its duties are similar to those of the closing round bracket, it is given the same priority (i.e. one). Like the closing round bracket this delimiter is not stacked and does not appear in the Reverse Polish expression.

Example

$$a + 3 \times (b - c) - i \times (-j/e \uparrow 2 \times (b + 3 \times i) + C);$$

Figure 8 shows how the number of items in the stack fluctuates during the re-ordering of the above arithmetic expression. The extent of the Reverse Polish generated at each delimiter is also shown.

3.2.1.2 Simple Boolean Expressions

In order to translate Boolean expressions the logical and relational operators must be fitted into the priority table.

A relation consists of a simple arithmetic expression on either side of a relational operator (see Revised ALGOL Report section 3.4.1). Thus when the relational operator is to be stacked its priority must be such as to first unstack all delimiters belonging to the arithmetic expression which precedes it. Therefore the relational operators must have a priority less than those of any of the arithmetic operators. The relational operators may all have the same priority.

Similarly a logical operator, which may be preceded by a relation, must unstack any delimiters belonging to the relation before it is itself stacked. As a result the logical operators have priorities lower than those of the relational or arithmetic operators.

However, round brackets may be used to enclose Boolean expressions as well as arithmetic expressions and so the priorities of the opening and closing brackets must remain at zero and one respectively. The closing bracket will then be able to unstack the remains of the part of the arithmetic or Boolean expression contained within the brackets.

The table of priorities is thus expanded to include the operators of a Boolean expression.

Delimiter	*Priority*
(	*0*
$\equiv$)	*1*
$\supset$	*2*
$\vee$	*3*
$\wedge$	*4*

Delimiter	*Priority*
$\neg$	*5*
$> \geq < \leq = \neq$	*6*
$+ -$	*7*
$\times / \div$ *NEG*	*8*
$\uparrow$	*9*

Examples

1. $+A-3>-C\times D/E;$

is translated into the following Reverse Polish

A, *3*, $-$, *C*, *NEG*, *D*, $\times$, *E*, /, $>$ (the unary $+$ is ignored)

2. $(\neg A+B>3 \wedge C) \vee (A-G)=1;$

is re-ordered into

A, *B*, $+$, *3*, $>$, $\neg$, *C*, $\wedge$, *A*, *G*, $-$, *1*, $=$, $\vee$

3.2.1.3 Subscripted Variables

Subscripted variables can be dealt with using the present table of priorities by allowing the opening and closing square brackets to have the same priorities as the corresponding round brackets.

However the delimiter ',' used between the subscript expressions must act as a closing bracket for the expression preceding it and then as an opening bracket for the expression following it. In the first role as a closing bracket it must unstack using a priority of one and never be stacked itself, but in the second role as an opening bracket it must be stacked with a priority of zero without first unstacking.

There are two ways in which this problem can be overcome.

3.2.1.3.1 First Method. The first method treats comma as two separate delimiters and gives it two priorities. One is a 'compare' priority (in this case a priority of one) to control the unstacking of the delimiters of the preceding expression, the other is a 'stack priority' (i.e. zero) with which it may be stacked as if it were an opening bracket. Then at the delimiter ']' the number of commas at the top of the stack give the number of dimensions of the subscripted variable.

Example

$A\,[B-C,\ 2,\ D/E]\times F;$

At the first comma the stack contains

Delimiter	*Stack Priority*
–	*7*
[	*0*

The compare priority of comma (i.e. one) causes the minus sign to be unstacked into the Reverse Polish and the comma is stacked with its stack priority of zero.
At the second comma the stack contains

Delimiter	*Stack Priority*
,	*0*
[	*0*

and the Reverse Polish generated so far is

A, B, C, −, 2

Since the top item of the stack has a priority of zero no unstacking takes place at the second comma. This comma is then stacked. At the closing square bracket the Reverse Polish consists of

A, B, C, −, 2, D, E

and the stack contains

Delimiter	*Stack Priority*
/	*8*
,	*0*
,	*0*
[	*0*

The priority of ']' causes the delimiter '/' to be unstacked into the generated expression and two commas are left in the stack above the corresponding opening bracket. A count is kept of the number of commas that are cleared from the stack, to give the number of dimensions (i.e. the number of commas plus one) for the subscripted variable. The delimiter '[' is then unstacked and discarded.
Finally when the delimiter ';' has been dealt with the completed Reverse Polish expression will consist of

A, B, C, −, 2, D, E, /,], F, ×

No elements of the expression remain in the stack.

The delimiter ']' appears in the Reverse Polish to indicate that the results calculated for the preceding subscript expressions have to be processed to enable the correct element of array *A* to be delivered to the expression. In the object program the operation which is generated for this task is *INDR*. The number of dimensions could be given as a parameter to the *INDR* oper-

ation for use by the Control Routine, but in fact in the Whetstone Compiler the number of dimensions is determined dynamically (see section 2.3.2).

3.2.1.3.2 Second Method. The second method allows comma to have only one priority (namely the compare priority of the first method) and does not require that the comma be stacked. After the delimiters have been unstacked under the control of the priority of one, the top of the stack must be the opening square bracket. This is left in the stack and translation proceeds for the expression following the comma. Thus the opening square bracket is made to act as the opening bracket for each of the expressions of the subscripted variable.

A disadvantage of this method is that at the closing square bracket there is no way of knowing how many dimensions the subscripted variable contains. However, this can be overcome by storing a counter along with the stacked opening square bracket and updating the counter every time this delimiter rises to the top of the stack because of the unstacking by a comma. Then at the closing square bracket the counter will contain the number of dimensions.

Example

$A\,[B - C,\ 2,\ D/E] \times F;$

The delimiter '[' is stacked along with its priority of zero and a dimension counter set at one.
At the first comma the stack contains

Delimiter	*Stack Priority*
–	*7*
[, *1*	*0*

After this comma has been dealt with the stack contains

Delimiter	*Stack Priority*
[, *2*	*0*

After the second comma the stack contains

Delimiter	*Stack Priority*
[, *3*	*0*

and the reverse Polish generated is

A, B, C, –, 2

At the closing square bracket the stack contains

Delimiter	*Stack Priority*
/	*8*
[, *3*	*0*

As before, the delimiter '/' is unstacked using the priority of the closing

bracket. The top of the stack contains '[' and its dimension counter, which is used to check the number of dimensions. This delimiter is then unstacked and discarded.
Finally the conversion of the expression is completed by the use of the delimiter ';' and the Reverse Polish generated is

$$A, B, C, -, 2, D, E, /,], F, \times$$

This second method has been chosen for the Whetstone Compiler. The reason for this choice is to help the Translator to identify the particular use to which the comma is being put. In ALGOL a comma has many uses and so if, after unstacking, the item at the top of the stack turns out to be another comma the preceding expression could have been a subscript expression, an actual parameter expression or a for list element. However, if the top of the stack was the delimiter '[' the preceding expression could only have been a subscript expression.

A subscripted variable can be used in a designational, Boolean or arithmetic expression. In each case the conversion of the surrounding expression is suspended whilst the arithmetic expressions forming its subscripts are converted. At the closing square bracket the conversion of the surrounding expression is resumed. For checking purposes the Translator needs information regarding the type of this surrounding expression. This information must be preserved, when the conversion of this expression is suspended, for use when the subscripted variable has been dealt with. To do this the state variable *TYPE* is used. This state variable contains details of the type of the expression that is being converted. It is stacked along with the delimiter '[' and then reset to arithmetic for the conversion of the subscript expression. Then at the delimiter ']' the value of *TYPE* stacked with the delimiter '[' (now at the top of the stack) is returned to the state variable.

Example

$$A > B[C, D - E, 2] \wedge F;$$

the delimiter '[' will be stacked with the value of the state variable *TYPE* (i.e. algebraic) and the counter set initially at one. The counter will be updated at each comma and so at the closing square bracket the stack contains

Delimiter	*Stack Priority*
[, algebraic, *3*	*0*
$>$	*6*

The delimiter '[' is then unstacked and *TYPE* reset from the value stacked with this delimiter. The counter is used to check the number of dimensions at this use of the array identifier *B* against details of previous uses of *B*, given in the name list entry for *B*. The Reverse Polish is

$$A, B, C, D, E, -, 2,], >, F, \wedge$$

3.2.1.4 Conditional Expressions

The algebraic or arithmetic expressions considered so far can be further complicated, as conditional expressions are allowed in ALGOL. The table of priorities must therefore be expanded to include the delimiters **if, then,** and **else.**

The delimiter **if** acts as an opening bracket for the Boolean expression following it and so **if** can be stacked with the same priority as the other opening brackets, namely zero. Since the conversion of the surrounding expression is suspended whilst this algebraic expression is dealt with the current value of *TYPE* is stacked with **if.** *TYPE* is then set to algebraic.

The closing bracket for the algebraic expression is **then** but this delimiter is also the opening bracket for the expression following it. The delimiter **then** plays a dual role similar to comma and the two methods discussed earlier to deal with this problem are again considered to decide the one suitable here. The one adopted for **then** is that which allows this delimiter to have two priorities. The compare priority controls the unstacking, leaving **if** at the top of the stack. *TYPE* is reset from the value stacked with **if,** since the expression following **then** must be of the same type as that which preceded **if;** the top of the stack is replaced by **then** and its stack priority.

Similarly the delimiter **else** has a dual role as the closing bracket of one expression and the opening bracket of another. It is given two priorities and after unstacking under the control of its compare priority the top of the stack must be **then** which is replaced by **else** along with its stack priority. The delimiter **then** must clear from the stack the remains of the preceding algebraic expression and this expression could have been conditional. The compare priority of **then** cannot therefore be greater than the stack priority of **else.** This means that the compare priority of **else** has to be two and that the priorities of the arithmetic, relational and logical operators have to be increased. The table of priorities now becomes

Delimiter	*Stack Priority*	*Compare Priority*
[(	*0*	void
if	*0*	void
then	*0*	*1*
else	*1*	*2*
]) , ;	void	*1*
$\equiv$	*3*	*3*
$\supset$	*4*	*4*
$\vee$	*5*	*5*
$\wedge$	*6*	*6*
$\neg$	*7*	*7*
$> \geq < \leq = \neq$	*8*	*8*
$+ -$	*9*	*9*
$\times / \div$ *NEG*	*10*	*10*
$\uparrow$	*11*	*11*

The delimiters '[' and '(' have no compare priority as they are stacked without

first clearing anything from the stack. Similarly the delimiters ']', ')', ';' and comma do not need a stack priority as they are never stacked. The conditional expression uses the value of the algebraic expression (between **if** and **then**) to choose one of two expressions, namely the one between **then** and **else** or that between **else** and the delimiter at the end of the conditional expression. It is not permissible to omit the delimiter **else** from a conditional expression.

Example

$y + (\textbf{if } a \geq 0 \textbf{ then } 3)$;

This is not legal ALGOL and in fact for negative values of a the following incomplete expression would result

$y +$;

By allotting a stack priority of zero to **then**, the delimiter at the end of the conditional expression is not able to unstack it and so the illegal ALGOL shown in the above example will be discovered. However, the 'end of expression' delimiter must be able to unstack **else** and for this reason the stack priority of **else** must be at least equal to one.

The algebraic expression situated between **if** and **then** will have as its value either **true** or **false**. If its value is **true** the expression following **then** must be used and the expression following **else** bypassed. Similarly if its value is **false** the expression following **then** is bypassed and that following **else** used.

Example

$y + (\textbf{if } a > 0 \textbf{ then } 3 \textbf{ else } a)$;

At the delimiter **if** the contents of the stack are

Delimiter	*Stack Priority*
(	*0*
+	*9*

The delimiter **if** is added to the stack along with the current value of *TYPE* (i.e. arithmetic) and the stack priority of zero. At the delimiter **then** the stack contains

Delimiter	*Stack Priority*
>	*8*
if, arithmetic	*0*
(	*0*
+	*9*

The compare priority of **then** (i.e. one) causes the relational operator '>' to be unstacked into the Reverse Polish, which then consists of

$y, a, 0, >$

The delimiter **if** is unstacked and *TYPE* reset from the value preserved at **if**.

The delimiter **then** is placed in the stack. An operation *IFJ* (If False Jump) must now be added to the Reverse Polish expression to cause the Reverse Polish expression following to be bypassed should the preceding algebraic expression have the value **false**. It is not possible to complete the operation *IFJ* as the extent of the expression between **then** and **else** is not known. Thus an 'incomplete operation' *IFJ* is generated at this point and the program counter is stacked with the delimiter **then** as a record of the obligation to complete the operation later. The Reverse Polish will then consist of

0	*y*
1	*a*
2	*0*
3	>
4	*IFJ* ( )
5	...

Note: The position of each of the constituents of the Reverse Polish is shown and no attempt is made here to allow for these constituents to vary in size.

At the delimiter **else** the stack contains

Delimiter	*Stack Priority*
then, *4*	*0*
(	*0*
+	*9*

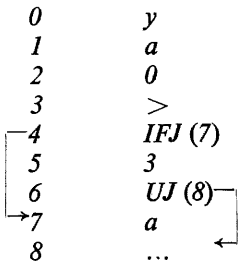
The Reverse Polish generated is

0	*y*
1	*a*
2	*0*
3	>
4	*IFJ* ( )
5	*3*
6	...

The compare priority of **else** will not cause any unstacking for this example as **then** is already at the top of the stack. Similarly, the size of the expression following **else** is not known and so an incomplete *UJ* (Unconditional Jump) operation is generated and a reminder to complete it will be stacked along with the delimiter **else**. The *IFJ* operation at the program position preserved in the stack with **then** is completed to the current program position and the delimiter **then** unstacked. The Reverse Polish is

```
0       y
1       a
2       0
3       >
4       IFJ (7)
5       3
6       UJ ( )
7       ...
```

The delimiter **else** is stacked with its stack priority of one and the program position of the *UJ* operation (i.e. *6*). The closing round bracket is the closing bracket for the preceding conditional expression, and so, after it has been dealt with, the generation of the Reverse Polish form of the conditional expression will have been completed.
Then the delimiter **else** is unstacked under the control of the compare priority of the closing bracket, and the *UJ* operation at the program position stacked with **else** is completed to the current program position.
Thus when the closing round bracket has been processed the Reverse Polish consists of



The stack contains only the delimiter '+' with its priority of nine.
The delimiter ';' completes the arithmetic expression. When it has been processed the Reverse Polish form of the expression will have been completely generated, as follows

```
0       y
1       a
2       0
3       >
4       IFJ (7)
5       3
6       UJ (8)
7       a
8       +
9       ...
```

The system of using the translator stack to preserve obligations as well as

delimiters allows the most complex conditional expression to be converted without requiring more than the current section of the ALGOL to be present.

Example

y + (**if if** *a* > *0* **then** *b* = *1* **else** *c* = *1* **then** *3* **else if** *a* > *1* **then** *4* **else** *5*);

Here the value *3* or the value of the conditional expression '**if** *a* > *1* **then** *4* **else** *5*' is added to *y* depending on whether the conditional algebraic expression '**if** *a* > *0* **then** *b* = *1* **else** *c* = *1*' is **true** or **false.**
At the first **then** the stack contains

Delimiter	*Stack Priority*
>	*8*
if, algebraic	*0*
if, arithmetic	*0*
(	*0*
+	*9*

When this delimiter has been processed the Reverse Polish consists of

0	*y*
1	*a*
2	*0*
3	>
4	*IFJ* ( )
5	...

and the stack contains

Delimiter	*Stack Priority*
then, *4*	*0*
if, arithmetic	*0*
(	*0*
+	*9*

At the second **then** the stack contains

Delimiter	*Stack Priority*
=	*8*
else, *8*	*1*
if, arithmetic	*0*
(	*0*
+	*9*

When this delimiter has been processed the Reverse Polish for the conditional algebraic expression will have been generated as follows.

0	*y*
1	*a*
2	*0*
3	>
4	*IFJ (9)*
5	*b*
6	*1*
7	=
8	*UJ (12)*
9	*c*
10	*1*
11	=
12	*IFJ ( )*
13	...

The stack will then contain

Delimiter	*Stack Priority*
then, *12*	*0*
(	*0*
+	*9*

At the last use of **if** the following items will have been added to the Reverse Polish.

13	*3*
14	*UJ ( )*

The delimiter **if** is stacked and the stack will then contain

Delimiter	*Stack Priority*
if, arithmetic	*0*
else, *14*	*1*
(	*0*
+	*9*

When the last **then** has been processed the following items will have been added to the Reverse Polish

15	*a*
16	*1*
17	>
18	*IFJ ( )*

the stack contains

Delimiter	*Stack Priority*
then, *18*	*0*
else, *14*	*1*
(	*0*
+	*9*

When the last **else** has been processed the stack will contain

Delimiter	*Stack Priority*
else, *20*	*1*
else, *14*	*1*
(	*0*
+	*9*

The *IFJ* operation at position *18* in the Reverse Polish will have been completed and the following items added

19	*4*
20	*UJ* ( )

The closing round bracket acts as a closing bracket for the two conditional expressions. The compare priority of this delimiter causes both **else** delimiters to be unstacked and their corresponding incomplete *UJ* operation filled in with the current program position. Finally the delimiter ';' completes the arithmetic expression and when this delimiter has been processed the complete Reverse Polish will have been generated and is as follows

0	*y*
1	*a*
2	*0*
3	>
4	*IFJ* (*9*)
5	*b*
6	*1*
7	=
8	*UJ* (*12*)
9	*c*
10	*1*
11	=
12	*IFJ* (*15*)
13	*3*
14	*UJ* (*22*)
15	*a*
16	*1*
17	>
18	*IFJ* (*21*)
19	*4*
20	*UJ* (*22*)
21	*5*
22	+
23	...

Figure 9 shows the generation of the Reverse Polish and the contents of the stack at each stage of the re-ordering of the conditional expression in this example.

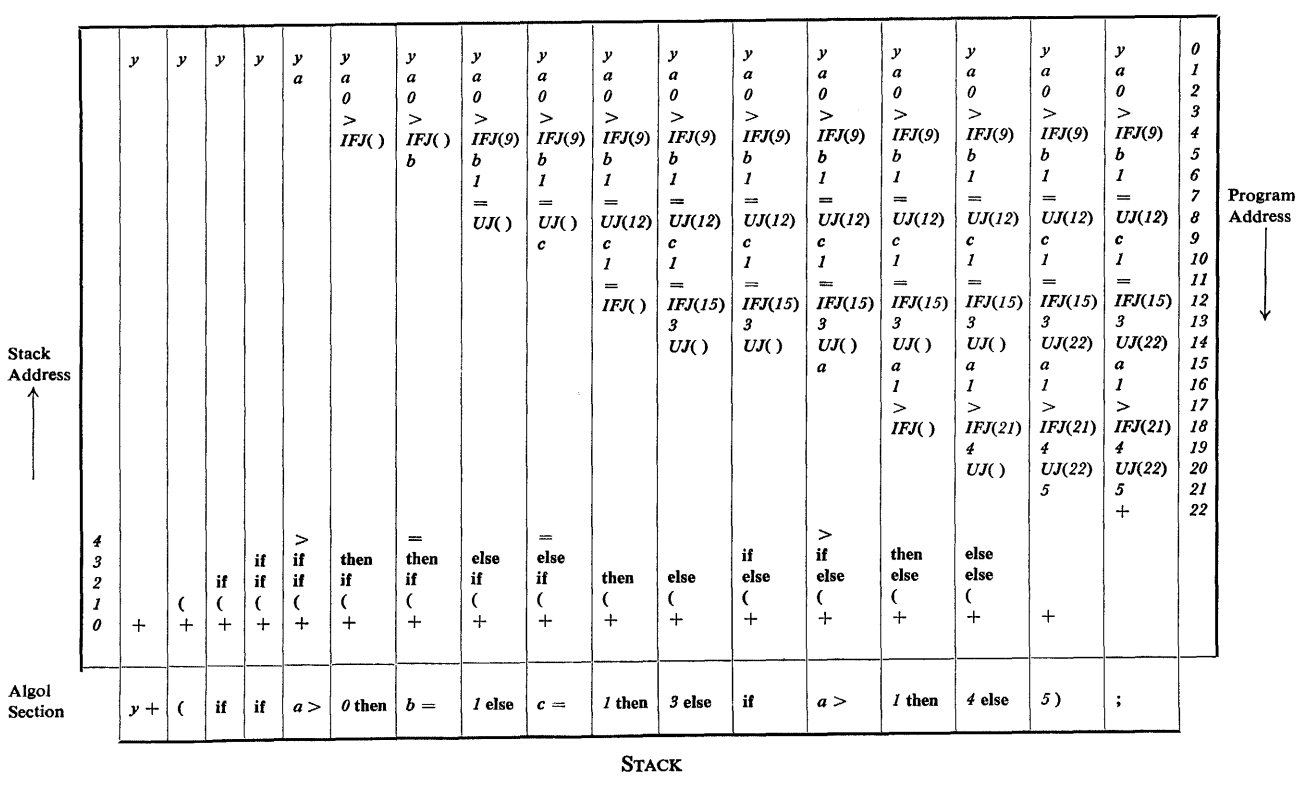


FIG. 9. Stylised Representation of Translator Stack and Reverse Polish.

3.2.2 Translation of Statements

This section illustrates how the translator stack and the system of priorities can also be used to translate statements. The translation of procedure statements and for statements are not considered in this section but are dealt with in sections 3.4.3 and 3.4.4 respectively.

3.2.2.1 Assignment Statements

The value of the expression on the right hand side of the delimiter ':=' is assigned to the simple or subscripted variable on the left hand side.

The state variable E is set to statement level for the translation of the left hand side to ensure that object program operations of the form 'Take Address' rather than 'Take Result' are generated for the simple or subscripted variable preceding the delimiter ':='. The state variable E is then set to expression level for the translation of the right hand side.

The system of priorities developed for the translation of expressions can be used for the translation of an assignment statement by including the delimiter ':=' in the table of priorities. The priority of ':=' must be such that it is unstacked only by the 'end of statement' delimiter (i.e. the delimiter ';'). This is achieved by giving ':=' a priority of two.

Example 1.

$y := a - 2;$

The Reverse Polish form of this statement is

$y, a, 2, -, :=$

and at the delimiter ';' the stack contains

Delimiter	*Stack Priority*
$-$	*9*
$:=$	*2*

The delimiter ';' with its priority of one causes these two items to be unstacked into the Reverse Polish.

Example 2.

$A[j, k] := 0;$

At the delimiter ']' the stack contains

Delimiter	*Stack Priority*
$[, 2$	*0*

and the Reverse Polish consists of

A, j, k

The delimiter '[' is unstacked after checking the number of dimensions of the array identifier *A*. The delimiter ']' is then added to the Reverse Polish. In fact the operation *INDA* is used for this delimiter in the object program whenever the subscripted variable appears on the left hand side of an assignment statement. At the delimiter ':=' the stack does not contain any elements of the statement to be unstacked before the delimiter ':=' is stacked with its priority of two.
At the delimiter ';' the delimiter ':=' is unstacked and the Reverse Polish generated is

A, j, k,], 0, :=

3.2.2.1.1 Multi-Assignment Statements. Multi-assignments are allowed in ALGOL and the translation of these poses a slight problem which is solved by means of the translator stack.

Example

a := A := 0;

Until the delimiter after the identifier *A* is reached it is not known whether this identifier is in the left part list, so that its address is required, or is on the right hand side, so that its value is required.

This presents no problem in the above example where *A* is a simple variable since '*A*:=' forms one section of the ALGOL program.
However, this is not the case when *A* is a subscripted variable.

Example

a := A [j, k] := 0

At the delimiter ']' the current section of the ALGOL contains this delimiter and the preceding identifier *k*. The stack contains

Delimiter	*Stack Priority*
[, 2	*0*
:=	*2*

The delimiter '[' is unstacked and the number of dimensions checked. Then the delimiter ']' is added to the Reverse Polish. However, it is not known until the delimiter following the closing square bracket is reached whether the object program operation for ']' should be *INDA* or *INDR*. If the delimiter following is ':=' *INDA* must be generated otherwise *INDR* is required. As a result neither *INDA* nor *INDR* is generated at this point but *INDA* is stacked with a high priority (i.e. twelve) as a reminder to the delimiter following to generate the correct operation. For this reason the delimiter ':=' is given a compare priority of twelve.

The Reverse Polish resulting from the example given above is

a, A, j, k,], 0, :=, :=

Two object program operations *ST* and *STA* correspond to the delimiter ':=' in the Reverse Polish. The operation *ST* is used for the last of the sequence of ':=' delimiters and *STA* for the others. Thus before stacking the delimiter ':=' the top of the stack is inspected and if it is ':=' the current delimiter is stacked as *STA*, otherwise as *ST*.

Example

The contents of the stack at the delimiter ';' in the example above are

Delimiter	*Stack Priority*
:= (*STA*)	*2*
:= (*ST*)	*2*

3.2.2.2 Go To Statements

A go to statement consists of the delimiter **go to** followed by a designational expression, but the Reverse Polish form required by the object program has the **go to** delimiter after the converted form of the expression. To achieve this re-ordering the stack priority of **go to** must be sufficiently low for it to remain in the stack throughout the designational expression following, and for it to be unstacked by the 'end of statement' delimiter. Thus **go to** is given a stack priority of two to enable it to act as the opening bracket of the designational expression.

Example

go to if *b* **then** *L1* **else** *L2*;

The Reverse Polish form of this statement is

0	*b*
1	*IFJ (4)*
2	*L1*
3	*UJ (5)*
4	*L2*
5	**go to**
6	...

3.2.2.3 Conditional Statements

The earlier section on the translation of conditional expressions has shown how ALGOL uses an if clause (i.e. **if** ⟨Boolean expression⟩ **then**) to decide which of two expressions shall be obeyed at run time. Similarly an if clause can be allowed to decide which one of two statements shall be obeyed. It can

also precede one statement and so decide whether this statement shall be obeyed or bypassed. This means that the delimiter **else** may be omitted from a conditional statement, thus forming an if statement.

Example

if $a > 0$ **then** $x := 0$ **else** $x := 1$;
if $a \geq 0$ **then** $x := 0$;

Both of these statements are allowed in ALGOL. The second statement acts as a dummy statement for negative values of *a*.

The end of statement delimiter ';' may therefore expect to find either **else** or **then** in the stack; it should be able to unstack **then.** For this reason the stack priority of zero given to the delimiter **then** used in conditional expressions will not work here. A way out of this difficulty is to have two internal representations for the delimiter **then**, namely '**then** *E*' used for conditional expressions and '**then** *S*' for conditional statements. Similarly two representations of **else**, namely '**else** *E*' and '**else** *S*', are used. The table of priorities given in section 3.2.1.4 refers to '**then** *E*' and '**else** *E*'.

The delimiter **then** must be able to unstack back to its corresponding **if** through an algebraic expression which might be conditional. Thus the compare priority of '**then** *S*' has to be less than or equal to the stack priority '**else** *E*' (i.e. one). Similarly the unconditional statement between **then** and **else** may contain a conditional expression. Therefore the compare priority of '**else** *S*' must be less than or equal to the stack priority of '**else** *E*'. But the stack priority of '**then** *S*' must be less than the compare priority of '**else** *S*', as **else** should be able to unstack down to, but not including, its corresponding **then.**

To fit in the delimiters '**then** *S*' and '**else** *S*' to the existing table of priorities and still obey the rules given above is not possible and so the priorities of the delimiters are revised and given below.

Delimiter	*Stack Priority*	*Compare Priority*
[(	*0*	void
] , ;	void	*1*
)	void	*2*
if	*0*	void
then *S*	*1*	*2*
then *E*	*0*	*2*
else *S*	*1*	*2*
else *E*	*2*	*2*
go to	*2*	void
:=	*2*	*12*
$\equiv$	*3*	*3*
$\supset$	*4*	*4*
$\vee$	*5*	*5*

Delimiter	*Stack Priority*	*Compare Priority*
$\wedge$	*6*	*6*
$\neg$	*7*	*7*
$> \geq < \leq = \neq$	*8*	*8*
$+ -$	*9*	*9*
$\times / \div$ *NEG*	*10*	*10*
$\uparrow$	*11*	*11*

The compare priority of the closing round bracket has been increased to two as round brackets may enclose only expressions, and ')' should not be able to unstack '**then** *S*' or '**else** *S*'. Normally the state variable *E* (which distinguishes between statement and expression levels) is used to disallow the use of '**then** *S*' or '**else** *S*' whilst in an expression, but *E* is set to statement level whilst an actual parameter list is being converted. This is because the object program requires operations for finding the address of parameters which are subscripted variables, rather than their values.

Example

P (*A* [*I*])

The actual parameter is converted into

A, *I*,]

Since the state variable *E* is set to statement level the delimiter ']' will appear in the object program as *INDA*.

The translation of conditional statements follows a similar pattern to that of conditional expressions. The priorities allotted to each of the delimiters ensure that the ALGOL is re-ordered into the correct Reverse Polish form.

3.2.2.4 Compound Statements

Several statements may be enclosed by the brackets **begin** and **end**. By this means they appear to the surrounding ALGOL as a single statement. This can be useful if, for instance, one of two sets of statements is to be executed.

Example

```
if a > 0 then begin x := 0;
                    y := 1
              end
         else begin y := 0;
                    x := 1
              end;
```

The Translator treats these statement brackets in a similar way to arithmetic round brackets, using them to re-order the conditional statements

into the correct Reverse Polish form without putting them into the object program. The delimiter **begin** does not cause any items to be unstacked (i.e. it has no compare priority). It is stacked with a priority of zero and it shields the delimiter '**then** *S*' below.

The last statement in a compound statement does not have to be followed by a semi-colon. Thus the delimiter **end** takes over the duties of semi-colon for this statement and is given a compare priority of one.

3.3 NAME LIST

The name list contains the names and details of all the identifiers with a currently valid declaration. This list is divided into blocks so that when the end of a block is reached the entries for identifiers declared within the block can be discarded. Before being discarded a copy is preserved in a backing store for use by the Control Routine, should a failure occur in the object program at run time.

There is one entry in the part of the name list corresponding to the current block for each identifier appearing in the block.

Example 1.

```
begin real a;
      a := 0;
      go to L;
      begin integer b;
            b := a + 1;
            go to M
      end;
M: L: ...
```

Before processing the delimiter **end** the name list will contain

no	*name*
1	*a*
2	*L*
3	*b*
4	*a*
5	*M*

At the occurrence of an identifier the name list is searched for an entry corresponding to this identifier in order to obtain details concerning its declaration or previous use. However, only the part of the name list appertaining to the current block is searched. This is because at the use of an identifier it cannot be assumed to be non-local merely because its declaration has not yet been met.

Example 2.

```
L: x := 0;
   begin real a;
         go to L;
         ...
   end
```

At the statement '**go to** *L*' it is not known whether the label *L* refers to the label preceding the statement '*x* := *0*' in the outer block or whether *L* will label a statement further on in the inner block. (The appearance of a label preceding a statement and separated from it by a colon is said to be the 'declaration' of the label.)

There are two types of entries in the name list. One is a 'declared entry', showing that the identifier has been declared in the current block and giving details of the declaration. The other is a 'used entry'; this shows that an identifier has appeared, but has not been declared, in the current block and gives information gleaned from the various uses of the identifier.

Example

The contents of the name list at the delimiter **end** in Example 1 above are shown again with the type of each entry marked.

no	*name*	*used/declared*
1	*a*	(declared)
2	*L*	(used)
3	*b*	(declared)
4	*a*	(used)
5	*M*	(used)

The information giving the type of the entry is stored in a part of the name list called the '*d* column'. For a used entry this is set to zero and for a declared entry it is set to one.

A used entry in the name list will also contain the program addresses of the start and finish of the chain of skeleton operations associated with the uses of the identifier.

Example

```
begin real x, y;
        ...
    begin integer i;
        ...
        y := 2;
        x := i + 1;
        y := x × x + y
    end
```

The object program produced for the assignment statements is given below. For comparison the corresponding Reverse Polish is given alongside.

Reverse Polish	*Syll*	*Op*	*Par*	*Remarks*
y	*50*	*AO*	( )	
2	*53*	*TIC*	*'2'*	(assuming the program has
:=	*60*	*ST*		reached *50* at this point and
x	*61*	*AO*	( )	that the level of the inner
i	*64*	*TIR*	*(2,3)*	block is two)
1	*67*	*TIC1*		
+	*68*	+		
:=	*69*	*ST*		
y	*70*	*AO*	*(51)*	The chain links the first
x	*73*	*RO*	*(62)*	syllable of the parameters
x	*76*	*RO*	*(74)*	of the skeleton operations
×	*79*	×		
y	*80*	*RO*	*(71)*	
+	*83*	+		
:=	*84*	*ST*		
	85	...		

The chain of skeleton operations for *x* starts at program syllable *77* which contains *74*, the address of the next link in the chain. This chain finishes at syllable *62*. Similarly the chain for *y* starts at syllable *81* and finishes at syllable *51*.

The name list for the inner block will contain

no	*name*	*type*	*d*	*np*	*start*	*finish*
1	*i*	**integer**	*1*	*2,3*		
2	*y*	arithmetic	*0*		*81*	*51*
3	*x*	arithmetic	*0*		*77*	*62*

The information gleaned from the various uses of the identifiers *x* and *y* show that they must be arithmetic (i.e. either a scalar or a function designator with zero parameters and of type **real** or **integer**). The declared entry for *i* gives details of the block level *n*, and *p*, the position reserved for *i* in the first order working storage of the block. These are packed together and stored in the '*np*' column.

3.3.1 Declaration of an Identifier

At the declaration of an identifier the current part of the name list is searched for an entry corresponding to the identifier.

(*i*) If no entry exists a declared entry is created and details concerning the declaration are added to the entry.

(*ii*) If a used entry exists its type information, which shows the expected type of declaration, is checked. It is also necessary to check that the identifier has not been used in the bound expression of an array declaration in the current block as this would indicate that a local variable had been used in an array declaration (see Revised ALGOL Report, section 5.2.4.2). To do this a marker called the '*exp*' column is incorporated into used

entries in the name list and this marker is set if the identifier is used in a bound expression of an array declaration.

Example

begin array *A* [*1*: *n*];
integer *n*;

When the declaration of *n* is reached the current part of the name list contains a used entry for *n* and this entry will have the *exp* marker set. Thus a failure indication is given.

If the used entry is compatible with the declaration of the identifier the used entry is changed into a declared entry by setting the *d* column marker, and then the *np* column is filled in.

The chain of skeleton operations associated with the uses of the identifier must then be followed through and the skeleton operations replaced by the appropriate object program operations. To do this the program address of the start of the chain is taken from the name list. The address, if any, given in the parameter position of this skeleton operation, which points to the next skeleton operation in the chain, is noted. The operation and its parameter are then replaced by the appropriate object program operation. This process is repeated for each skeleton operation in the chain. An example of this process of 'unchaining' is given in section 3.3.3.

(*iii*) If a declared entry already exists a failure indication is given as it indicates that the identifier has already been declared in the current block.

3.3.2 Use of an Identifier

When an identifier is used the current part of the name list is searched.

(*i*) If no entry exists a used entry is created. Details of the expected type of the identifier are put in the *type* column of this entry. The program address of the skeleton operation generated for this use of the identifier is placed in the position allotted for the start and finish of the chain of skeleton operations. The 'chain' consists of only one element and the skeleton operation just generated lies at both the start and finish of the chain.

Example

if *B* **then**

The expected type of *B* is **Boolean** and the skeleton operation generated is

Syll	*Op*	*Par*
50	*RO*	( )

The used entry created for *B* is

no	*name*	*type*	*d*	*start*	*finish*	*exp*
1	*B*	**Boolean**	*0*	*51*	*51*	

In this case the type of the identifier has been restricted to Boolean although it is still not known whether *B* is a Boolean variable or a Boolean procedure with no parameters. (This has been discussed in more detail in section 3.1.1.1.)
If the identifier is being used in a subscript bound expression of an array declaration the *exp* column of the used entry is set to one. Then if the identifier later proves to be local a failure can be given.

(*ii*) If a declared entry is found, the type of use of the identifier is checked against the declaration information contained in the entry. The appropriate operation is generated in the object program. However, if the identifier is being used in an array declaration a failure must be indicated since a local variable cannot be used in an array declaration (see Revised Report section 5.2.4.2).

Example

```
procedure A (B); real B;
begin integer i;
      array F [1: B, 1: i];
      . . .
```

It is allowed to use *B* in the declaration of array *F*, but a failure is indicated when *i* is reached.

(*iii*) If a used entry is found the type of the current use of the identifier is checked against the information contained in the entry, which has been gleaned from previous uses. Also, if the current use increases the information concerning the expected type of the identifier this extra information is placed in the *type* column of the entry. Thus the *type* column of a used entry contains the 'logical sum' of all information concerning the various uses of the identifier. A skeleton operation is generated and added to the chain. This is done by giving in its parameter position the address given in the chain start position of the entry. The address of the parameter of this new skeleton operation is then stored in the chain start position of the name list.

Example

```
÷ x;
```

The name list contains a used entry for *x* as follows

no	*name*	*type*	*d*	*np*	*start*	*finish*	*exp*
1	*x*	arithmetic	*0*		*63*	*14*	

Now the expected type of this identifier can be further restricted from arithmetic (i.e. real or integer) to integer.

The skeleton operation generated for this use is

Syll	*Op*	*Par*
112	*RO*	*(63)*

The name list entry for x is changed to

no	*name*	*type*	*d*	*np*	*start*	*finish*	*exp*
1	*x*	**integer**	*0*		*113*	*14*	

3.3.3 The End of a Block

The declared entries in the part of the name list corresponding to the current block are deleted from the name list. They refer to local identifiers, and since the block is about to be left these identifiers have no further significance in the program.

The used entries in this part of the name list refer to non-local identifiers. These are dealt with in turn by searching the name list for the surrounding block for a corresponding entry as follows.

(*i*) If there is no entry in the name list of the surrounding block, the used entry is transferred to this part of the name list. The *exp* column of the used entry is set to zero, if necessary, before transferring it. The identifier has been shown to be non-local and so there is no further need for the marker denoting that the identifier has been used in a subscripted bound expression. In fact if the marker is left in the used entry and the identifier turns out to be local to the surrounding block a failure would be given.

Example

```
begin integer i, j;
    . . .
    begin real n;
        . . .
        begin integer procedure ADD;
                    ADD := i + j;
            array B [1:j];
            . . .
        end
    end
end
```

When the array declaration has been translated the name list contains the following.

no	*name*	*type*	*d*	*np*	*syll*	*dim*	*exp*
1	*i*	i	*1*	*1,3*			
2	*j*	i	*1*	*1,4*			
3	*n*	r	*1*	*2,3*			
4	*ADD*	i p	*1*	*4,0*	*50*	*0*	
5	*i*	arith	*0*	*68*	*68*		
6	*j*	arith	*0*	*78*	*71*		*1*
7	*B*	r a	*1*	*3,3*		*1*	

It should be noted that two extra items of information have been introduced for the name list entries given above. The entry for the procedure identifier *ADD* required a '*syll*' column in which is stored the syllable address of the object program for the start of the procedure body. The second extra item of information is the '*dim*' column which is required for both the procedure identifier *ADD* and the array identifier *B* (it is also required for a switch identifier). In this part of the name list the number of dimensions (in the case of an array or switch identifier) or the number of parameters (in the case of a procedure identifier) is stored. At each use of an array identifier as a subscripted variable or each explicit call of a procedure the number of dimensions or parameters is checked. Entry number *5* has *np* and *syll* columns set to *68*. In fact the *start* and *finish* columns used in earlier examples are combined with the *np* and *syll* columns.

This is possible because used entries do not require *np* and *syll* columns and declared entries do not require *start* and *finish* columns. Thus the start and finish of the chain (consisting of a single operation, at syllable *67*, say) for *i* are given in the *np* and *syll* columns respectively of its name list entry.

For convenience, entries in the *type* columns have been abbreviated as follows

type	*type column*	
integer	i	
real	r	
procedure	p	
array	a	
arithmetic	arith	(i.e. **real** or **integer**)

Other possible abbreviations are

type	*type column*	
Boolean	B	
string	st	
switch	sw	
label	l	
algebraic	alg	(i.e. **real**, **integer** or **Boolean**)

When the inner block is left, the corresponding part of the name list is collapsed. The declared entries for *ADD* and *B* are discarded; the used entries for *i* and *j* are added to the name list of the surrounding block after setting the *exp* column of the entry for *j* to zero. The name list will then contain

no	*name*	*type*	*d*	*np*	*syll*	*dim*	*exp*
1	*i*	i	*1*	*1,3*			
2	*j*	i	*1*	*1,4*			
3	*n*	r	*1*	*2,3*			
4	*i*	arith	*0*	*68*	*68*		
5	*j*	arith	*0*	*78*	*71*		

(*ii*) If a declared entry exists in the name list for the surrounding block, its information is used to replace the skeleton operations in the chain corresponding to the used entry in the current block. The *type* columns of the declared and used entries are checked for compatibility.
The way in which the chain is followed through has been described in section 3.3.1. However, in this case the *exp* column of the used entry can be ignored as the identifier is non-local and can thus be used in a subscript bound expression.

Example

```
begin real x;
      begin real i;
            x := i := 1;
            x := x + i
      end;
      ...
```

Before processing the delimiter **end** the name list contains

no	*name*	*type*	*d*	*np*	*syll*	*dim*	*exp*
1	*x*	r	*1*	*1,3*			
2	*i*	r	*1*	*2,3*			
3	*x*	arith	*0*	*27*	*15*		

and the object program that has been generated is

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	Unconditional jump around outer block
4	*BE*	( )	
7	*CBL*		
8	*UJ*	( )	Unconditional jump around inner block
11	*BE*	( )	
14	*AO*	( )	*x*
17	*TRA*	(*2,3*)	*i*
20	*TIC1*		
21	*STA*		*i* := *1*
22	*ST*		*x* := *1*
23	*AO*	(*15*)	*x*
26	*RO*	(*24*)	*x*
29	*TRR*	(*2,3*)	*i*
32	+		
33	*ST*		*x* := *x* + *i*
34	...		

It will be noted that the parameters for the operation *BE* have been left blank. These cannot be filled in until the end of a block, when the extent of the first order storage is known. This is dealt with in more detail later when the translation of blocks is discussed.

The delimiter **end** causes the name list for the inner block to be collapsed as follows.

(a) The declared entry for *i* is discarded.

(b) The name list for the surrounding block is searched for an entry for the identifier *x*. In this case a declared entry is found and the information contained in it is checked against that contained in the corresponding used entry for *x* in the inner block.

Using the information taken from the *type* and *np* columns of the declared entry the chain of skeleton operations is followed through. Each element of the chain is checked before being replaced to ensure that each use of the identifier is compatible with the type information now available. The used entry gives the address of the start of the chain as *27*. This points to the parameter of the first skeleton operation in the chain, which in turn points to the parameter of the next skeleton operation (i.e. *24*). The address of the next element in the chain is noted, then the operation *RO* at address *26* is checked against the type information for *x* and completed as *TRR*. The contents of the *np* column of the declared entry are then stored as the parameter of this operation (i.e. at address *27*). Each element of the chain is completed in a similar fashion until the last element has been dealt with.

When all the skeleton operations for *x* have been filled in the object program will be

Syll	*Op*	*Par*
0	*CBL*	
1	*UJ*	( )
4	*BE*	( )
7	*CBL*	
8	*UJ*	( )
11	*BE*	( )
14	*TRA*	(*1,3*)
17	*TRA*	(*2,3*)
20	*TIC1*	
21	*STA*	
22	*ST*	
23	*TRA*	(*1,3*)
26	*TRR*	(*1,3*)
29	*TRR*	(*2,3*)
32	+	
33	*ST*	
34	...	

(*iii*) If a used entry exists in the name list for the surrounding block the two used entries are checked for compatibility and then the two separate chains (each used entry has a chain associated with it) are combined into one chain.

Example

```
begin real x;
      x := y × y;
      begin integer i;
            y := y ÷ 2;
            . . .
      end;
      . . .
```

At the delimiter **end** the name list contains the following entries for x, y and i

no	*name*	*type*	*d*	*np*	*syll*	*dim*	*exp*
1	*x*	r	*1*	*1,3*			
2	*y*	arith	*0*	*57*	*54*		
3	*i*	i	*1*	*2,3*			
4	*y*	i	*0*	*65*	*62*		

The chain of operations for the entry for y in the current block is

Syll	*Op*	*Par*
61	*AO*	( )
64	*RO*	(*62*)

The chain of operations for the entry for y in the surrounding block is

Syll	*Op*	*Par*
53	*RO*	( )
56	*RO*	(*54*)

After combining the two chains the single chain for y is

Syll	*Op*	*Par*
53	*RO*	( )
56	*RO*	(*54*)
61	*AO*	(*57*)
64	*RO*	(*62*)

To do this, the information given in the *np* column of the entry for y for the surrounding block is placed in the parameter at the program address given in the *syll* column of the entry for y for the current block.
The contents of the *np* column of the entry for y for the current block are transferred to the corresponding column of the other entry for y.

Thus after the name list for the current block has been collapsed the name list will contain

no	*name*	*type*	*d*	*np*	*syll*	*dim*	*exp*
1	*x*	r	*1*	*1,3*			
2	*y*	i	*0*	*65*	*54*		

The used entry for *y* has its *type* column limited from arithmetic (i.e. real or integer) to **integer** since the use of *y* in the inner block has provided this further information.
It should be noted that by including in the name list entry for *y* the addresses of both the start and finish of the chain, it was possible to combine the two chains without having to follow through one of them to find its finish.

3.3.4 The End of a Program

When the end of the program is reached the only used entries remaining in the name list should be those referring to the standard functions. These identifiers are allowed in an ALGOL program without explicit declaration. The standard functions are used as if they have been declared in an outer block inside which the ALGOL program is contained.

If, however, there are used entries for any other identifiers a failure is given. A possible cause of this type of failure is the omission of a delimiter between two identifiers.

Example

```
begin real procedure P (A, B); value A, B;
                      real A, B;
                      P := A B;
...
```

In the body of the procedure *P* an implied multiplication sign has been used between *A* and *B*. However the Translator assumes that the procedure identifier has been assigned the value of a non-local variable *AB*. It is not until the end of the program that this mistake is revealed, when a used entry for an identifier *AB* will still remain in the name list.

A further item of information is stored in used entries in the name list which is useful when the above type of mistake occurs. This is the '*line*' column which contains the line number of the ALGOL text where the identifier was first used. The *line* column of a declared entry in the name list contains the line number of its declaration, for use at a failure in the object program (see Appendix 5).

3.3.5 Procedure Block

A block is created for a procedure body regardless of whether the body is a block or not. If, however, the procedure body is an unlabelled block there is no need to create an extra block in the object program and so the formal parameters to the procedure and the variables declared local to the body occupy positions in the first order working storage of the same procedure block.

It is possible in ALGOL to re-declare an identifier that has appeared in the formal parameter part as a variable local to the procedure body and thus render the particular formal parameter inaccessible. To allow for this the formal parameters are separated from the local identifiers in the name list, thus effectively creating an extra block in the name list for a procedure body which is an unlabelled block.

Example

```
begin integer i;
      real procedure A (a, b, c); value b, c; real b; integer a, c;
      begin integer i, j; real B;
            i := a + c;
            . . .
      end;
```

After translating the statement '$i := a + c$' inside the procedure body the name list contains.

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*
1	*i*	i	*1*	*1,3*					
2	*A*	r p	*1*	*2,0*	*65*			*3*	
3	*a*	i	*1*	*2,3*		*1*			
4	*b*	r	*1*	*2,5*			*1*		
5	*c*	i	*1*	*2,7*			*1*		
6	*i*	i	*1*	*2,9*					
7	*j*	i	*1*	*2,10*					
8	*B*	r	*1*	*2,11*					
9	*a*	arith	*0*	*75*	*75*				
10	*c*	arith	*0*	*78*	*78*				

The entries for the formal parameters contain extra information to show that they are indeed formal parameters and also to indicate whether they have been called by name or by value. This information is given in two extra columns – the '*f*' column and '*v*' column. The *f* column is set to one for a formal parameter called by name, whilst the *v* column is set to one for a formal parameter called by value. (In the case of a label called by value both the *f* and the *v* columns of its name list entry are set since the object program operation *TFL* is used for a formal label whether called by name or by value – see section 2.5.6.3.) Although the formal

parameters and the local identifiers are treated in the object program as being local to the same block by having the same block level (in this example '2'), the formal parameters are treated as non-local variables during the translation of the procedure body. Then at the end of the procedure body the name list entries for the procedure body and also those for the formal parameters are collapsed.

If the formal parameters were not separated from the local variables in the name list it would be possible to produce an incorrect translation.

Example

```
procedure P (a, b); real a, b;
begin switch S := if a > 0 then L1 else L2, L3;
      integer a;
      . . .
end
```

At the appearance of the identifier *a* in the switch declaration it would be assumed that this was the formal parameter *a* and not the local variable *a* if the current part of the name list contained entries for the formal parameters.

However, it is unlikely that a programmer would wish to render a formal parameter inaccessible by declaring the identifier as a local variable in the body of the procedure and so a warning message is printed when this is found (i.e. when collapsing the name list for the procedure body block). The printing of this warning message does not affect the continuation of the translation of the ALGOL text.

3.3.5.1 Assignment to a Procedure Identifier

If a procedure has been declared to be a type procedure (i.e. a real, integer or Boolean procedure) one or more explicit assignments to the procedure identifier may occur anywhere within the body of the procedure declaration. For the procedure to be called by a function designator at least one such assignment must have been made. The Translator can check whether the procedure identifier has had a value assigned to it, but it cannot check that this assignment is not bypassed at run time.

Example

```
real procedure P;
    if B then P := x + y
```

Here the procedure identifier *P* is assigned the value of the expression '$x + y$' if, and only if, the non-local variable *B* has the value **true**.

The following devices are used to enable the Translator to do as much checking as possible on the legality of function designators.

(*i*) If, at the collapse of the name list for a type procedure block, no assignment has been made to the procedure identifier a failure is given. This means that in the Whetstone Compiler a restriction has been placed on the ALGOL to the extent that a type procedure must be capable of being called by a function designator (see Appendix 2).

(*ii*) An extra item is created as the first item of the part of the name list for a block and this is not set up in the normal way. If the block is a 'procedure block' (i.e. the block which contains the formal parameters, if any) the first item contains a duplicate copy of the entry for the corresponding procedure identifier. For other types of blocks the first item is left blank.

In place of the *line* column for the duplicate entry another piece of information, namely the '*FD*' marker is stored. This marker is set up when an assignment to the procedure identifier occurs. The assignment may occur whilst this block is being translated or during an inner block. For the first a declared entry, namely the duplicate entry, will be found in the name list. In the second case a used entry is either found or created in the name list corresponding to the inner block. Eventually, the procedure block will be regained after collapsing the name list for any inner blocks and the used entry will correspond to the duplicate entry at the head of the procedure block list. This will cause the chain corresponding to the used entry to be followed through. An assignment to the procedure identifier will have been marked by the generation of an *AO* operation. If such an operation is found in the chain the *FD* marker is set.

It should be noted that the special first item of the name list for a block described here has not been shown on pictures of the name list for examples given earlier in this section.

Example

```
begin real procedure A (B, C); value C; real B, C;
    begin real i;
        i := B + C;
        begin real D;
            D := i × i;
            A := D + i
        end
    end;
```

Before processing the delimiter **end** the name list contains

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*A*	r p	*1*	*2,0*	*10*			*2*		*1*
3	*A*	r p	*1*	*2,0*	*10*			*2*		*0*
4	*B*	r	*1*	*2,3*		*1*				*1*

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
5	*C*	r	*1*	*2,5*			*1*			*1*
6										
7	*i*	r	*1*	*2,7*						*2*
8	*B*	arith	*0*	*20*	*20*					*3*
9	*C*	arith	*0*	*23*	*23*					*3*
10										
11	*D*	r	*1*	*3,3*						*4*
12	*i*	arith	*0*	*52*	*38*					*5*
13	*A*	alg	*0*	*46*	*46*					*6*

The object program that has been generated to this point is

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	Jump around outer block
4	*BE*	( )	
7	*UJ*	( )	Jump around procedure body
10	*PE*	( ), *2*	Start of procedure *A*
14	*CA*		
15	*CSR*		
16	*TRA*	(*2,7*)	
19	*RO*	( )	*B*
22	*RO*	( )	*C*
25	+		
26	*ST*		$i := B + C$
27	*CBL*		
28	*UJ*	( )	Jump around inner block
31	*BE*	( )	
34	*TRA*	(*3,3*)	
37	*RO*	( )	*i*
40	*RO*	(*38*)	*i*
43	×		
44	*ST*		$D := i \times i$
45	*AO*	( )	*A*
48	*TRR*	(*3,3*)	
51	*RO*	(*41*)	*i*
54	+		
55	*ST*		$A := D + i$
56	...		

At the collapse of the name list for the inner block the declared entry for *D* is discarded, the used entry for *A* is placed in the name list for the containing block and the used entry for *i*, which corresponds to a declared entry in the containing block, is discarded after unchaining and replacing the skeleton operations for *i*. At the end of the block forming the procedure body the skeleton operations for uses of the parameters

B and *C* are replaced. The used entry for *A* is found to correspond with a declared entry which is a duplicate entry (as the first entry in the name list for that block) and this indicates that the identifier *A* is the procedure identifier. Thus, on working through the chain of skeleton operations for *A* any occurrence of the skeleton operation *AO* indicates that an assignment has been made to the procedure identifier and the *FD* marker is set in the duplicate entry. At the end of the procedure body (in this case at the delimiter ';' following the delimiter **end**) the name list for the procedure block is collapsed. Before discarding the first item it is inspected and if it is a duplicate copy of an entry for a type procedure a failure is given if the *FD* marker has not been set.

The above example has shown how a duplicate copy of a procedure identifier can be used to check that an assignment has been made to a procedure identifier inside the body of a type procedure. However, the Translator must indicate a failure if an assignment is made to the procedure identifier outside its body. This is done by giving a failure if a 'Take Address' operation is to be generated for a procedure identifier whose declared entry is not the first item of the current part of the name list.

Example 1.

```
begin real procedure P;
          P := y − x;
      real x, y;
      P := x := 0;
      . . .
```

At the second assignment to *P* the name list for the procedure body has been collapsed and so the name list contains

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*P*	r p	*1*	*2,0*	*10*			*0*		*1*
3	*y*	r	*1*	*1,4*						*3*
4	*x*	r	*1*	*1,3*						*3*

Thus when '*P* := ' is encountered the name list is searched and a declared entry for *P* is found. Since this entry shows *P* to be a real procedure, and since this entry is not the first item of the current part of the name list a failure is given.

Example 2.

```
begin real procedure P;
          P := y − x;
      real x, y;
      begin real z;
            P := z := 0
      end;
      . . .
```

In this case the second assignment to *P* occurs in the inner block and the operation *AO* will have been generated. At the delimiter **end** the name list contains

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*P*	r p	*1*	*2,0*	*10*			*0*		*1*
3	*y*	r	*1*	*1,4*						*3*
4	*x*	r	*1*	*1,3*						*3*
5										
6	*z*	r	*1*	*2,3*						*4*
7	*P*	arith	*0*	*34*	*34*					*5*

The object program generated to this point is

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	Jump around outer block
4	*BE*	( )	
7	*UJ*	(*26*)	Jump around procedure body
10	*PE*	(*2,0*), *0*	Start of procedure *P*
14	*TRA*	(*2,0*)	
17	*TRR*	(*1,4*)	
20	*TRR*	(*1,3*)	
23	–		
24	*ST*		$P := y - x$
25	*RETURN*		
26	*CBL*		
27	*UJ*	( )	Jump around inner block
30	*BE*	( )	
33	*AO*	( )	*P*
36	*TRA*	(*2,3*)	
39	*TIC0*		
40	*STA*		$z := 0$
41	*ST*		$P := 0$
42	...		

At the collapse of the inner block the chain of operations for *P* contains an operation *AO*. The entry in the surrounding block shows that *P* is a real procedure and, since this declared entry is not the first item of the list for the surrounding block, the use of *AO* is not allowed and a failure is given.

3.3.6 *Dim* Column

This part of the name list entry contains the number of dimensions or parameters associated with the identifier (e.g. the *dim* column for a scalar is set to zero). However, it is not always possible to fill in the *dim* column when

an entry is created for an identifier. For example when a declared entry is created for a formal parameter specified to be an array or a procedure it is not known how many dimensions or parameters will be associated with the formal parameter. Similarly, when a used entry is created for an identifier which is being used as an actual parameter, no details are known as to the type or number of dimensions or parameters.

For this reason a value of '-1' is stored in the *dim* column of an entry if no details are available concerning the number of dimensions or parameters associated with the identifier. This will be replaced as soon as details are available from subsequent uses of the identifier. Thus the *dim* column is used for checking the validity of the uses of an identifier.

Example

```
procedure P (A, B); array A; real procedure B;
begin integer i;
      for i := 1 step 1 until n do A [i] := B (X, i)
end;
```

Declared entries are created for *A* and *B* when the procedure heading is translated and '-1' is stored in the *dim* column of each entry. The use of *A* as a subscripted variable in the procedure body gives information as to the dimensions of array *A* and so the value of '-1' is replaced by '*1*' in the *dim* column of the entry for *A*. Similarly *2* replaces '-1' in the *dim* column of the entry for *B* when the procedure call for this identifier is translated. A used entry is created for the actual parameter *X* and '-1' is stored in its *dim* column.

3.3.7 *U* Column

For purposes of checking, an extra item of information called the '*u*' column is stored in name list entries (this item has not been shown on earlier examples).

This part of a name list entry is used to indicate whether the identifier has appeared in a statement or an expression. By this means it is possible to print out a warning message if, at the end of a block, an identifier has been declared but not used.

This check is not extended to labels which are often declared (i.e. precede a colon) for the purpose of marking a section of the program and not used in a designational expression.

When an identifier is used in a statement or an expression the name list for the current block is searched for an entry corresponding to the identifier. If a declared entry exists the *u* column marker is set in this entry. If no entry exists a used entry is created and the *u* column marker is set.

At the end of a block the name list for the current block is collapsed (see section 3.3.3). Before the declared entries are discarded the *u* column marker is inspected; if it is not set and the declared entry refers to a non-label a

warning message is printed. If a used entry in the current block corresponds to a declared entry in the containing block, the *u* column marker is set on the declared entry.

Example

Start of program: **begin integer** *i, j*; **array** *A* [*1*: *5*];
for *i* := *1* **step** *1* **until** *5* **do** *A* [*i*] := *i*
end

Declared entries are created for the identifiers *Start of program*, *i*, *j* and *A*. The *u* column markers are set on the entries for *i* and *A* when these identifiers are used in the for statement. However, the identifier *j* is declared but not used and so at the collapse of the name list for the block a message is printed to warn the programmer of a possible error in the program. In the case of the label *Start of program* no such message is printed as this label is being used to identify the block.

3.3.8 Summary

The size of the name list fluctuates throughout the translation of the ALGOL. At the end of a block all entries for identifiers local to that block are deleted. Thus the name list contains entries only for those identifiers which have a currently valid declaration.

At each occurrence of an identifier only the part of the name list which has been set up for the current block is searched for an entry corresponding to the identifier. Similarly, at the end of a block only the part of the name list corresponding to the surrounding block is searched in order to deal with the used entries of the current block.

The system of chaining employed in the Whetstone Compiler enables the generation of the object program to proceed whether the declaration details of an identifier are available or not. However, either of the following two slight modifications to the method of translation used would eliminate the need for chaining.

3.3.8.1 First Method

The first method uses a preliminary scan through the ALGOL whilst it is being read on to a suitable backing store. During this scan, lists are constructed of all identifiers that are used in a block giving declaration details for those which are local. The ALGOL is then translated by means of a single scan through the stored program using the appropriate list for the block that is currently being translated. However, a label may be used to jump forward to an unknown program address. In this case a pseudo-address is generated in the object program and details of this pseudo-address are preserved in a list. When the destination of the jump is known the list is used to change the pseudo-address into an actual address.

3.3.8.2 Second Method

The second method requires the ALGOL to be restricted slightly in order that the declaration of any identifier other than a label should occur before the identifier is used in a statement or in an expression. This can be achieved by insisting on a certain ordering of the declarations at a block head (with the scalar declarations first and the procedure declarations last) and by banning various facilities such as mutual recursion of parallel procedures, and the use of local function designators in a switch list.

The problem of a label being used to jump forward is solved after the manner of the first method.

Example

```
begin procedure P;
                  begin
                  . . .
                  end;
      procedure Q;
                  begin
                  . . .
                  end;
      . . .
```

In this example P may be called from within the body of Q but the procedure Q may not be called inside the body of P.

The method used in a one-pass translator for the ZEBRA Computer, which has been described by van der Mey [69], is a cross between this second method and the one used in the Whetstone Compiler. In the ZEBRA ALGOL 60 Translator scalar and array declarations must precede any switch or procedure declarations in a block head. Each time a label, switch or procedure identifier is used before the occurrence of its declaration a used entry (called a 'contra-declaration' by van der Mey) is added to the name list. At the end of a block the list of contra-declarations is processed to complete the skeleton operations corresponding to those identifiers which have since been declared.

3.4 TRANSLATION TECHNIQUES

3.4.1 Translation of Declarations

A compound statement consists of a set of statements preceded by **begin** and followed by **end**. However, a block has one or more declarations between the delimiter **begin** and the set of statements which are again followed by the delimiter **end**. Thus at the delimiter **begin** it is not known whether a compound statement or a block is to follow. A state variable V is used to discriminate between the two. It is set to zero at the delimiter **begin**, to one whilst translating declarations and to two whilst translating statements. Whenever a new declaration occurs, the state variable V is inspected. If V is zero this declaration is the first of a block and so the corresponding object program operations are generated for the start of the new block. If V is equal to one this declaration is not the first in the head of the current block and so the object program operations for setting up the current block will have been already generated. However, if V is equal to two a failure is given as declarations must be placed before the first statement in a block.

To avoid unnecessary complication the examples given in this section assume that no entries exist in the current part of the name list for the identifiers being declared.

3.4.1.1 Scalar Declarations

The state variable T is set up with a bit pattern which gives details of the type of the declaration (i.e. either real, integer or Boolean) and the marker D is set to one if it is an own declaration. At the end of the declaration (i.e. at the delimiter ';') the state variable T is cleared and D reset to zero.

Example

begin real *A*, *B*;
own integer *C*, *D*;

The various stages in the translation of this block head are shown below.

(*i*) The delimiter **begin** is stacked with its priority of zero and the state variable V is set to zero.

(*ii*) At the delimiter **real**, T is set up accordingly after first checking that it was zero. The fact that V is zero shows that this is the first declaration of a block and so a subroutine called 'BLOCK BEGIN' is entered. This subroutine causes the following object program to be generated

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	A jump around the block
4	*BE*	( )	
7	...		

The delimiter **begin** at the top of the stack is replaced by the following three items

Stacked Items	*Priority*
UJ, 2	*13*
2, L, NL	
begin (bl)	*0*

The first of these shows that an incomplete unconditional jump has been generated and the address (*2*) stacked with this item points to the parameter of the incomplete *UJ* operation; the stack priority of the item is thirteen so that when it is at the top of the stack it will be unstacked by any of the delimiters which have a compare priority other than void.

The second of these items contains the values of *V*, *L* and *NL*. *L* and *NL* are state variables which have not previously been mentioned. *L* is used to count the number of local scalars and array words; when the end of the block is reached *L* will give the extent of the first order working storage (see section 2.2.3).

NL is always set to indicate the first item of the current section of the name list.

Since the delimiter **begin** heralds the start of a statement the value of two is stacked for *V*.

These state variables will then be used to hold information concerning the translation of the current block. At the end of this block they are reset from the stacked values so that the translation of the surrounding block can proceed.

After stacking this item, *L* is set to zero, *NL* reset to point to the next free entry in the name list (i.e. *NLP*), and *V* is set to one for the translation of the declarations.

The third of these items consists of the delimiter **begin** with a marker to show that a block is being translated.

Other duties performed by BLOCK BEGIN are the updating of the block level count *n* and the reserving of a blank entry at the head of the part of the name list for the current block.

(*iii*) At the delimiter ',' the state variable *T* shows that a scalar declaration is being translated and that an identifier must precede this delimiter. Since the current part of the name list contains no entry for the identifier *A*, a declared entry is made and its *type* column filled from the state variable *T*. The values *n* and '*L* + *3*' are packed together and stored in the *np* column of this entry (where *n* and *L* indicate the values of the block level count *n* and the state variable *L*). The contents of *L* are then increased by one.

(*iv*) At the delimiter ';' the identifier *B* is dealt with as in step (*iii*) above and then the state variable *T* is cleared. The marker *D* will already be zero as this is not an own declaration.
(*v*) At the delimiter **own** the marker *D* is set to one, after checking that it was previously zero. The state variable *T* must also be zero. Since this is not the first declaration of the current block (as shown by *V*) the subroutine BLOCK BEGIN can be bypassed.
(*vi*) The delimiter **integer** causes the state variable *T* to be set up for the current declaration.
(*vii*) At the delimiter ',' the name list is searched for an entry corresponding to the identifier *C* in the current block. No entry is found and so a declared entry is created and using the current value of *T* its *type* column is set up to be integer. As *C* is declared to be an own variable, 'zero' (since block level *n* is given as zero for own variables) and L_p are packed together and stored in the *np* column of its entry. (L_p is a state variable which gives the extent of first order working storage for own scalars and own array words.) L_p is then increased by one.
(*viii*) At the delimiter ';' the identifier *D* is declared in a similar manner to that described in step (*vii*) for the identifier *C*. The state variable *T* and the marker *D* are then cleared.
The name list will then contain the following items for these identifiers (assuming that these declarations are placed at the head of the program block).

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*A*	r	*1*	*1,3*						*1*
3	*B*	r	*1*	*1,4*						*1*
4	*C*	i	*1*	*0,1*						*2*
5	*D*	i	*1*	*0,2*						*2*

3.4.1.2 Array Declarations

An array declaration contains one or more array segments. Each segment is treated as a separate declaration and is translated separately although all the segments share the same type delimiters.

The state variable *T* is used to contain the bit pattern for the type of the declaration (e.g. real array) and the marker *D* is set if it is an own array declaration.

Thus at the end of an array segment, a delimiter ',' will indicate that another segment is to follow and *T* and *D* will be needed for the translation of this segment. A delimiter ';' at the end of a segment indicates that it is also the end of the array declaration and so *T* and *D* are cleared.

If the array declaration is the first declaration after the delimiter **begin**, object program operations must be generated to set up the block. This has been discussed in the previous section on scalar declarations.

3.4.1.2.1 Translation of an Array Segment. In the Whetstone Compiler, translation of an own array segment is somewhat different to that of a non-own segment. Thus the two types of segments are considered in turn.

3.4.1.2.1.1 Non-Own Array Segment. The segment contains one or more array identifiers which are to be declared. The name list entries which will be generated for these identifiers need the type information, which is contained in the state variable *T*, and also details as to the number of dimensions, which are not known until the bound pair list has been translated. Thus the name list entry for the array identifier could be given the type details as soon as the identifier has been reached in the array segment and then at the end of the bound pair list have the dimension information added. However, this means that details of the numbers of the name list entries for all of the array identifiers in the segment must be kept until the dimension information is available. For this reason, in the Whetstone Compiler, the array identifiers are not 'declared' until the end of the segment when details of the type and the number of dimensions are known. The names of the array identifiers are held in the translator stack until the bound pair list has been translated.

The delimiter ',' can be used in many different ways in an array declaration and the Translator must be able to differentiate between these uses.

Example

begin array *A, B* [*0*: *2, 1*: *10*], *C, D* [*0*: *5*];

The way in which the Translator recognizes the various uses of comma is listed below.

(*i*) The comma between array identifiers. At the start of the array segment an operation *MSF* is stacked with a priority of zero and a counter which is used to give the number of array identifiers in the current array segment. Thus this use of comma is recognized by the fact that *MSF* is at the top of the stack. The array identifier preceding the comma is placed in the stack beneath the operation *MSF*. To do this *MSF* is unstacked, the array identifier placed in the stack and then the operation *MSF* is re-stacked after its counter has been updated.

Example

array *A, B, C* [*1*: *10*];

After the first comma the top of the stack contains

Stacked Item	*Priority*
MSF, 1	*0*
A	

Then, after the second comma, the top of the stack contains

Stacked Item	*Priority*
MSF, 2	*0*
B	
A	

(*ii*) The comma between array segments. At the end of an array segment the operation *MSF* and all the identifiers are unstacked and dealt with. Thus the comma between array segments will find a delimiter **begin** at the top of the stack.

(*iii*) The comma between bound pairs. This use can be recognized as the top of the stack will contain neither *MSF* nor **begin**.

The delimiter '[' at the start of the bound pair list is stacked as '$[_D$' to distinguish it from the opening square bracket used in a subscripted variable. It has a stack priority of zero. A counter and a marker are also stacked with this delimiter. The counter is used to give the number of dimensions; it is initially set to one and then increased by one at each comma separating the bound pairs. The marker is used to ensure that the delimiters ':' and ',' are used alternately between the subscript bound expressions of the bound pair list. The marker is set to zero at the delimiter '[', changed from zero to one at the delimiter ':' and then reset to zero at the delimiter ','. The delimiters ':' and ',' are not stacked and the delimiter '[', stacked as '$[_D$', is used as the opening bracket for the translation of all the subscript bound expressions in a similar way to that described for subscript expressions in section 3.2.1.3.2. However, for their role as the closing bracket of a subscript bound expression the delimiters ':' and ',' are given a compare priority of one.

The subscript bound expressions are translated into object program operations in the usual way.

Example

begin array *A*, *B* [$-1:0, +1:3$];

The object program generated for the bound pair list of this declaration is

Syll	*Op*	*Par*
0	*TIC1*	
1	*NEG*	
2	*TIC0*	
3	*TIC1*	
4	*TIC*	*'3'*
11	...	

At the closing square bracket of an array segment, after the last bound expression has been dealt with, the top of the stack will contain '$[_D$' with a

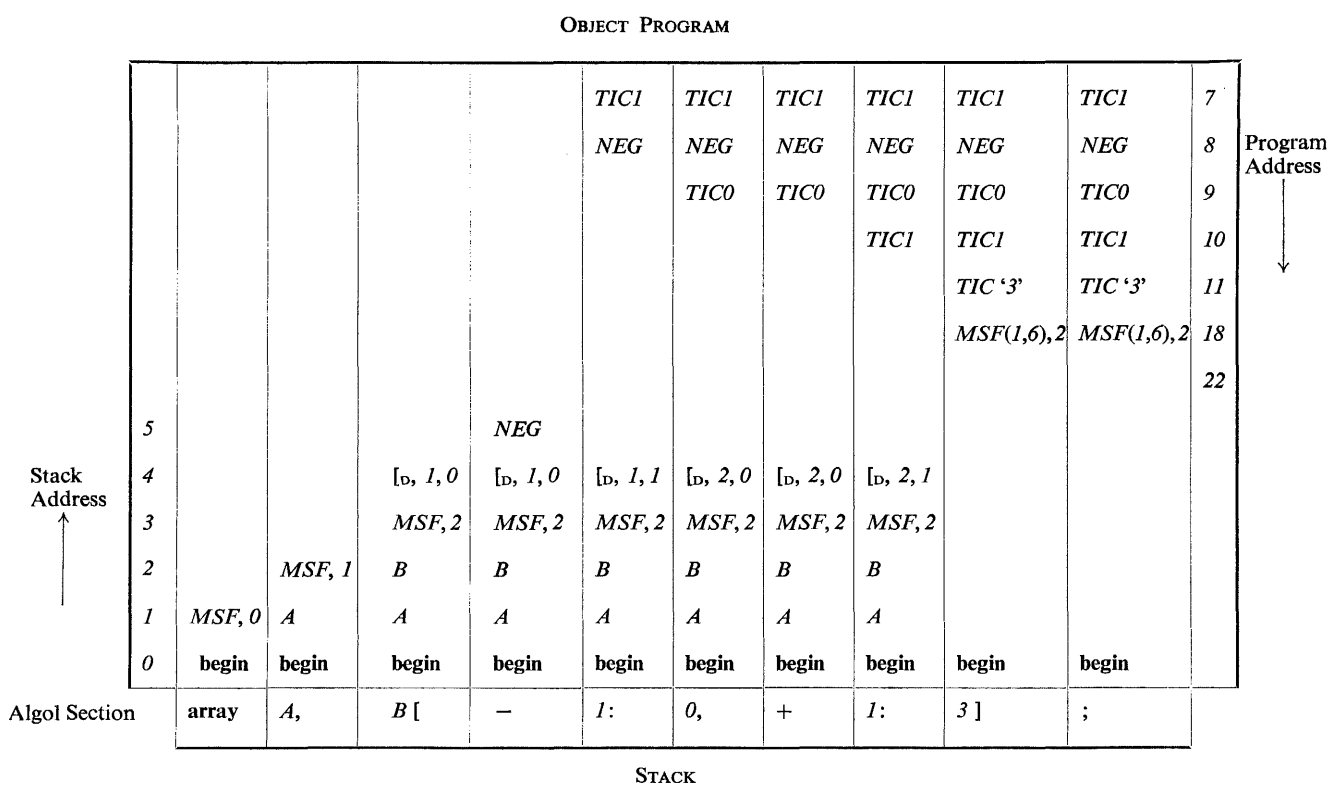


FIG. 10. Representation of Object Program and Stack at each stage during the translation of an array declaration.

counter containing *d*, the number of dimensions of the segment, and a marker set to one. The value *d* contained in the counter is noted and '$[_D$' is deleted from the stack. The top of the stack now contains *MSF* with a counter giving *N*, the number of array identifiers declared in the current segment. An operation *MSF* is generated in the object program having as its two parameters the *np* address of the last array identifier declared in the segment and the value *N*. Thus, since the state variable *L* contains the number of scalars and array words already declared in the current block, the following operation is generated

$$MSF\,(n,\ L + 2 + N),\ N$$

The value *N* is noted and the item '*MSF,N*' at the top of the stack is unstacked, leaving *N* identifiers at the top of the stack to be declared.

The identifier at the top of the stack is declared using the type information contained in the state variable *T*, the number of dimensions *d* and (*n*,*L* + 2 + *N*) as its *np* address. All the other array identifiers are declared in a similar way and then the state variable *L* is increased by *N*.

Example

begin real *x*, *y*;
array *A*, *B* [−*1*: *0*, + *1*: *3*];

Figure 10 shows how the stack is used to translate the above array declaration. The translator stack and the object program are shown after the completion of each delimiter routine. When the array declaration has been completed the name list will contain the following entries (assuming that the block level is one).

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*x*	r	*1*	*1,3*				*0*		*1*
3	*y*	r	*1*	*1,4*				*0*		*1*
4	*B*	r a	*1*	*1,6*				*2*		*2*
5	*A*	r a	*1*	*1,5*				*2*		*2*

3.4.1.2.1.2 Own Array Segment. The bound expressions of an own array segment are restricted by the Whetstone Compiler to be signed or unsigned integers. This is because the Translator must calculate the size of each own array to enable the required amount of space to be set aside for it at run time. It was not considered worthwhile allowing the bound expressions to be more complicated constant expressions. The translation of an own segment is similar to that described above for a non-own segment with the following four exceptions.

(*i*) *MOSF* is used instead of *MSF*.

(*ii*) The value zero is used as the block level for the *np* addresses instead of the value of the state variable *n*, and the state variable L_p is used instead of *L* to contain the number of own array words (*N*).

OBJECT PROGRAM

			AOA( )	*AOA( )*	*AOA( )*	*AOA( )*	*AOA( )*	*AOA( )*	*AOA( )*	*AOA( )*	*AOA( )*	*AOA(25)*	*7*
							TIC1	*TIC1*	*TIC1*	*TIC1*	*TIC1*	*TIC1*	*10*
							NEG	*NEG*	*NEG*	*NEG*	*NEG*	*NEG*	*11*
								TIC0	*TIC0*	*TIC0*	*TIC0*	*TIC0*	*12*
										TIC1	*TIC1*	*TIC1*	*13*
											TIC '3'	*TIC '3'*	*14*
											MOSF(0,2), 2	*MOSF(0,2), 2*	*21*
													25
Stack Address	*4*				*MOSF,2*	*MOSF,2*	*MOSF,2*	*MOSF,2*	*MOSF,2*	*MOSF,2*			
	3			*MOSF,1*	*B*	*B*	*B*	*B*	*B*	*B*			
	2		*MOSF,0*	*A*	*A*	*A*	*A*	*A*	*A*	*A*			
	1		*UJ,8*	*UJ,8*	*UJ,8*	*UJ,8*	*UJ,8*	*UJ,8*	*UJ,8*	*UJ,8*	*UJ,8*		
	0		**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	
Algol Section		**own**	**real**	**array**	*A,*	*B* [	—	*1:*	*0,*	+	1:	*3*]	;

Program Address ↓

STACK

FIG. 11. Representation of Object Program and Stack at each stage during the translation of an own array declaration.

(*iii*) At the delimiter **array** an incomplete operation '*AOA* ( )' is generated and '*UJ*, $i - 2$' is stacked as a reminder to fill in the parameter of the *AOA* operation at the end of the own array declaration. (The counter '*i*' gives the extent of the generated object program and hence '$i - 2$' gives the syllable address of the parameter of the incomplete operation *AOA*.)

Thus the comma used between own array segments will find the item *UJ* at the top of the stack instead of **begin**.

(*iv*) Since the bound pair list of an own array declaration has been restricted and the size of the own array has to be calculated at translation time, a single routine is used to translate it instead of using the normal technique of a separate routine for each delimiter. Whilst translating the bound pair list the size *s* of the array is calculated and then at the end of the array segment the state variable L_0 is increased by *d*, the number of dimensions, to allow for the storage mapping function, and by '$s \times N$' to allow for the *N* arrays each of size *s* that have been declared in this segment.

Example

begin real *x*, *y*;
own real array *A*, *B* [-1: 0, $+$ 1: 3];

Figure 11 shows the generation of the object program and the contents of the stack during the translation of the own array declaration.
At the end of the bound pair list the state variable L_0 is increased by $(2 + 2 \times 6)$ to allow for the storage mapping function containing two items and for the two arrays each containing six elements. The name list will contain the following entries for this block head, which is assumed to be at level two, when the array declaration has been translated

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*x*	r	*1*	*2,3*				*0*		*1*
3	*y*	r	*1*	*2,4*				*0*		*1*
4	*B*	r a	*1*	*0,2*				*2*		*2*
5	*A*	r a	*1*	*0,1*				*2*		*2*

3.4.1.2.2 Translation of Comma Used Between Array Segments. When the previous segment has been translated, the top of the stack will contain either the delimiter **begin** or the operation *UJ* (if it is an own array declaration). In both cases the state variable *T* and the own marker *D* will already have been set up to deal with all segments of the current array declaration. Thus '*MOSF*, *0*' is stacked if it is an own array declaration, otherwise '*MSF*, *0*' is stacked. The translation of the next segment is similar to that of the previous segment.

3.4.1.3 Switch Declarations

As with the scalar and array declarations the state variable *T* is used to

contain the bit pattern giving the type of the declaration. In this case the own marker *D* must be zero as an identifier cannot be declared to be an own switch.

At the start of the switch declaration an unconditional jump operation *UJ* is generated so that the object program operations generated for the declaration are bypassed upon entry to the block in which the declaration occurs. However, since the extent of the switch declaration is not known at its start, the operation *UJ* is generated as an incomplete operation and '*UJ*, *i* − *2*' is stacked as a reminder to complete the operation at the end of the declaration. The switch declaration is treated as a block with one item of first order working storage and so the following block entry operation is generated

BE (*n*,*I*)

(where *n* gives the level of this block)

After the switch identifier has been declared as a switch the delimiter **begin** is stacked with a priority of zero and a marker showing that it is a switch block. This type of **begin** is unstacked at the delimiter ';' at the end of the declaration whereas a normal **begin** requires the delimiter **end** to be unstacked.

Two program addresses are also stacked with the **'switch begin'**. These are used to point to the parameters of the last *DSI* and *UJ* operations, (these operations are generated at the start and finish respectively of each designational expression). The operations *DSI* and *UJ* are generated as incomplete operations and then filled in as soon as possible. In the case of the operation *DSI* it can be filled in at the end of the current designational expression before the next *DSI* is generated, but the unconditional jump operations all point to the end of the declaration (i.e. the operation *EIS*). Therefore the *UJ* operations are chained together and the address of the start of the chain is stacked with the **'switch begin'**.

Example

begin switch *S* := *L1*, *L2*, **if** *a* > *0* **then** *L3* **else** *L1*;

Since this declaration is the first in the current block head, operations are generated for setting up the block. At the delimiter ':=' the identifier *S* is declared as a switch and the state variables *E* and *TYPE* are set to show that a designational expression is to follow.

The item **'switch begin,** , *i* + *4*' is stacked with a priority of zero. The first of the program addresses will contain the address of the parameter of the operation *UJ* placed at the end of the first designational expression. The second of the program addresses points to the parameter of the first *DSI* operation to be generated.

The state variable *n* is increased by one to allow the switch declaration to be treated as a block.

The following object program operations are generated

Syll	*Op*	*Par*	*Remarks*
7	*UJ*	( )	Unconditional jump around switch declaration
10	*BE*	(*2,1*)	
13	*DSI*	( )	
16	...		

At the first comma a skeleton operation *RO* is generated for *L1*, followed by the operation *DUMMY* since the state variable *TYPE* shows the type of expression to be designational. This is because this skeleton operation (which is a three-syllable operation) will be replaced by the four-syllable operation *TL* later and so the *DUMMY* operation will be overwritten. An incomplete operation *UJ* is generated and the address of its parameter stored with '**switch begin**' at the top of the stack. The first *DSI* operation can now be completed and a new incomplete *DSI* operation is generated. Thus the object program generated is

Syll	*Op*	*Par*	*Remarks*
7	*UJ*	( )	
10	*BE*	(*2,1*)	
13	*DSI*	(*23*)	
16	*RO*	( )	*L1*
19	*DUMMY*		
20	*UJ*	( )	Jump to *EIS*
23	*DSI*	( )	
26	...		

The item at the top of the stack is now

Stacked Item	*Priority*
switch begin, *21*, *24*	*0*

After the operations '*RO* ( )' and *DUMMY* have been generated for *L2* a further *UJ* will be generated. This is chained to the operation *UJ* generated for the first designational expression by having as its parameter the address of the parameter of the first *UJ*. The address of the new parameter is stacked with the item '**switch begin**' and the following operations are added to the object program

Syll	*Op*	*Par*	*Remarks*
26	*RO*	( )	*L2*
29	*DUMMY*		
30	*UJ*	(*21*)	
33	*DSI*	( )	
36	...		

The counters stacked with '**switch begin**' now have the values *31* and *34* respectively.

The conditional expression for the last designational expression is translated in the normal way. The operation *DUMMY* is again used to leave space for the extra syllable of the operation *TL*. Thus the following object program operations are generated for this designational expression.

Syll	*Op*	*Par*	*Remarks*
36	*RO*	( )	*a*
39	*TIC0*		
40	>		
41	*IFJ*	(*51*)	
44	*RO*	( )	*L3*
47	*DUMMY*		
48	*UJ*	(*55*)	
51	*RO*	(*17*)	*L1*
54	*DUMMY*		
55	*UJ*	(*31*)	Jump to *EIS*, chained to parameter of last operation *UJ* to *EIS*
58	...		

Since this is the last designational expression of the declaration the operations *ESL* (End Switch List) and *EIS* (End Implicit Subroutine) are generated. All the incomplete jump operations to the *EIS* operations are then filled in and the last *DSI* operation filled in to point to *ESL*. The item **'switch begin'** is unstacked from the top of the stack and the unconditional jump around the switch declaration can then be filled in. Also the state variable *n* is decreased by one. Thus the object program generated for the block head is

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	
4	*BE*	( )	
7	*UJ*	(*60*)	Jump around switch declaration
10	*BE*	(*2,1*)	
13	*DSI*	(*23*)	
16	*RO*	( )	*L1*
19	*DUMMY*		
20	*UJ*	(*59*)	Jump to *EIS*
23	*DSI*	(*33*)	
26	*RO*	( )	*L2*
29	*DUMMY*		
30	*UJ*	(*59*)	Jump to *EIS*
33	*DSI*	(*58*)	
36	*RO*	( )	*a*
39	*TIC0*		
40	>		
41	*IFJ*	(*51*)	

Syll	*Op*	*Par*		*Remarks*
44	*RO*	( )	}	*L3*
47	*DUMMY*		}	
48	*UJ*	(*55*)		
51	*RO*	(*17*)	}	*L1*
54	*DUMMY*		}	
55	*UJ*	(*59*)		Jump to *EIS*
58	*ESL*			
59	*EIS*			
60	...			

The switch declaration is made into a block but only the switch identifier can be declared within it and so there is no need to set up a separate part in the name list for a switch block. Thus the name list will contain the following entries for the switch declaration.

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*S*	sw	*1*		*10*			*1*		*1*
3	*L1*	l	*0*	*52*	*17*					*1*
4	*L2*	l	*0*	*27*	*27*					*1*
5	*a*	arith	*0*	*37*	*37*					*1*
6	*L3*	l	*0*	*45*	*45*					*1*

The information contained in used entries has already been described in section 3.3.

The declared entry for *S* contains the syllable address of the start of the object program for the switch declaration; the *dim* column is set to one since the switch designator is similar to a subscripted variable containing one dimension.

3.4.1.4 Procedure Declarations

3.4.1.4.1 Translation of the Procedure Heading. The translation of a procedure heading is handled by a single routine; this is the second departure from the normal technique of using a separate routine for each delimiter (the other being the translation of a bound pair list of an own array segment). The main reasons for this are that a procedure heading cannot involve any recursive definitions (i.e. a procedure heading cannot contain another procedure heading) and to avoid confusion between specifications and declarations. A specification for arrays contains only the list of array identifiers but the declaration must contain the bound pair list as well, and identifiers may be specified to be labels or strings but cannot be declared as such.

Example

begin array *A* [*0*: *31*];
procedure *Q* (*X*, *Y*, *Z*); **array** *X*; **label** *Y*; **string** *Z*;
...

A procedure declaration is treated as a block with the formal parameters appearing as identifiers local to this block. If, however, the procedure body is an unlabelled block a new block is not created and so the 'procedure block' will contain both the formal parameters and the identifiers declared as local to this block, (however, the formal parameters are separated from the local variables in the name list as described in section 3.3.5).

Example

```
procedure Q (X,Y); array X; label Y;
begin integer i, j;
      . . .
end;
```

The procedure block created for *Q* contains the formal parameters *X* and *Y* as well as the identifiers *i* and *j*.

As has been described in section 3.3.5.1 the first item of the name list for the procedure block contains a duplicate copy of the entry for the procedure identifier. The *line* column of this duplicate copy is used for the *FD* marker. This marker is used for checking that an assignment to the procedure identifier has taken place in the case of a type procedure.

A procedure heading consists of up to four parts.

(*i*) The type delimiters and the procedure identifier. The name list entry for the procedure identifier requires both the type information and the number of parameters (stored in the *dim* column of the name list entry) and so the declaration of the procedure identifier is held over until the formal parameter part has been dealt with.

(*ii*) Formal parameter part. In this section of the procedure heading declared entries are set up for the formal parameters in the name list corresponding to the procedure block. The *f* and *np* columns are filled in for these entries but the *type* column is left blank for specification details. The procedure identifier is then declared using the type information given in the state variable *T* and the number of parameters found from the translation of the formal parameter part.

(*iii*) Value part. This section of the heading gives a list of identifiers whose name list entries require the *f* column to be set to zero and the *v* column to be set to one. There may be no value part in the procedure heading.

(*iv*) Specification part. The *type* columns of the name list entries for the parameters are filled in from the details given in the specifications. Each specification is translated in turn and the state variable *T* is used to contain the bit pattern giving the type of the specification. At the end of the specification part there should be no entries for the parameters with the *type* column still blank. The *dim* columns of any parameters specified to be arrays or procedures are set to '-1' as the number of dimensions or parameters is not given in the specification part (see section 3.3.6).

For a procedure with no parameters the formal parameter, value, and specification parts are omitted.

At the end of the procedure heading the following object program operations are generated

(*i*) An *UJ* operation is generated to enable the procedure declaration to be bypassed at run time. Since the extent of the declaration is not yet known this operation is generated as an incomplete operation and an item is stacked as a reminder to complete it as soon as possible.

(*ii*) An operation '*PE* ( ), *m*' is generated for the start of the procedure block. As the first order storage for this block can include local variables, the first parameter of this operation is left blank and then filled in with the values of *n* and *L* at the end of the block. The second parameter contains the number of formal parameters. A delimiter **begin** is stacked with a priority of zero and a marker '**procedure**' to show the type of the block that is being translated. The program address of the parameter of the *PE* operation is also stacked with the **begin** as a reminder to fill it in at the end of the block.

(*iii*) An appropriate parameter operation is then generated for each of the formal parameters in turn. These operations have been described in detail in section 2.5.6.

The *f* column of an entry corresponding to a label called by value is reset to one. This is because the object program requires the formal operation *TFL* to be used for labels whether called by name or by value. The reason for the use of *TFL* for labels called by value has been discussed in section 2.5.6.3.

Example

begin real procedure *R* (*U*, *V*, *W*, *X*, *Y*, *Z*); **value** *X*, *Z*; **real** *U*, *W*;
real procedure *V*; **integer** *X*; **label** *Y*, *Z*;

The name list will contain the following entries at the end of the above procedure heading

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*R*	r p	*1*	*2,0*	*10*			*6*		*1*
3	*R*	r p	*1*	*2,0*	*10*			*6*		
4	*U*	r	*1*	*2,3*		*1*				*1*
5	*V*	r p	*1*	*2,5*		*1*		*−1*		*1*
6	*W*	r	*1*	*2,7*		*1*				*1*
7	*X*	i	*1*	*2,9*			*1*			*1*
8	*Y*	l	*1*	*2,11*		*1*				*1*
9	*Z*	l	*1*	*2,13*		*1*	*1*			*1*

3.4.1.4.2 Translation of the Procedure Body. If the body of the procedure is an unlabelled block the Translator does not set up a new block but a

marker (bl) is added to the **begin** at the top of the stack. Then a corresponding **end** must be found to cancel this marker before the delimiter ';' ending the procedure declaration can be found. If the body of the procedure is a labelled block the name list for the block created for the procedure will contain entries for the parameters and this local label. A block will then be set up for the body in the normal way.

Example

begin procedure *P* (*A*); **value** *A*; **integer** *A*;
L: **begin real** *x*;
. . .

The name list contains

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*P*	p	*1*		*10*			*1*		*1*
3	*P*	p	*1*		*10*			*1*		
4	*A*	i	*1*	*2,3*			*1*			*1*
5	*L*	l	*1*	*2,0*	*15*					*2*
6										
7	*x*	r	*1*	*3,3*						*2*

(The *np* column of a label is set up with the block level only, as operations corresponding to a label require the block level and the contents of the *syll* column as parameters.)
The following object program operations will have been generated when the label *L* has been dealt with

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	Jump around block
4	*BE*	( )	
7	*UJ*	( )	Jump around procedure
10	*PE*	( ), *1*	Start of procedure *P*
14	*CSI*		
15	. . .		Syll address of label *L*

If the top of the stack contains **begin** with the marker **'procedure'** but not the extra marker (bl) when a semi-colon is reached then the translation of a procedure declaration has been completed. The *PE* operation can be filled in using the address stacked with the item **'procedure begin'** which is then deleted from the stack. The jump operation *UJ* around the declaration can then also be completed using the item now at the top of the stack.

3.4.1.5 Bypassing Switch and Procedure Declarations at Run Time

The previous two sections have shown how procedure and switch declarations can be bypassed at run time by means of an *UJ* operation. This is generated as an incomplete operation at the start of the declaration and then completed at the end of the declaration. However, this means that successive switch or procedure declarations in a block head would have to be bypassed one at a time.

Example

```
begin procedure P; begin
                     . . .
                   end;
    switch S := L1, L2, L3;
    real procedure Q; begin
                        . . .
                      end;
. . .
```

The object program would be of the form

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	Jump around block
4	*BE*	( )	
7	*UJ*	(*30*)	Jump around *P*
10	. . .		Procedure *P*
	. . .		
30	*UJ*	(*68*)	Jump around *S*
33	. . .		Switch *S*
	. . .		
68	*UJ*	(*115*)	Jump around *Q*
71	. . .		Procedure *Q*
	. . .		
115	. . .		

If the incomplete *UJ* operation generated at the start of the declaration is not filled in when the end of the declaration is reached it is possible to allow this operation to bypass successive declarations in one jump. At the end of the procedure or switch declaration the reminder to complete the *UJ* operation is left at the top of the stack to be found by the next declaration or statement. Thus, should the top of the stack contain *UJ* at the start of a procedure or switch declaration, there is no need to generate a further incomplete *UJ* operation in the object program. This means that at the start of an array declaration (which cannot be bypassed) or the first statement of a block, the top of the stack must be inspected. If it contains *UJ* the jump operation around the last declaration or group of declarations must be completed.

Example

Using a single *UJ* operation to bypass the three declarations given in the previous example, the object program would be

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	( )	
4	*BE*	( )	
7	*UJ*	(*109*)	Jump around the three declarations
10	. . .		Procedure *P*
	. . .		
30	. . .		Switch *S*
	. . .		
65	. . .		Procedure *Q*
	. . .		
109	. . .		

3.4.2 Translation of Switch Designators

A switch designator is written in a similar form to a subscripted variable with one dimension.

Example

S [*I*]

If the declaration details of the switch identifier are not available it is not always possible for the Translator to know whether a switch designator or a subscripted variable of a vector is being translated.

Example

P (*B* [*I*])

Without checking the specification of the formal parameter of procedure *P* the identifier *B* could be an array or a switch identifier.

Thus, the Translator will generate similar object program operations for both a switch designator and a subscripted variable. The translation of subscripted variables has been discussed in section 3.2.1.3. The operation *INDR* will be generated if the switch designator is used at expression level (i.e. in a designational expression inside a switch declaration or go to statement) otherwise *INDA* will be generated (i.e. a switch designator used as an actual parameter).

Example

go to *S* [*1*];

At the delimiter **go to** the state variable *TYPE* is set to indicate that the expression following is a designational expression. Thus in this case the subscripted variable is recognized as a switch designator and *S* must prove to be a switch identifier.

Example

P (*B* [*1*])

In this case the identifier *B* could be an array or a switch identifier. No check is made on the correspondence between the type of the actual parameter and the specification of the corresponding formal parameter. The object program operations generated for this use of a switch designator are

Syll	*Op*	*Par*	*Remarks*
0	*TSA*	*B*	The parameter contains the address of the switch *B*
3	*TIC1*		
4	*INDA*		
5	...		

However, the Translator checks that a switch designator has only one dimension. It also checks that an array or switch identifier is used consistently.

Example

if *A* [*0*] > *0* **then go to if** *b* **then** *A* [*1*] **else** *S* [*1*, *2*];

The above statement would fail for either of two reasons. The identifier *A* is used both as an array identifier and as a switch identifier and the switch designator '*S* [*1*, *2*]' has two dimensions.

3.4.3 Translation of a Procedure Call

A procedure call may occur as a procedure statement or as a function designator in an expression. The state variable *E* distinguishes between the two types of procedure call. If the procedure is called by a function designator it must be declared as a type procedure. The number of actual parameters in the procedure call is checked against the number of formal parameters in the heading of the declaration of the procedure identifier.

3.4.3.1 Procedure Call With no Parameters

A call of a procedure with no parameters by means of a procedure statement can be recognized as such by the Translator. But if a procedure with no

parameters is called by a function designator it cannot be distinguished from a simple variable without checking the *type* column of the name list entry for the procedure identifier. If an entry for the procedure identifier exists in the current part of the name list its *type* and *dim* columns are checked against the current use. Otherwise a used entry is created and details of the current use are inserted.

3.4.3.2 Procedure Call With Parameters

A call of a procedure with a non-empty parameter part is recognized by the occurrence of a left round bracket immediately following an identifier. The validity of this use of the procedure identifier cannot be fully checked until the number of actual parameters is known. Thus, the procedure identifier is preserved in the stack until the actual parameter part has been dealt with. Then the validity of the current use is checked against the *type* and *dim* columns of the entry for the procedure identifier.

Example

$$X := X \div P(A, 3, B + C);$$

The state variable E shows that this procedure call appears in an expression. At the end of the actual parameter part it is found that the declaration for P must have three formal parameters. Thus, at this point the name list for the current block is searched for an entry corresponding to P. If an entry exists, the *type* and *dim* columns are checked against details of the current use, otherwise a used entry is created and these details added to the entry.

At the opening bracket of the actual parameter part an incomplete *UJ* operation is generated and a reminder to complete it as soon as possible is stacked. This *UJ* operation will lead to the *CF* or *CFF* operation to be generated at the end of the operations for the procedure call. The name of the procedure identifier is stacked (as described above) and finally the delimiter '(' is also stacked. A state variable *PROC* is used to indicate that a procedure call is being translated.

Since an actual parameter can include another procedure call the current value of *PROC* (i.e. for the text surrounding this procedure call) is preserved in the stack with the delimiter '('. Then the state variable *PROC* is set for the translation of this procedure call and reset at the end of this call using the value stacked with the delimiter '('. Two other items are stacked with '(', namely the current values of the state variables E and *TYPE*. This is because the translation of the surrounding expression is suspended whilst a function designator contained in the expression is being translated (a similar situation occurs for the delimiter **if** – see section 3.2.1.4). These two state variables are then used to contain details of the type of the current actual parameter.

Example

$X := X + P(A, 3, B + C);$

At the left round bracket an incomplete operation *UJ* ( ) is generated and the following items are stacked

Stacked Item	*Priority*	*Remarks*
UJ, $i - 2$	*13*	
P		The procedure identifier
PROC, *E*, *TYPE*		The current values of the state variables *PROC*, *E* and *TYPE* packed together into a word.
(	*0*	

3.4.3.2.1 Actual Parameter Part. An 'actual operation' is generated for each actual parameter of a procedure call. If no declaration details are available for an actual parameter the skeleton 'actual operation' *PO* is generated. Since a procedure call is translated independently of the corresponding procedure declaration it is not known which type of 'actual operation' will replace the skeleton operation. If the actual parameter should prove to be a label the skeleton operation will be replaced by

PL (*a*), *m*

However, this is a four-syllable operation and so four syllables must be allowed for the skeleton 'actual operation' *PO*. To simplify matters each type of 'actual operation' is stored in the object program as a four-syllable operation although only *PL* uses the fourth syllable. This does not cause any difficulties for the Control Routine which inspects the 'actual operations' under the control of the appropriate *CF* or *CFF* routine.

The 'actual operations' are preceded in the object program by operations generated to calculate an actual parameter expression, the value of a constant actual parameter, or the characters of a string parameter. Thus the 'actual operations' are stacked with a priority of zero until they can be unstacked into the object program at the closing round bracket of the actual parameter part. For this reason the 'actual operations' appear in the object program in the reverse order to the actual parameters to which they refer.

Example

P (*A*, *B*, *C*);

is translated into

Syll	*Op*	*Par*
0	*UJ*	(*15*)
3	*PR*	*C*, –
7	*PR*	*B*, –
11	*PR*	*A*, –
15	*CF*	*P*, *3*
19	...	

(It is assumed that *A*, *B* and *C* have been declared to be real.) In this example all the actual parameters are single identifiers and so there is no object program between the *UJ* operation and the first of the 'actual operations'.

3.4.3.2.1.1 Implicit Subroutine. An implicit subroutine is generated whenever an actual parameter is proved to be other than a single identifier, constant or string. Thus, if the state variable *PROC* is set, the last delimiter was a parameter delimiter and the current delimiter is not a parameter delimiter, an implicit subroutine is generated. A state variable *LD* (i.e. the last delimiter) contains the delimiter previous to the current delimiter. The end of an implicit subroutine is recognized if, at a parameter delimiter, the last delimiter is not another parameter delimiter. However, the Translator must be able to recognize whether a delimiter is in fact a parameter delimiter. The left and right round brackets may be used as parameter delimiters or as expression brackets inside a parameter expression; a comma may be used as a parameter delimiter or as a delimiter between subscript expressions in a subscripted variable which is used in an actual parameter part.

For this reason the value of *PROC* is preserved in the stack with '(', used as an expression bracket, or '[' for a subscripted variable. *PROC* is then set to zero for the translation of the expression between the round brackets or the subscript expressions, respectively. (This has not been shown in earlier examples.)

Since an implicit subroutine is treated as a block, the block level count *n* is increased by one and the operation '*BE* (*n*,*0*)' is generated as the first operation of the subroutine. This operation can be generated as a complete operation as an actual parameter cannot have any identifiers declared within it and so the working space required by the block is known to be zero. For this reason it is not necessary to create a new block in the name list for an implicit subroutine.

At the end of an implicit subroutine the operation *EIS* (End of Implicit Subroutine) is generated.

3.4.3.2.1.2 Parameter Comment Convention. The basic cycle routine is used to remove comments from the ALGOL text.

When the state variable *PROC* is set, this routine will deliver ',' as the current delimiter instead of

) ⟨letter string⟩ : (

However, in the case of a subscripted variable used in an actual parameter part it is not permissible to make use of this type of comment between the subscript expressions.

This is a further reason for preserving the value of *PROC* at a left square bracket and setting *PROC* to zero for the translation of the subscript expressions.

Example

P (*A* [*0*) this is the first element of matrix *A* : (*0*])

This is not legal ALGOL as a parameter comment is being used between the two subscript expressions.

3.4.3.2.1.3 Types of Actual Parameter

3.4.3.2.1.3.1 Identifier. The appropriate 'actual operation' (see section 2.5) is stacked for this actual parameter.

This identifier is recognized as an actual parameter if the state variable *PROC* has been set and if two parameter delimiters are separated by only this identifier.

If the current part of the name list contains a used entry for the actual parameter identifier a skeleton 'actual operation' *PO* is stacked. The name list entry number of the used entry for this identifier is also stacked with the skeleton 'actual operation'. When this skeleton operation is added to the object program the contents of the *np* column of the name list entry for the identifier will be stored in the parameter position of this operation. By this means the skeleton operation will point to the next element in the chain of operations associated with this identifier. The address of the parameter of this operation is then stored in the *np* column.

This skeleton 'actual operation' cannot be added on to the chain of used entries when it is stacked as the position it will eventually occupy in the object program is not known until the unstacking occurs. By stacking the name list entry number, rather than the contents of the *np* column of the name list entry of the identifier, the possibility of more than one use of the identifier in the actual parameter part is allowed for.

Example

P (*B*, *B* + *1*)

(Assuming that the *np* column of the used entry for *B* contains the address *50* when this procedure call is reached.) If the skeleton 'actual operation *PO*, *50*' is stacked for the first use of *B* the *np* column cannot be given the address in the object program that will be occupied by this operation. Then at the use of *B* in the second parameter another skeleton operation has to be generated but the *np* column has not yet been set up with the address of the last skeleton operation in the chain (i.e. the skeleton 'actual operation' still in the stack).

3.4.3.2.1.3.2 Constant. The six-syllable representation of the value of the constant is stored in the object program. The appropriate 'actual operation' ('*PRC*, *i* − *6*', '*PIC*, *i* − *6*' or '*PBC*, *i* − *6*') whose parameter gives the address of the constant, is then stacked.

3.4.3.2.1.3.3 Subscripted Variable. An implicit subroutine is generated for the operations required to evaluate the address of the subscripted variable and the appropriate 'actual operation' is stacked.

Normally the state variable *E* is set to expression level at the start of an implicit subroutine but if the actual parameter is a subscripted variable, *E* is left at statement level. This ensures that *INDA* will be generated for this variable as the implicit subroutine is required to deliver the address rather than the value of the subscripted variable.

Example

P (*A* [*1*, *2*])

is translated into

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*23*)	
3	*BE*	(*n*,*0*)	
6	*TRA*	*A*	Assuming *A* to be of type **real**
9	*TIC1*		
10	*TIC*	'*2*'	
17	*INDA*		
18	*EIS*		
19	*PSR*	(*3*), –	This operation points to the operation *BE* at the start of the subroutine
23	*CF*	*P*, *1*	
27	...		

At the left square bracket an implicit subroutine is set up and the 'actual operation *PSR*, *3*' is stacked. Then at the right round bracket (which is recognized as a parameter bracket since *PROC* is set) the last delimiter is ']' and so the operation *EIS* is generated in the object program.

3.4.3.2.1.3.4 Expression. An implicit subroutine is generated for an actual parameter which is an expression, in a similar way to that shown in the previous section, and an 'actual operation' *PSR*, whose parameter gives the starting address of this subroutine, is stacked.

An actual parameter expression could be a designational, algebraic or arithmetic expression and it is not always possible to decide the type of the expression.

Example

P (**if** *b* **then** *L1* **else** *L2*);

If the declarations for *L1* or *L2* are not available it is not possible to decide the type of the above actual parameter expression without referr-

ing to the specification of the formal parameter corresponding to the procedure *P*.

Thus any skeleton result operation *RO* that is generated for a parameter expression of unknown type could be replaced eventually by the following four-syllable operation

TL (*a*), *m*

rather than a more usual three-syllable operation, such as

TRR (*n,p*), *TIR* (*n,p*), etc.

To allow for this the normal three-syllable skeleton operation is generated and followed by a one-syllable operation *DUMMY*. This operation will be overwritten by the fourth syllable of a 'Take Label' operation or left as *DUMMY* if the skeleton operation is replaced by any other form of 'Take Result' or *CFZ* or *CFFZ* operation.

In the above example the object program that would result would be (assuming skeleton operations are generated for *L1* and *L2*)

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	(*28*)	
3	*BE*	(*n,0*)	
6	*TBR*	*b*	
9	*IFJ*	(*19*)	
12	*RO*	( )	*L1*
15	*DUMMY*		
16	*UJ*	(*23*)	
19	*RO*	( )	*L2*
22	*DUMMY*		
23	*EIS*		
24	*PSR*	(*3*), –	
28	*CF*	*P, 1*	
32	...		

If the identifiers *L1* and *L2* prove to be labels the skeleton operations will be replaced by 'Take Label' operations and the object program for the above statement becomes

Syll	*Op*	*Par*
0	*UJ*	(*28*)
3	*BE*	(*n,0*)
6	*TBR*	*b*
9	*IFJ*	(*19*)
12	*TL*	*L1, m*
16	*UJ*	(*23*)
19	*TL*	*L2, m*
23	*EIS*	
24	*PSR*	(*3*), –
28	*CF*	*P, 1*
32	...	

However, if the identifiers *L1* and *L2* are declared to be of type **real** the object program for the procedure call becomes

Syll	*Op*	*Par*
0	*UJ*	*(28)*
3	*BE*	*(n,0)*
6	*TBR*	*b*
9	*IFJ*	*(19)*
12	*TRR*	*L1*
15	*DUMMY*	
16	*UJ*	*(23)*
19	*TRR*	*L2*
22	*DUMMY*	
23	*EIS*	
24	*PSR*	*(3), –*
28	*CF*	*P, 1*
32	...	

Any delimiter which could start an expression is checked to see if it is the start of a parameter expression. In this case the state variable *PROC* would be set and the state variable *LD* would contain a left round bracket or comma.

The parameter delimiter comma is not stacked because the 'actual operation' with its priority of zero, which was stacked for the parameter, acts as the opening bracket for the parameter expression. The state variable *E* is set to expression level as soon as possible in a parameter expression. This is done at the first delimiter unless this delimiter is '['. However, should the subscripted variable be the first item in an expression the operation *INDR* must be generated instead of *INDA* as in the previous section. Thus, if the state variable *E* is set to statement level neither *INDA* nor *INDR* is generated but *INDA* is stacked with a high priority (namely twelve). This item is found and unstacked by the next delimiter and the appropriate operation (i.e. *INDA* or *INDR*) is generated. If the next delimiter is a parameter delimiter the operation *INDA* is generated; otherwise *INDR* is generated and the state variable *E* set to expression level for the translation of the remainder of the parameter expression.

Example

; *R* (*A* [*0*] – *1*);

The left round bracket is recognized as a parameter bracket as it is preceded by the identifier *R*. When this delimiter has been dealt with the top of the stack will contain

Stacked Item	*Priority*	*Remarks*
(	*0*	
0, 1, 0		This item contains the values of the state variables *PROC*, *E* and *TYPE*
R		
UJ, i – 2	*13*	

At the delimiter '[' an implicit subroutine is generated because the last delimiter was '(' and *PROC* is set. Since this actual parameter might be a subscripted variable rather than an expression containing the subscripted variable as a first item, the state variable *E* is left at statement level. The operation *TIC0* is generated for the subscript expression and then at the delimiter ']' the item *INDA* is stacked with a stack priority of twelve. At the delimiter '−' the state variable *E* is set to expression level, *INDA* unstacked and an *INDR* operation is generated (since the state variable *E* indicates that an expression is being translated). The minus is then stacked with its priority of nine.

The closing round bracket is recognized as a parameter bracket as *PROC* is set. After unstacking the minus into the object program the top of the stack contains '*PSR, 3*'.

Since the last delimiter is not a parameter delimiter (i.e. either a left round bracket or a comma) an operation *EIS* is generated to complete the implicit subroutine for the previous parameter. The 'actual operations' (in this case there is only one) are then unstacked, counted and added to the object program and this leaves '(' at the top of the stack. This is unstacked and discarded. The state variables *PROC*, *E* and *TYPE* are then reset from the values at the top of the stack. Using the type information contained in *TYPE* and the information concerning the number of dimensions given by the number of 'actual operations' unstacked into the object program, the name list entry for the procedure identifier *R* at the top of the stack is checked. The identifier *R* is then unstacked and discarded and the *UJ* operation, at the object program address given with the item *UJ* at the top of the stack, is completed to the current program address (*i*). The item (*UJ*) at the top of the stack is then unstacked and discarded. A *CF* operation is generated for *R* (unless *R* is a formal parameter, when the operation *CFF* is generated) and the parameters of this operation are the syllable address of the declaration for *R* (this is given in the *syll* column of a declared name list entry for *R*) and the number of actual parameters in this procedure call. Thus the object program generated for this procedure call is

Syll	*Op*	*Par*
0	*UJ*	(*18*)
3	*BE*	(*n,0*)
6	*TRA*	*A*
9	*TIC0*	
10	*INDR*	
11	*TIC1*	
12	−	
13	*EIS*	
14	*PSR*	(*3*), −
18	*CF*	*R, 1*
22	...	

It should be noted that an actual parameter which consists of a plus sign followed by either a number or an identifier is treated as an expression and an implicit subroutine is generated as described above. This is the only case when the unary plus sign has an effect on the object program.

Example

$P(+A, +2{\cdot}5_{10}\ 1)$

The object program produced from the procedure call is

Syll	*Op*	*Par*
0	*UJ*	(*29*)
3	*BE*	(*n*,*0*)
6	*TRR*	*A*
9	*EIS*	
10	*BE*	(*n*,*0*)
13	*TRC*	'$2{\cdot}5_{10}\ 1$'
20	*EIS*	
21	*PSR*	(*10*), –
25	*PSR*	(*3*), –
29	*CF*	*P*, *2*
33	...	

3.4.3.2.1.3.5 ***String.*** An actual parameter which is a string is dealt with by the delimiter routine STRING, which is entered when the opening string quote is reached.

The basic symbols forming a string are stored in the object program as a sequence of 8-bit characters. Before storing the opening string quote and again after storing the closing string quote the syllable count *i* is increased to point to the first syllable of the next word. In this way the author of a procedure declaration requiring the string can use the appropriate number of words of object program knowing that they contain only the basic symbols comprising the string.

The 'actual operation' *PST* is stacked for this type of actual parameter. The parameter of *PST* points to the first syllable of the word containing the first string character (i.e. the opening string quote).

Finally a check is made that the closing string quote is immediately followed by a parameter delimiter or a closing round bracket.

3.4.3.2.2 ***Translation of Closing Parameter Bracket.*** After dealing with the last parameter, the top of the stack will contain as many 'actual operations' as there are parameters in this procedure call.

Whilst unstacking the 'actual operations' into the object program a note is kept of the number of 'actual operations' unstacked; when the last 'actual operation' has been unstacked the top of the stack will contain '(', and the number of parameters (*X*) in this procedure call will be known. Any skeleton

'actual operation' is dealt with as described in section 3.4.3.2.1.3.1. The delimiter '(' is unstacked and discarded and then the state variables *PROC*, *E* and *TYPE* are reset from the values at the top of the stack. These values are then unstacked and discarded.

The current part of the name list is checked for an entry corresponding to the procedure identifier which is at the top of the stack. If an entry exists, information concerning the current use of the procedure identifier (given by the state variables *E* and *TYPE* and the number of parameters *X*) is checked against the details given in the name list entry; if no entry exists, a used entry is created and details concerning the current procedure call are placed in this entry. The appropriate 'Call Function' operation is generated depending on the type of name list entry for the procedure identifier.

(*i*) If it is a declared entry with the *f* column not set, the operation *CF* is generated; this operation has as its two parameters the contents of the *syll* column of the declared entry and the number of parameters (*X*) respectively.

(*ii*) If it is a declared entry with the *f* column set, the operation *CFF* is generated; this operation has as its two parameters the contents of the *np* column of the declared entry and the number of parameters (*X*) respectively.

(*iii*) If it is a used entry the skeleton operation *FO* is generated. The first of the two parameters of this operation is used for the chain link by storing the contents of the *np* column of the used entry in this parameter space and then placing the address of this parameter in the *np* column. The number of parameters (*X*) is stored as the second parameter of this skeleton operation. The procedure identifier is then unstacked and discarded. If the state variable *E* is set to statement level a further object program operation is generated, namely *REJECT*. In this case the procedure has been called by means of a procedure statement and so there should be no ALGOL between this closing round bracket and the end of statement delimiter (i.e. ';', **end** or **else**).

Example

$$; R (A, B [0, 1], C, 2{\cdot}5_{10} 1, B [0, 1] + 1);$$

Figure 12 shows how the stack is used during the translation of this procedure statement. The generation of the object program is also shown for each stage of the translation.

It is assumed that *A* and *B* are of type **real** and that *C* is a formal parameter.

A brief description is given below of the items that are stacked and the operations that are generated at each delimiter (reference should be made to the previous sections for the reasons for stacking the various items).

(*i*) The delimiter '('. At this point an *UJ* operation is generated as an incomplete operation; its parameter will be filled with the address of the *CF* operation at the corresponding closing bracket.

Four items are stacked; these are a reminder to complete the *UJ* opera-

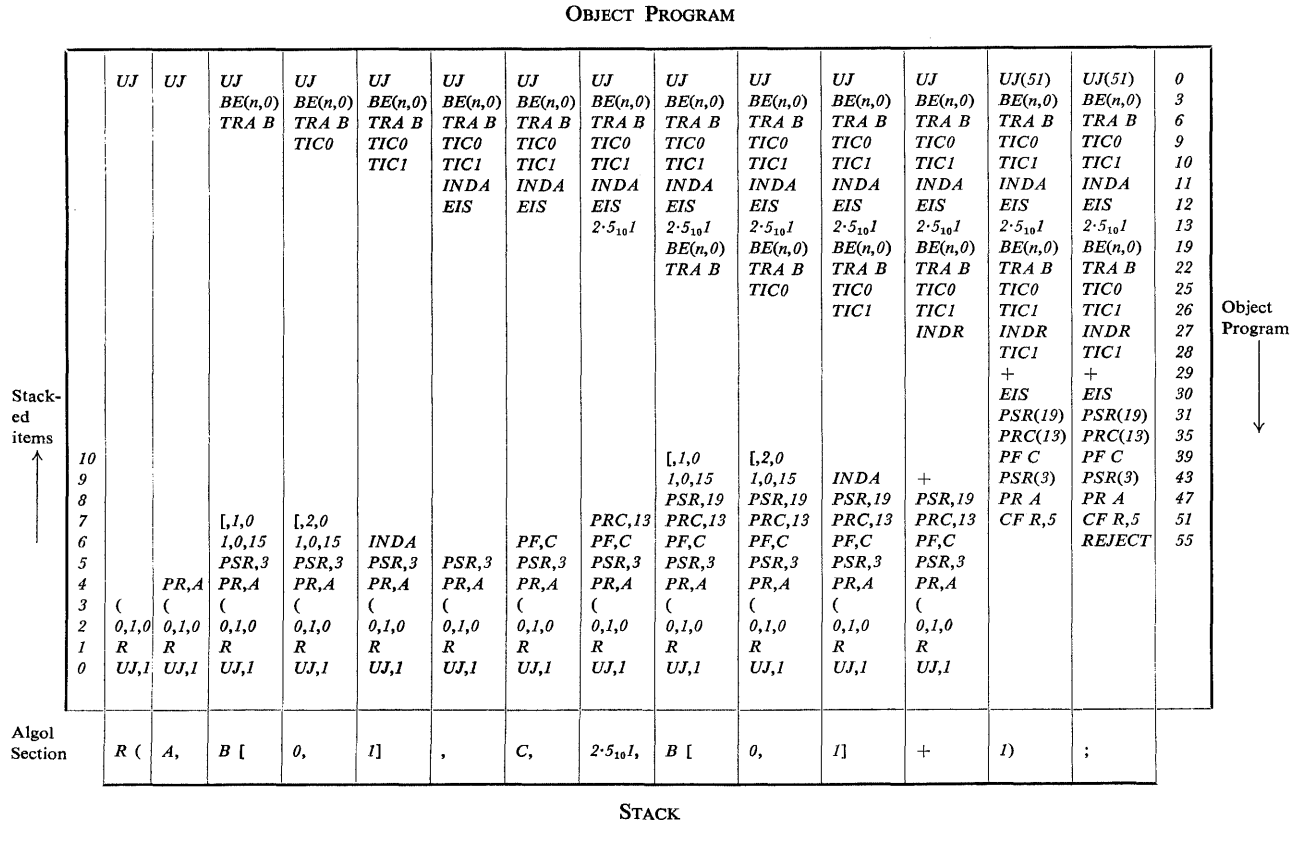


FIG. 12. Generation of the Object Program and use of the Stack for a procedure statement.

tion, the procedure identifier *R*, a word containing the values of the state variables *PROC*, *E* and *TYPE*, and finally the current delimiter '('.

(*ii*) The first parameter comma. An 'actual operation' *PR* is stacked for the actual parameter *A*.

(*iii*) Left square bracket. At the left square bracket an implicit subroutine is created for the second parameter and a *BE* operation is generated. The operation *TRA* is generated for the array identifier *B*. Three items are added to the stack; these are the 'actual operation' *PSR*, a word containing the values of *PROC*, *E* and *I* (this contains the number of the name list entry for the array identifier *B*, which is assumed to be fifteen, at this point), and the delimiter '['.

(*iv*) The subscript delimiter comma. An operation is generated for the preceding subscript expression (namely the constant zero) and the dimension counter which is stacked with the delimiter '[' is increased by one.

(*v*) Right square bracket. The operation *TIC1* is generated for the second subscript expression; '[' and the preserved values of *PROC*, *E* and *I* are unstacked. Thus the operation *INDA* is stacked to be found and unstacked by the next delimiter.

(*vi*) The second parameter comma. At this point the item *INDA* is unstacked into the object program and the operation *EIS* is generated to complete the operations for the implicit subroutine.

(*vii*) The third parameter comma. The 'actual operation' *PF* is stacked for the third actual parameter *C*.

(*viii*) The fourth parameter comma. The six-syllable representation of the constant is added to the object program and the corresponding 'actual operation' *PRC* is stacked.

(*ix*) Left square bracket. Similar to (*iii*).

(*x*) Subscript comma. Similar to (*iv*).

(*xi*) Right square bracket. Similar to (*v*).

(*xii*) The delimiter '+'. *INDA* is unstacked and the operation *INDR* is generated. The current delimiter is then stacked.

(*xiii*) Closing parameter bracket. The operation *TIC1* is generated for the last item in the preceding actual parameter expression, the delimiter '+' is unstacked into the object program and the operation *EIS* is generated to complete the set of operations for the implicit subroutine.

The five 'actual operations' at the top of the stack are then unstacked into the object program, a *CF* operation is generated and the four items stacked at the opening parameter bracket are unstacked.

(*xiv*) The statement delimiter ';'. At this point the operation *REJECT* is generated.

3.4.4 Translation of For Statements

A for statement consists of a for clause followed by a statement (called the controlled statement). The controlled statement is turned into a block by the Translator even if it is a simple statement or a compound statement. This

ensures that the controlled statement is always entered by means of the for clause and cannot be entered by a go to statement (Revised ALGOL Report section 4.6.6). If in fact this statement is an unlabelled block the Translator does not create an extra block.

3.4.4.1 Translation of the For Clause

At the delimiter **for** the state variable *E* is inspected to ensure that it is set to statement level. If the state variable *V* shows that a declaration has just been translated, *V* is set to two to show that no more declarations are allowed in the current block. Then if the top of the stack contains *UJ*, the incomplete operation *UJ* at the address given in the stacked item is completed to cause a jump to the current address in the object program.

A marker '*F*' is used to indicate that a for clause is being translated. Since the definition of a for clause is not recursive (i.e. a for clause cannot contain another for clause inside it) the marker *F* is never stacked.

At the delimiter **for** the marker *F* is set and the delimiter **for** is stacked with a stack priority of zero. The value of the object program counter *i* is also stacked with **for** in order to indicate the start of the object program for the controlled variable. An incomplete operation *UJ* is generated and its parameter will be filled in when the delimiter ':=' is reached. The state variable *TYPE* is then set to arithmetic for the translation of the controlled variable since the controlled variable may be of type **real** or **integer** but not of type **Boolean.**

The state variable *E* remains at statement level for the translation of the controlled variable. This is to ensure that the object program operation generated will cause the address, rather than the value, of the controlled variable to be calculated.

At the delimiter ':=' the marker *F* shows that this delimiter follows the controlled variable of a for clause. If the delimiter is preceded by an identifier, this is the controlled variable (which is a simple variable) and the appropriate 'Take Address' operation is generated for it. Otherwise this delimiter will be preceded by a right square bracket (as the controlled variable must either be a simple or a subscripted variable); in this case the item *INDA* at the top of the stack is unstacked into the object program.

The top of the stack will then contain **for** together with an address. The parameter of the *UJ* operation at this address is filled in with the value '*i* + *1*' to cause the object program operations for the controlled variable to be bypassed upon entry to the for statement. The following object program operations are generated.

Op	*Par*
LINK	
CFZ	( )
FORS1	

A second address '*i* − *3*' is added to the delimiter **for** at the top of the

stack as a reminder to fill in the parameter of the *CFZ* operation when the delimiter **do** is reached.

The for list element following the delimiter ':=' is assumed to be a step-until type of element. A reminder to correct this assumption, if necessary, is stacked (i.e. '*FORS1*, $i - 1$') with a stack priority of zero.

The state variable E is then set to expression level for the translation of the for list elements.

3.4.4.1.1 Arithmetic Expression Element. The arithmetic expression is translated in the normal way, using the stack to re-order it into the Reverse Polish form required for the object program.

The for list element delimiter following this element (i.e. comma or **do**) has a compare priority of two to complete the translation of the arithmetic expression.

Since this for list element was assumed to be a step-until element the top of the stack must now contain *FORS1* with the address of the *FORS1* operation generated for this element. This operation is changed to *FORA* and the item at the top of the stack is unstacked and discarded.

If the current for list element delimiter is a comma, the operations *LINK* and *FORS1* are generated. This is because the next for list element is also assumed to be a step-until element and once again '*FORS1*, $i - 1$' is stacked as a reminder to correct the assumption, if necessary. If, however, the current for list element delimiter is **do** only the operation *LINK* is generated as this is the last for list element of the for clause.

Example

for $V := A + 1$,

When the delimiter ':=' has been dealt with, the following object program operations will have been generated

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	*(7)*	
3	*TIA*	*V*	*V* is assumed to be of type **integer**
6	*LINK*		
7	*CFZ*	( )	
10	*FORS1*		
11	...		

The stack contains the following items for this portion of the for clause.

Stacked Item	*Priority*
FORS1, 10	*0*
for *3, 8*	*0*

The state variable E is set to expression level for the translation of the arithmetic expression following the delimiter ':='. The state variable

I

TYPE is left at arithmetic for the translation of this arithmetic expression. At the delimiter ',' the delimiter '+' will be at the top of the stack and the appropriate operations will have been generated for *A* and *1*. The comma acts as the closing bracket for the preceding arithmetic expression and its compare priority of two clauses the '+' to be unstacked into the object program, which will then contain

Syll	*Op*	*Par*	*Remarks*
0	*UJ*	*(7)*	
3	*TIA*	*V*	
6	*LINK*		
7	*CFZ*	( )	
10	*FORS1*		
11	*TRR*	*A*	*A* is assumed to be of type **real**
14	*TIC1*		
15	+		
16	...		

The top of the stack contains the reminder to replace the operation *FORS1* at syllable *10* by the operation *FORA*.
The object program for this for list element is completed by the generation of the operation *LINK*.
Since the comma precedes another for list element the operation *FORS1* is generated and '*FORS1*, $i - 1$' is stacked as a reminder to change it, if necessary.
The state variable *TYPE* is reset to arithmetic for the next for list element as it may have been restricted to **integer** during the translation of the current for list element.

3.4.4.1.2 Step-Until Element. This for list element consists of

⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until** ⟨arithmetic expression⟩

In this case the closing bracket of the first arithmetic expression is the delimiter **step** which will thus have a compare priority of two.

When **step** has dealt with the preceding arithmetic expression, the reminder *FORS1* at the top of the stack may be discarded as the assumption made at the start of this for list element has proved to be correct. The operation *LINK* is generated.

The delimiter **step** also acts as the opening bracket for the second arithmetic expression. *FORS1* is stacked with a stack priority of zero and the operation *FORS2* is generated.

This time a parameter of minus one is stacked with *FORS1* to indicate that this item was stacked at **step** and that the second arithmetic expression in this for list element is to be followed by **until**.

The state variable *TYPE* is reset to arithmetic for the translation of the second arithmetic expression.

Similarly, the delimiter **until** has a compare priority of two since it is a closing bracket of an arithmetic expression. After dealing with the preceding arithmetic expression the top of the stack must contain '*FORS1*, -1' to show that the delimiter **step** preceded the expression. This item at the top of the stack is then replaced by *FORS2* with a stack priority of zero.

The stacked *FORS2* acts as the opening bracket for the third and final arithmetic expression of this for list element. Once again *TYPE* is reset to arithmetic for the translation of the arithmetic expression following this delimiter.

The action of the for list element delimiter, which follows this for list element, is similar to that described for the arithmetic expression element except that the item *FORS2* is left at the top of the stack. This item at the top of the stack is then unstacked and discarded.

Example

, A [I] **step** $x \div 2$ **until** $m \times n - 2$,

At the comma preceding this element the operation *FORS1* is generated and '*FORS1*, $i - 1$' is stacked with a priority of zero. *TYPE* is also set to arithmetic.

The translation of the subscripted variable has been described before. In this case the operation *INDR* is generated as the state variable *E* is already set to expression level and there is no doubt as to whether *INDA* or *INDR* is required. The delimiter **step** causes no unstacking as *FORS1* (with its priority of zero) is already at the top of the stack. This item is replaced by '*FORS1*, -1' also with a priority of zero and the operations *LINK* and *FORS2* are generated. *TYPE* is reset to arithmetic although, in this example, it will still be at arithmetic at the end of the preceding expression (i.e. the subscripted variable). The object program for the for list element at this point contains

Syll	*Op*	*Par*
0	*FORS1*	
1	*TRA*	*A*
4	*TIC1*	
5	*INDR*	
6	*LINK*	
7	*FORS2*	
8	...	

At the delimiter '$\div$' *TYPE* is restricted to **integer** as the operand following must be of type **integer**.

Thus, at **until**, operations for x and *2* will have been generated and '$\div$' will be at the top of the stack. The delimiter **until** causes '$\div$' to be unstacked into the object program, the operation *LINK* to be generated and the item '*FORS1*, -1' now at the top of the stack to be replaced by *FORS2*, also with a priority of zero. The object program now contains

Syll	*Op*	*Par*
0	*FORS1*	
1	*TRA*	*A*
4	*TIC1*	
5	*INDR*	
6	*LINK*	
7	*FORS2*	
8	*TIR*	*x*
11	*TIC*	'*2*'
18	÷	
19	*LINK*	
20	...	

The state variable *TYPE* is once more reset to arithmetic and the last arithmetic expression is translated in the normal way. At the comma following this arithmetic expression the top of the stack will contain the delimiter '−'. This delimiter is unstacked into the object program as its stack priority is greater than the compare priority of the comma. *FORS2*, which is then at the top of the stack, is unstacked and discarded. The operation *LINK* is generated to complete this for list element and *FORS1* is generated as the first operation for the next for list element. As before, a reminder to change this operation if necessary is stacked as '*FORS1*, $i - 1$' with a priority of zero.
The complete object program for this for list element is

Syll	*Op*	*Par*
0	*FORS1*	
1	*TRA*	*A*
4	*TIC1*	
5	*INDR*	
6	*LINK*	
7	*FORS2*	
8	*TIR*	*x*
11	*TIC*	'*2*'
18	÷	
19	*LINK*	
20	*TRR*	*m*
23	*TRR*	*n*
26	×	
27	*TIC*	'*2*'
34	−	
35	*LINK*	
36	*FORS1*	
37	− −	

3.4.4.1.3 While-Element. This type of for list element consists of
⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩

Thus the delimiter **while** acts as a closing bracket to the arithmetic expression and needs a compare priority of two.

When the arithmetic expression has been dealt with the top of the stack will contain *FORS1*. This is the reminder that the operation *FORS1* was generated as the first operation for this element. In this case the operation *FORS1* is replaced by *FORW* and the reminder is unstacked and discarded.

To complete the set of operations for the arithmetic expression a *LINK* operation is generated.

FORW is stacked to act as the opening bracket for the Boolean expression following the delimiter **while**, and as such it has a stack priority of zero.

The state variable *TYPE* is set to algebraic to allow for the real, integer or Boolean variables in the Boolean expression following the delimiter **while**.

The for element delimiter following this element has a compare priority of two to enable the translation of the Boolean expression to be completed.

At this point, the top of the stack contains *FORW*, which is unstacked and discarded; the operation *LINK* is generated.

If the for list element delimiter is a comma, a *FORS1* operation is generated for the start of the next for list element and once again a reminder to change it, if necessary, is stacked.

Example

,*V* + *1* **while** *V* < *10*,

FORS1 has been generated as the first of the set of operations for this element and the item *FORS1* stacked.
When the delimiter **while** is reached the operations for *V* and *1* will have been added to the object program, which will then contain

Syll	*Op*	*Par*
0	*FORS1*	
1	*TIR*	*V*
4	*TIC1*	
5	...	

The delimiter '+' will be at the top of the stack and this is unstacked by **while** which has a compare priority of two. This leaves the following item at the top of the stack

Stacked Item	*Priority*
FORS1, 0	*0*

Since this element has turned out to be a while-element the *FORS1* operation, whose address is given by the item at the top of the stack, is replaced by *FORW*, and the item at the top of the stack is discarded.
A *LINK* operation is generated, *FORW* is stacked with a stack priority of zero and the state variable *TYPE* is set to algebraic.
At the delimiter '<', *TYPE* is restricted from algebraic to arithmetic as

the identifier or constant preceding this delimiter cannot be of type **Boolean**. An operation is then generated for *V* and the delimiter '<' is stacked (no unstacking takes place before '<' is stacked as the item at the top of the stack, *FORW*, has a stack priority of zero). At the delimiter comma, an operation is generated for *10*, '<' is unstacked into the object program and the operation *LINK* is generated.
Thus the object program for this element is

Syll	*Op*	*Par*
0	*FORW*	
1	*TIR*	*V*
4	*TIC1*	
5	+	
6	*LINK*	
7	*TIR*	*V*
10	*TIC*	'*10*'
17	<	
18	*LINK*	
19	...	

The item *FORW* is then unstacked and discarded and since the comma signifies that this for clause contains another for list element *FORS1* is stacked and generated in the usual way.

3.4.4.1.4 Translation of the Delimiter **do.** The first duty of **do** is that of a closing bracket for the preceding for list element.

When the translation of this element has been completed the translation of the for clause has also been completed and the marker *F* is cleared. The item at the top of the stack must then be **for**. This item has two addresses stacked with it. The first of these is required as the second parameter of the operation *FBE* which is generated at this delimiter. The second is the address of the parameter of the *CFZ* operation generated at the start of the for clause. This enables the address of the *FBE* operation to be stored as the parameter of the *CFZ* operation. The item **for** can then be discarded.

The following two incomplete operations are generated

Op	*Par*	*Remarks*
FSE	( )	
FBE	( ), (*a*)	Where '*a*' is the first of the two addresses stacked with **for**.

The *FSE* operation acts as an extra for list element and is used to jump to the statement following the for statement when all the elements in the for clause have been obeyed. Thus it requires, as a parameter, the address of the first operation in the object program after the set of operations generated for the for statement; this address is not known until the controlled statement has been translated.

The *FBE* operation is a special type of *BE* operation and it requires (*n*,*L*) as its first parameter. Although the level *n* of the block created for the controlled statement is known, the extent of the first order working storage is not and so this parameter is filled in when the controlled statement has been translated. In fact the value of *L* will be known at the end of the declarations for the block created for the controlled statement but for convenience the *FBE* and *FSE* operations are both filled in at the end of the for statement.

Since the controlled statement is made into a block the following items are stacked:

Stacked Item	*Priority*
V, *L*, *NL*	
for begin, *i*−4	*0*

The first of these items consists of a word containing the values of the state variables *V*, *L* and *NL* which must be preserved whilst the block created for the controlled statement is being translated.

The second of these items is the delimiter **begin** which has the marker **for** attached to show the type of block that is being translated. The program address of the first parameter of the *FBE* operation is also stacked with the **'for begin'** so that *FBE* and the preceding operation *FSE* can be completed at the end of the translation of the controlled statement. A new block is created in the name list by setting *NL* equal to *NLP*. *NLP* is then increased by one to ensure that the first entry for the new block is left blank since the block created for the controlled statement is not a procedure block.

Finally, the state variable *E* is set to statement level and *L* is set to four to allow space for the two accumulators required by step-until elements (see section 2.6.2.3).

3.4.4.2 Translation of the Controlled Statement.

A block is created for the controlled statement and **'for begin'** is stacked regardless of whether the controlled statement is a single statement, compound statement or block.

3.4.4.2.1 A Single Statement as the Controlled Statement. In this case the end of statement delimiter (i.e. ';' or **end**) must cause the block created for the controlled statement to be collapsed and the item **'for begin'** to be unstacked and discarded.

Example

for *i* := *1* **step** *1* **until** *n* **do** *A* [*i*] := *0*;

The delimiter ';' completes the translation of the preceding assignment statement and the top of the stack will then contain **'for begin'**, which was stacked at the delimiter **do**. This shows that the current delimiter ';' must also complete the translation of a for statement as follows

The operation *FR* is generated and the name list for the block created for the controlled statement is collapsed (see section 3.3.3). Using the address stacked with **'for begin'** the parameter of the *FBE* operation is filled in with the values (*n*,*L*) (i.e. the values of state variables *n* and *L* packed together into sixteen bits). The parameter of the *FSE* operation preceding the *FBE* operation is filled in with the value of the object program counter (*i*). The item **'for begin'** is unstacked and discarded; the top of the stack then contains the values of *V*, *L* and *NL* appertaining to the surrounding block. These are unstacked and used to reset the state variables.
Finally, the block level count (*n*) is reduced by one.

It is possible for several for statements to finish at the same end of statement delimiter. Thus, after completing the translation of a for statement, the top of the stack must be inspected once more to see if it contains another **'for begin'**. If it does, the translation of this for statement must be completed and the block set up for its controlled statement collapsed. This process continues until the top of the stack no longer contains **'for begin'**.

Example

for *i* := *1* **step** *1* **until** *n* **do**
for *j* := *1* **step** *1* **until** *m* **do** *A* [*i*, *j*] := *0*;

In this case the delimiter ';' is the end of statement delimiter for both of the for statements. The delimiter ';' causes the translation of both the inner and outer for statements to be completed.

3.4.4.2.2 An Unlabelled Compound Statement as the Controlled Statement. If the first delimiter of a controlled statement is **begin**, the item **'for begin'** at the top of the stack has a marker (bl) added to it. This marker must be deleted by the corresponding delimiter **end** before an end of statement delimiter can cause the translation of the for statement to be completed.

Example

begin
for *i* := *1* **step** *1* **until** *n* **do begin** *A* [*i*] := *0*;
S := *S* + *B* [*i*]
end
end

At the delimiter **do** the item **'for begin'** is stacked. A marker (bl) is added at the next delimiter, which is **begin**, to change **'for begin'** into **'for begin** (bl)'. The delimiter ';' separating the two statements in the compound statement will find the item **'for begin** (bl)' at the top of the stack after dealing with the preceding assignment statement. Since the marker (bl) is present on the **'for begin'** the delimiter ';' cannot cause the translation

of the for statement to be completed. After the first delimiter **end** has caused the translation of the preceding statement to be completed, the top of the stack will be '**for begin** (bl)'. The marker (bl) is deleted to leave '**for begin**' at the top of the stack.

The second delimiter **end** then finds the item '**for begin**' at the top of the stack. Since this item has no (bl) marker, the delimiter **end** must be the end of statement delimiter for the for statement and so the translation of the for statement is completed. As in the previous section, allowance must be made for several for statements ending at the same delimiter.

3.4.4.2.3 An Unlabelled Block as the Controlled Statement. Since a block has already been created for the controlled statement there is no need to create an extra block when the controlled statement is an unlabelled block.

At the delimiter **begin** the marker (bl) is added to '**for begin**' at the top of the stack and the state variable V is set to zero.

At the first declaration in the block, V is changed from zero to one and the translation of the declarations is commenced, since operations for setting up the new block were generated at the delimiter **do**.

The delimiter **end** at the end of the block causes the marker (bl) to be deleted from '**for begin** (bl)' so that the end of statement delimiter following can cause the translation of this for statement (and any other for statement ending at this delimiter) to be completed.

3.4.4.2.4 A Labelled Block or Compound Statement as a Controlled Statement. If the controlled statement is a labelled block, an extra block must be created for it (the reasons for this are given in section 2.6.1). The name list for the block created at the delimiter **do** will contain only one identifier, namely the label.

The delimiter **begin** is treated in the normal way when it follows the delimiter ':'. Thus, the delimiter **begin** is stacked with a stack priority of zero and the state variable V is set to zero.

If the delimiter following **begin** indicates a declaration, a new block is created in the usual way (see section 3.4.1). Then at the end of the controlled statement the delimiter **end** will find '**begin** (bl)' in the stack and so cause the translation of the block to be completed. However, if the delimiter following **begin** indicates a statement, the controlled statement is a compound statement. The item **begin** is left unaltered in the stack, to be found and discarded by the corresponding delimiter **end** at the end of the controlled statement.

The top of the stack will then contain the item '**for begin**' so that the end of statement delimiter following the delimiter **end** can cause the translation of the for statement to be completed.

Once again, the translation of any other for statement ending at the same delimiter must also be completed.

3.4.4.2.5 A Conditional Statement as the Controlled Statement. In the original ALGOL 60 Report the use of a for statement to control a conditional state-

ment could give rise to ambiguities. This situation has been remedied in the Revised ALGOL Report by changes to the syntax given in sections 4.1.1 and 4.5.1.

Example

if *b1* **then for** *i* := *a* **do if** *b2* **then** *S1* **else** *S2*;

According to the syntax of the original ALGOL Report it was not clear whether this was equivalent to

if *b1* **then for** *i* := *a* **do begin if** *b2* **then** *S1*
else *S2* **end**;

or

if *b1* **then begin for** *i* := *a* **do if** *b2* **then** *S1* **end**
else *S2*;

The Revised ALGOL Report makes it clear that the former is the correct interpretation. The latter interpretation requires a construction

⟨if clause⟩ ⟨for statement⟩ **else** ⟨statement⟩

which implies that the construction

⟨if clause⟩ ⟨for statement⟩

is a valid if statement. This is not the case since a for statement is not an unconditional statement.

The set of priority values which is used to control the stacking of delimiters ensures the correct translation of such a statement. The delimiter **for** is stacked with a zero priority, so shielding **then**. At the delimiter **do** the **for** is replaced in the stack by '**for begin**', again with a zero priority. This '**for begin**' is unstacked by the ';' after **else** has unstacked the second **then**. Thus the controlled statement is correctly treated as a conditional statement.

3.4.5 Translation of Code Procedures

Section 5.4.6 of the Revised ALGOL Report states that procedure bodies may be expressed in non-ALGOL code. In the Whetstone Compiler such code procedures are written in KDF9 User Code, according to the rules described earlier, in section 2.7.1.

Communication between the code procedure and the surrounding ALGOL text is by means of pseudo-instructions, similar in form to normal User Code instructions but referring to the formal parameters given in the heading of the code procedure. The Translator replaces these pseudo-instructions with normal User Code instructions which in the simplest cases fetch or store the contents of the appropriate formal accumulator. In certain cases, notably when the formal parameter is specified to be scalar and is called by name, the

Translator generates a short sequence of object program operations, and replaces the pseudo-instructions by normal User Code instructions which use the object program operations as a subroutine.

The object program operations which are generated are placed after the object program representation of the procedure heading, but the instructions of the code procedure are stored separately. At the end of translation the User Code Compiler is used to assemble together the various code procedures and a User Code version of the Control Routine, replacing all symbolic addresses by actual machine addresses, etc., thus forming an integrated machine code program. In this way the code procedures are transformed into subroutines of the Control Routine.

3.4.5.1 Procedure Heading

The procedure heading, which is written in ALGOL, is translated in the normal way (see section 3.4.1.4.1); name list entries are set up for the formal parameters, and *PE* and parameter list operations for the formal parameters are generated in the object program.

3.4.5.2 Procedure Body

The first delimiter of the procedure body is the delimiter **KDF9** and this indicates that a code procedure is being translated.

3.4.5.2.1 Translation of the Delimeter **KDF9.** Since the only first order working storage required for a code procedure is that required by the formal parameters, the *PE* operation can be filled in at this point. The address of the space reserved for the parameter of the *PE* operation is given with the item '**procedure begin**' at the top of the stack; the values of the block level n and the state variable L are packed together and stored as the parameter of *PE*.

An operation '*DOWN m*' is generated in the object program; at run time this causes the Control Routine to enter the m^{th} code procedure. The name list entry for each formal parameter is then inspected in order to generate the appropriate object program subroutines for the scalar or label parameters that have been called by name. The addresses of these subroutines are stored in the *syll* column of the name list entry. This can be done since the *syll* column is not otherwise used for formal parameters of code procedures.

Details are given below of the object program subroutines generated for these formal parameters.

3.4.5.2.2 Real Parameter Called by Name. Two subroutines are generated for this type of parameter; one is used to fetch the value of the parameter and the other to assign a value to the parameter. Each subroutine contains two object program operations; the address of the first of the four opera-

tions generated is stored in the *syll* column of the name list entry for the parameter. The operations generated are

TFR (*n*,*p*)
UP1
TFAR (*n*,*p*)
UP2

The contents of the *np* column of the name list entry for the parameters are stored in the parameter position of the operations *TFR* and *TFAR*.

3.4.5.2.3 Integer Parameter Called by Name. This is dealt with in a similar way. The four object program operations generated are

TFI (*n*,*p*)
UP1
TFAI (*n*,*p*)
UP2

3.4.5.2.4 Boolean Parameter Called by Name. This is also similar to a real or integer parameter except that the four object program operations generated are

TFB (*n*,*p*)
UP1
TFA (*n*,*p*)
UP2

3.4.5.2.5 Label Parameter Called by Name. One subroutine, comprising two object program operations, is generated for this type of parameter. The address of this subroutine is stored in the *syll* column of the name list entry for this parameter.

The two operations generated for this subroutine are

TFL (*n*,*p*)
GTA

3.4.5.2.6 Translation of Remainder of Procedure Body. When the name list entry for each of the formal parameters has been inspected and the appropriate object program subroutine generated, the rest of the procedure body is translated as follows.

Certain pseudo-instructions are allowed in the User Code procedure body to refer to the formal parameters. They are as follows.

'*a*'
= '*a*'
J '*a*'

The first of these signifies that the value of the parameter is required and it is replaced by instructions for fetching this value either directly, or in the

case of a scalar called by name, using the subroutine of object program operations.

The second of these instructions (='*a*') indicates that an assignment is to be made to the formal parameter and so the appropriate instructions are generated to replace this pseudo-instruction.

Finally, *J* '*a*' is replaced by instructions which will cause the procedure body to be left by means of a jump to the parameter.

Another pseudo-instruction that is used in code procedure bodies is the delimiter **EXIT**. This is replaced by instructions which will cause a return to be made to the Control Routine.

The body of the code procedure, with its pseudo-instructions replaced as shown above, is stored in a backing store. The end of a code body is indicated by the delimiter **ALGOL**. At this point the name list for the procedure is collapsed, the block level *n* is decreased by one and the item **'procedure begin'** at the top of the stack is unstacked and discarded. The state variables *V*, *L* and *NL* are reset from the values which are then at the top of the stack before these too are unstacked and discarded.

3.4.5.3 End of a Program Containing Code Procedures

At the end of the translation of an ALGOL program a check is made to find whether it contained any code procedures; if it did, these are copied from the backing store, where they were preserved, and are processed, along with a User Code version of the Control Routine, by the User Code Compiler to form a single integrated program.

3.4.6 Translation of a Program

3.4.6.1 Start of a Program

An ALGOL program may be a compound statement or a block (see section 4.1.1 of the Revised ALGOL Report). However, for convenience, a program is always considered to be a block and so at the first delimiter of a program, which must be **begin** or ':', the appropriate operations are generated and items stacked for setting up the block. Thus the following operations are generated

Syll	*Op*	*Par*
0	*CBL*	
1	*UJ*	( )
4	*BE*	( )
7	...	

and the following items are stacked

Stacked Item	*Priority*
UJ, $i+2$	*13*
x	
begin (bl)	*0*

The item x contains the values of the state variables V, L and NL packed together. If the first delimiter is ':' the program may be a labelled compound statement or a labelled block. In the first case the program is made into a block as described above but in the second case an extra block is created so that the outermost block contains only the label or labels preceding the first **begin** delimiter. In this way a jump to any one of the labels will cause the inner block to be left and re-entered again, and so causing, at run time, the storage of the block to be collapsed and then set up again. To do this the delimiter **begin**, which must follow immediately after the set of label declarations, is stacked with a marker (l). Then, if the next delimiter indicates a declaration, this item at the top of the stack has a marker (bl) added and operations for setting up an extra block are generated. However, if the next delimiter indicates a statement the item '**begin** (l)' is left in the stack as the opening bracket for the compound statement and the compound statement is then translated in the normal way.

3.4.6.2 End of a Program

The last delimiter of a program is **end** and its corresponding **begin** could have been stacked in any of the following three forms

begin (bl)
begin (bl) (l)
begin (l)

In the first case the program is an unlabelled block (or an unlabelled compound statement which has been made into a block) and this block is terminated. In the second case the program is a labelled block and so this delimiter **end** must cause the termination of two blocks, the outer one containing the label or labels of the program.

In the third case the program is a labelled compound statement and this item is unstacked and discarded whereupon '**begin** (bl)' will be at the top of the stack and then the block, set up for the program, is terminated.

When the name list is collapsed for the program block the only used entries should be those for standard functions.

3.4.7 Program Checking

The system of error checking has been briefly described in section 1.4.2.1. The basis of the system is to scan through the rest of a block in which an error has been found without doing any checking. Any inner blocks which are encountered during this scanning process are checked, but on return to the block being scanned the name list entries for the inner block are discarded, rather than collapsed in the normal way. At the end of the block containing the error, a full analysis of the ALGOL text is resumed. Thus a fairly simple standard process, which is brought into action at the finding of any kind of error, is used to enable a fair proportion of the errors in an ALGOL program to be found, without producing any spurious error messages.

Further errors in the same block, or errors concerning identifiers not declared or specified in any inner blocks, will not be found by this process.

When an error has been found, and the appropriate error message has been printed, the following actions are taken

(*i*) The marker *FAIL* is set. This marker is checked when the end of the program is reached, and if set, prevents entry to the Control Routine. It also has the effect of inhibiting the printing of warning messages about identifiers which have been declared but not used, and the checking that type procedures contain an assignment to the procedure identifier.

(*ii*) Starting at the top of the stack, items are discarded until a '**begin** (bl)' is reached. During this process the number of **begin** items is counted, and the result placed in the variable *LEVEL*. The name list entries for the current block are also discarded, using the current value of *NL* and the values of *NL* given with any '**for begin**' or '**procedure begin**' items that are discarded from the stack.

(*iii*) The ALGOL text is scanned until a **begin** or **end** delimiter (other than in a comment or a string) is reached.

If a **begin** is found, the next symbol is checked to find whether it is the start of a block or a compound statement. If it starts a compound statement one is added to *LEVEL* and scanning continues, otherwise *LEVEL* and a '**begin** (bl)' are stacked, with a marker to indicate that the containing block contains an error, and normal analysis of the ALGOL text is resumed. When the end of this inner block is reached the marker indicates that the name list is to be discarded rather than collapsed, and that the variable *LEVEL* is to be reset using the stacked value, before resuming the scan. If an **end** delimiter is reached during the scanning *LEVEL* is decreased by one, and if it is then zero, normal analysis of the ALGOL text is resumed, otherwise the scanning continues.

3.5 TRANSLATOR ROUTINES

The Translator considers an ALGOL program to be composed of sections which are bounded by the delimiters. In general each different delimiter is processed by a separate delimiter routine. A routine called the basic cycle routine is responsible for reading in the ALGOL and delivering it, a section at a time, to the appropriate routine.

Some of the tasks which are common to two or more delimiter routines are dealt with by a further set of subroutines.

3.5.1 Basic Cycle Routine

ALGOL programs that are to be translated by the Whetstone Compiler are prepared on a Flexowriter, which produces both a paper tape version and a printed copy of the program. The code punched on the paper tape is a case dependent six-bit code with underlined characters being used to form basic symbols; the full character set of the paper tape code is given in Appendix 4. The paper tape version of the program is used by the Compiler although the ALGOL program is considered to be that shown on the printed copy. If this printed copy is correct ALGOL the paper tape must be accepted by the Compiler. This means that any hidden features on the paper tape such as superfluous case definition characters are allowed for by the Compiler.

The six-bit input characters are converted by the basic cycle routine into a case independent eight-bit code and this internal code is used by the delimiter routines. The various typographical characters and any comments in the ALGOL are eliminated by the basic cycle routine. Another of the tasks undertaken by this routine is the processing of the constituents of numbers and identifiers.

3.5.1.1 Read Routine

This routine is responsible for reading in the ALGOL text, removing the typographical features and delivering the next ALGOL basic symbol to the basic cycle routine.

Two buffer areas are set aside in the core storage for use by the read routine so that the six-bit input characters can be read into one buffer whilst those already stored in the other buffer are processed. By this means it is possible to translate the ALGOL whilst it is being read in, provided that the characters stored in the one buffer can be processed in the same, or less, time than it takes to fill the other buffer.

A one-bit 'case register' is used to keep a record of the case of the current character and this register is adjusted whenever a case definition character is processed. Similarly an 'underline register' is set by the underline character

to determine whether the current character is underlined. Using the current character and the values of the case and underline registers a table look up is carried out to determine the equivalent eight-bit internal code. When the equivalent code has been found for an underlined character the underline register is cleared.

By means of the case and underline registers it is possible to allow for superfluous case definition characters and multiple underlining of a character. This is necessary if the ALGOL program is considered to be that shown on the printed copy produced by the Flexowriter.

Example

'case shift _ *b*'

or

'case shift case normal case shift _ _ _ *b*'

Both appear on the printed copy as '$\underline{b}$'.

The underline key on the Flexowriter is a 'dead key' (i.e. it does not move the carriage on) and so an underlined character is produced by punching first the underline and then the character. It is possible for a non-printing character which also does not move the carriage, such as a case definition character, to be punched between the underline and the character and this must be allowed for.

Example

'_ case shift *b*'

This will appear as '$\underline{b}$' on the printed copy and so this arrangement of input characters must be accepted as the constituents of $\underline{b}$.

However a character which moves the carriage of the Flexowriter cannot be punched between the underline and the character as this would not appear to be correct on the hard copy.

Example

'CRLF _ CRLF *b*'

This is not allowed as the constituents of '$\underline{b}$' as these characters would appear on the printed copy as

$\overline{b}$

(CRLF signifies 'Carriage Return Line Feed')
Similarly '_space *b*' is not allowed as this would appear on the printed copy as

$_{b}$

Since the underline character is produced by a dead key on the Flexowriter there is, allowing for spurious spaces, case shifts, etc., only one way to produce a basic symbol composed of underlined characters (e.g. b e g i n). However, if the ALGOL programs are prepared on a teleprinter which has an underline key that is not a dead key, a back space key would have to be provided in order to be able to produce an underlined character on the printed copy. The use of a backspace leads to complications as it is difficult to interpret the paper tape without reading to a CRLF and then creating a picture of the printed line in the machine. This is because there are many ways of punching the constituents of a basic symbol such as **begin** and, if the printed copy of the program looks right, any of these combinations must be accepted. To reduce the complications involved in the use of a backspace character it is possible to give a rule as to the order in which the constituents of an underlined character must be punched. This solves the problem for the Compiler but creates a problem for the programmer who cannot tell whether this rule has been obeyed by looking at the printed copy of the program. For this reason a character code which includes a backspace character should be avoided if possible, as should separate carriage return and line feed characters.

Example

If the underline key is not a dead key and a backspace key is used to produce an underlined character the following show some of the ways of producing the basic symbol **if**

i f backspace backspace _ _
_ backspace *i* _ backspace *f*
_ _ backspace backspace *i f*

The result of the table look up routine is an eight-bit internal code. This might be the code of a basic symbol such as '+' or merely that of a constituent of a basic symbol such as an underlined letter. Thus, it is still not possible to know which basic symbol is being processed in certain cases until the remaining constituents have been dealt with. To allow for this a code buffer of eight-bit codes is used. The latest code produced by the table look up routine is put in at one end of the code buffer and all the codes in the buffer are moved up so that the code that was at the other end of the buffer is taken out to be processed. In this way a code is not processed straight from the table look up routine but must first work its way to the other end of the code buffer. The code buffer is of sufficient length to hold all the constituent underlined characters of the longest basic symbol (i.e. p r o c e d u r e). If the code taken from the code buffer is an underlined character then the contents of the code buffer can be checked against a list of the various underlined basic symbols. This process is taken care of by another table look up routine called the DICT routine, which produces a single eight-bit code for the basic symbol. As the codes are taken out of the code buffer they are replaced by further

codes at the other end by the table look up routine, which forms eight-bit case independent characters from the input characters. The use of this code buffer, which in fact holds twelve eight-bit codes, means that characters are converted into their equivalent eight-bit code, stored in the code buffer and then processed only when they arrive at the other end. If the code taken from the code buffer is ':' the next code in the buffer is inspected to see whether it is '='. If this is the case these are the constituents of the delimiter ':=' and so these two codes are replaced by the eight-bit code for ':='.

The read routine also deals with the parameter comment convention. When the code taken from the buffer is ')' the state variable *PROC* is checked to find whether a parameter part is being translated. If this is so and if the delimiter ')' is not a character inside a string, the next code in the buffer is checked; if it is a letter then a parameter comment has to be ignored. To do this a marker '*pc*' is used to trace the passage through the comment. The marker *pc* is set to one at the code ')' and this code is discarded. Another character is then fetched from the input buffer, converted by the table look up routine and added to the bottom of the code buffer. The code which leaves the top of the code buffer must be either a letter or a ':' whilst *pc* is set to one. If it is a letter it is discarded and the process is repeated to fetch further characters until in fact a ':' emerges from the code buffer. This code is also discarded and the next code in the buffer is inspected to make sure that it is the delimiter '(' before it too is discarded. The codes which were discarded are replaced by the single eight-bit code for the delimiter ','.

Although the read routine is used primarily as a subroutine within the basic cycle routine it is also used as a subroutine by the delimiter routine STRING. Characters inside a string are stored as basic symbols and the read routine is used to deliver the basic symbols. For this reason care must be taken that the read routine does not discard any basic symbols as part of a parameter comment when it is delivering string characters.

3.5.2 ALGOL Section Routine

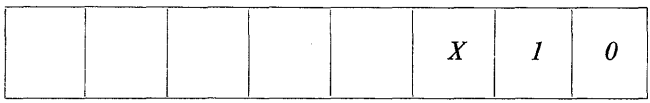
This routine processes the ALGOL program a section at a time and uses the read routine as a subroutine to produce the next basic symbol. The basic symbol could be a delimiter or a constituent of an identifier or a number. If it is a letter then it must be the first character of an identifier and so the read routine is re-entered to fetch the remaining letters or digits of the identifier and these are packed together in a 48-bit word. Although the internal code for the basic symbols is an eight-bit code it is arranged for the letters and digits to occupy codes in the range *0–63* so that the characters of an identifier can be packed as a six-bit code. In this way eight characters can be packed into a 48-bit word to represent an identifier; if an identifier has more than eight characters a message is printed to warn the programmer that only the first eight characters are being used for the identifier. The characters are packed into the identifier word from one end and when another letter or digit is to be added to the identifier those already stored are moved up six places. Thus

it is possible to distinguish between the identifier *X1* and *X10*, say, even if the binary code for the digit *0* is zero.

Example

identifier *X1* is stored as

whereas the identifier *X10* is stored as



If the basic symbol is a constituent of a number the rest of the constituents are fetched and the number is evaluated and stored in a 48-bit word for use by the delimiter routines. The delimiters '.' and '$_{10}$' are processed by the number conversion part of the ALGOL section routine as constituents of a number instead of being treated as delimiters by a delimiter routine.

Another delimiter that is processed within the ALGOL section routine is the delimiter **comment**. After checking the state variable *LD* (last delimiter) to make sure that this delimiter follows either the delimiter **begin** or the delimiter ';' the delimiter **comment** causes the read routine to be entered to fetch the next basic symbol. If this basic symbol is not the delimiter ';' it is discarded and the read routine is re-entered. This process is repeated until the delimiter ';' is delivered.

The ALGOL section routine sets the state variable *m* to indicate the type of ALGOL section that is being delivered to the appropriate delimiter routine. If the section consists of just a delimiter *m* is set to zero; if the section consists of an identifier followed by a delimiter *m* is set to one; finally, if the section, consists of a constant followed by a delimiter *m* is set to two. In the case of the comment convention described above

comment ⟨ any basic symbol not including ; ⟩ ;

is reduced to the delimiter ';' and *m* is set to zero since the section consists of just the delimiter.

One final task is performed by the ALGOL section routine before delivering the next ALGOL section to the appropriate delimiter routine and that is to check the type of the ALGOL section. The basic cycle routine has a parameter *p* associated with it; when the basic cycle routine is called from within a delimiter routine to fetch the next ALGOL section the parameter *p* is set to indicate the value that the state variable *m* must have before returning to the delimiter routine.

Example

After the delimiter **procedure** has been processed the basic cycle routine is entered to fetch the next ALGOL section. It is known that this section must consist of an identifier (i.e. the procedure identifier) and a delimiter (i.e. either '(' or ';') and so the parameter p is set to one. Thus before returning to the 'procedure heading' routine the state variable m is checked to make sure that it is equal to one also.

The parameter p is allowed to have two further values, namely three and four. If p is set to three there is no check on the type of the ALGOL section. The central loop of the translator consists of a call on the basic cycle routine with the parameter p set to three and in general a delimiter routine returns to this central loop when it has finished it tasks. The value of four is used for the parameter p to indicate that the basic cycle routine has been entered from the **end** delimiter routine to allow for the **end** comment convention. If p has the value four any constituents of identifiers or numbers are discarded immediately without wasting time processing them.

The **end** comment convention is not dealt with completely within the basic cycle routine as are the other comment conventions. This is because a common form of error in ALGOL programs is the omission of the delimiter ';' after the last of a series of one or more **end** delimiters, causing the statement following to be treated as a comment.

To draw attention to this possibility a message is printed to warn a programmer if a comment after an **end** contains a delimiter, before continuing the process of translation. The more usual forms of **end** comment, namely identifiers and letter strings, thus do not cause the printing of a warning message. The checking and discarding of any **end** comments is done by the **end** delimiter routine, using the basic cycle routine with the parameter p set to four.

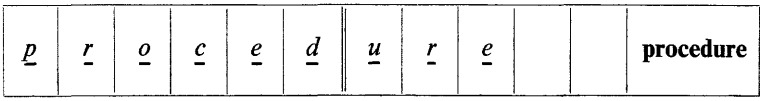
3.5.2.1 DICT Routine

The DICT routine is a table look up routine, which compares the characters in the code buffer with the sets of characters given in a table of basic symbols and their constituent underlined characters. When the code buffer is found to contain the constituent characters of a basic symbol, these characters are removed from the buffer, and an eight-bit representation of the basic symbol is used in their place.

The items in the list are stored as double-words with the internal code for the basic symbol given in the least significant eight bits of the second word, and the constituent underlined characters starting at the most significant end of the first word.

Considering the contents of the items in the list to be double-length integers (i.e. integers which are stored in a double-word store), the items are listed in descending order of magnitude. The content of the code buffer is also considered as a double-length integer and as such it is compared with each of the

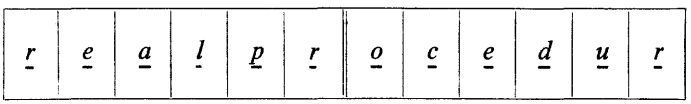
Example



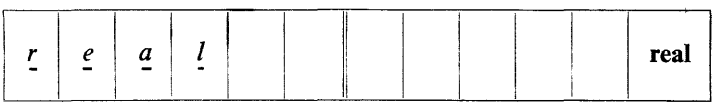
items in the list in turn until the list item is reached that is less than the integer value of the code buffer. This is the list item that is required and the code buffer should contain the underlined constituent characters of this basic symbol. These characters are taken from the code buffer one at a time and checked against the equivalent character in the list item before being discarded. Finally the eight-bit representation of the basic symbol is taken from the least significant part of the second word of the list item for delivering to the ALGOL section routine.

Example

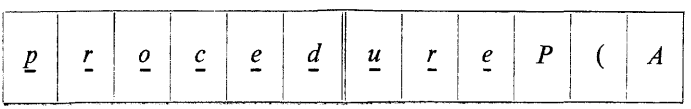
The code buffer might contain



This is compared with each of the items down the list until the following item is reached and recognized as the required item



Underlined characters are taken from the code buffer and the list item one at a time and compared until there are no underlined characters remaining in the list item. The eight-bit code for the basic symbol **real** is then extracted from the list item for use by the ALGOL section routine. Since the code buffer is refilled from the other end as the underlined characters forming the delimiter **real** were removed the buffer might now contain



It is possible to allow for two different versions of a basic symbol (e.g. **Boolean** or **boolean**) by storing each different form as a separate list item.

Naturally, spaces, changes to a new line, spurious case definition characters, etc., are allowed between the underlined characters of any basic symbol. In fact these are removed before the underlined characters are added to the code buffer, as described earlier.

3.5.3 Delimiter Routines

In general each delimiter is dealt with by a separate routine; the exceptions being the procedure heading routine which deals with the complete procedure heading, the own array routine which deals with the whole of the bound pair list of an own array segment and the delimiters **comment**, '.' and '$_{10}$' which are dealt with inside the basic cycle routine.

The delimiter routines are entered from the central loop of the Translator, which consists of a call of the basic cycle routine for the next ALGOL section and then an exit to the appropriate delimiter routine. Brief descriptions of the translator routines are given with the flow diagrams in Appendix 11.

REFERENCES

1. Allmark, R. H. and Lucking, J. R. (1963). Design of an Arithmetic Unit incorporating a Nesting Store. "Information Processing 1962", Proceedings of IFIP Congress 62, pp. 694–698. North-Holland Publishing Co., Amsterdam.
2. Arden, B. and Graham, R. (1959). On GAT and the Construction of Translators. *Comm. A.C.M.* **2**, 7, pp. 24–26.
3. Arden, B. and Graham, R. (1959). Letter to the Editor. *Comm. A.C.M.* **2**, 11, pp. A10–A11.
4. Arden, B., Galler, B. A. and Graham, R. M. (1961). Criticisms of ALGOL 60. *Comm. A.C.M.* **4**, 7, p. 309.
5. Arden, B., Galler, B. A. and Graham, R. M. (1962). An Algorithm for Translating Boolean Expressions. *J. A.C.M.* **9**, 2, pp. 222–239.
6. Backus, J. W. *et al.* (1957). The FORTRAN Automatic Coding System. "Proceedings of the WJCC", pp. 188–198. Institute of Radio Engineers, New York.
7. Backus, J. W. (1959). The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. "Information Processing", Proceedings of ICIP Paris, pp. 125–132. UNESCO, Paris.
8. Barton, R. S. (1961). A New Approach to the Functional Design of a Digital Computer. "Proceedings of the WJCC", pp. 393–396. Association for Computing Machinery, New York.
9. Baumann, R. (1961, 1962). ALGOL-Manual der ALCOR-Gruppe. *Elektronische Rech.* 5/6 (1961) and 2 (1962).
10. Bottenbruch, H. (1962). Structure and Use of ALGOL 60. *J. A.C.M.* **9**, 2, pp.161–221.
11. Bottenbruch, H. and Grau, A. A. (1962). On Translation of Boolean Expressions. *Comm. A.C.M.* **5**, 7, pp. 384–386.
12. Dahlstrand, I. and Naur, P. (1960). Integers as Labels. ALGOL Bulletin No. 10. Regnecentralen, Copenhagen, 10.3.
13. Davis, G. M. (1960). The English Electric KDF9 Computer System. *Comp. Bull.* **4**, 3, pp. 119–120.
14. Dijkstra, E. W. (1960). Recursive Programming. *Num. Math.* **2**, 5, pp. 312–318.
15. Dijkstra, E. W. (1961). An ALGOL 60 Translator for the X1 (translated from the German in *MTW* **2** (1961), pp. 54–56 and *MTW* **3** (1961), pp. 115–119, by M. Woodger.) ALGOL Bull. Suppl. No. 10.
16. Dijkstra, E. W. (1961). Making a Translator for ALGOL 60. *A.P.I.C. Bull.* 7, pp. 3–11.
17. Dijkstra, E. W. (1961). Defense of ALGOL 60. *Comm. A.C.M.* **4**, 11, pp. 502–503.

18. Dijkstra, E. W. (1962). "A Primer of ALGOL 60 Programming". Academic Press, London.
19. Duncan, F. G. (1962). Implementation of ALGOL 60 for the English Electric KDF9. *Comp. J.* **5**, 2, pp. 130–132.
20. Duncan, F. G. (1963). Input and Output for ALGOL 60 on KDF9. *Comp. J.* **5**, 4, pp. 341–344.
21. Evans, A., Perlis, A. J. and van Zoeren, H. (1961). The Use of Threaded Lists in Constructing a Combined ALGOL and Machine-Like Assembly Processor. *Comm. A.C.M.* **4**, 1, pp. 36–41.
22. Floyd, R. W. (1961). An Algorithm for Coding Efficient Arithmetic Operations. *Comm. A.C.M.* **4**, 1, pp. 42–51.
23. Floyd, R. W. (1961). A Descriptive Language for Symbol Manipulation. *J. A.C.M.* **8**, 4, pp. 579–584.
24. Floyd, R. W. (1962). On Syntactic Analysis and Operator Precedence. Report CA–62–2, Computer Associates, Inc., Woburn, Mass.
25. Genuys, F. (1962). Commentaires sur le langage ALGOL. *Chiffres*, **5**, 1, pp. 29–53.
26. Grau. A. A. (1961). Recursive Processes and ALGOL Translation. *Comm. A.C.M.* **4**, 1, pp. 10–15.
27. Grau, A. A. (1962). A Translator-Oriented Symbolic Programming Language. *J. A.C.M.* **9**, 4, pp. 480–487.
28. Halstead, M. H. (1962). "Machine-Independent Computer Programming". Spartan Books, Washington D.C.
29. Hamblin, C. L. (1957). Computer Languages. *Austr. J. Sci.* (Dec), pp. 135–139.
30. Hamblin, C. L. (1962). Translation to and from Polish Notation. *Comp. J.* **5**, 3, pp. 210–213.
31. Hawkins, E. N. and Huxtable, D. H. R. (1963). A Multi-Pass Translation Scheme for ALGOL 60. "Annual Review in Automatic Programming", vol. 3, pp. 163–205. Pergamon Press, Oxford.
32. Higman, B. (1963). Towards an ALGOL Translator. "Annual Review in Automatic Programming", vol. 3, pp. 121–162. Pergamon Press, Oxford.
33. Higman, B. (1963). What EVERYBODY Should Know about ALGOL. *Comp. J.* **6**, 1, pp. 50–56.
34. Hill, U., Langmaack, H., Schwarz, H. R. and Seegmüller, G. (1962). Efficient Handling of Subscripted Variables in ALGOL 60 Compilers. "Proceedings of the Symposium on Symbolic Languages in Data Processing, Rome", pp. 331–340. Gordon & Breach, New York.
35. Huskey, H. D. (1961). Compiling Techniques for Algebraic Expressions. *Comp. J.* **4**, 1, pp. 10–19.
36. Huskey, H. D. and Wattenburg, W. H. (1961). Compiling Techniques for Boolean Expressions and Conditional Statements in ALGOL 60. *Comm. A.C.M.* **4**, 1, pp. 70–75.
37. Ingerman, P. Z. (1961). Thunks—A way of compiling procedure statements with some comments on procedure declarations. *Comm. A.C.M.* **4**, 1, pp. 55–58.

38. Ingerman, P. Z. (1961). Dynamic Declarations. *Comm. A.C.M.* **4**, 1, pp. 59–65.
39. Irons, E. T. (1961). A Syntax Directed Compiler for ALGOL 60. *Comm. A.C.M.* **4**, 1, pp. 51–55.
40. Irons, E. T. (1963). The Structure and Use of the Syntax Directed Compiler. "Annual Review in Automatic Programming", vol. 3, pp. 207–227. Pergamon Press, Oxford.
41. Irons, E. T. and Feurzeig, W. (1961). Comments on the Implementation of Recursive Procedures and Blocks in ALGOL 60. *Comm. A.C.M.* **4**, 1, pp. 65–69.
42. Jensen, J., Mondrup, P. and Naur, P. (1961). A Storage Allocation Scheme for ALGOL 60. *Comm. A.C.M.* **4**, 10, pp. 441–445.
43. Jensen, J. and Naur, P. (1961). An Implementation of ALGOL 60 Procedures. *Nord. Tidskr. Inform.-Behand.* **1**, 1, pp. 38–47.
44. Kanner, H. (1959). An Algebraic Translator. *Comm. A.C.M.* **2**, 10, pp. 19–22.
45. Knuth, D. E. (1959). RUNCIBLE—Algebraic Translation on a Limited Computer. *Comm. A.C.M.* **2**, 11, pp. 18–21.
46. Knuth, D. E. and Merner, J. N. (1961). ALGOL 60 Confidential. *Comm. A.C.M.* **4**, 6, pp. 268–272.
47. Ledley, R. S. and Wilson, J. B. (1962). Automatic-Programming-Language Translation through Syntactical Analysis. *Comm. A.C.M.* **5**, 3, pp. 145–155.
48. Lucas, P. (1961). The Structure of Formula-Translators. ALGOL Bull. Suppl. No. 16.
49. Lynch, W. C. (1960). Coding Isomorphisms. *Comm. A.C.M.* **3**, 2, pp. 84–85.
50. McCarthy, J. (1961). A Basis for a Mathematical Theory of Computation, Preliminary Report. "Proceedings of the WJCC", pp. 225–238. Association for Computing Machinery, New York.
51. McCracken, D. D. (1962). "A Guide to ALGOL Programming". Wiley, New York.
52. Milnes, H. W. (1957). Logical Programming and Algebraic Interpretation. *Indust. Math.* **8**, pp. 17–26.
53. Naur, P. (ed.). (1960). "Report on the Algorithmic Language ALGOL 60". Regnecentralen, Copenhagen.
54. Naur, P. (1961). "A Course of ALGOL 60 Programming". Regnecentralen, Copenhagen.
55. Naur, P. (ed.), with amendments by Woodger, M. (ed.). (1962). "Revised Report on the Algorithmic Language ALGOL 60". International Federation for Information Processing. (Also in *Comm. A.C.M.* (1963) **6**, 1, pp. 1–17, *Comp. J.* (1963) **5**, 4, pp. 349–367, and *Num. Math.* (1963) **4**, 5, pp. 420–453.)
56. Perlis, A. J. and Samelson K. (ed.) (1958). Preliminary Report—International Algebraic Language. *Comm. A.C.M.* **1**, 12, pp. 8–22.
57. Perlis, A. J. and Thornton, C. (1960). Symbol Manipulation by Threaded Lists. *Comm. A.C.M.* **3**, 4, pp. 195–204.

58. Randell, B. (1964). The Whetstone KDF9 ALGOL Translator. "Introduction to System Programming" (ed. by P. Wegner). Academic Press, London.
59. Randell, B., Duncan, F. G. and Huxtable, D. H. R. (1963). KDF9 ALGOL Note 1. Data Processing and Control Systems Division, The English Electric Company Ltd., Kidsgrove, Staffordshire.
60. Randell, B. and Russell, L. J. (1963). Single Scan Techniques for the Translation of Arithmetic Expressions in ALGOL 60. (To be published.)
61. Rutishauser, H. (1952). Automatische Rechenplanfertigung bei Programmgesteuerten Rechenmaschinen (Automatic Programming of Programme-Controlled Computers). *Mitt. Inst. Angew. Math. ETH Zurich*, No. 3.
62. Samelson, K. and Bauer, F. L. (1959). Sequentielle Formelübersetzung. *Elektronische Rech.* **1**, pp. 176–182. (Published in English as—Sequential Formula Translation. *Comm. A.C.M.* **3**, 2, pp. 76–83.)
63. Samelson, K. and Bauer, F. L. (1962). The ALCOR Project. "Proceedings of the Symposium on Symbolic Languages in Data Processing, Rome", pp. 207–217. Gordon & Breach, New York.
64. Sattley, K. (1961). Allocation of Storage for Arrays in ALGOL 60. *Comm. A.C.M.* **4**, 1, pp. 60–65.
65. Sheridan, P. B. (1959). The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System. *Comm. A.C.M.* **2**, 2, pp. 9–21.
66. Steel, T. B. (1961). A First Version of UNCOL. "Proceedings of the WJCC", pp. 371–377. Association for Computing Machinery, New York.
67. Strong, J., Wegstein, J., Trittee, A., Olsztyn, J., Mock, O. and Steel, T. (1958). The Problem of Programming Communication with Changing Machines—A Proposed Solution. Report of the Share Ad-Hoc Committee on Universal Languages. *Comm. A.C.M.* **1**, 8, pp. 12–18; *Comm. A.C.M.* **1**, 9, pp. 9–15.
68. Takahashi, S., Nishino, H., Yoshihiro, K. and Fuchi, K. (1962). Systems Design of the E.T.L. Mk6 Computer. "Information Processing 1962", Proceedings of IFIP Congress 62, pp. 690–693. North-Holland Publishing Co., Amsterdam.
69. van der Mey, G. (1962). Process for an ALGOL Translator. Report 164 MA. Dr Neher Laboratorium, Staatsbedrijf der Posterijen, Telegrafie en Telefonie, Leidshendam.
70. van der Poel, W. L. (1962). The Construction of an ALGOL Translator for a Small Computer. "Proceedings of the Symposium on Symbolic Languages in Data Processing, Rome", pp. 229–236. Gordon & Breach, New York.
71. Wegstein, J. H. (1959). From Formulas to Computer Oriented Language. *Comm. A.C.M.* **2**, 3, pp. 6–8.
72. KDF9 ALGOL Manual (1963). Data Processing and Control Systems Division, The English Electric Co. Ltd., Kidsgrove, Staffordshire.
73. KDF9 Programming Manual (1963). Data Processing and Control Systems Division, The English Electric Co. Ltd., Kidsgrove, Staffordshire.

Publishers of the Journals listed in the References are as follows:

A.P.I.C. Bull. The Automatic Programming Information Centre, Brighton College of Technology, England.

Aust. J. Sci. Australian National Research Council, Sydney, Australia.

Chiffres. The Association Française de Calcul et de Traitement de l'Information, Paris, France.

Comm. A.C.M. The Association for Computing Machinery, New York, U.S.A.

Comp. Bull. The British Computer Society, London, England.

Comp. J. The British Computer Society, London, England.

Elektronische Rech. Oldenbourg, Munich, Germany.

Indust. Math. Industrial Mathematics Society, Detroit, U.S.A.

J. A.C.M. The Association for Computing Machinery, New York, U.S.A.

Mitt. Inst. Angew. Math. ETH Zurich. Eidgenössischen Technischen Hochschule, Zurich, Switzerland.

Nord. Tidskr. Inform.-Behand. Regnecentralen, Copenhagen, Denmark.

Num. Math. Springer-Verlag, Berlin, Germany.

APPENDIX 1

A Worked Example

'A brief case history of one job done with a system seldom gives a good measure of its usefulness, particularly when the selection is made by the authors of the system.'

Backus [6]

The single example chosen to illustrate the workings of the Whetstone ALGOL Compiler is based on the procedure *GPS* (General Problem Solver) in the article 'ALGOL 60 Confidential' by Knuth and Merner [46]. In this article the possibilities of recursive procedures and parameters called by name are demonstrated by using *GPS* in several assignment statements, one of which has the effect of multiplying two matrices together, and which is presented as a challenge to compiler writers.

This use of the procedure *GPS*, which is rightly described by Knuth and Merner as 'just ALGOL for ALGOL's sake', nevertheless demonstrates several fundamental features of ALGOL which are of great practical value, and hence is worth using to demonstrate translation techniques. In order to reduce the task of description to more manageable proportions, the working of the Whetstone Compiler is demonstrated on a program using *GPS* to set up a matrix *A*, in which the element '*A* [*i*, *j*]' has the value '*i* + *j*'. In addition the full object program representation of the assignment statement which uses *GPS* to perform matrix multiplication is given. Though it is not claimed that the method of implementation is in any way optimum, the example shows that the standard mechanisms of the Whetstone Compiler for handling recursive procedures and parameters called by name can easily deal with the apparent complications of General Problem Solver.

```
begin real i, j;
      array A [1: 2, 1: 3];
      real procedure GPS (I, N, Z, V); real I, N, Z, V;
      begin for I := 1 step 1 until N do Z := V;
            GPS := 1
      end;
      i := GPS (j, 3·0, i, GPS (i, 2·0, A [i, j], i + j) )
end
```

The translation of this program is best illustrated by showing the contents of the stack and name list at various stages during translation. (The abbreviations used are as described in section 3.3.3.)

(*i*) At the end of the array declaration. Operations have been generated to enter the program block and to set up the storage mapping function for the array. Fig. 13(a) shows the contents of the stack and the name list.

NAME LIST

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*i*	r	*1*	*1,3*						*1*
3	*j*	r	*1*	*1,4*						*1*
4	*A*	r a	*1*	*1,5*				*2*		*2*

no	*Stacked Item*	*Stack Priority*
2	**begin** (bl)	*0*
1	*1, 0, 1*	
0	*UJ, 2*	*13*

STACK

FIG. 13(a). At the end of the array declaration.

The object program contains

Syll	*Op*	*Par*
0	*CBL*	
1	*UJ*	( )
4	*BE*	( )
7	*TIC1*	
8	*TIC*	'*2*'
15	*TIC1*	
16	*TIC*	'3'
23	*MSF*	(*1,5*), *1*
27	...	

(*ii*) At the end of the procedure heading. The name list, shown with the stack in Fig. 13(b), now contains entries for the procedure identifier and the formal parameters.

NAME LIST

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*i*	r	*1*	*1,3*						*1*
3	*j*	r	*1*	*1,4*						*1*
4	*A*	r a	*1*	*1,5*				*2*		*2*
5	*GPS*	r p	*1*	*2,0*	*30*			*4*		*3*
6	*GPS*	r p	*1*	*2,0*	*30*			*4*		*0*
7	*I*	r	*1*	*2,3*		*1*				*3*
8	*N*	r	*1*	*2,5*		*1*				*3*
9	*Z*	r	*1*	*2,7*		*1*				*3*
10	*V*	r	*1*	*2,9*		*1*				*3*

no	*Stacked Item*	*Stack Priority*
5	**procedure begin**, *30*	*0*
4	*1, 3, 1*	
3	*UJ, 28*	*13*
2	**begin** (bl)	*0*
1	*1, 0, 1*	
0	*UJ, 2*	*13*

STACK

FIG. 13(b). At the end of the procedure heading.

The following operations have been added to the object program

Syll	*Op*	*Par*
27	*UJ*	( )
30	*PE*	( ), *4*
34	*CA*	
35	*CA*	
36	*CA*	
37	*CA*	
38	...	

(*iii*) At the end of the procedure body. The for statement has been translated and the name list and stack are as shown in Fig. 13(c). The formal parameter identifiers have been discarded from the name list and the '**procedure begin**' items unstacked. The parameters *n* and *L* of the *PE* operation at syllable *30* have been filled in with (*2,8*) and the following operations added to the object program.

NAME LIST

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*i*	r	*1*	*1,3*						*1*
3	*j*	r	*1*	*1,4*						*1*
4	*A*	r a	*1*	*1,5*				*2*		*2*
5	*GPS*	r p	*1*	*2,0*	*30*			*4*		*3*

no	*Stacked Item*	*Stack Priority*
3	*UJ, 28*	*13*
2	**begin** (bl)	*0*
1	*1, 0, 1*	
0	*UJ, 2*	*13*

STACK

FIG. 13(c). At the end of the procedure body.

Syll	*Op*	*Par*
38	*UJ*	(*45*)
41	*TFAR*	(*2,3*)
44	*LINK*	
45	*CFZ*	(*61*)
48	*FORS1*	
49	*TIC1*	
50	*LINK*	
51	*FORS2*	
52	*TIC1*	
53	*LINK*	
54	*TFR*	(*2,5*)
57	*LINK*	
58	*FSE*	(*74*)
61	*FBE*	(*3,4*) (*41*)
66	*TFAR*	(*2,7*)
69	*TFR*	(*2,9*)
72	*ST*	
73	*FR*	
74	*TRA*	(*2,0*)
77	*TIC1*	
78	*ST*	
79	*RETURN*	
80	...	

(*iv*) Before processing the first closing round bracket of the assignment statement. The name list has remained unchanged, but a large number of items have been stacked (Fig. 13(d)). These items include those stacked

NAME LIST

no	*name*	*type*	*d*	*np*	*syll*	*f*	*v*	*dim*	*exp*	*line*
1										
2	*i*	r	*1*	*1,3*						*1*
3	*j*	r	*1*	*1,4*						*1*
4	*A*	r a	*1*	*1,5*				*2*		*2*
5	*GPS*	r p	*1*	*2,0*	*30*			*4*		*3*

no	*Stacked Item*	*Stack Priority*
20	+	*9*
19	*PSR, 118*	*0*
18	*PSR, 104*	*0*
17	*PRC, 98*	*0*
16	*PR, (1,3)*	*0*
15	*(, 1*	*0*
14	*0, 0, 0*	
13	*GPS*	
12	*UJ, 96*	*13*
11	*PSR, 92*	*0*
10	*PR, (1,3)*	*0*
9	*PRC, 86*	*0*
8	*PR, (1,4)*	*0*
7	*(, 0*	*0*
6	*0, 0, (ABG)*	
5	*GPS*	
4	*UJ, 84*	*13*
3	*ST*	*2*
2	**begin** (bl)	*0*
1	*1, 0, 1*	
0	*UJ, 2*	*13*

STACK

FIG. 13(d). Before processing the first closing round bracket of the assignment statement.

at each opening procedure bracket, and the two sets of 'actual operations'. The object program operations that have been added are

Syll	*Op*	*Par*
80	*TRA*	(*1,3*)
83	*UJ*	( )
86	'*3·0*'	
92	*BE*	(*2,0*)
95	*UJ*	( )
98	'*2·0*'	
104	*BE*	(*3,0*)
107	*TRA*	(*1,5*)
110	*TRR*	(*1,3*)
113	*TRR*	(*1,4*)
116	*INDA*	
117	*EIS*	
118	*BE*	(*3,0*)
121	*TRR*	(*1,3*)
124	*TRR*	(*1,4*)
127	...	

(*v*) After processing the final **end**. This delimiter completes the program; when it has been processed the name list and the stack will be empty. Various operations have been added to the object program, including the two sets of 'actual operations'. The complete object program is

Syll	*Op*	*Par*	*Remarks*
0	*CBL*		
1	*UJ*	(*172*)	
4	*BE*	(*1,3*)	Program entry
7	*TIC1*		
8	*TIC*	'*2*'	
15	*TIC1*		Array declaration
16	*TIC*	'*3*'	
23	*MSF*	(*1,5*), *1*	*A*
27	*UJ*	(*80*)	Jump around procedure
30	*PE*	(*2,8*), *4*	Entry to procedure *GPS*
34	*CA*		
35	*CA*		
36	*CA*		
37	*CA*		
38	*UJ*	(*45*)	
41	*TFAR*	(*2,3*)	*I*
44	*LINK*		
45	*CFZ*	(*61*)	
48	*FORS1*		

Syll	*Op*	*Par*	*Remarks*
49	*TIC1*		
50	*LINK*		
51	*FORS2*		
52	*TIC1*		
53	*LINK*		
54	*TFR*	(*2,5*)	*N*
57	*LINK*		
58	*FSE*	(*74*)	
61	*FBE*	(*3,4*), (*41*)	Entry to for statement
66	*TFAR*	(*2,7*)	*Z*
69	*TFR*	(*2,9*)	*V*
72	*ST*		*Z* := *V*
73	*FR*		
74	*TRA*	(*2,0*)	*GPS*
77	*TIC1*		
78	*ST*		*GPS* := *1*
79	*RETURN*		
80	*TRA*	(*1,3*)	*i*
83	*UJ*	(*166*)	
86	'*3·0*'		
92	*BE*	(*2,0*)	Entry to subroutine for '*GPS* (*i*, *2·0*, *A* [*i*,*j*], *i*+*j*)'
95	*UJ*	(*145*)	
98	'*2·0*'		
104	*BE*	(*3,0*)	
107	*TRA*	(*1,5*)	*A* — Subroutine for '*A* [*i*,*j*]'
110	*TRR*	(*1,3*)	*i* — Subroutine for '*A* [*i*,*j*]'
113	*TRR*	(*1,4*)	*j* — Subroutine for '*A* [*i*,*j*]'
116	*INDA*		*A* [*i*, *j*] — Subroutine for '*A* [*i*,*j*]'
117	*EIS*		Subroutine for '*i* + *j*'
118	*BE*	(*3,0*)	Subroutine for '*i* + *j*'
121	*TRR*	(*1,3*)	*i* — Subroutine for '*i* + *j*'
124	*TRR*	(*1,4*)	*j* — Subroutine for '*i* + *j*'
127	+		*i* + *j* — Subroutine for '*i* + *j*'
128	*EIS*		Subroutine for '*i* + *j*'
129	*PSR*	(*118*), –	
133	*PSR*	(*104*), –	
137	*PRC*	(*98*), –	
141	*PR*	(*1,3*), –	*i*
145	*CF*	(*30*), *4*	*GPS*
149	*EIS*		
150	*PSR*	(*92*), –	
154	*PR*	(*1,3*), –	*i*
158	*PRC*	(*86*), –	
162	*PR*	(*1,4*), –	*j*

Syll	*Op*	*Par*		*Remarks*
166	*CF*	*(30), 4*	*GPS*	
170	*ST*			
171	*RETURN*			
172	*FINISH*			

The working of this object program is demonstrated by showing the contents of the stack at various selected points.

Figure 14(a) shows the contents of the stack after the following actions have been performed

(1) Entry to the program.
(2) Setting up of the array *A*.
(3) Stacking of an accumulator, containing the address of *i*.
(4) Entry to procedure *GPS*.
(5) Checking of the four parameters to *GPS*.

Figure 14(b) shows the stack when the controlled statement of the for statement has been entered. This has caused the assignment of *1·0* to *j*, the stacking of an accumulator containing the address of *i*, and entry to a subroutine which has caused a recursive activation of *GPS*, whose parameters have been checked.

In Fig. 14(c) the for statement has been activated recursively, the value *1·0* has been assigned to *i*, and the subroutine for evaluating the address of '*A* [*i*, *j*]', which corresponds to the formal parameter *Z*, has been entered. The next operation, *INDA*, will replace the top three accumulators with an accumulator containing the address of '*A* [*1*, *1*]'.

In Fig. 14(d) the result of the subroutine, i.e. the address of '*A* [*1*, *1*]', is given in what was the result accumulator of the subroutine, and a second subroutine, for evaluating '*i*+*j*', has been entered. Accumulators containing the current values of *i* and *j* are at the top of the stack.

Finally, in Fig. 14(e) this second subroutine has been completed, and at the top of the stack are two accumulators, ready for the operation *ST* which will perform the assignment

$$A[i, j] := i + j; \qquad (\text{for } i = j = 1)$$

i.e. $A[1, 1] := 2{\cdot}0;$

The assignment

$$A[2, 1] := 3{\cdot}0;$$

is performed next, as a result of an assignment of *2·0* to *i* by the for statement and a second call on the two subroutines. The for block and the recursive activation of *GPS* are left and the outer for statement then assigns *2·0* to *j*;

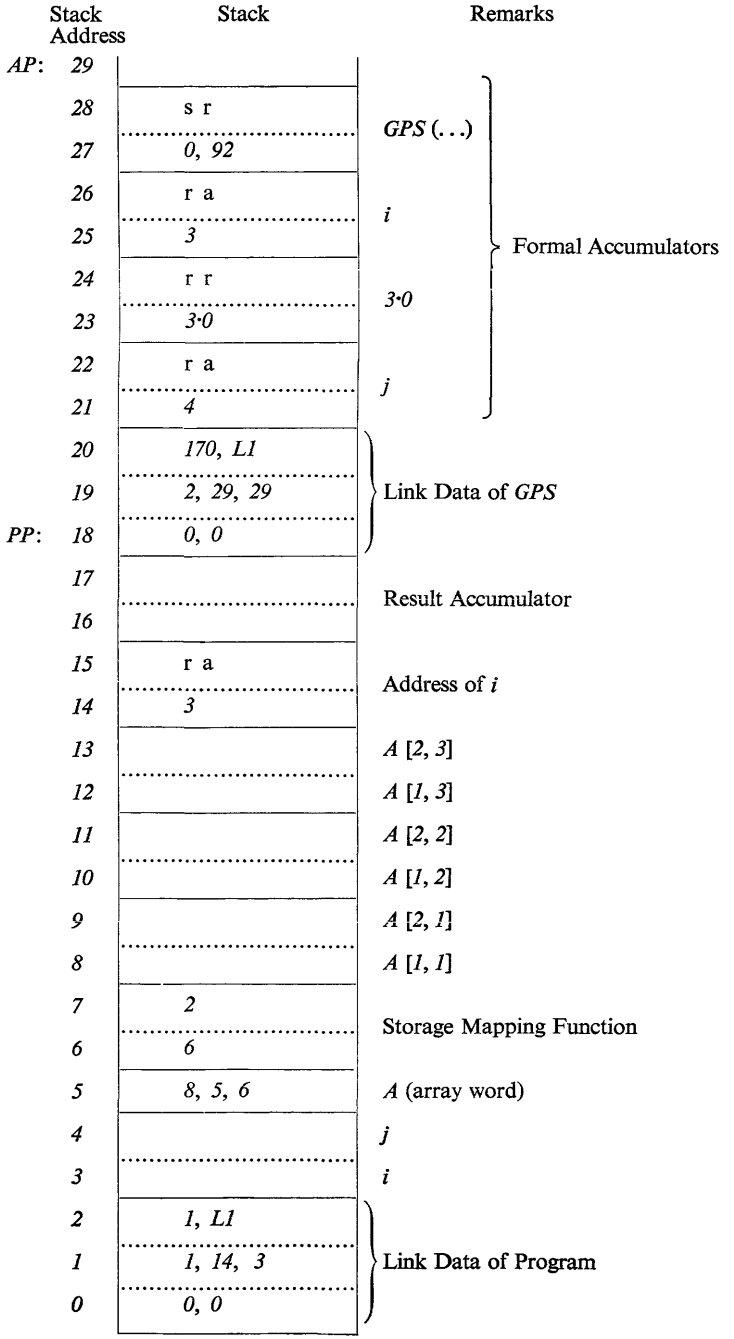


FIG. 14(a). After completing checking of parameters at first entry to *GPS*.

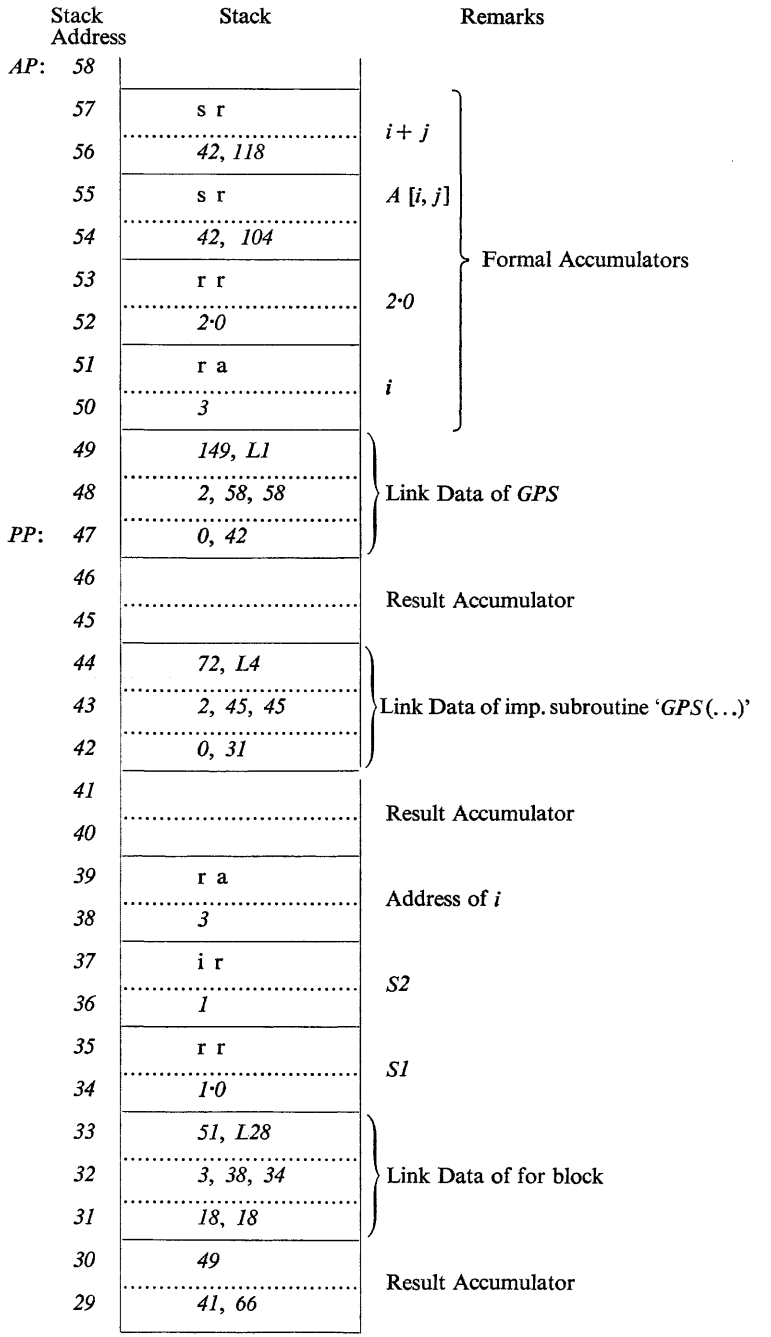


FIG. 14(b). After completing checking of parameters at second entry to *GPS*.

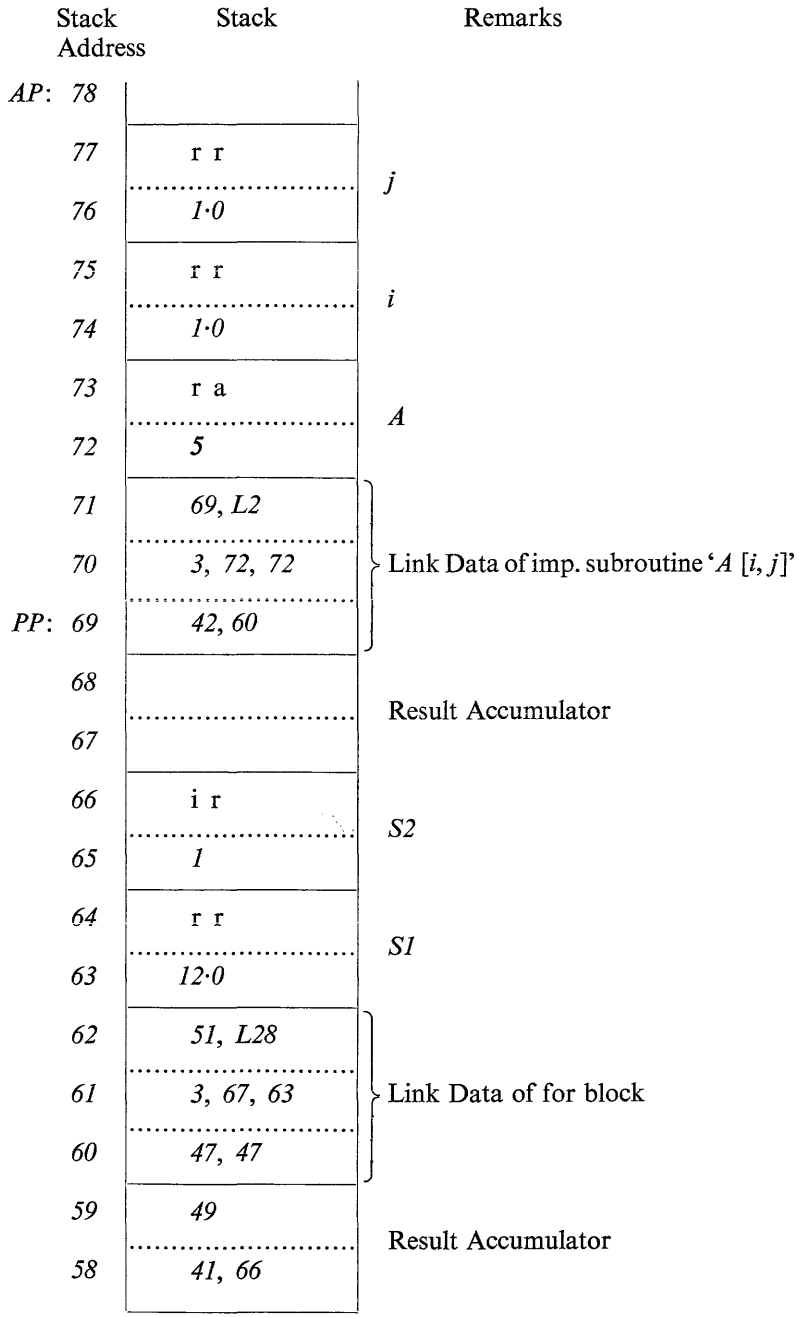


FIG. 14(c). About to calculate the address of '*A* [*I*, *I*]'.

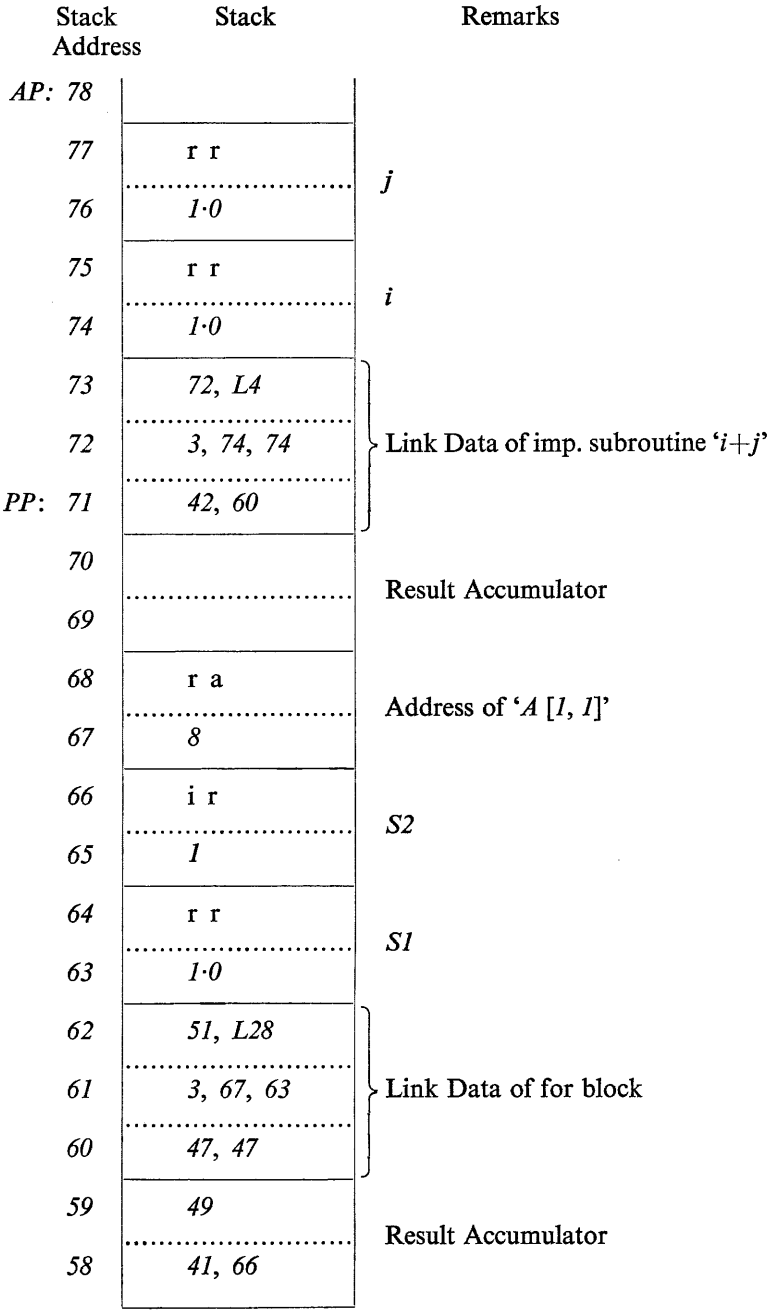


FIG. 14(d). About to calculate '$i+j$'.

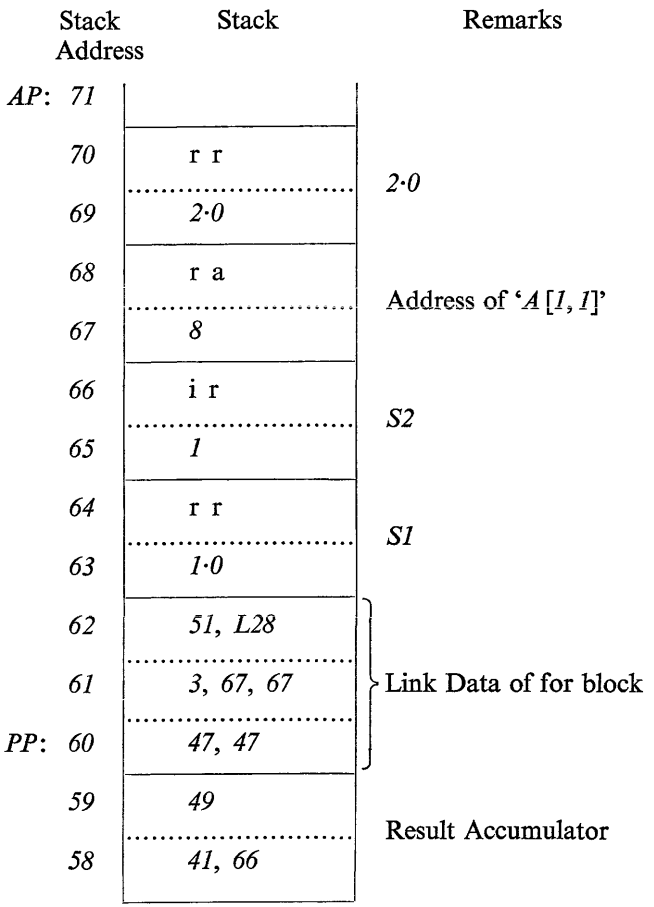


FIG. 14(e). About to perform '$A\,[i, j] := i+j$', for $i = j = 1$.

the process of activating *GPS* and the for statement recursively is repeated, to perform

$$A\,[1, 2] := 3{\cdot}0;$$
$$A\,[2, 2] := 4{\cdot}0;$$

The third and final assignment to j by the outer for statement eventually causes the assignments

$$A\,[1, 3] := 4{\cdot}0;$$
$$A\,[2, 3] := 5{\cdot}0;$$

to be made.

Then the for statement and the original activation of *GPS* are left, and the value of *GPS* (i.e. *1·0*) is assigned to *i*. Finally, when the end of the program

is reached the values of the elements of array A are lost, since no output statements have been included in the program.

The assignment statement given by Knuth and Merner is

$$I := GPS\,(I,\ 1{\cdot}0,\ C\,[1,\ 1],\ 0{\cdot}0)$$
$$\times\ GPS\,(I,\ (m-1)\ \times\ GPS\,(J,\ (p-1)\ \times\ GPS\,(K, n,\ C[I, J],$$
$$C\,[I, J] + A\,[I, K] \times B\,[K, J]),\ C\,[I, J+1],\ 0{\cdot}0),\ C\,[I + 1, 1],\ 0{\cdot}0);$$

which multiplies the array '$A\ [1{:}\ m,\ 1{:}\ n]$' by '$B\ [1{:}\ n,\ 1{:}\ p]$' and stores the result in '$C\ [1{:}\ m, 1{:}\ p]$'.

The object program representation of this assignment statement is

Syll	*Op*	*Par*		*Remarks*
0	*TRA*	*I*		
3	*UJ*	*(44)*		
6	*'1·0'*			
12	*BE*	*(2,0)*		
15	*TRA*	*C*		
18	*TIC1*			
19	*TIC1*			
20	*INDA*		*C [1,1]*	
21	*EIS*			
22	*'0·0'*			
28	*PRC*	*(22), –*	*0·0*	'actual operations'
32	*PSR*	*(12), –*	*C [1, 1]*	
36	*PRC*	*(6), –*	*1·0*	
40	*PR*	*I , –*	*I*	
44	*CF*	*GPS, 4*		
48	*UJ*	*(225)*		
51	*BE*	*(2,0)*		
54	*TRR*	*m*		
57	*TIC1*			
58	*–*		*m − 1*	
59	*UJ*	*(183)*		
62	*BE*	*(3,0)*		
65	*TRR*	*p*		
68	*TIC1*			
69	*–*		*p − 1*	
70	*UJ*	*(139)*		
73	*BE*	*(4,0)*		
76	*TRA*	*C*		
79	*TRR*	*I*		
82	*TRR*	*J*		
85	*INDA*		*C [I, J]*	
86	*EIS*			

Syll	*Op*	*Par*	*Remarks*	
87	*BE*	(*4,0*)		
90	*TRA*	*C*		
93	*TRR*	*I*		
96	*TRR*	*J*		
99	*INDR*		*C* [*I, J*]	
100	*TRA*	*A*		
103	*TRR*	*I*		
106	*TRR*	*K*		
109	*INDR*		*A* [*I, K*]	
110	*TRA*	*B*		
113	*TRR*	*K*		
116	*TRR*	*J*		
119	*INDR*		*B* [*K, J*]	
120	×			
121	+			
122	*EIS*			
123	*PSR*	(*87*), –	$C[I,J]+A[I,K]\times B[K,J]$	'actual operations'
127	*PSR*	(*73*), –	$C[I,J]$	
131	*PR*	*n*, –	*n*	
135	*PR*	*K*, –	*K*	
139	*CF*	*GPS*, *4*		
143	×			
144	*EIS*			
145	*BE*	(*3,0*)		
148	*TRA*	*C*		
151	*TRR*	*I*		
154	*TRR*	*J*		
157	*TIC1*			
158	+			
159	*INDA*		*C* [*I, J*+*1*]	
160	*EIS*			
161	'*0·0*'			
167	*PRC*	(*161*), –	*0·0*	'actual operations'
171	*PSR*	(*145*), –	*C* [*I, J*+*1*]	
175	*PSR*	(*62*), –	$(p-1)\times GPS(\ldots)$	
179	*PR*	*J*, –	*J*	
183	*CF*	*GPS*, *4*		
187	×			
188	*EIS*			
189	*BE*	(*2,0*)		
192	*TRA*	*C*		
195	*TRR*	*I*		
198	*TIC1*			
199	+			
200	*TIC1*			

Syll	*Op*	*Par*	*Remarks*	
201	*INDA*		$C\,[I+1,\ 1]$	
202	*EIS*			
203	*'0·0'*			
209	*PRC*	(*203*), –	*0·0*	'actual operations'
213	*PSR*	(*189*), –	$C\,[I+1,\ 1]$	
217	*PSR*	(*51*), –	$(m-1) \times GPS\,(\ldots)$	
221	*PR*	*I*, –	*I*	
225	*CF*	*GPS*, *4*		
229	×			
230	*ST*		$I := GPS\,(\ldots) \times GPS\,(\ldots)$	

APPENDIX 2

Restrictions Imposed on ALGOL 60 in KDF9 ALGOL

KDF9 ALGOL is being implemented using two separate compilers. The writers of each compiler have placed slight restrictions on ALGOL 60 – the logical sum of these restrictions forms the complete set of restrictions used to define the subset of ALGOL 60 which is called KDF9 ALGOL. These restrictions are detailed below, together with a brief description of the reasons for each restriction, and remarks on the changes necessary to the Whetstone Compiler in order to remove the restrictions.

(*i*) All Formal Parameters must have Specifications

This restriction allows the Whetstone Compiler to generate, at each use of an identifier, the appropriate object program operations, in which information regarding the type of use being made of the identifier is given implicitly.

Example

procedure *P* (*x*, *a*, *b*); **real** *x*; **integer** *a*; **real procedure** *b*;
x := *a* + *b*;

The body of this procedure is translated into

Syll	*Op*	*Par*	*Remarks*
0	*TFAR*	*x*	(Take Formal Address Real)
3	*TFI*	*a*	(Take Formal Integer)
6	*CFFZ*	*b*	(Call Formal Function Zero)
9	+		
10	*ST*		
11	...		

If parameters did not have specifications the Translator would have to produce less detailed operations, whose action is dependent on the contents of the appropriate formal accumulator.

The above example would be translated into

Syll	*Op*	*Par*	*Remarks*
0	*TA*	*x*	(Take Address)
3	*TR*	*a*	(Take Result)
6	*TR*	*b*	(Take Result)
9	+		
10	*ST*		
11	...		

Thus the indecision (with regard to the meaning of identifiers) at translation time, which is resolved when a set of skeleton operations are unchained, would in this case be carried forward into the Control Routine.

(*ii*) Labels must not be Unsigned Integers

The reason for this is concerned with the use of a designational expression as an actual parameter.

Example

```
procedure Pop (Q); procedure Q; begin . . . Q(3); . . . end;
procedure Pip (A); label A; begin . . . go to A; . . . end;
procedure Pap (B); real B; begin . . . := B; . . . end;
Pop (Pap);
Pop (Pip);
```

(Taken from ALGOL Bulletin No 10 – October 1960 [12].)

The number *3* appearing within the body of procedure *Pop* will in the first of the procedure statements be used as a number, while in the second it will be used as a label.

Integer labels could be allowed if the Translator, in cases of doubt, produced operations which could be used either as 'Take Integer Constant' (or 'Parameter Integer Constant') or 'Take Label' (or 'Parameter Label') operations. The result of such an operation would be an accumulator containing both the integer and the (*PP,a*) representation of the label. If such an accumulator was used by a *GTA* (Go To Accumulator) operation the integer would be ignored; alternatively the (*PP,a*) would be ignored.

(*iii*) Go To an Undefined Switch Designator Produces an Error Indication

A technique for allowing a go to statement, which uses a switch designator whose value is undefined because of an out-of-range switch index, to act as a dummy statement, has been described in section 2.4.2.

(*iv*) A Type Procedure must Contain an Assignment to the Procedure Identifier

This restriction prevents the extremely trivial case of the declaration of a procedure which is not intended for use by a function designator being preceded by a type declarator.

Example

```
real procedure P; x := 3;
```

This procedure could only be called by a procedure statement, and hence the delimiter **real** is redundant. The reason for the restriction is to facilitate the checking, by the Translator, of whether a given procedure can be called by a function designator; the easiest way of removing the restriction would be to relax this checking.

(*v*) The Actual Parameters Corresponding to a Formal Parameter Specified to be a Procedure must be Procedures with Identical Specification Parts

This restriction arises from the system of procedure classification used in the Kidsgrove Compiler (see section 1.2.4.1). This restriction is not necessary to the Whetstone Compiler, and in fact cannot be checked.

(*vi*) Dynamic Own Arrays are not Allowed

This restriction arises because the size of own arrays is determined at translation time, in order that storage space for the elements can be allocated at the start of the run time stack.

The difficulty of implementing dynamic own arrays is that the size and shape of an own array, whose elements must be retained when the block in which it is declared is left, may change on subsequent entries to the block. A proposed technique for implementing dynamic own arrays is described in a paper by Ingerman [38].

APPENDIX 3

The KDF9 Computer

The KDF9 Computer has been designed and manufactured at the Data Processing and Control Systems Division of the English Electric Company, Kidsgrove, England. It is a medium-sized computer, with a magnetic core storage of up to 32,768 words, each of 48 binary digits. The basic input/output medium is 8-hole paper tape. Magnetic tape is also used for input/output and as backing storage. Other input/output devices can include line printers, punched card equipment, etc.

One of the most distinctive features of the KDF9 is its 'nesting store accumulator' which allows the evaluation of arithmetic expressions to be programmed in what is essentially a Reverse Polish notation. The nesting store (a form of push-down store or stack) comprises 16 cells. In general, arithmetic operations, logical operations, shifts, etc., are all addressless and work on the top one or two words of the nesting store. A similar set of 16 cells, comprising the 'subroutine jump nesting store' is used for the storage of links during the activation of subroutines. Finally, there is a set of 15 Q-stores (index registers) which can be used to modify fetch and store instructions, control shifts, etc., and which can also be used as fixed point accumulators. Fetch and Store operations are used to transmit information between the core store and the top cell of the nesting store.

The KDF9 Computer has an extensive order code, made up of one-, two- and three-syllable instructions (a syllable is 8 binary digits). An assembly language, called User Code, is used to prepare programs for conversion into binary, and can be regarded, for most purposes, as being the actual machine code of KDF9.

Example

42 ; *Y6* ; (Fetch contents of store *Y6* to top of nesting store)
Y106 ; (Fetch contents of store *Y106* to top of nesting store)
+F ; (Floating point addition of top two cells of nesting store)
=Y7 ; (Store contents of top cell of nesting store in *Y7*)
J42 ; (Jump to instruction with reference *42*)

This example demonstrates a repeated loop, each time forming

$$Y7 := Y6 + Y106$$

A full description of User Code is given in the KDF9 Programming Manual [73], and a more complete description of the KDF9 Computer itself has been given by Davis [13].

APPENDIX 4

KDF9 ALGOL Hardware Representations

Basic Symbol (Reference Language)	8-Channel (Flexowriter) Version	5-Channel (Creed) Version
0 to *9*	*0* to *9*	*0* to *9*
A to *Z*	A to Z	A to Z
a to *z*	a to z	
+	+	+
−	−	−
×	×	×
/	/	/
÷	÷	*DIV
↑	↑	**
<	<	*≥
≤	≤	*>
=	=	=
≥	≥	≥
>	>	>
≠	≠	≠
≡	eqv	*EQV
⊃	imp	*IMP
∨	or	*OR
∧	and	*AND
¬	not	*NOT
,	,	,
.	.	.
10	10	*v*
:	:	→
;	;	*,
:=	:=	*=
␣	*	£
(	(	(
)	)	)
[	[	*(
]	]	*)
‘	[	*Q
’	]	*U
begin	begin	*BEGIN

The KDF9 Flexowriter produces 8-hole paper tape, using 6 holes as information channels, according to the following table

Value			
Dec.	Octal	Function	
0	*00*	Space	
1	*01*		
2	*02*	CRLF	
3	*03*		
4	*04*	Tab	
5	*05*		
6	*06*	Case shift	
7	*07*	Case normal	
8	*10*		
9	*11*		
10	*12*		
11	*13*		
12	*14*		
13	*15*		
14	*16*		
		Symbol	
		Normal	Shifted
15	*17*	/	:
16	*20*	*0*	↑
17	*21*	*1*	[
18	*22*	*2*	]
19	*23*	*3*	<
20	*24*	*4*	>
21	*25*	*5*	=
22	*26*	*6*	×
23	*27*	*7*	÷
24	*30*	*8*	(
25	*31*	*9*	)
*26**	*32*		
27	*33*	$\overline{}_{10}$	£
*28***	*34*	;	;
29	*35*	+	≠
30	*36*	−	*
31	*37*	.	,
32	*40*		
33	*41*	A	a
34	*42*	B	b
35	*43*	C	c

* Non-escaping key (i.e. no carriage motion).
** Separator symbol.

Value		Symbol	
Dec.	Octal	Normal	Shifted
36	*44*	D	d
37	*45*	E	e
38	*46*	F	f
39	*47*	G	g
40	*50*	H	h
41	*51*	I	i
42	*52*	J	j
43	*53*	K	k
44	*54*	L	l
45	*55*	M	m
46	*56*	N	n
47	*57*	O	o
48	*60*	P	p
49	*61*	Q	q
50	*62*	R	r
51	*63*	S	s
52	*64*	T	t
53	*65*	U	u
54	*66*	V	v
55	*67*	W	w
56	*70*	X	x
57	*71*	Y	y
58	*72*	Z	z
59	*73*		
60	*74*		
*61****	*75*	→	→
*62*****	*76*		
63	*77*		

*** End Message character.
**** *ESCAPE* character.

APPENDIX 5

Implementation of Program Testing Facilities

The position identifiers which are used to provide the trace and retroactive trace facilities are placed in the object program as parameters to a special operation *'TRACE'*. This operation is generated at each procedure or label declaration if the ALGOL program was preceded by the appropriate message to the Translator.

At run time the action of this operation is to place its parameter, i.e. a representation of the characters forming the position identifier, in a 'circular store' (a 32-word vector which is addressed cyclically). The value of the store used by the procedure *TEST* controls whether this position identifier is also printed; at a failure the contents of the circular store are printed, thus giving a retroactive trace.

The standard failure information (line number, last position identifier, etc.) and the post-mortem facility use the information stored on the 'failure tape' as a byproduct of translation. Two sets of information are stored on the failure tape; the first is a table of line number or position identifier against object program counter. Intermingled with the entries to this table are blocks of information, containing name list entries, which are produced when the Translator collapses the name list.

At a failure the failure tape is scanned, in order to perform a table look-up, using the current value of the program counter. This enables the line number and last position identifier corresponding to the point in the ALGOL text at which the failure occurred to be found and printed out. Scanning of the tape then continues until a group of name list entries with block levels equal to the current value of *BN* (Block Number) is encountered. These entries will be for the current block, and allow the identifiers corresponding to all the scalars in the current first order working storage to be found. These identifiers and the values of the scalars are printed out. By scanning the tape until the next group of name list entries with block levels which are less than the current value of *BN* is found, the process can be repeated for the containing block. This technique is used to provide post-mortem information for as many block levels as have been requested.

APPENDIX 6

Implementation of Segmentation

A 'complete procedure' which contains no own variables can be translated without any need for reference to the program in which it is contained as at the end of the procedure both the name list and the translator stack will have returned to the state they were in before translation of the procedure was started. Furthermore the generated object program will be complete, and will not contain any skeleton operations that need unchaining. Hence such a procedure could be translated in isolation from the program in which it is embedded.

However, if this is done the object program addresses and the block levels (used in parameters to the object program operations) will be given relative to the start of the procedure, or 'segment', instead of to the start of the program. This will not matter if the Control Routine makes the appropriate corrections to program addresses during the activation of the procedure (block levels need not be modified since there can be no access to non-locals). Since the Control Routine must modify object program addresses in this way, the position at which the sequence of object program operations generated from the procedure is placed is no longer of importance. Thus this sequence of operations can be kept on magnetic tape, to be brought down into core storage only when needed, and placed wherever convenient, in fact at the top of the stack.

At the call of a segment its object program operations are placed at the top of the stack, before allocating space for its working storage. At the completion of the activation of the segment both its operations and its working storage are deleted from the stack. The address at which the first syllable of the sequence of object program operations is stored is used to modify all the object program addresses used within the segment.

Since a segment can itself contain further segments it is necessary to be able to have several segments in the stack at any one time, and to be able to use the appropriate address modifier. This is done by storing these modifiers in the stack, and setting up chains linking the positions at which the modifiers are stored. Two chains are needed, one for ordinary exit from a segment by reaching its end, the other for temporary exit during use of a parameter called by name, or for exit by means of a go to statement. (These two chains may be compared with the dynamic and static chains for ordinary blocks and procedures.) This technique could be extended, so that segments called recursively would not be stored repeatedly, using a second *DISPLAY* vector to give details of the segments that are already in the stack.

APPENDIX 7

Object Program Operations

		Parameters	*Section No.*
AOA	Avoid Own Array	(*a*)	2.3.3
BE	Block Entry	(*n*,*L*)	2.2.3
CA	Check Arithmetic		2.5.6.1
CAB	Check Array Boolean		2.5.6
CAI	Check Array Integer		2.5.6
CAR	Check Array Real		2.5.6
CB	Check Boolean		2.5.6.2
CBFA	Copy Boolean Formal Array		2.5.6.4
CBL	Call Block		2.2.3
CF	Call Function	(*a*), *m*	2.2.3
CFB	Check Function Boolean		2.5.6
CFF	Call Formal Function	(*n*,*p*), *m*	2.5.5.6
CFFZ	Call Formal Function Zero	(*n*,*p*)	2.5.5.6
CFI	Check Function Integer		2.5.6
CFR	Check Function Real		2.5.6
CFZ	Call Function Zero	(*a*)	2.2.2.3
CIFA	Copy Integer Formal Array		2.5.6.4
CL	Check Label		2.5.6.3
CPR	Check Procedure		2.5.6
CRFA	Copy Real Formal Array		2.5.6.4
CSB	Check and Store Boolean		2.5.6.2
CSI	Check and Store Integer		2.5.6.1
CSL	Check and Store Label		2.5.6.3
CSR	Check and Store Real		2.5.6.1
CST	Check String		2.5.6
CSW	Check Switch		2.5.6
DOWN		*m*	2.7.2
DSI	Decrement Switch Index	(*a*)	2.4.2
DUMMY			2.5.4.3
EIS	End Implicit Subroutine		2.5.4.3
ESL	End Switch List		2.4.2
FBE	For Block Entry	(*n*,*L*), (*a*)	2.6.1
FINISH			2.2.8
FORA	For Arithmetic		2.6.2.1
FORS1	For Step – 1st Entry		2.6.2.3
FORS2	For Step – 2nd Entry		2.6.2.3
FORW	For While		2.6.2.2

		Parameters	*Section No.*
FR	For Return		2.6.1
FSE	For Statement End	(*a*)	2.6.1
GTA	Go to Accumulator		2.4.1
IFJ	If False Jump	(*a*)	2.1.5
INDA	Index Address		2.3.2
INDR	Index Result		2.3.2
LINK			2.6.1
MOSF	Make Own Storage Function	(*n*,*p*), *m*	2.3.3
MSF	Make Storage Function	(*n*,*p*), *m*	2.3.1.1
NEG	Negate		2.1.1
PB	Parameter Boolean	(*n*,*p*), –	2.5.4.1
PBA	Parameter Boolean Array	(*n*,*p*), –	2.5.4.4
PBC	Parameter Boolean Constant	(*a*), –	2.5.4.2
PE	Procedure Entry	(*n*,*L*), *m*	2.2.3
PF	Parameter Formal	(*n*,*p*), –	2.5.4.10
PFB	Parameter Function Boolean	(*a*), –	2.5.4.8
PFI	Parameter Function Integer	(*a*), –	2.5.4.8
PFR	Parameter Function Real	(*a*), –	2.5.4.8
PI	Parameter Integer	(*n*,*p*), –	2.5.4.1
PIA	Parameter Integer Array	(*n*,*p*), –	2.5.4.4
PIC	Parameter Integer Constant	(*a*), –	2.5.4.2
PL	Parameter Label	(*a*), *m*	2.5.4.6
PPR	Parameter Procedure	(*a*), –	2.5.4.8
PR	Parameter Real	(*n*,*p*), –	2.5.4.1
PRA	Parameter Real Array	(*n*,*p*), –	2.5.4.4
PRC	Parameter Real Constant	(*a*), –	2.5.4.2
PSR	Parameter Subroutine	(*a*), –	2.5.4.3
PST	Parameter String	(*a*), –	2.5.4.9
PSW	Parameter Switch	(*a*), –	2.5.4.7
REJECT			2.2.6
RETURN			2.2.4
ST	Store		2.1.1
STA	Store Also		2.1.1
TBA	Take Boolean Address	(*n*,*p*)	2.1.4
TBCF	Take Boolean Constant False		2.1.4
TBCT	Take Boolean Constant True		2.1.4
TBR	Take Boolean Result	(*n*,*p*)	2.1.4
TFA	Take Formal Address	(*n*,*p*)	2.5.5.2
TFAI	Take Formal Address Integer	(*n*,*p*)	2.5.5.1.1
TFAR	Take Formal Address Real	(*n*,*p*)	2.5.5.1.1
TFB	Take Formal Boolean	(*n*,*p*)	2.5.5.2
TFI	Take Formal Integer	(*n*,*p*)	2.5.5.1.2
TFL	Take Formal Label	(*n*,*p*)	2.5.5.4
TFR	Take Formal Real	(*n*,*p*)	2.5.5.1.2

		Parameters	*Section No.*
TIA	Take Integer Address	(*n*,*p*)	2.1.1
TIC	Take Integer Constant	'const'	2.1.2
TIC0	Take Integer Constant Zero		2.1.2
TIC1	Take Integer Constant One		2.1.2
TIR	Take Integer Result	(*n*,*p*)	2.1.1
TL	Take Label	(*a*), *m*	2.4.1
TRA	Take Real Address	(*n*,*p*)	2.1.1
TRC	Take Real Constant	'const'	2.1.2
TRR	Take Real Result	(*n*,*p*)	2.1.1
TSA	Take Switch Address	(*a*)	2.4.3
UJ	Unconditional Jump	(*a*)	2.1.5
UP1			2.7.2
UP2			2.7.2
$+$	Add		2.1.1
$-$	Subtract		2.1.1
$\times$	Multiply		2.1.1
$/$	Divide		2.1.1
$\div$	Integer Divide		2.1.1
$\uparrow$	Power		2.1.1.1
$>$	Greater Than		2.1.4
$\geq$	Greater Than or Equal		2.1.4
$=$	Equal		2.1.4
$\neq$	Unequal		2.1.4
$\leq$	Less Than or Equal		2.1.4
$<$	Less Than		2.1.4
$\neg$	Not		2.1.4
$\wedge$	And		2.1.4
$\vee$	Or		2.1.4
$\supset$	Implies		2.1.4
$\equiv$	Equivalent		2.1.4

Here 'Section No.' gives the section where the action of the operation is described. The notation used for parameters is as follows:

(*a*)	two-syllable program address
(*n*,*p*)	two-syllable dynamic stack address
(*n*,*L*)	block number, and extent of working storage, packed into two syllables
m	one-syllable parameter
'const'	six-syllable (i.e. one word) representation of a constant
–	one-syllable space

APPENDIX 8

State Variables

Details of the state variables mentioned in section 3 of the book are given below, together with the number of the section in which they were introduced.

Name	*Section No.*	*Description*
E	3.1.2	Statement/Expression marker. Set to one for statements and to zero for expressions.
V	3.4.1	Declaration/Statement marker. Set to zero at **begin**, to one for declarations and to two for statements.
T	3.4.1.1	Contains a bit pattern representation of the type of the current declaration.
TYPE	3.1.2	Contains a bit pattern representation of the type of expression, etc.
D	3.4.1.1	Own marker. Set to one whilst an own declaration is being processed.
m	3.5.2	ALGOL section marker. Set to zero if the section contains merely a delimiter, to one if it contains an identifier and a delimiter, and to two if it contains a constant and a delimiter.
n	3.4.1.1	Block level count.
NLP	3.4.1.1	Name List Pointer. This points to the next free space in the name list.
NL	3.4.1.1	Value of *NLP* at the start of the current block.
L	3.4.1.1	First order storage count.
L_p	3.4.1.1	First order storage count for own identifiers.
L_0	3.4.1.2.1.2	Second order storage count for own identifiers.
ARITH	3.1.2	Arithmetic expression marker.
F	3.4.4.1	For clause marker.
LD	3.4.3.2.1.1	Last Delimiter (i.e. the delimiter preceding the delimiter currently being processed).

Details of further state variables and markers are given with the Translator flow diagrams in Appendix 11.

APPENDIX 9

Details of the Various Implementations of the Whetstone Compiler

In this Appendix a few brief details are given of the four versions of the Whetstone Compiler which have been written to date (July, 1963). These are for the English Electric KDF9 and DEUCE, the Ferranti PEGASUS and the N.P.L. ACE computers. (Work has just started on a fifth version, for the Ferranti MERCURY computer, at the Systems Engineering Department of Associated Electrical Industries, Ltd, Manchester.)

No estimates are given of the speed of translation and of running of translated programs on the various computers since they would not be very meaningful without detailed knowledge of the computers. However, a paper by Ryder, describing experience with the PEGASUS version of the compiler (which was the first to be completed), and giving detailed comparisons of its speeds and capabilities with those of other methods of programming the computer, is to be published shortly.

(*i*) KDF9

A brief description of the KDF9 computer has been given in Appendix 3. The Control Routine and the Translator are essentially as described in sections 2 and 3 of this book, and occupy 2500 and 1100 48-bit words, respectively. (There are on average 3 instructions per word.)

(*ii*) DEUCE

The DEUCE version of the compiler was written as a joint project by Liverpool University and English Electric Co., Ltd. The Translator was written by J. M. Watt, D. S. Collens and G. M. Gillow of the Computer Laboratory, Liverpool University, and the Control Routine by M. A. Batty of the Atomic Power Division, The English Electric Co., Ltd.

The compiler accepts KDF9 ALGOL, subject to the following restrictions:

(a) no own arrays;

(b) no strings (but there is a facility for copying strings from the input to the output);

(c) code procedures are allowed in a restricted form but require a knowledge of the internal workings of the Control Routine.

Three different paper tape hardware representations of ALGOL are allowed – a 5-hole modified Telex code can be used in addition to the 5 and 8-hole representations given in Appendix 4. Data input and output is on 5-hole tape, using a set of procedures which are available without explicit declaration. The Translator and Control Routine (including input/output and standard functions) occupy approximately 6000 and 4000 32-bit words, respectively (one instruction per word).

(*iii*) PEGASUS

The PEGASUS ALGOL compiler was written by K. L. Ryder and Miss M. J. MacDonald of the Mathematical Services Section, De Havilland Aircraft Co., Ltd.

The compiler accepts KDF9 ALGOL subject to the following restrictions:

(a) no own variables or arrays;

(b) no strings (but there is a facility for copying strings from the input to the output);

(c) array parameters can only be called by name, and not by value;

(d) as with the DEUCE ALGOL compiler, a knowledge of the internal workings of the Control Routine is required in order to write code procedures.

The compiler accepts the 5-hole ALGOL representation given in Appendix 4. Procedures for data input and output, which use 5-hole tape, and also the standard functions, are provided by means of code procedures. The Translator and the Control Routine occupy approximately 3100 and 1100 39-bit words, respectively (two instructions per word).

(*iv*) ACE

The ACE ALGOL compiler has been written by M. Woodger and C. W. Nott of the Mathematics Division of the National Physical Laboratory.

The compiler is intended as a very close copy of the KDF9 version, for use until a KDF9 computer is installed at the N.P.L.

The main difference is that code procedures have to be written in ACE machine code.

The compiler accepts the 8-hole ALGOL representation given in Appendix 4. Input and output is provided on 80-column punched cards using code procedures. The Translator and Control Routine occupy approximately 3900 and 2400 48-bit words, respectively (one instruction per word).

APPENDIX 10

Control Routine Flow Diagrams

The flow diagrams consist of a set of separate routines corresponding to the various object program operations, and a set of subroutines.

The subroutines are headed by the name of the corresponding object program operation and its parameters. In general, each routine ends by reaching 'Control' which causes the routine corresponding to the object program operation indicated by the current value of the program counter to be obeyed.

Subroutines are headed by a name in a rectangular box, followed by a list of any parameters to the subroutine. Each subroutine ends by reaching 'EXIT'. A subroutine is called by giving its name and any parameters, and returns at EXIT to the instruction following the one that called the subroutine.

In general all quantities which are manipulated are stored in a complete word and are local to the routine or subroutine in which they appear. Packing and unpacking items into parts of words is given explicitly.

Quantities which have a global scope rather than being local to a routine or subroutine are:

i	program counter (in terms of syllables)
AP	accumulator pointer
PP	procedure pointer
L_p	initially extent of first order own working storage, then used by the *MOSF* operation
L_0	extent of second order own working storage
S	the stack (a vector)
prog	the object program (a vector)
DISPLAY	(a vector)

Certain other quantities temporarily retain their values between different routines and subroutines – when this is the case it is mentioned explicitly. The parameters to operation routines are automatically expanded to full words, and both they and the parameters to subroutines are regarded as being 'called by value', (i.e. the parameters are evaluated on entry to the routine or subroutine and thereafter used as local variables).

The identifiers *L1*, *L2*, etc., are used to label certain instructions in the routines.

Notation

Advantage is taken of the fact that multiple subscripting is not used in order to compress instructions such as

$$S[AP] := X;\ S[AP-1] := Y$$

into the single instruction

$$S\,[AP,\ AP-1] := X,\ Y$$

A vertical line is used to separate a set of alternatives. For example at a branch point controlled by the test

$$S\,[AP] = \text{'real'} \mid \text{'integer'}\ ?$$

the path labelled 'YES' would be followed if '$S\,[AP]$' contained the bit pattern representing either 'real' or 'integer'.

The function designator D is such that

$$D\,(n,p) \equiv DISPLAY\,[n] + p$$

The following notation is used to denote the expansion of a section of a word into a complete word

$h1\,(X)$	–	1st half of word X
$h2\,(X)$	–	2nd half of word X
$s1\,(X)$	–	1st syllable of word X
$s23\,(X)$	–	2nd and 3rd syllables of word X

The notation

word $(X,\ Y)$ or *word* $(X,\ Y, Z)$

denotes the word made by combining the parameters, as two half-words or three double-syllables, respectively. The reverse process uses the notation

words $(X,\ Y)$ or *words* $(X,\ Y, Z)$

For example

words $(a,\ b) := S\,[AP]$

indicates that the contents of '$S\,[AP]$' are split into two half-words which are expanded into complete words and assigned to a and b.

Bit Patterns

The various bit patterns and their meanings are

0 1 1 0 0 0 0 0 0	'real'
0 0 1 0 0 0 0 0 0	'integer'
0 0 0 1 0 0 0 0 0	'Boolean'
0 1 1 0 0 1 0 0 0	'real address'
0 0 1 0 0 1 0 0 0	'integer address'
0 0 0 1 0 1 0 0 0	'Boolean address'
1 1 1 0 0 1 0 0 0	'real proc. address'
1 0 1 0 0 1 0 0 0	'integer proc. address'
1 0 0 1 0 1 0 0 0	'Boolean proc. address'
0 1 1 0 0 1 0 0 1	'real array'

0 0 1 0 0 1 0 0 1	'integer array'
0 0 0 1 0 1 0 0 1	'Boolean array'
0 1 1 0 0 0 1 0 0	'real procedure'
0 0 1 0 0 0 1 0 0	'integer procedure'
0 0 0 1 0 0 1 0 0	'Boolean procedure'
0 0 0 0 1 0 0 0 0	'label' or 'formal switch'
0 0 0 0 1 0 0 0 1	'switch'
0 0 0 0 0 0 1 0 0	'procedure'
0 0 0 0 0 0 1 1 0	'implicit subroutine'
0 0 0 0 0 0 0 0 0	'string'

The term 'arith' (arithmetic) is used to mean 'real' or 'integer' and the term 'alg' (algebraic) to mean 'arith' or 'Boolean'.

Index to Control Routine Flow Diagrams

AT ENTRY TO CONTROL ROUTINE

This uses the values contained in L_0 and L_p, which have been set up by the Translator. The program counter *i*, *PP* and '*DISPLAY* [*0*]' are set to zero. *AP* is set up to indicate the extent of own storage. The final action is to reach 'Control', which causes the routine corresponding to the operation given in the first syllable of the object program to be obeyed.

CBL (Call Block)

This uses the subroutine GO, and uses the local variable *a* to contain the address of the corresponding *BE* operation before this is placed in the object program counter. The machine code link *L1* (the address of 'Control') is stacked.

CF (Call Function)

Similar to *CBL*, but hands the value of its parameter *m* to the subroutine GO, and places its parameter *a* in the object program counter.

CFZ (Call Function Zero)

A version of *CF*, in which *m* is automatically set to zero.

CFF (Call Formal Function)

This routine uses its (*n*,*p*) parameter (evaluated using the local variable *s*) to locate the appropriate formal accumulator, and then uses the subroutine GO FORMAL.

CFFZ (Call Formal Function Zero)

As *CFF*, but *m* is automatically set to zero.

BE (Block Entry)

All the work of this routine is performed by the subroutine ENTER.

PE (Procedure Entry)

After checking that its parameter *m* is equal to the variable *m'*, which has been set up by the subroutine GO in the operation which called the *PE* operation, the routine uses the subroutine ENTER.

AT ENTRY TO CONTROL ROUTINE

$i := 0$
$PP := DISPLAY[0] := 0$
$AP := L_o + L_p$
Control

CBL
$i := i + 1$
$AP := AP - 2$
$a := i + 3$

CFZ (a)
$i := i + 3$

CF (a), m
$i := i + 4$

$m := 0$

GO (m)

$i := a$
$h2(S[PP+2]) := L1$
L1: Control

CFFZ (n,p)
$i := i + 3$
$s := D(n,p)$
$m := 0$

CFF (n,p), m
$i := i + 4$
$s := D(n,p)$

words (P,a) $:= S[s]$

GO FORMAL $(m,P,a,L1)$

L1: Control

BE (n,L)
$i := i + 3$

PE (n,L), m
$i := i + 4$
$m = m'$?

YES

NO

ENTER (n,L)
Control

FAIL

RETURN

This routine, which uses the subroutine LEAVE, has a local variable *LINK* which is set up with the machine code link given in the current set of stacked link data. The final action of *RETURN* is not to call 'Control' for the next operation, but to jump to the machine code instruction whose address is given in *LINK*.

EIS (End Implicit Subroutine)

After copying the contents of the top accumulator into the current result accumulator, this routine joins *RETURN*.

REJECT

Removes the top accumulator, by decreasing *AP* by two.

GTA (Go To Accumulator)

This routine uses two local variables *q* and *a* (set up with the value of *PP* and the program address, respectively) which characterize a stacked label. When *GTA* causes the current block to be left, the subroutine UDD is used and *AP* and *PP* are reset. The final action is to reset the program counter.

DUMMY

Performs no action.

IFJ (If False Jump)

The routine checks that the top accumulator contains a Boolean value, and resets the program counter using the value of the parameter *a*, if the accumulator has the value **false**.

UJ (Unconditional Jump)

Resets the program counter, using the value of its parameter *a*.

RETURN

$i := i + 1$

EIS

$i := i + 1$
$S[PP-2, PP-1] := S[AP-2, AP-1]$

$i := h1(S[PP+2])$
$LINK := h2(S[PP+2])$

LEAVE

EXIT TO LINK

REJECT

$i := i + 1$
$AP := AP-2$
Control

GTA

$i := i + 1$
$words\ (q,a) := S[AP-2]$
$q = PP$?

YES

NO

$AP := AP - 2$

UDD $(q,s1(S[PP+1]))$

$PP := q$
$AP := s23(S[PP+1])$

$i := a$
Control

DUMMY

$i := i + 1$
Control

IFJ (a)

$i := i + 3$
$S[AP-1]$?

other

Boolean

FAIL

$S[AP-2] =$ **true** ?

NO

YES

$i := a$

$AP := AP-2$
Control

UJ (a)

$i := i + 3$
$i := a$
Control

TFR (Take Formal Real)

This routine uses the subroutine TAKE FORMAL, which in simple cases will act as a normal subroutine, producing either a value or an address in the top accumulator and returning directly to the point at which the routine calls the subroutine COND TAKE. In more complicated cases TAKE FORMAL will involve the use of a subroutine of object program operations or a call on a procedure. In such a case a return is made to the routine *TFR* from the operation *EIS* or *RETURN*, at the position *L4* (the address of the machine code instruction calling the subroutine COND TAKE) which had been stacked by TAKE FORMAL. Thus in all cases COND TAKE is reached, and finally the contents of the top accumulator are checked and, if necessary, converted to type **real**. The local variable *s* is used during the evaluation of the (*n,p*) address of the formal accumulator.

TFI (Take Formal Integer)

As *TFR*, except that the conversion is to type **integer**.

TFB (Take Formal Boolean)

As *TFR*, but without the need for any conversion.

TFL (Take Formal Label)

As *TFB*, but does not call COND TAKE.

TFA (Take Formal Address)

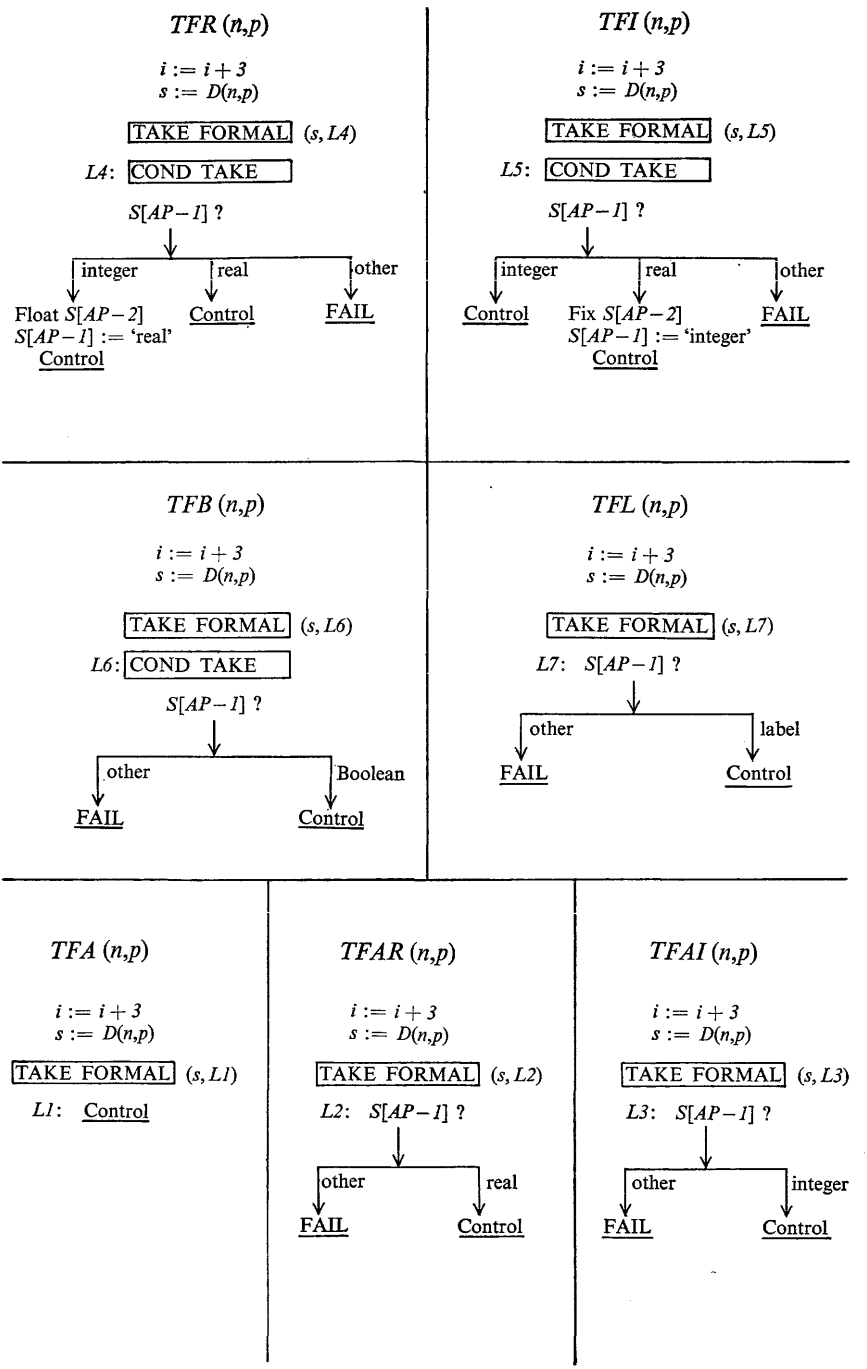
In this case, as the action of the routine after calling TAKE FORMAL is to return to 'Control', *L1* (the address of 'Control') is given as a parameter to TAKE FORMAL.

TFAR (Take Formal Address Real)

After calling TAKE FORMAL, the routine checks that the resulting top accumulator contains a real address.

TFAI (Take Formal Address Integer)

As *TFAR*, but checks that the top accumulator contains an integer address.


TFR (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L4)
L4: COND TAKE
S[AP−1] ?
integer
real
other
Float S[AP−2]
S[AP−1] := 'real'
Control
Control
FAIL
TFI (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L5)
L5: COND TAKE
S[AP−1] ?
integer
real
other
Control
Fix S[AP−2]
S[AP−1] := 'integer'
Control
FAIL
TFB (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L6)
L6: COND TAKE
S[AP−1] ?
other
Boolean
FAIL
Control
TFL (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L7)
L7: S[AP−1] ?
other
label
FAIL
Control
TFA (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L1)
L1: Control
TFAR (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L2)
L2: S[AP−1] ?
other
real
FAIL
Control
TFAI (n,p)
i := i + 3
s := D(n,p)
TAKE FORMAL (s, L3)
L3: S[AP−1] ?
other
integer
FAIL
Control

CSR (Check and Store Real)

This routine is similar to the operation *TFR* in that it uses the subroutines TAKE FORMAL and COND TAKE to evaluate an actual parameter which might be represented by a subroutine of object program operations. However *CSR* uses the stacked formal pointer (*FP*) to address the required formal accumulator. The final actions, after any necessary conversion to type **real**, are to store the value of the actual parameter in the formal accumulator, and to increase *FP* by two, using the local variable *s*.

CSI (Check and Store Integer)

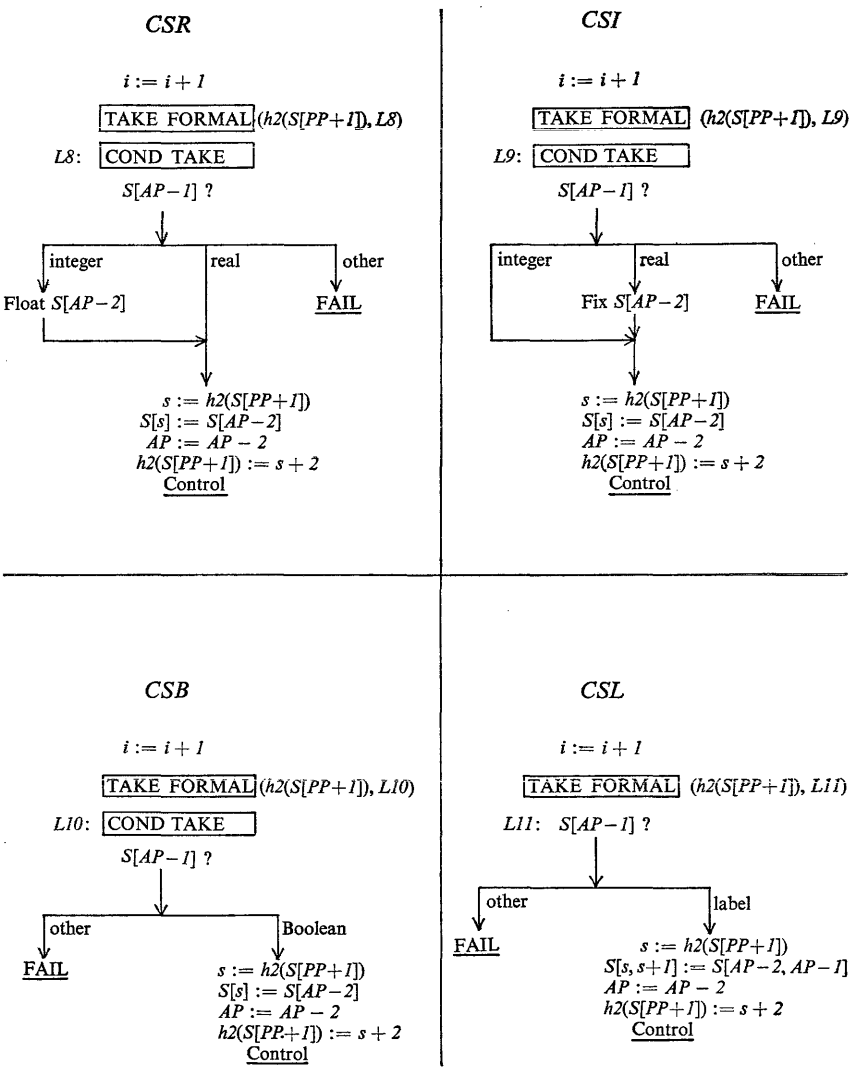
As *CSR*, except that the conversion is to type **integer**.

CSB (Check and Store Boolean)

As *CSR*, but without the need for any conversion.

CSL (Check and Store Label)

As *CSB*, but does not call COND TAKE, and replaces the bit pattern in the second word of the formal accumulator with the bit pattern from the accumulator produced by TAKE FORMAL.


CSR
i := i + 1
TAKE FORMAL (h2(S[PP+1]), L8)
L8: COND TAKE
S[AP−1] ?
integer
real
other
Float S[AP−2]
FAIL
s := h2(S[PP+1])
S[s] := S[AP−2]
AP := AP − 2
h2(S[PP+1]) := s + 2
Control
CSI
i := i + 1
TAKE FORMAL (h2(S[PP+1]), L9)
L9: COND TAKE
S[AP−1] ?
integer
real
other
Fix S[AP−2]
FAIL
s := h2(S[PP+1])
S[s] := S[AP−2]
AP := AP − 2
h2(S[PP+1]) := s + 2
Control
CSB
i := i + 1
TAKE FORMAL (h2(S[PP+1]), L10)
L10: COND TAKE
S[AP−1] ?
other
Boolean
FAIL
s := h2(S[PP+1])
S[s] := S[AP−2]
AP := AP − 2
h2(S[PP+1]) := s + 2
Control
CSL
i := i + 1
TAKE FORMAL (h2(S[PP+1]), L11)
L11: S[AP−1] ?
other
label
FAIL
s := h2(S[PP+1])
S[s, s+1] := S[AP−2, AP−1]
AP := AP − 2
h2(S[PP+1]) := s + 2
Control

CRFA (Copy Real Formal Array)

This routine copies an actual parameter array into local working storage, and sets up a suitable array word in the space occupied by the formal parameter.

Variables local to the routine are

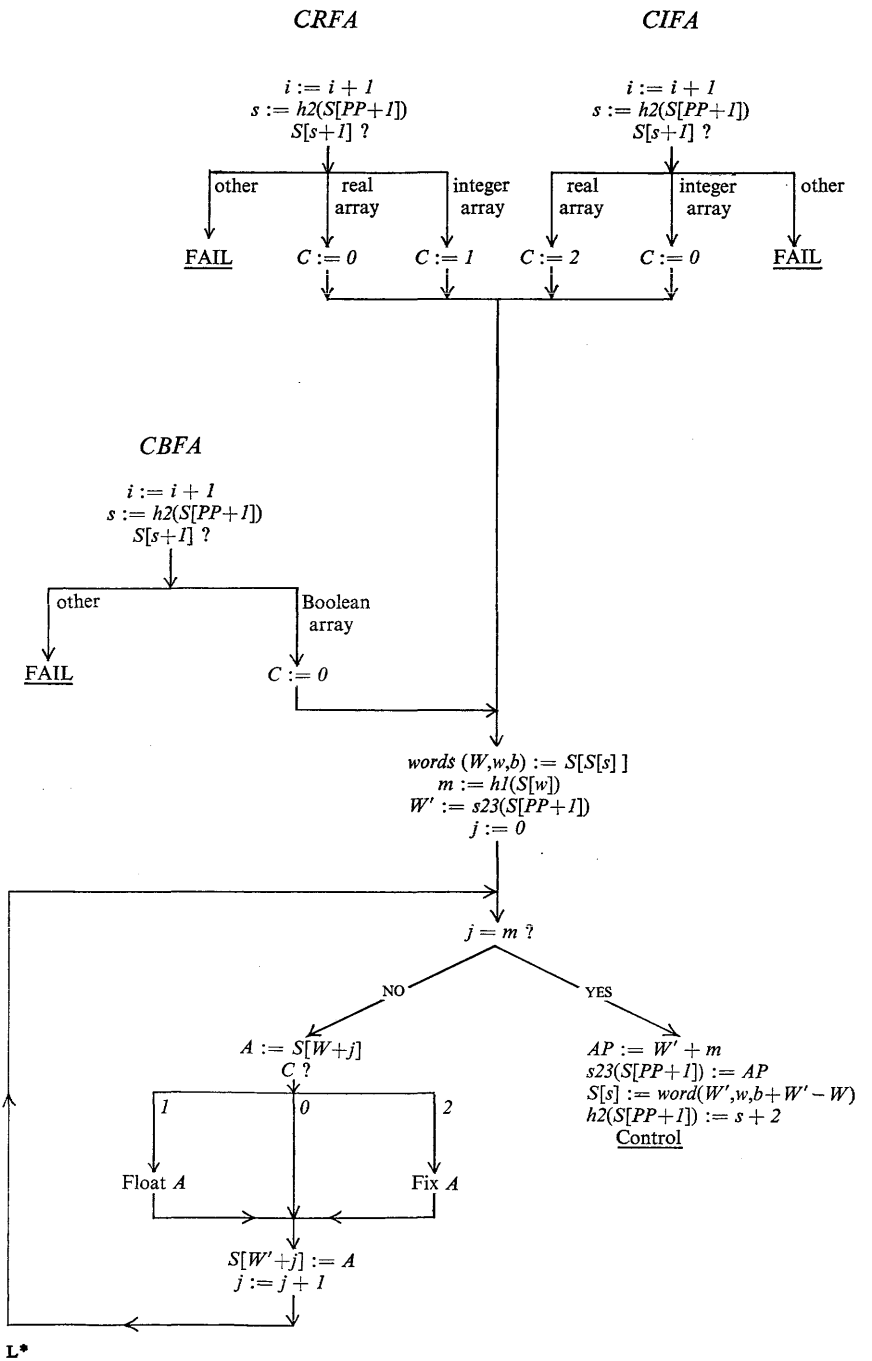
s	set up with the value of the formal pointer
C	used to control any possible conversions from type **integer** to type **real**
W	starting address of the array
w	address of the storage mapping function of the array
b	base address of the array
W'	starting address of the new copy of the array
m	number of elements in the array
A	temporary store for array elements
j	counter used during copying of the array

CIFA (Copy Integer Formal Array)

As *CRFA*, but with possible conversions from type **real** to type **integer**.

CBFA (Copy Boolean Formal Array)

As *CRFA*, but without any conversions.


CRFA
CIFA
i := i + 1
s := h2(S[PP+1])
S[s+1] ?
other
real array
integer array
FAIL
C := 0
C := 1
C := 2
C := 0
FAIL
CBFA
i := i + 1
s := h2(S[PP+1])
S[s+1] ?
other
Boolean array
FAIL
C := 0
words (W,w,b) := S[S[s]]
m := h1(S[w])
W' := s23(S[PP+1])
j := 0
j = m ?
NO
YES
A := S[W+j]
C ?
1
0
2
Float A
Fix A
S[W'+j] := A
j := j + 1
AP := W' + m
s23(S[PP+1]) := AP
S[s] := word(W',w,b+W'−W)
h2(S[PP+1]) := s + 2
Control
L*

TRA (Take Real Address)

This routine sets up an accumulator with an evaluated address calculated, using the local variable *s*, from the dynamic stack address given as a parameter to the *TRA* operation, and the bit pattern 'real address'. If *p* is zero, then the address is adjusted to be that of the result accumulator, and the bit pattern to 'real procedure address'.

TIA (Take Integer Address)

Similar to *TRA*.

TBA (Take Boolean Address)

Similar to *TRA*.

TRR (Take Real Result)

Stacks an accumulator with the contents of the word whose dynamic address is given as a parameter, and the bit pattern 'real'.

TIR (Take Integer Result)

Similar to *TRR*.

TBR (Take Boolean Result)

Similar to *TRR*.

TRC (Take Real Constant)

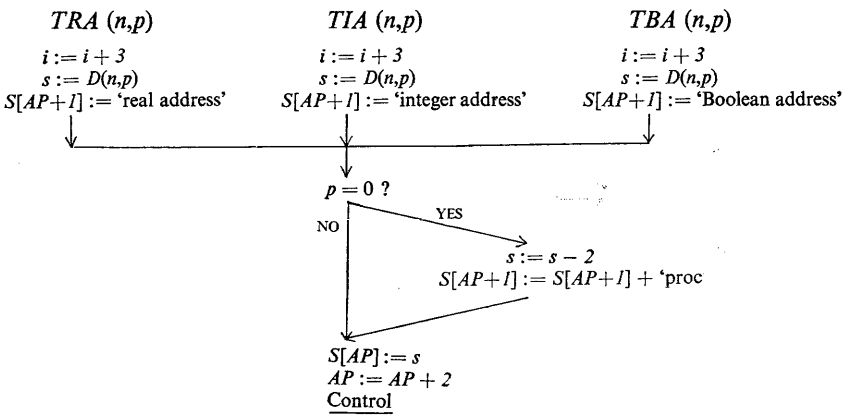
Stacks an accumulator containing the constant given in the six syllables following the operation *TRC*.

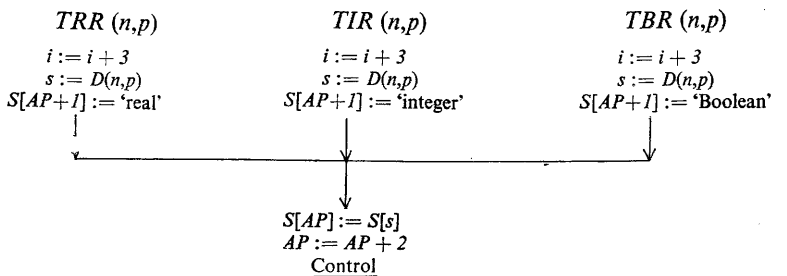
TIC (Take Integer Constant)

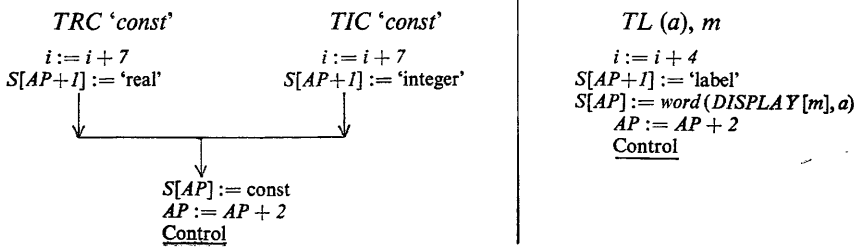
Similar to *TRC*.

TL (Take Label)

Stacks an accumulator containing the (*PP*, *a*) representation of a label.


TRA (n,p)
i := i + 3
s := D(n,p)
S[AP+1] := 'real address'
TIA (n,p)
i := i + 3
s := D(n,p)
S[AP+1] := 'integer address'
TBA (n,p)
i := i + 3
s := D(n,p)
S[AP+1] := 'Boolean address'
p = 0 ?
NO
YES
s := s - 2
S[AP+1] := S[AP+1] + 'proc
S[AP] := s
AP := AP + 2
Control


TRR (n,p)
i := i + 3
s := D(n,p)
S[AP+1] := 'real'
TIR (n,p)
i := i + 3
s := D(n,p)
S[AP+1] := 'integer'
TBR (n,p)
i := i + 3
s := D(n,p)
S[AP+1] := 'Boolean'
S[AP] := S[s]
AP := AP + 2
Control


TRC 'const'
i := i + 7
S[AP+1] := 'real'
TIC 'const'
i := i + 7
S[AP+1] := 'integer'
S[AP] := const
AP := AP + 2
Control
TL (a), m
i := i + 4
S[AP+1] := 'label'
S[AP] := word (DISPLAY [m], a)
AP := AP + 2
Control

TIC1 (Take Integer Constant One)
Stacks an accumulator with the value '*1*'.

TIC0 (Take Integer Constant Zero)
Similar to *TIC1*.

TBCT (Take Boolean Constant True)
Stacks an accumulator with the value **'true'**.

TBCF (Take Boolean Constant False)
Similar to *TBCT*.

TSA (Take Switch Address)
Stacks an accumulator with a switch address.

DSI (Decrement Switch Index)
Uses the local variable *x* whilst decreasing the switch index by one. If the switch index is then zero the program counter is replaced by the value given as a parameter to *DSI*.

AOA (Avoid Own Array)
This operation replaces itself by the operation *UJ*.

ESL (End Switch List)
This operation, reached in the case of an out-of-range switch index, leads to the failure routine.

Check operations

These operations check the bit pattern of the appropriate formal accumulator, addressed (using the local variable *s*) by means of the stacked formal pointer, according to the following table

CA	'arith' | 'arith address' | 'arith procedure' | 'imp. s.r.'
CB	'Boolean' | 'Boolean address' | 'Boolean procedure' | 'imp. s.r.'
CL	'label' | 'imp. s.r.'
CAR	'real array'
CAI	'integer array'
CAB	'Boolean array'
CPR	'procedure'
CFR	'real procedure'
CFI	'integer procedure'
CFB	'Boolean procedure'
CST	'string'
CSW	'switch'

TIC1

$i := i + 1$
$S[AP] := 1$

TIC0

$i := i + 1$
$S[AP] := 0$

$S[AP+1] :=$ 'integer'
$AP := AP + 2$
Control

TBCT

$i := i + 1$
$S[AP] :=$ **true**

TBCF

$i := i + 1$
$S[AP] :=$ **false**

$S[AP+1] :=$ 'Boolean'
$AP := AP + 2$
Control

TSA (a)

$i := i + 3$
$S[AP+1] :=$ 'switch'
$S[AP] := word(0,a)$
$AP := AP + 2$
Control

DSI (a)

$i := i + 3$
$x := S[AP + 3] - 1$
$x = 0$?

YES

NO

$S[PP+3] := x$
$i := a$

Control

AOA (a)

$i := i + 3$
$prog[i-3] :=$ 'UJ'
Control

ESL

$i := i + 1$
FAIL

CA, CB, CL, CAR, CAI, CAB, CPR, CFR, CFI, CFB, CST, CSW

$i := i + 1$
$s := h2(S[PP+1])$
$S[s+1]$?

other

see table

FAIL

$h2(S[PP+1]) := s + 2$
Control

$+, -, \times$

The routines for these three operations are given together, using the notation 'op' to indicate where the appropriate operation is required. The subroutine TYPE CHANGE, which is used to set up the operands in the stores *X* and *Y*, has two different exits, according to the type of the two operands.

NEG

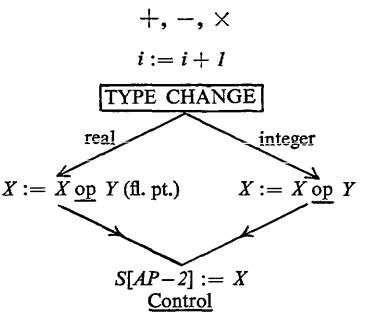
In this operation, which uses the local variable *X*, a failure is indicated if the top accumulator is not arithmetic.

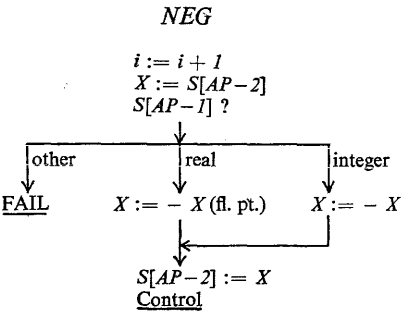
/

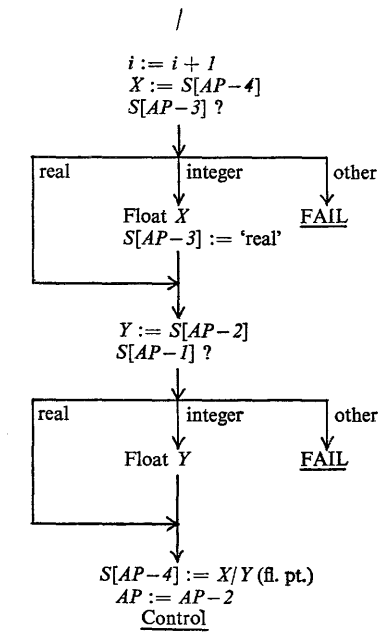
The local variables *X* and *Y* are used to contain the values of the two operands, converted to type **real** if necessary. A failure is indicated if either operand is not real or integer.

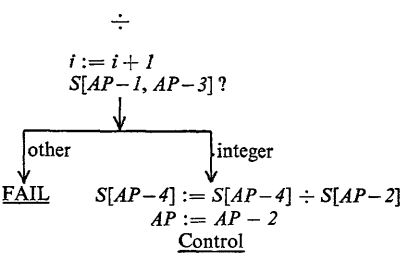
÷

This operation, which performs integer division, using the top two accumulators as operands, indicates a failure if either of the accumulators does not contain an integer.


+, −, ×
i := i + 1
TYPE CHANGE
real
integer
X := X op Y (fl. pt.)
X := X op Y
S[AP−2] := X
Control


NEG
i := i + 1
X := S[AP−2]
S[AP−1] ?
other
real
integer
FAIL
X := − X (fl. pt.)
X := − X
S[AP−2] := X
Control


/
i := i + 1
X := S[AP−4]
S[AP−3] ?
real
integer
other
Float X
S[AP−3] := 'real'
FAIL
Y := S[AP−2]
S[AP−1] ?
real
integer
other
Float Y
FAIL
S[AP−4] := X/Y (fl. pt.)
AP := AP−2
Control


÷
i := i + 1
S[AP−1, AP−3] ?
other
integer
FAIL
S[AP−4] := S[AP−4] ÷ S[AP−2]
AP := AP − 2
Control

↑ (Exponentiation)

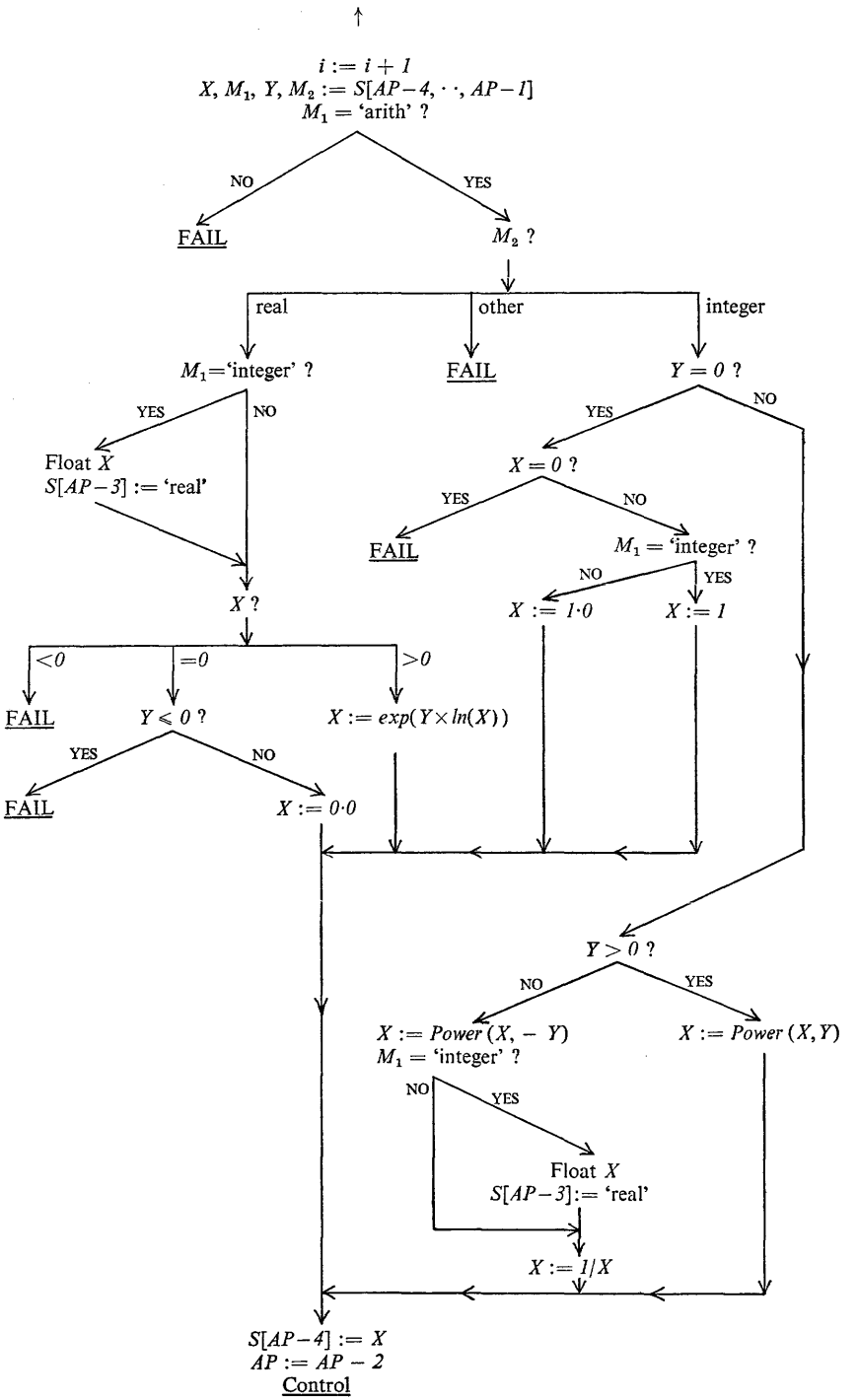
This routine performs exponentiation according to the rules given in section 3.3.4.3 of the Revised Algol Report. The routines *exp* (exponential) and *ln* (natural logarithm), which operate on either real numbers or integers, are used. The subroutine '*Power* (*a*, *b*)' performs the repeated multiplication

$$a \times a \times \ldots \times a \ (b \text{ times})$$

where *a* is of type **real** or **integer** and *b* is a positive integer, by repeated shifting down of the exponent and squaring of the base.

The local variables *X* and *Y* are used to contain the two operands, and the local variables M_1 and M_2 to contain bit patterns representing the type of *X* and *Y*, respectively.

The routine reaches FAIL if *X* and *Y* are such that '$X \uparrow Y$' is undefined according to section 3.3.4.3 of the Revised Algol Report.


↑
i := i + 1
X, M1, Y, M2 := S[AP−4, · ·, AP−1]
M1 = 'arith' ?
NO
YES
FAIL
M2 ?
real
other
integer
M1='integer' ?
FAIL
Y = 0 ?
YES
NO
YES
NO
Float X
S[AP−3] := 'real'
X = 0 ?
YES
NO
FAIL
M1 = 'integer' ?
NO
YES
X := 1·0
X := 1
X ?
<0
=0
>0
FAIL
Y ⩽ 0 ?
X := exp(Y×ln(X))
YES
NO
FAIL
X := 0·0
Y > 0 ?
NO
YES
X := Power (X, − Y)
M1 = 'integer' ?
X := Power (X, Y)
NO
YES
Float X
S[AP−3]:= 'real'
X := 1/X
S[AP−4] := X
AP := AP − 2
Control

$<, \leqslant, =, \geqslant, >, \neq$

The routines for the various relational operations have been combined, using the notation 'op' to indicate the particular operation required. The local variables *X* and *Y* are set up by the subroutine TYPE CHANGE.

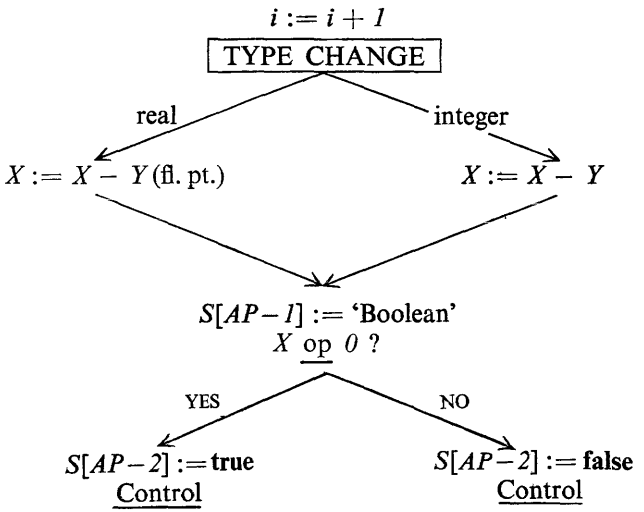
$\wedge, \vee, \supset, \equiv$

The routines for these Boolean operations have been combined in a similar way. A failure is indicated if either of the operands is not Boolean.

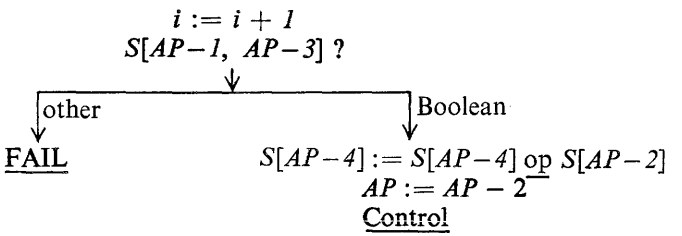
$\neg$

This routine indicates a failure if the top accumulator is not of type **Boolean**, otherwise it replaces it with the opposite Boolean value.

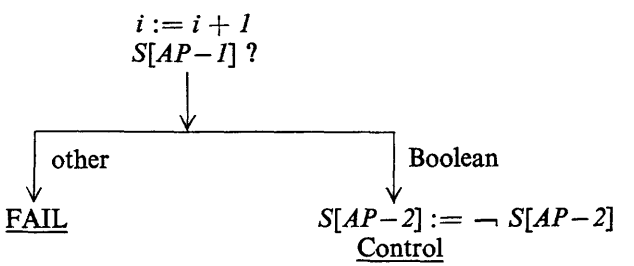
$<, \leqslant, =, \geqslant, >, \neq$



$\wedge, \vee, \supset, \equiv$



$\neg$



ST (Store)

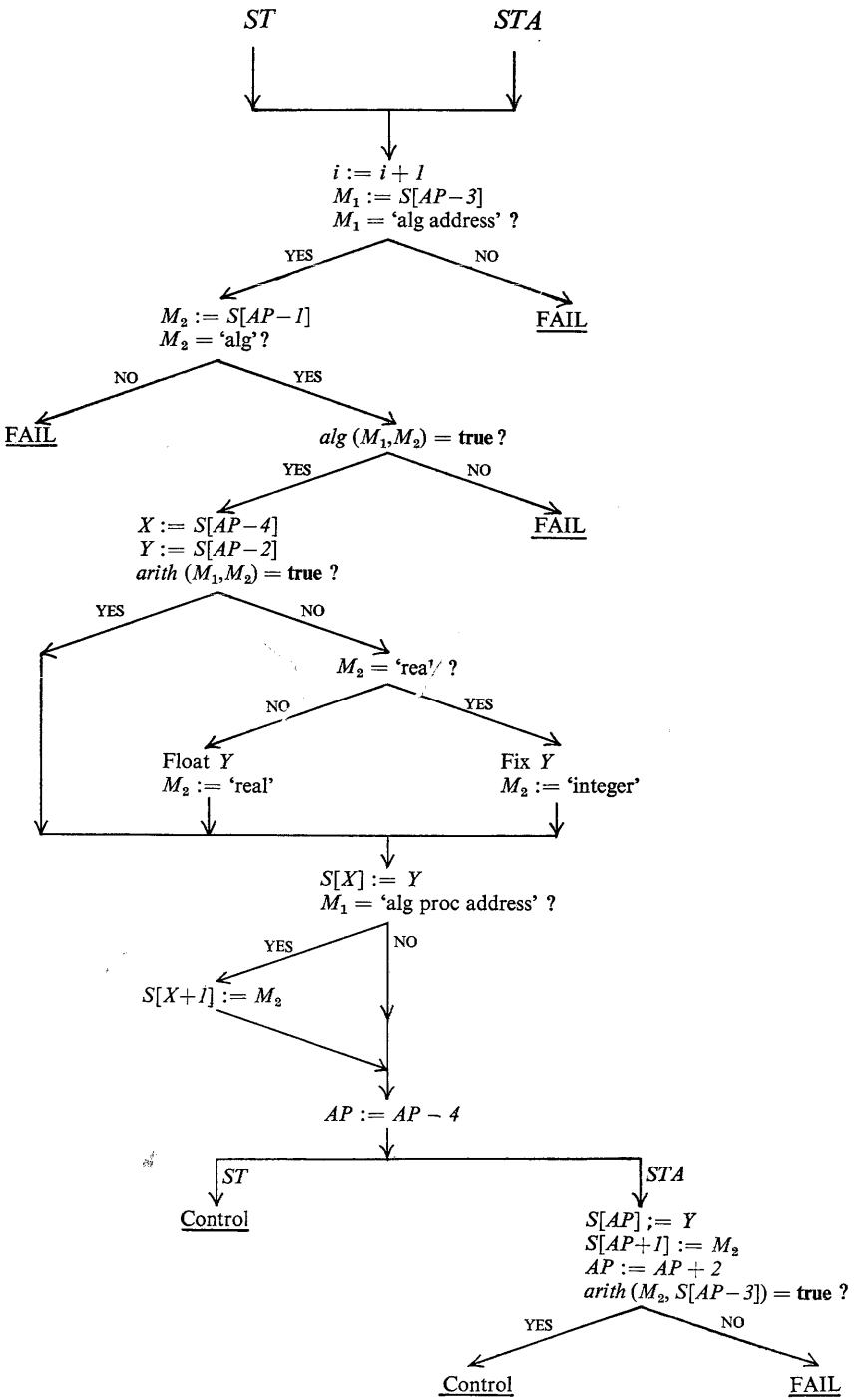
That part of the flow diagram which is common to both *ST* and *STA* is referred to elsewhere as STORE. The variables X and Y are used for the values of the top two accumulators, and M_1 and M_2 for their accompanying bit patterns. Of these Y and M_2 are retained for use outside STORE.

The notation '*alg* (*a*, *b*)' denotes a Boolean function designator, which has the value **true** when the sections of the bit patterns *a* and *b* which denote type information both represent 'Boolean' or both represent 'arithmetic' (i.e. real or integer). Similarly '*arith* (*a*, *b*)' has the value **true** if the type information sections of the bit patterns are identical, but has the value **false** if one represents a real quantity and the other an integer quantity.

Using these function designators, STORE checks that the contents of the top two accumulators are acceptable, and determines whether any real-integer conversions are necessary, before performing the required storage.

STA (Store Also)

After using STORE the operation *STA* moves the top accumulator down one place, and checks that the next variable in the left part list is of identical type.


ST
STA
i := i + 1
M1 := S[AP−3]
M1 = 'alg address' ?
YES
NO
M2 := S[AP−1]
M2 = 'alg'?
FAIL
NO
YES
FAIL
alg (M1,M2) = true ?
YES
NO
X := S[AP−4]
Y := S[AP−2]
arith (M1,M2) = true ?
FAIL
YES
NO
M2 = 'real' ?
NO
YES
Float Y
M2 := 'real'
Fix Y
M2 := 'integer'
S[X] := Y
M1 = 'alg proc address' ?
YES
NO
S[X+1] := M2
AP := AP − 4
ST
STA
Control
S[AP] := Y
S[AP+1] := M2
AP := AP + 2
arith (M2, S[AP−3]) = true ?
YES
NO
Control
FAIL

MSF (Make Storage Function)

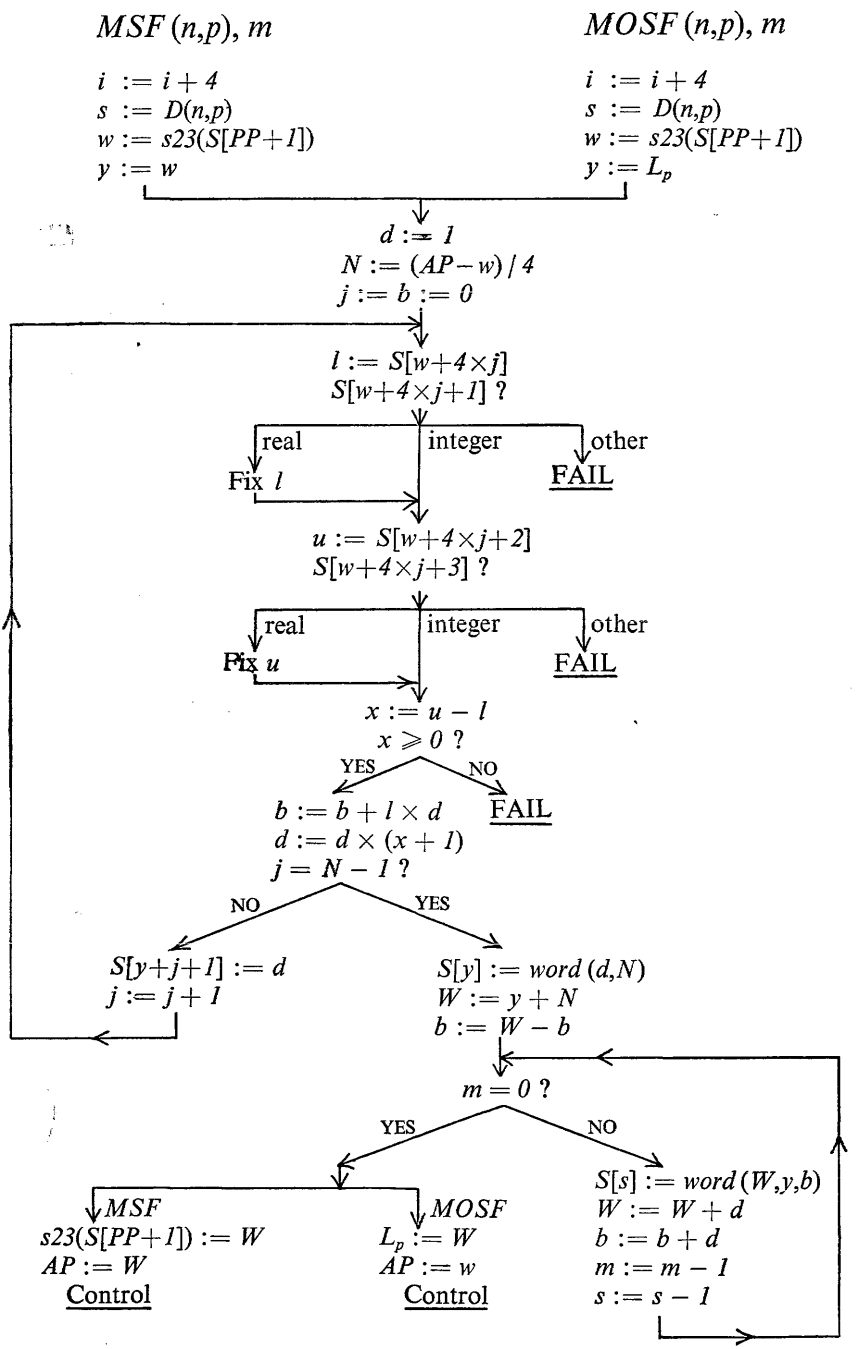
This routine sets up a storage mapping function and the array words for m arrays, the last one of which has the dynamic address (n,p), given with this operation. The various counters which are local to this routine are

- j addresses the stacked accumulators
- b base address
- w set up with the value of the working storage pointer
- y storage position for the mapping function
- d used in the calculation of the mapping function
- N number of dimensions
- W starting address
- s used in the evaluation of the address of the array word
- u upper bound
- l lower bound

The various checks incorporated in the routine are that the stacked accumulators contain integers or real numbers, and that no lower bounds exceed the corresponding upper bounds.

MOSF (Make Own Storage Function)

Most of this routine is common to *MSF*, but the storage mapping function and space for the arrays are set up in own storage under control of the pointer L_p, rather than in local working storage.


MSF (n,p), m
i := i + 4
s := D(n,p)
w := s23(S[PP+1])
y := w
MOSF (n,p), m
i := i + 4
s := D(n,p)
w := s23(S[PP+1])
y := L_p
d := 1
N := (AP − w) / 4
j := b := 0
l := S[w+4×j]
S[w+4×j+1] ?
real
integer
other
Fix l
FAIL
u := S[w+4×j+2]
S[w+4×j+3] ?
real
integer
other
Fix u
FAIL
x := u − l
x ⩾ 0 ?
YES
NO
FAIL
b := b + l × d
d := d × (x + 1)
j = N − 1 ?
NO
YES
S[y+j+1] := d
j := j + 1
S[y] := word (d,N)
W := y + N
b := W − b
m = 0 ?
YES
NO
MSF
s23(S[PP+1]) := W
AP := W
Control
MOSF
L_p := W
AP := w
Control
S[s] := word (W,y,b)
W := W + d
b := b + d
m := m − 1
s := s − 1

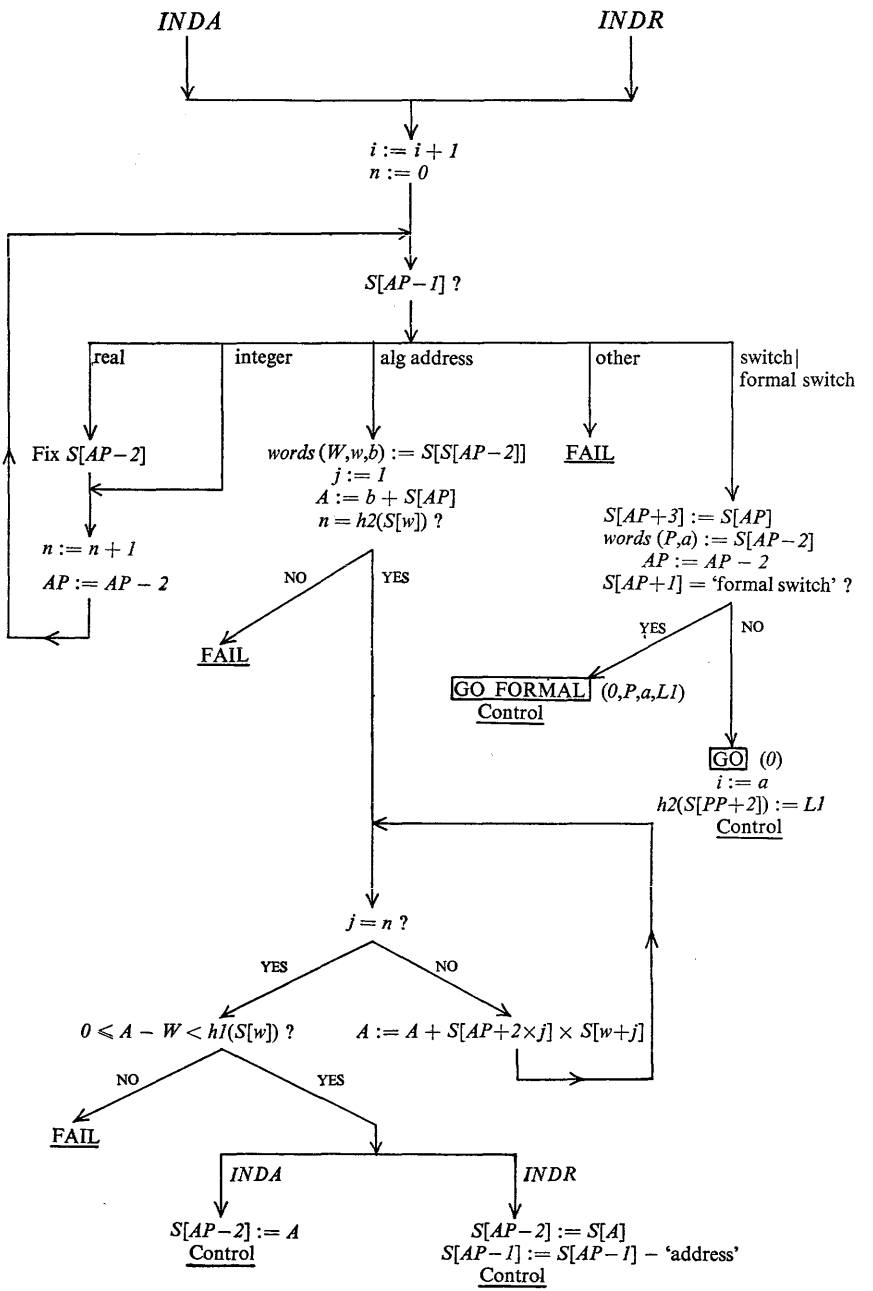
INDA (Index Address)

The local variable *n* is used to determine the number of accumulators containing integers or real numbers at the top of the stack. If an accumulator containing an algebraic address is found then the operation is being used for a subscripted variable. The local variables *W*, *w* and *b* are set up with the starting address of the array, the address of the storage mapping function, and the base address of the array, respectively. Two further local variables, *A* and *j*, are used in the calculation of the stack address of the array element specified by the subscripted variable, and for controlling the loop of instructions used for the case of a multi-dimensional array, respectively. Checks are made that the resulting address is within the confines of the array, and that the subscripted variable has the correct number of subscript expressions.

In the case of the operation being used for a switch designator, an accumulator containing the address of a switch or a formal switch will be found. The local variables *P* and *a* are used for the value of *PP* and the program address, which characterize the switch. The subroutine GO or GO FORMAL is used to activate the switch block.

INDR (Index Result)

The flow diagram of *INDR* is almost identical with that of *INDA*, except that finally, in the case of its being used for a subscripted variable, the value rather the address of the subscripted variable is stacked, and the accompanying bit pattern modified accordingly.


INDA
INDR
i := i + 1
n := 0
S[AP−1] ?
real
integer
alg address
other
switch| formal switch
Fix S[AP−2]
words (W,w,b) := S[S[AP−2]]
j := 1
A := b + S[AP]
n = h2(S[w]) ?
FAIL
S[AP+3] := S[AP]
words (P,a) := S[AP−2]
AP := AP − 2
S[AP+1] = 'formal switch' ?
n := n + 1
AP := AP − 2
NO
YES
FAIL
YES
NO
GO FORMAL (0,P,a,L1)
Control
GO (0)
i := a
h2(S[PP+2]) := L1
Control
j = n ?
YES
NO
0 ⩽ A − W < h1(S[w]) ?
A := A + S[AP+2×j] × S[w+j]
NO
YES
FAIL
INDA
INDR
S[AP−2] := A
Control
S[AP−2] := S[A]
S[AP−1] := S[AP−1] − 'address'
Control

FOR W (For While)

The start of this routine is similar to *FOR A*. However after using STORE the second subroutine of object program operations is used to evaluate the Boolean expression *F*. If this has the value **true** then the program counter is set so that the controlled statement will be obeyed, otherwise the current value of the program counter (which indicates the next for list element) is stored.

FOR A (For Arithmetic)

This routine stores the program counter in '*S* [*PP*−*1*]' whilst it uses the subroutine of object program operations to obtain the address of the controlled variable. The program counter is then reset so that the value of the arithmetic expression can be found. After using the subroutine STORE to perform '*V* := *A*' the program counter is set up with the address of the first operation of the controlled statement.

FBE (For Block Entry)

In addition to performing the tasks of *BE* (Block Entry) this routine stores the addresses of the first of the set of object program operations generated from the controlled variable and the controlled statement, and resets the program counter to point at the first for list element.

FSE (For Statement End)

This routine causes the for block to be left, and a jump to be made to the operation following the *FR* (For Return) operation.

FR (For Return)

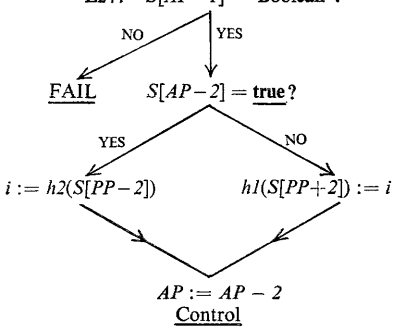
This routine resets the program counter to point at the current for list element.

LINK

This routine jumps to the machine instruction whose address has been stored by a for list element operation.

FOR W

$i := i + 1$
$S[PP-1] := i$
$i := h1(S[PP-2])$
$h2(S[PP+2]) := L22$
Control
[Take Address of V / *LINK*]
L22: $i := S[PP-1]$
$h2(S[PP+2]) := L23$
Control
[Take E / *LINK*]
L23: STORE
$h2(S[PP+2]) := L24$
Control
[Take F / *LINK*]
L24: $S[AP-1]$ = 'Boolean'?



FOR A

$i := i + 1$
$S[PP-1] := i$
$i := h1(S[PP-2])$
$h2(S[PP+2]) := L20$
Control
[Take Address of V / *LINK*]
L20: $i := S[PP-1]$
$h2(S[PP+2]) := L21$
Control
[Take A / *LINK*]
L21: $h1(S[PP+2]) := i$
STORE
$i := h2(S[PP-2])$
Control

FR

$i := i + 1$
$i := h1(S[PP+2])$
Control

LINK

$i := i + 1$
$LINK := h2(S[PP+2])$
Control

FSE (a)

$i := i + 3$
$i := a$
LEAVE
$AP := AP - 2$
Control

FBE (n,L), (a)

$i := i + 5$
$S[PP-2] := word(a,i)$
$i := h1(S[PP+2])$
ENTER (*n,L*)
Control

FOR S1, FOR S2 (For Step, 1st and 2nd Entries)

These routines use the notation

$\{S1\}$ for $S\,[PP+3, PP+4]$
$\{S2\}$ for $S\,[PP+5, PP+6]$
$\{R\}$ for $S\,[AP-2, AP-1]$

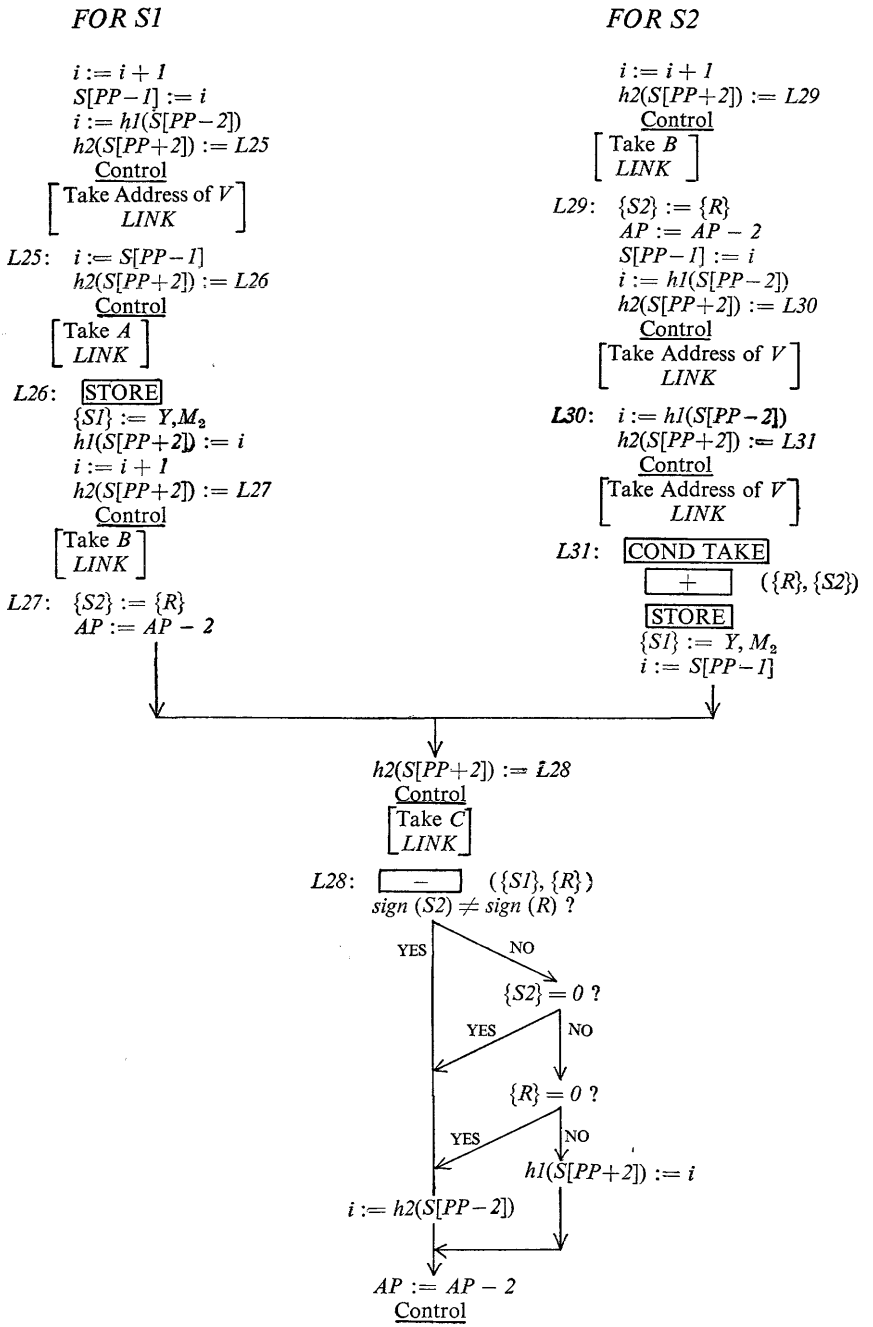
The subroutine STORE (the common part of *ST* and *STA*) has values of Y and M_2 as byproducts. The subroutines '+' and '−' perform addition and subtraction, respectively, on their two parameters (accumulators) and replace the top accumulator, ($\{R\}$), with the result.

FOR S1 performs

$S1 := V := A;\ S2 := B;$
if *sign* $(S2) \times (S1-C) > 0$ **then go to** *exit*;

FOR S2 performs

$S2 := B;\ S1 := V := V + S2;$
if *sign* $(S2) \times (S1-C) > 0$ **then go to** *exit*;


FOR S1
i := i + 1
S[PP−1] := i
i := h1(S[PP−2])
h2(S[PP+2]) := L25
Control
Take Address of V
LINK
L25: i := S[PP−1]
h2(S[PP+2]) := L26
Control
Take A
LINK
L26: STORE
{S1} := Y,M2
h1(S[PP+2]) := i
i := i + 1
h2(S[PP+2]) := L27
Control
Take B
LINK
L27: {S2} := {R}
AP := AP − 2
FOR S2
i := i + 1
h2(S[PP+2]) := L29
Control
Take B
LINK
L29: {S2} := {R}
AP := AP − 2
S[PP−1] := i
i := h1(S[PP−2])
h2(S[PP+2]) := L30
Control
Take Address of V
LINK
L30: i := h1(S[PP−2])
h2(S[PP+2]) := L31
Control
Take Address of V
LINK
L31: COND TAKE
+
({R}, {S2})
STORE
{S1} := Y, M2
i := S[PP−1]
h2(S[PP+2]) := L28
Control
Take C
LINK
L28: −
({S1}, {R})
sign (S2) ≠ sign (R) ?
YES
NO
{S2} = 0 ?
YES
NO
{R} = 0 ?
YES
NO
h1(S[PP+2]) := i
i := h2(S[PP−2])
AP := AP − 2
Control

GO

This subroutine sets up various items of stacked link data, and then processes the 'actual operations' in order to set up the formal accumulators.

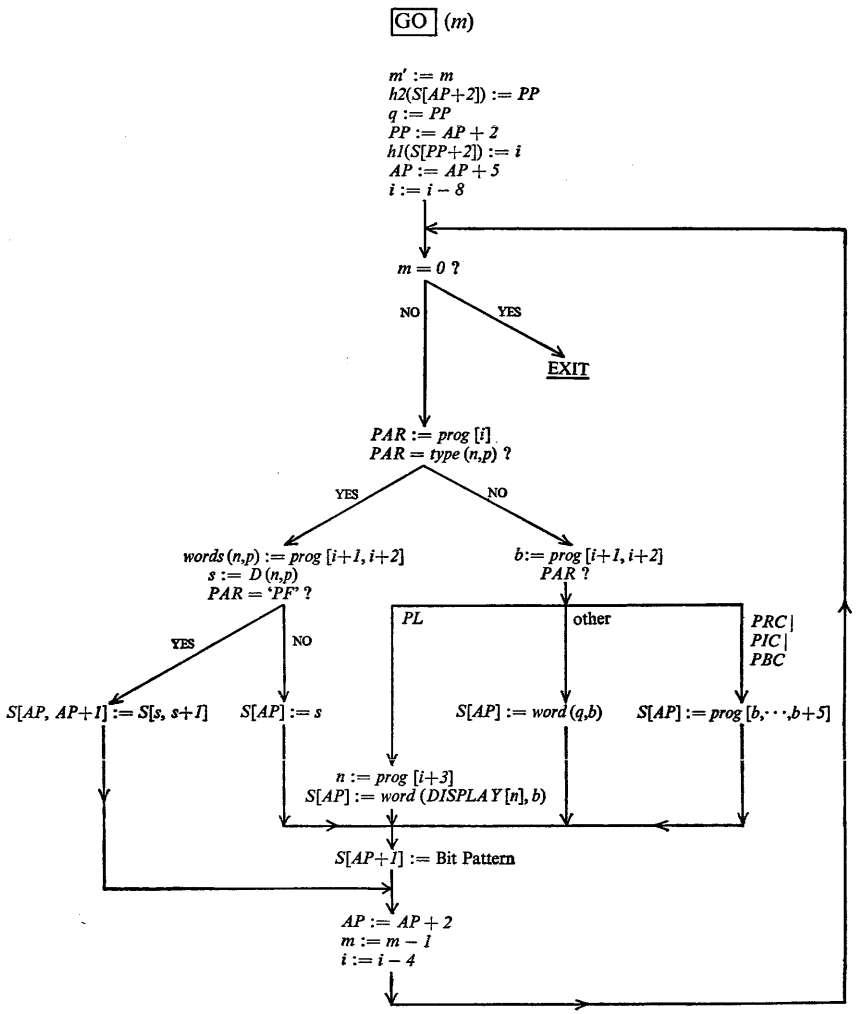
Variables that are local to this subroutine are

q	set up with the value of *PP* on entry to the subroutine
PAR	contains each of the 'actual operations' in turn
b	set up with a program address, if an 'actual operation' has such a parameter
s	set up with a stack address, evaluated from the dynamic address, if an 'actual operation' has such a parameter

The Boolean function designator '*type* (*n*,*p*)' has the value **true** if *PAR* contains an 'actual operation' which has an (*n*,*p*) parameter. (The fact that *n* occupies 6 bits and *p* occupies 10 bits has been ignored in using the procedure *words* to unpack them into complete words).

Each 'actual operation' causes the storing of a bit pattern in the appropriate formal accumulator, according to the following table

PR	'real address'
PI	'integer address'
PB	'Boolean address'
PRC	'real'
PIC	'integer'
PBC	'Boolean'
PRA	'real array'
PIA	'integer array'
PBA	'Boolean array'
PFR	'real procedure'
PFI	'integer procedure'
PFB	'Boolean procedure'
PL	'label'
PSW	'formal switch'
PPR	'procedure'
PSR	'implicit subroutine'
PST	'string'


GO (m)
m' := m
h2(S[AP+2]) := PP
q := PP
PP := AP + 2
h1(S[PP+2]) := i
AP := AP + 5
i := i - 8
m = 0 ?
NO
YES
EXIT
PAR := prog [i]
PAR = type (n,p) ?
YES
NO
words (n,p) := prog [i+1, i+2]
s := D (n,p)
PAR = 'PF' ?
b:= prog [i+1, i+2]
PAR ?
YES
NO
PL
other
PRC |
PIC |
PBC
S[AP, AP+1] := S[s, s+1]
S[AP] := s
S[AP] := word (q,b)
S[AP] := prog [b,···,b+5]
n := prog [i+3]
S[AP] := word (DISPLAY [n], b)
S[AP+1] := Bit Pattern
AP := AP + 2
m := m - 1
i := i - 4

UDD (Update *DISPLAY*)

This subroutine, which is used to ensure that *DISPLAY* coincides with the static chain, uses the local variable k to contain the value of the block level of each set of link data in turn until EXIT is reached.

ENTER

This subroutine completes the setting up of stacked link data.

LEAVE

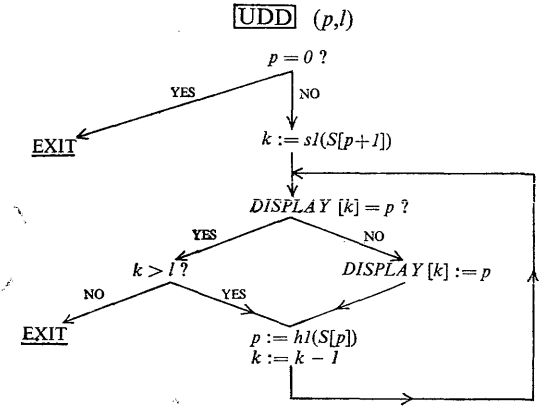
This subroutine resets *AP* and *PP* and then calls the subroutine UDD.

GO FORMAL

This subroutine uses the subroutine GO and UDD, resets the program counter and finally stacks the machine code link given as its final parameter.

TAKE FORMAL

This subroutine either stacks the contents of the formal accumulator (addressed by the parameter s), or uses the contents of the formal accumulator to provide the P and a parameters for GO FORMAL. In the former case the notation '*mask* (x)' indicates the operation of deleting the final two digits of the bit pattern x, thus changing 'real array' into 'real address', etc.



ENTER (*n*,*L*)

AP := *PP* + *L* + 3
s1(*S*[*PP*+*1*]) := *n*
s23(*s*[*PP*+*1*]) := *AP*
h1(*S*[*PP*]) := *DISPLAY* [*n*−*1*]
DISPLAY [*n*] := *PP*
h2(*S*[*PP*+*1*]) := *PP* + 3
EXIT

LEAVE

AP := *PP*
PP := *h2*(*S*[*PP*])
UDD (*PP*, *s1*(*S*[*AP*+*1*]))
EXIT

GO FORMAL (*m*,*p*,*a*,*LINK*)

GO (*m*)
UDD (*p*, *s1*(*S*[*q*+*1*]))
i := *a*
h2(*S*[*PP*+*2*]) := *LINK*
EXIT

TAKE FORMAL (*s*,*LINK*)

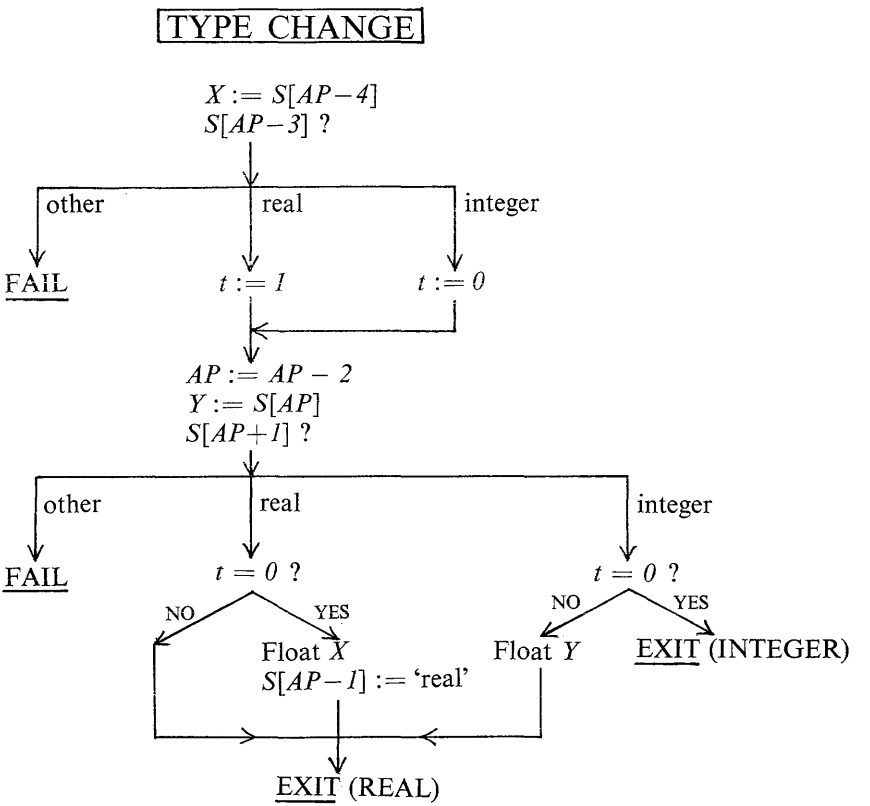
S[*s*+*1*] = 'imp. s.r.' | 'alg proc'?
YES
NO
words(*P*,*a*) := *S*[*s*]
GO FORMAL (*0*,*P*,*a*,*LINK*)
Control
S[*AP*] := *S*[*s*]
S[*AP*+*1*] := *mask* (*S*[*s*+*1*])
AP := *AP* + 2
EXIT

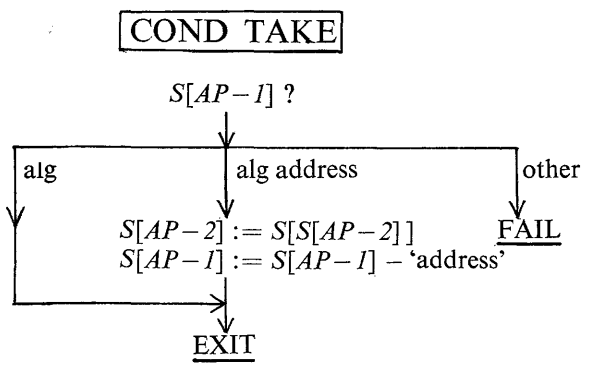
TYPE CHANGE

This subroutine sets up the stores *X* and *Y* with the contents of the top two accumulators, and if one of them is of type **real**, converts the other to be also of type **real** (the local variable *t* is used to control type conversions). A failure is indicated if either accumulator contains other than a real or integer result.

COND TAKE

This routine checks the top accumulator, indicates a failure unless it contains either an algebraic result or an algebraic address, and in the latter case replaces the accumulator with the contents of the address, altering the bit pattern appropriately.


TYPE CHANGE
X := S[AP−4]
S[AP−3] ?
other
real
integer
FAIL
t := 1
t := 0
AP := AP − 2
Y := S[AP]
S[AP+1] ?
other
real
integer
FAIL
t = 0 ?
t = 0 ?
NO
YES
NO
YES
Float X
S[AP−1] := 'real'
Float Y
EXIT (INTEGER)
EXIT (REAL)


COND TAKE
S[AP−1] ?
alg
alg address
other
S[AP−2] := S[S[AP−2]]
S[AP−1] := S[AP−1] − 'address'
FAIL
EXIT

APPENDIX 11

Translator Flow Diagrams

The flow diagrams consist of a set of separate routines corresponding to the various delimiters, and a set of subroutines.

The routines are headed by the corresponding delimiter; in general each routine ends by reaching 'out' which signifies a return to the central loop of the Translator in order to fetch the next ALGOL section. However certain routines, which deal with more than one ALGOL section, end by reaching 'OUT 2', rather than 'out'. This indicates that the last delimiter fetched has not been dealt with, and that the routine corresponding to this delimiter is to be entered.

The flow diagrams use the same basic notation as the Control Routine flow diagrams, given in Appendix 10. Quantities which have a global scope, rather than being local to a particular routine or subroutine, are

i	object program counter (syllables)
SP	Stack Pointer
LEVEL, *R*, *FAIL*	markers for use when an error has been found in the ALGOL text
I	set to indicate a name list entry of current interest
X	used to contain number of dimensions or parameters
Delimiter	current delimiter
identifier	current identifier
constant	current constant
CONS	indicates type of current constant
PROC	set to one whilst processing the parameters of a procedure
EM	End Message marker
case	case marker
u	underline marker
posn	used to indicate whether any printed characters have appeared on the current line
line no.	current line number
rel. line no.	line number relative to the last position identifier
posn. identifier	last position identifier
Y	used in conjunction with *TYPE* to set up *type* column of name list entries
item [*I*]	name list entry *I*
syll [*I*]	*syll* column of name list entry *I*

Similarly, the other constituent parts of name list entry *I* are indicated by

type [*I*], *np* [*I*], *f* [*I*], *v* [*I*], *u* [*I*],
exp [*I*], *dim* [*I*], *FD* [*I*], *line* [*I*],
name [*I*] and *d* [*I*].

The state variables listed in Appendix 8 are also treated as global variables. As with the Control Routine flow diagrams, details of any quantities which, though not global, temporarily retain their values between different routines and subroutines, are given in the accompanying text.

Notation

The action of taking the item at the top of the stack and distributing the various constituent parts of the item into fixed locations is denoted by the procedure '*restore*'. The parameters to this procedure correspond to some or all of the constituent parts of the item at the top of the stack. Those parts which are to be stored in fixed locations are indicated by a parameter, enclosed in square brackets, giving the name of the location. The final action of *restore* is to decrease *SP* by one.

For example

If the item at the top of the stack is

switch begin, *31*, *34*, *0*

then

restore (**switch begin**, [*x*], [*a*])

deletes this item, having set *x* to be *31* and *a* to be *34*.

The procedure '*prestore*' is a variant of *restore* which does not decrease *SP*.

The notation *TS* is used to indicate the item at the top of the stack.

The bit patterns indicating type are split into 12 columns, denoted by the letters *a*, *b*, *c*, *A*, *B*, . . . , *H*, *J*. One or more columns of a bit pattern can be specified using the appropriate letters.

For example

ABG (*TYPE*) := *1*

sets columns *A*, *B* and *G* of the state variable *TYPE* to one, and

aG (*type* [*I*]) = *1*

has the value **true** if columns *a* and *G* of the *type* column of name list entry *I* are equal to one.

The subroutine STACK has, as parameters given on separate lines and enclosed in square brackets, any items which are to be added to the top of the stack. The stack priorities are indicated by underlining.

For example

$$\boxed{\text{STACK}} \quad \begin{bmatrix} UJ,\ i-2,\ \underline{13} \\ MOSF,\ 0,\ \underline{\underline{0}} \end{bmatrix}$$

will stack the item '*UJ, i − 2, 13*' and then the item '*MOSF, 0, 0*'.

The subroutine COMPILE uses a similar notation to indicate any operations (and their parameters) to be added to the object program. The list of operations is preceded by an integer (enclosed in round brackets) which gives the total number of syllables to be added to the object program.

$$\boxed{\text{COMPILE}} \ (9) \ \begin{bmatrix} LINK \\ FSE\ (\ ) \\ FBE\ (\ ),\ (a) \end{bmatrix}$$

Bit Patterns

The various bit patterns and their meanings are

a b c A B C D E F G H J	
1 0 0 1 1 1 0 0 0 1 0 0	'real'
1 0 0 1 1 1 0 1 0 1 0 0	'integer'
1 0 0 1 1 0 1 0 0 1 0 0	'Boolean'
0 1 0 1 1 1 0 0 0 1 0 0	'real array'
0 1 0 1 1 1 0 1 0 1 0 0	'integer array'
0 1 0 1 1 0 1 0 0 1 0 0	'Boolean array'
0 0 1 1 1 1 0 0 0 1 0 0	'real procedure'
0 0 1 1 1 1 0 1 0 1 0 0	'integer procedure'
0 0 1 1 1 0 1 0 0 1 0 0	'Boolean procedure'
0 0 1 0 0 0 0 0 0 1 0 0	'procedure'
1 0 0 1 1 1 0 0 0 1 0 1	'real procedure zero'
1 0 0 1 1 1 0 1 0 1 0 1	'integer procedure zero'
1 0 0 1 1 0 1 0 0 1 0 1	'Boolean procedure zero'
1 0 0 0 0 0 0 0 0 1 0 1	'procedure zero'
0 1 0 1 0 0 0 0 1 0 0 0	'switch'
1 0 0 1 0 0 0 0 1 0 0 0	'label'
0 0 0 0 0 0 0 0 0 0 1 0	'string'

The bit patterns used for the internal representations of the delimiters, stacked items and object program operations can be chosen in order to facilitate the various manipulations and tests on the bit patterns used in the Compiler. However, the choice of bit patterns is largely governed by the facilities in the order code of the particular computer on which the compiler is being implemented; therefore the bit patterns used in the Whetstone Compiler are not given here.

Index to Translator Flow Diagrams

AT ENTRY TO TRANSLATOR

As this routine is the first one entered at the start of a translation its first task is to set the various markers and state variables to their initial values. The markers referred to by the statement 'Zero markers' are

R, *LEVEL*, *FAIL*, *buffer*, *line buffer*, *i*, *EM*, *case*, *u*, *posn identifier*, *rel. line no.*, *line no.*, *ARITH*, L_0, *n*, *F*, *PROC*, *D*, *T* and *TYPE*.

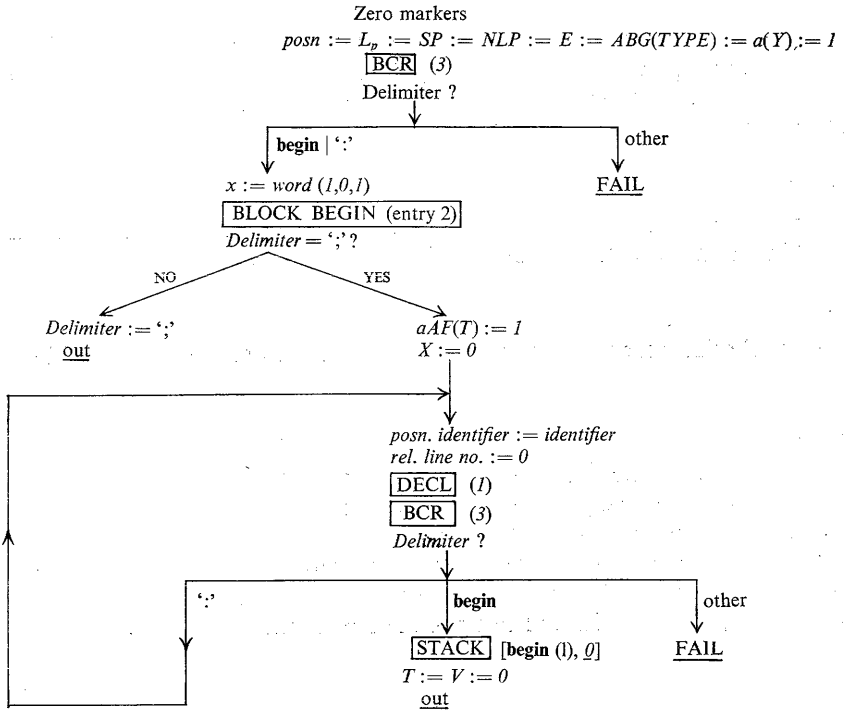
The routine then checks that the program commences either with **begin**, or **begin** preceded by one or more labels. The local variable *x* is set up for use in the subroutine BLOCK BEGIN, which forms the program block in the stack, name list and object program (even if the ALGOL program is a compound statement). The subroutine BLOCK BEGIN is entered by means of its second entry point to avoid any checking of the contents of the top of the stack.

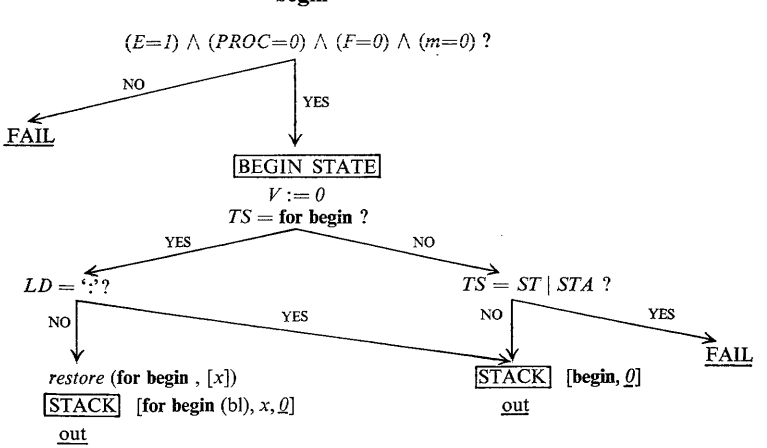
begin

This routine uses the subroutine BEGIN STATE to check whether the delimiter **begin** starts the first statement of a block.

If the **begin** is found to be the start of a controlled statement which is a labelled block or compound statement a marker (bl) is attached to the item **'for begin'** at the top of the stack (using the local variable *x*). The routine checks the validity of the use of **begin** from the current values of state variables. An extra check has to be made to ensure that **begin** does not follow the delimiter ':='.

AT ENTRY TO TRANSLATOR





real, integer, Boolean

These routines, after checking that the state variable *T* is zero, set it up with the appropriate bit pattern (using the local variable *y*). The subroutine DEC is then used to check whether the current delimiter indicates the start of the first declaration of a block.

switch

Similar to the above routines. The state variable *D* is also checked, and *TYPE* is set up to indicate a designational expression.

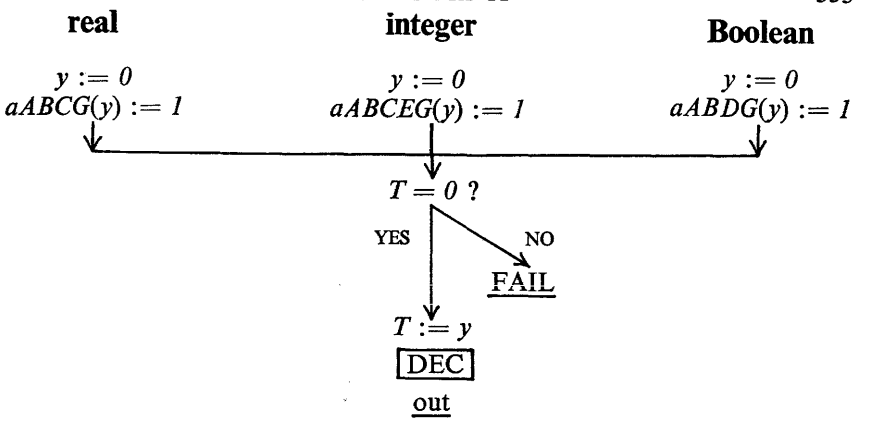
own

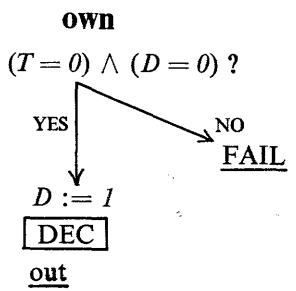
After checking the validity of the use of this delimiter the state variable *D* is set.

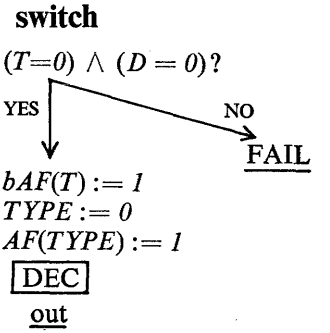
array

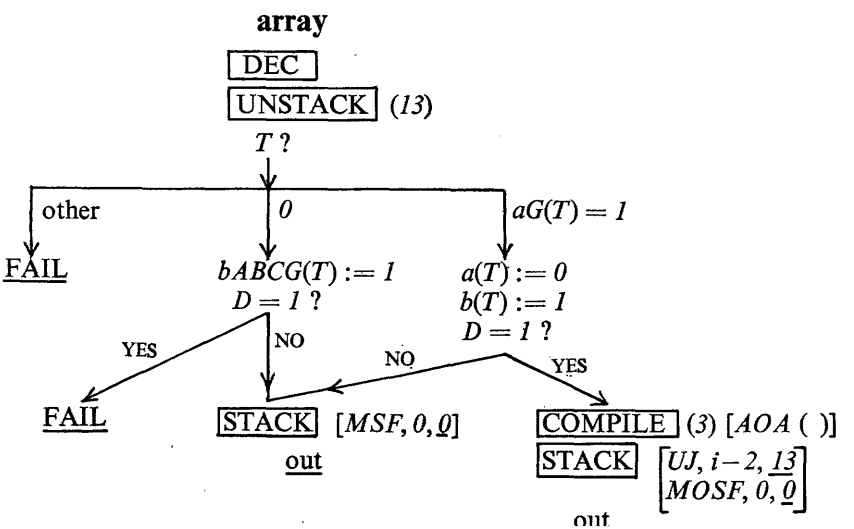
The subroutine DEC is used as described above. The subroutine UNSTACK completes the parameter of an *UJ* operation if this declaration follows a procedure or a switch declaration.

A check is made that the delimiter is not immediately preceded by **own**. The state variable *D* is inspected to decide whether to stack the operation *MSF* or to generate *AOA* and to stack the operations *UJ* and *MOSF*.


real
integer
Boolean
y := 0
aABCG(y) := 1
y := 0
aABCEG(y) := 1
y := 0
aABDG(y) := 1
T = 0 ?
YES
NO
FAIL
T := y
DEC
out


own
(T = 0) ∧ (D = 0) ?
YES
NO
FAIL
D := 1
DEC
out


switch
(T=0) ∧ (D = 0)?
YES
NO
FAIL
bAF(T) := 1
TYPE := 0
AF(TYPE) := 1
DEC
out


array
DEC
UNSTACK (13)
T ?
other
0
aG(T) = 1
FAIL
bABCG(T) := 1
D = 1 ?
a(T) := 0
b(T) := 1
D = 1 ?
YES
NO
NO
YES
FAIL
STACK [MSF, 0, 0]
out
COMPILE (3) [AOA ( )]
STACK
[UJ, i−2, 13
MOSF, 0, 0]
out

procedure

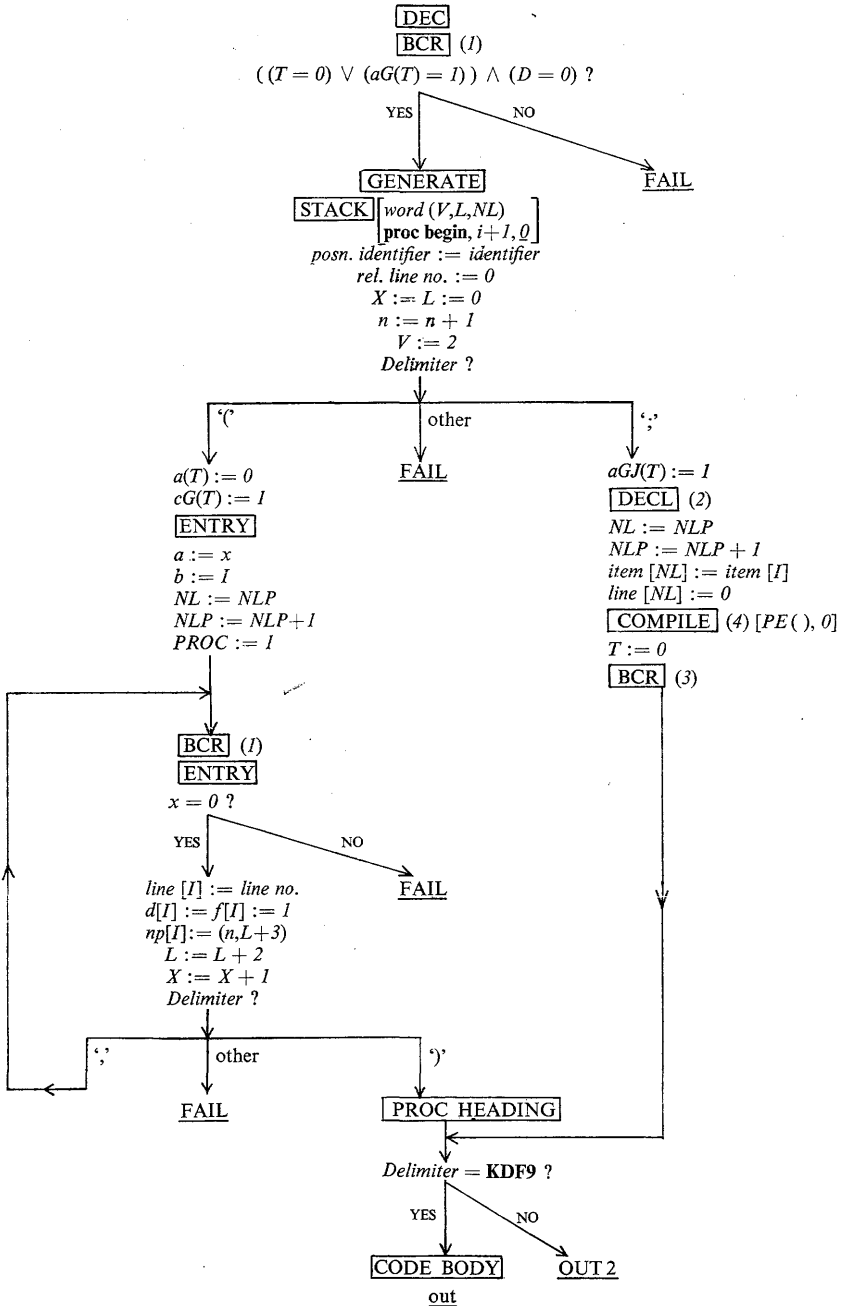
After using various state variables to check the validity of use of this delimiter, a procedure block is set up in the stack.

If the procedure has no parameters the procedure identifier is 'declared' using the subroutine DECL, and the operation *PE* is added to the object program. The next delimiter is then fetched in order to check whether the procedure body is in ALGOL or in User Code.

On the other hand, if the procedure has parameters the procedure identifier is not declared until the formal parameter part has been processed, when the number of parameters will be known (a check is made that an identifier does not appear twice in the list of formal parameters). The local variables *a* and *b* are used to preserve details of the name list entry for the procedure identifier, found using the subroutine ENTRY, until the identifier is 'declared' in the subroutine PROC HEADING, which also deals with the value and specification parts.

The subroutine CODE BODY processes a procedure body which is in User Code up to and including the semi-colon following the delimiter **ALGOL** (no flow diagram is given for this subroutine, since it is highly machine-dependent).

procedure


DEC
BCR (1)
((T = 0) ∨ (aG(T) = 1)) ∧ (D = 0) ?
YES
NO
FAIL
GENERATE
STACK
word (V,L,NL)
proc begin, i+1, 0
posn. identifier := identifier
rel. line no. := 0
X := L := 0
n := n + 1
V := 2
Delimiter ?
'('
other
';'
a(T) := 0
cG(T) := 1
ENTRY
a := x
b := I
NL := NLP
NLP := NLP+1
PROC := 1
FAIL
aGJ(T) := 1
DECL (2)
NL := NLP
NLP := NLP + 1
item [NL] := item [I]
line [NL] := 0
COMPILE (4) [PE( ), 0]
T := 0
BCR (3)
BCR (1)
ENTRY
x = 0 ?
YES
NO
FAIL
line [I] := line no.
d[I] := f[I] := 1
np[I] := (n,L+3)
L := L + 2
X := X + 1
Delimiter ?
','
other
')'
FAIL
PROC HEADING
Delimiter = KDF9 ?
YES
NO
CODE BODY
OUT 2
out

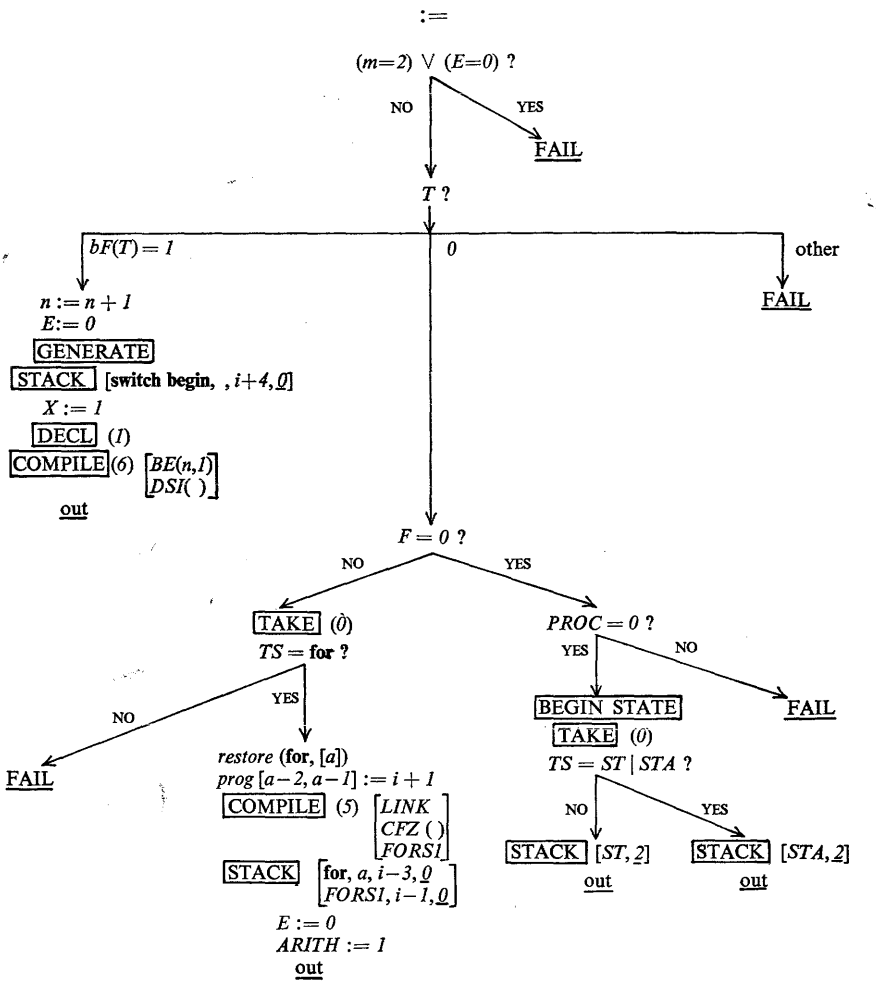
:=

This routine deals with the three possible uses of the delimiter ':='.

In the case of a switch declaration the switch block is set up in the stack and the object program, and the switch identifier is 'declared'.

If the delimiter is used in a for clause, operations are generated for the controlled variable, and the parameter of the *UJ* operation (generated as an incomplete operation at the delimiter **for**) is completed to the *CFZ* operation, using the local variable *a*. The state variables *E* and *ARITH* are set for the arithmetic expression which must follow the current delimiter.

The third possible use of the delimiter is in an assignment statement. A failure is given if the delimiter is used inside a procedure call. After using the subroutine TAKE to deal with the variable (simple or subscripted) which precedes this delimiter, the top of the stack is inspected to decide whether *ST* or *STA* is to be stacked.

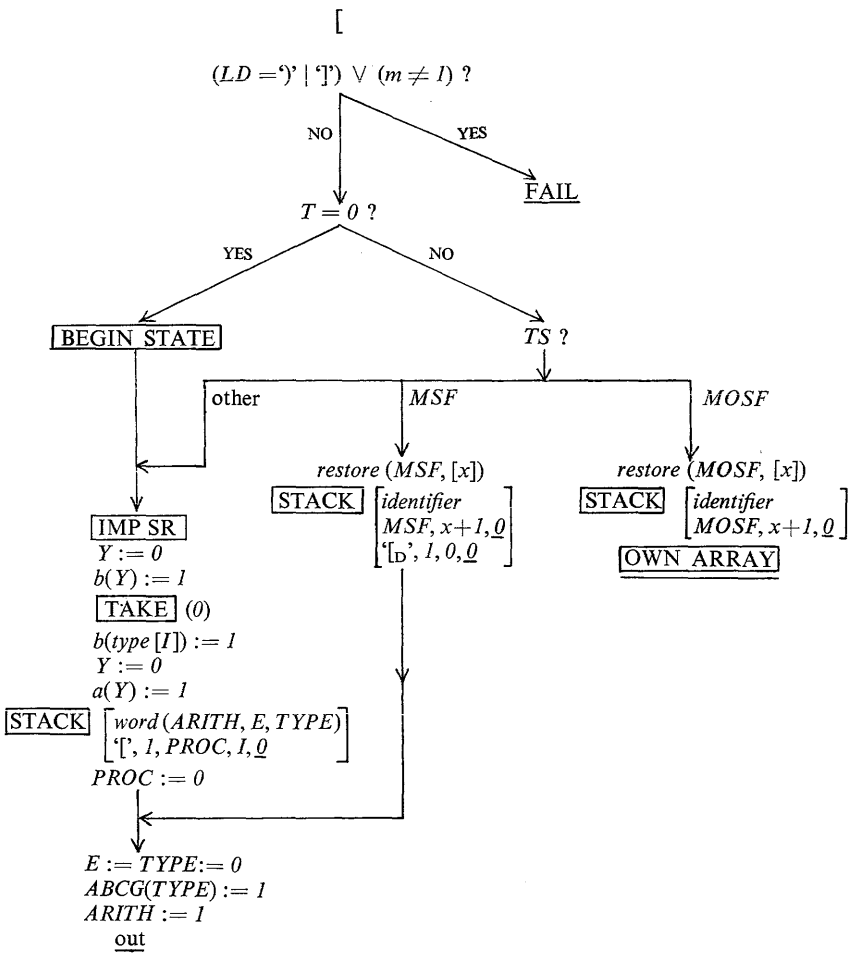

:=
(m=2) ∨ (E=0) ?
NO
YES
FAIL
T ?
bF(T) = 1
0
other
FAIL
n := n + 1
E := 0
GENERATE
STACK [switch begin, , i+4, 0]
X := 1
DECL (1)
COMPILE (6) [BE(n,1) DSI( )]
out
F = 0 ?
NO
YES
TAKE (0)
TS = for ?
YES
NO
FAIL
restore (for, [a])
prog [a−2, a−1] := i + 1
COMPILE (5) [LINK CFZ ( ) FORSI]
STACK [for, a, i−3, 0 FORSI, i−1, 0]
E := 0
ARITH := 1
out
PROC = 0 ?
YES
NO
BEGIN STATE
FAIL
TAKE (0)
TS = ST | STA ?
NO
YES
STACK [ST, 2]
out
STACK [STA, 2]
out

[

This routine deals with the use of this delimiter for a subscripted variable (or a switch designator) or as the start of a bound pair list in an array declaration.

In the first case the array or switch identifier is processed by the subroutine TAKE, after setting column *b* of the global variable *Y*, and the delimiter is stacked with certain state variables. Finally the state variables *E*, *TYPE* and *ARITH* are set up for the arithmetic subscript expression which must follow.

In the second case the array identifier is preserved in the stack beneath the *MSF* or *MOSF* item, using the local variable *x*. In the case of an own array the bound pair list is dealt with by the subroutine OWN ARRAY, which returns direct to the central loop of the Translator. For a non-own array the delimiter is stacked, with a marker D to identify its current use.

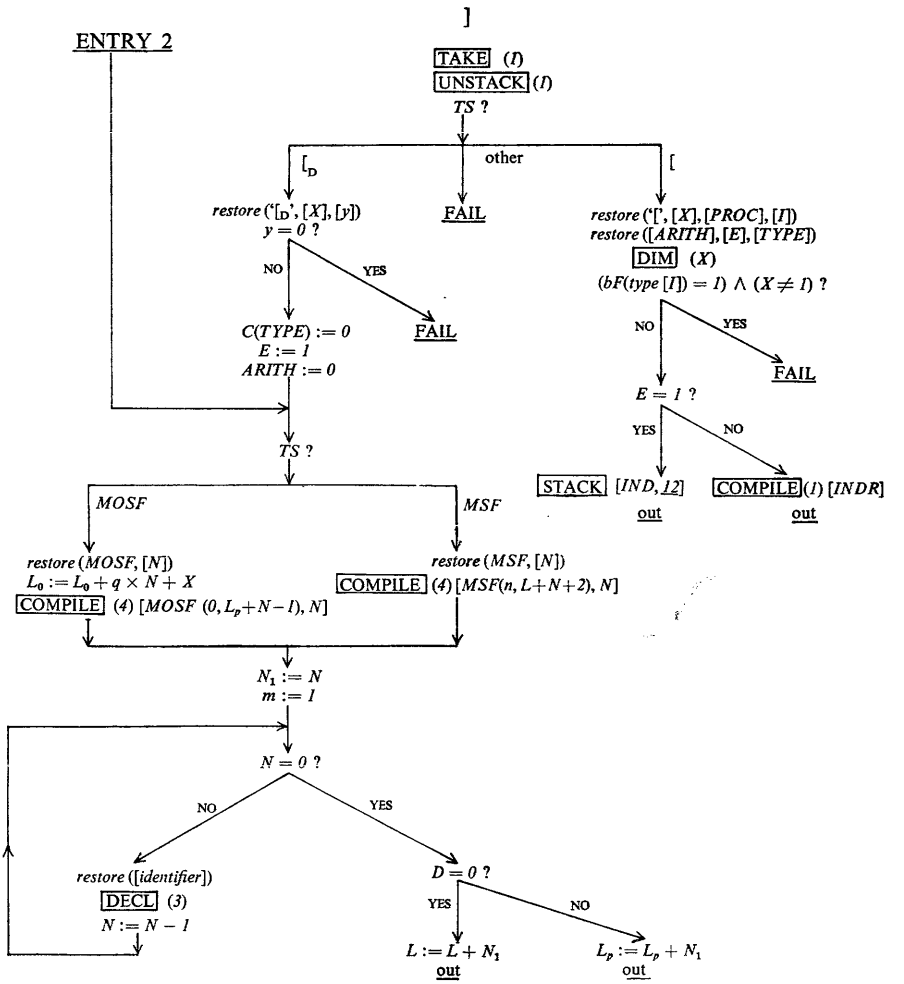

[
(LD =')' | ']') ∨ (m ≠ 1) ?
NO
YES
FAIL
T = 0 ?
YES
NO
BEGIN STATE
TS ?
other
MSF
MOSF
restore (MSF, [x])
STACK
identifier
MSF, x+1, 0
'[D', 1, 0, 0
restore (MOSF, [x])
STACK
identifier
MOSF, x+1, 0
OWN ARRAY
IMP SR
Y := 0
b(Y) := 1
TAKE (0)
b(type [I]) := 1
Y := 0
a(Y) := 1
STACK
word (ARITH, E, TYPE)
'[', 1, PROC, I, 0
PROC := 0
E := TYPE := 0
ABCG(TYPE) := 1
ARITH := 1
out

]

After dealing with the completion of the expression before this delimiter, the top of the stack indicates whether a subscript expression or an array bound pair list has been completed.

In the case of a subscript expression the number of dimensions is put into, or checked against, the *dim* column of the name list entry for the array or switch identifier. (A check is made that a switch designator contains only a single subscript expression.)

For an array declaration, the identifiers in the array list, which have been preserved in the stack beneath the *MSF* or *MOSF* items, are unstacked and 'declared', using the local variables N and N_1, and the information given in the state variables T and X. A check is made, using the local variable y, that the final subscript bound was preceded by the delimiter ':'. The subroutine OWN ARRAY, which processes the bound pair list of an own array declaration, reaches this routine via entry 2 in order to declare the identifiers in the array list. The variable q is set up by OWN ARRAY for use by this routine.

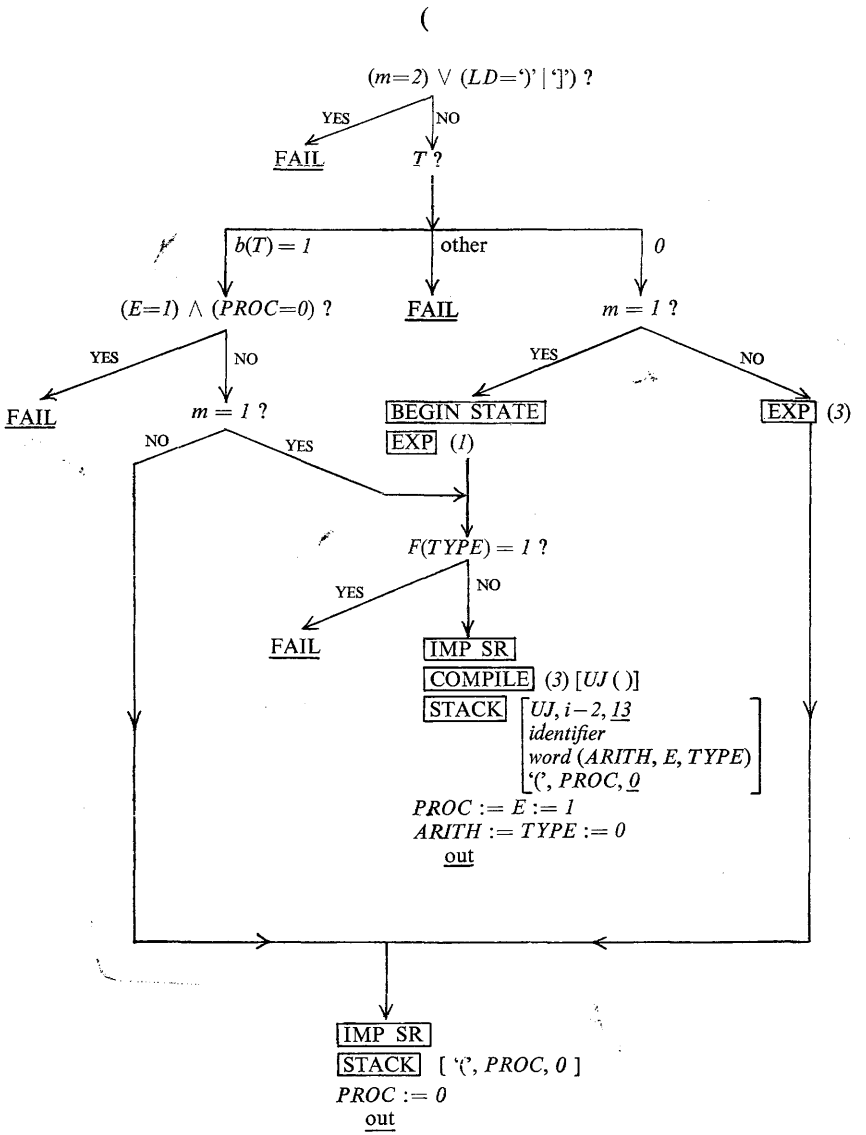

]
ENTRY 2
TAKE (I)
UNSTACK (I)
TS ?
[D
other
[
restore ('[D', [X], [y])
y = 0 ?
FAIL
restore ('[', [X], [PROC], [I])
restore ([ARITH], [E], [TYPE])
DIM (X)
(bF(type [I]) = I) ∧ (X ≠ I) ?
NO
YES
C(TYPE) := 0
E := 1
ARITH := 0
FAIL
E = 1 ?
YES
NO
STACK [IND, 12]
out
COMPILE (I) [INDR]
out
TS ?
MOSF
MSF
restore (MOSF, [N])
L0 := L0 + q × N + X
COMPILE (4) [MOSF (0, Lp+N−1), N]
restore (MSF, [N])
COMPILE (4) [MSF(n, L+N+2), N]
N1 := N
m := 1
N = 0 ?
NO
YES
restore ([identifier])
DECL (3)
N := N − 1
D = 0 ?
YES
NO
L := L + N1
out
Lp := Lp + N1
out

(

The state variable *m* indicates whether this delimiter is being used as an expression bracket or in a procedure call.

The expression bracket can be used in a statement or in an array or switch declaration. In the former case the subroutine EXP is used to change the state variable *E* from statement level to expression level if necessary. In the latter case a failure is indicated if *E* is set to statement level, unless *PROC* shows that an actual parameter is being processed. In either case the subroutine IMP SR is used to check whether this delimiter is the start of an actual parameter expression. Finally the delimiter is stacked.

In the case of a procedure call bracket, a check is made that the state variable *TYPE* is not set to 'designational'. An incomplete operation *UJ* is generated, to be completed later to the 'Call Function' operation, and four items are stacked. These are (*i*) a reminder to complete the *UJ* operation, (*ii*) the procedure identifier, (*iii*) values of the state variables *ARITH*, *E* and *TYPE*, and (*iv*) the current delimiter with the value of the state variable *PROC*. Finally these state variables are set to deal with the actual parameters of this procedure call.

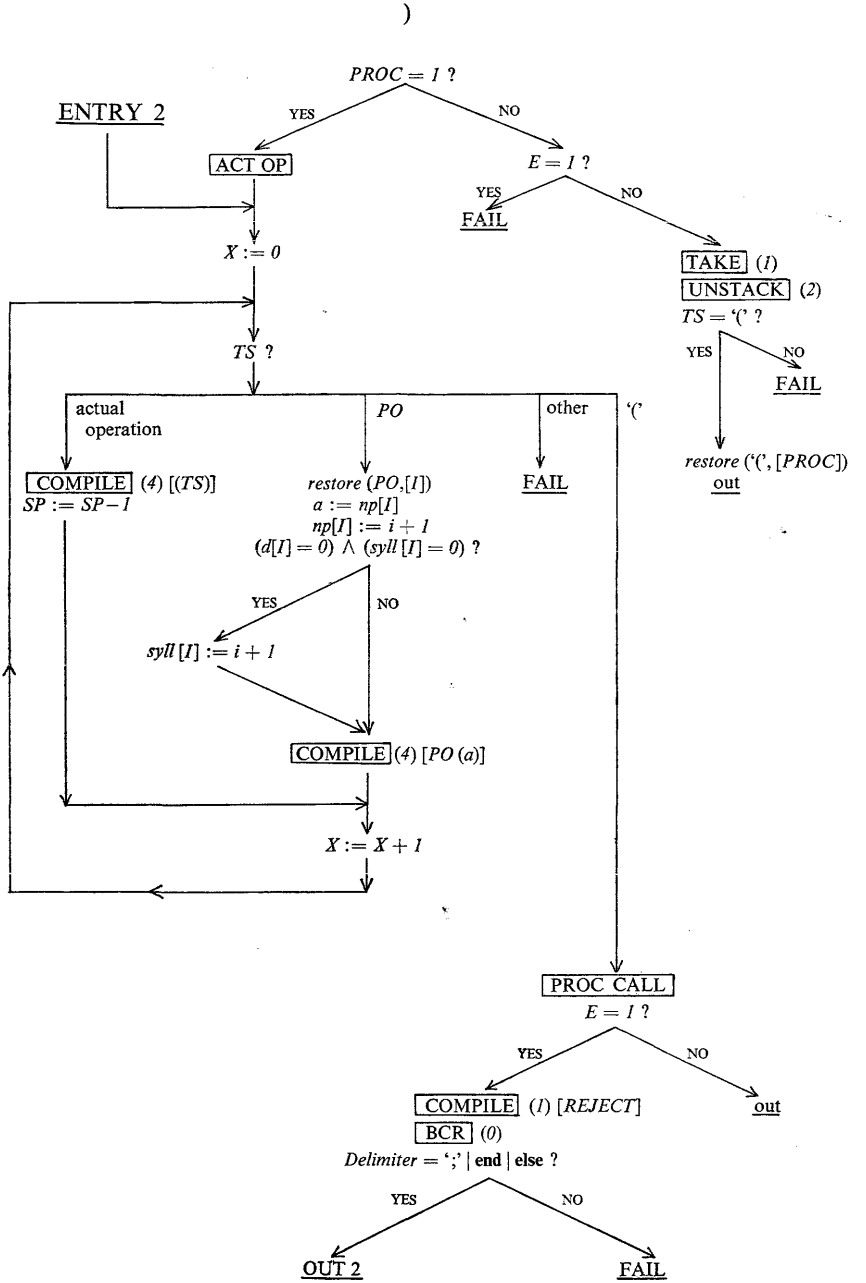

(
(m=2) ∨ (LD=')' | ']') ?
YES
NO
FAIL
T ?
b(T) = 1
other
0
(E=1) ∧ (PROC=0) ?
FAIL
m = 1 ?
YES
NO
YES
NO
FAIL
m = 1 ?
BEGIN STATE
EXP (1)
EXP (3)
NO
YES
F(TYPE) = 1 ?
YES
NO
FAIL
IMP SR
COMPILE (3) [UJ ( )]
STACK [UJ, i−2, 13
identifier
word (ARITH, E, TYPE)
'(', PROC, 0]
PROC := E := 1
ARITH := TYPE := 0
out
IMP SR
STACK ['(', PROC, 0]
PROC := 0
out

)

The state variable *PROC* indicates whether this delimiter is an expression bracket or a procedure call bracket.

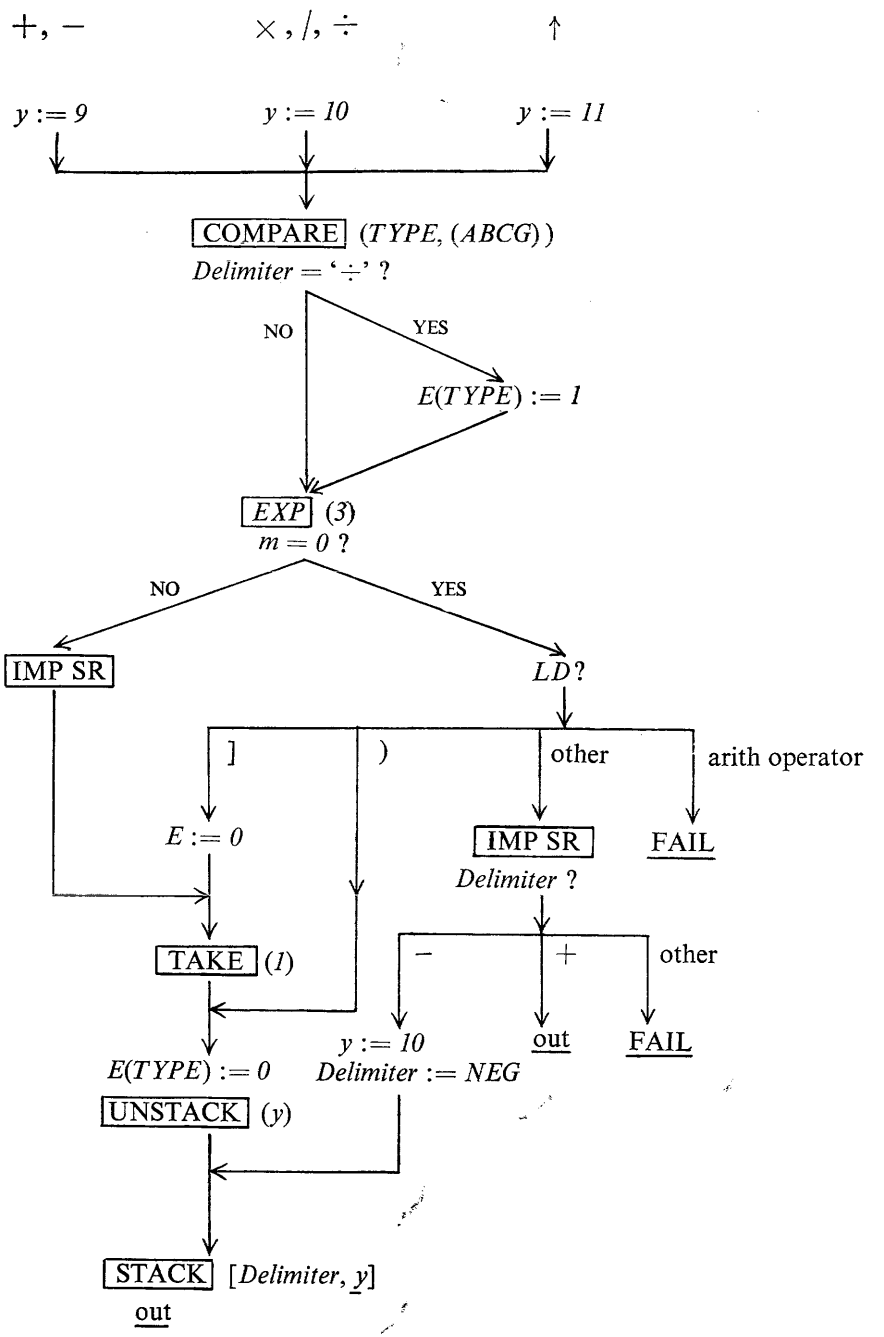
In the case of an expression bracket the translation of the preceding expression is completed by the subroutines TAKE and UNSTACK. The top of the stack should then contain the corresponding opening bracket, which is unstacked and discarded, after restoring the value of the state variable *PROC* stacked with it.

In the case of a procedure call bracket the translation of the preceding actual parameter is dealt with by the subroutine ACT OP. The 'actual operations' are unstacked into the object program, whereupon the stack should contain the corresponding opening bracket. During this unstacking the chaining of any skeleton 'actual operations' is dealt with, using the local variable *a*. The subroutine PROC CALL processes the procedure identifier and generates the appropriate 'Call Function' operation. Finally, if the state variable *E* is set to statement level the operation *REJECT* is generated and the next delimiter is checked to ensure that it is an 'end of statement' delimiter.


)
PROC = 1 ?
ENTRY 2
YES
NO
ACT OP
E = 1 ?
YES
NO
FAIL
X := 0
TAKE (1)
UNSTACK (2)
TS = '(' ?
YES
NO
FAIL
TS ?
actual operation
PO
other
'('
restore ('(', [PROC])
out
COMPILE (4) [(TS)]
SP := SP − 1
restore (PO,[I])
a := np[I]
np[I] := i + 1
(d[I] = 0) ∧ (syll[I] = 0) ?
FAIL
YES
NO
syll[I] := i + 1
COMPILE (4) [PO (a)]
X := X + 1
PROC CALL
E = 1 ?
YES
NO
COMPILE (1) [REJECT]
out
BCR (0)
Delimiter = ';' | end | else ?
YES
NO
OUT 2
FAIL

$+, -, \times, /, \div, \uparrow$

After setting up the local variable y with the appropriate stack priority for the current delimiter the validity of its use is checked by the subroutines COMPARE and EXP. If this delimiter is not preceded by an identifier or a constant the last delimiter is checked, and if the current delimiter proves to be a unary operator it is ignored in the case of '+' (after using IMP SR to check whether an implicit subroutine needs to be generated) and changed to *NEG* in the case of '−'. The subroutine UNSTACK is used, with the parameter y, to unstack any operators, with priorities greater than or equal to that of the current delimiter, into the object program. Finally the current delimiter is stacked.


+, −
×, /, ÷
↑
y := 9
y := 10
y := 11
COMPARE (TYPE, (ABCG))
Delimiter = '÷' ?
NO
YES
E(TYPE) := 1
EXP (3)
m = 0 ?
NO
YES
IMP SR
LD?
]
)
other
arith operator
E := 0
IMP SR
FAIL
Delimiter ?
TAKE (1)
−
+
other
y := 10
Delimiter := NEG
out
FAIL
E(TYPE) := 0
UNSTACK (y)
STACK [Delimiter, y]
out

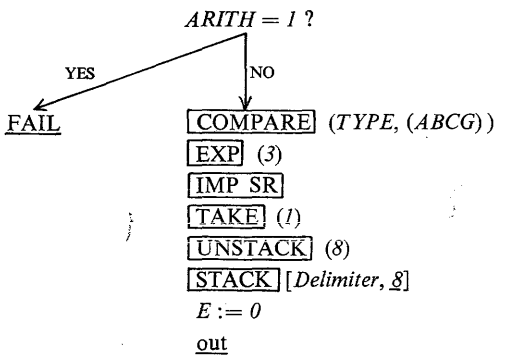
$<, \leqslant, =, \geqslant, >, \neq$

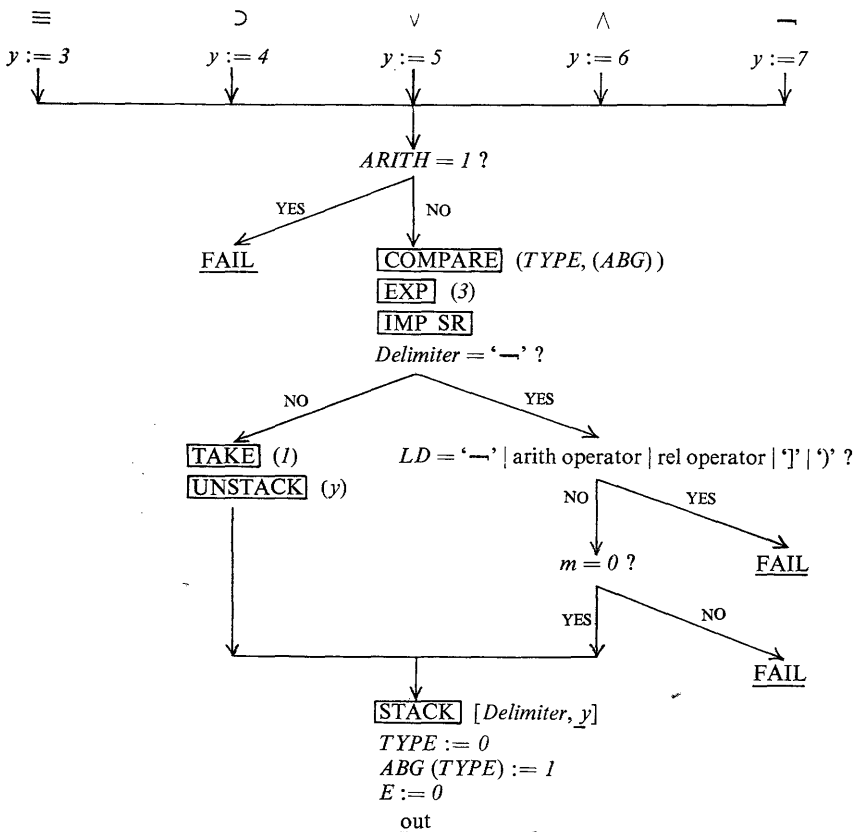
After testing the state variable *ARITH*, and using the subroutine COMPARE to check the validity of use of the current delimiter the subroutine EXP is used to change from statement to expression level, if necessary. The subroutine IMP SR is used to check whether it is necessary to set up an implicit subroutine, and then TAKE is used to process any identifier or constant preceding the delimiter. Finally UNSTACK is used to unstack any operators with priorities greater than or equal to the priority of the current delimiter, which is then stacked.

$\equiv, \supset, \vee, \wedge, \neg$

Except in the case of the delimiter '$\neg$' the action of these routines is similar to those for the relational operators (the local variable y being used to give the priority of the current delimiter). However extra checks as to the validity of use of '$\neg$' are incorporated in its routine. Finally the state variable *TYPE* is set to indicate 'algebraic' for the expression following.

$<, \leqslant, =, \geqslant, >, \neq$



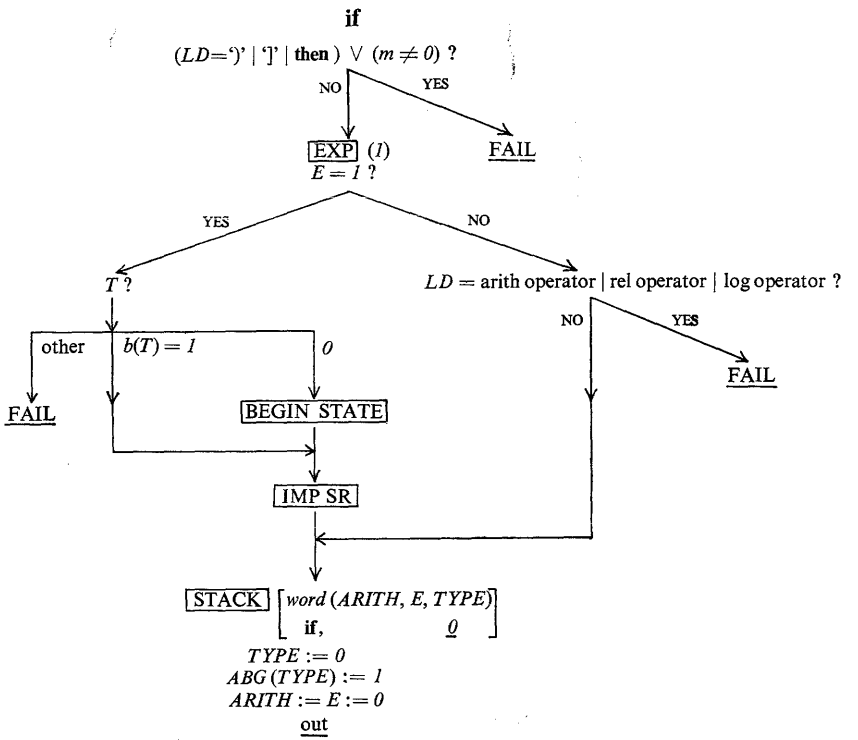


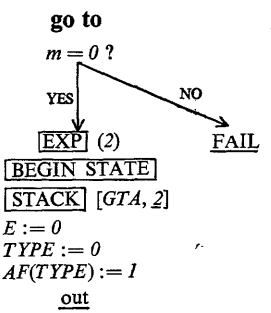
if

After checking the validity of use of the delimiter, the subroutine EXP is used to change the state variable *E* to expression level if the current delimiter follows the delimiter ':='. The state variable *E* then differentiates between the use of the current delimiter in a conditional expression or a conditional statement. (The special case of a conditional expression as an actual parameter is recognized by the subroutine IMP SR.) In the former case further checks are made on the use of the delimiter. Finally the delimiter is stacked, together with the values of *ARITH*, *E* and *TYPE*, before these state variables are set up for the algebraic expression following.

go to

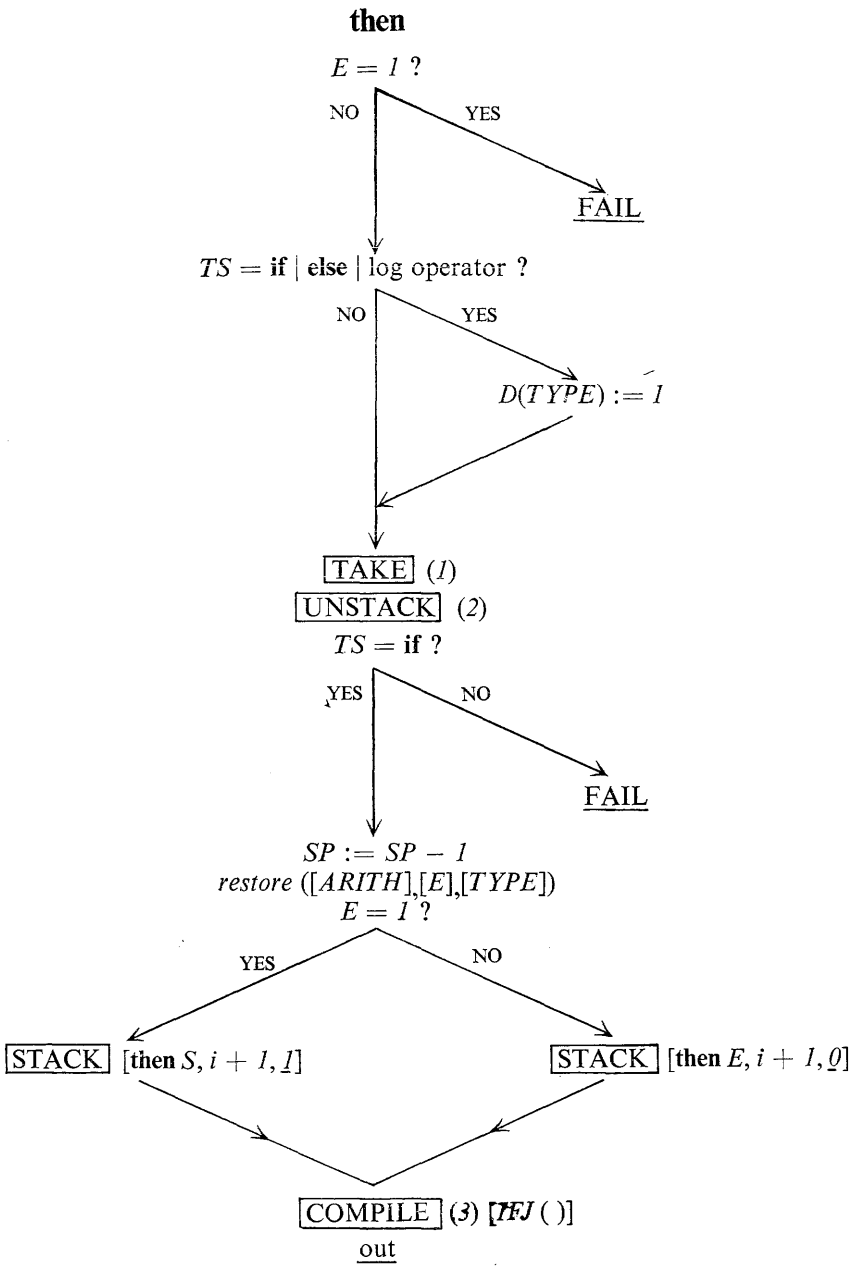
After checking that this delimiter is not preceded by an identifier or constant, EXP is used to check that the delimiter occurs at the start of a statement, and BEGIN STATE to check whether it is the first statement of a block or a compound statement. The delimiter is then stacked and the state variables *E* and *TYPE* are set up for the designational expression following.


if
(LD=')' | ']' | then) ∨ (m ≠ 0) ?
NO
YES
EXP (1)
E = 1 ?
FAIL
YES
NO
T ?
LD = arith operator | rel operator | log operator ?
NO
YES
other
b(T) = 1
0
FAIL
FAIL
BEGIN STATE
IMP SR
STACK
word (ARITH, E, TYPE)
if,
0
TYPE := 0
ABG (TYPE) := 1
ARITH := E := 0
out


go to
m = 0 ?
YES
NO
EXP (2)
FAIL
BEGIN STATE
STACK [GTA, 2]
E := 0
TYPE := 0
AF(TYPE) := 1
out

then

This routine completes the object program for the algebraic expression following the corresponding delimiter **if**. It then unstacks the item **if** and restores the values of the state variables *ARITH*, *E* and *TYPE* stacked with it. The state variable *E* is then used to decide whether the delimiter should be stacked as '**then** *S*' or '**then** *E*'. An incomplete *IFJ* operation is generated and a reminder to complete it at the delimiter **else** is stacked with the item '**then** *S*' or '**then** *E*'.


then
E = 1 ?
NO
YES
FAIL
TS = if | else | log operator ?
NO
YES
D(TYPE) := 1
TAKE (1)
UNSTACK (2)
TS = if ?
YES
NO
FAIL
SP := SP − 1
restore ([ARITH],[E],[TYPE])
E = 1 ?
YES
NO
STACK [then S, i + 1, 1]
STACK [then E, i + 1, 0]
COMPILE (3) [JEJ ( )]
out

N

else

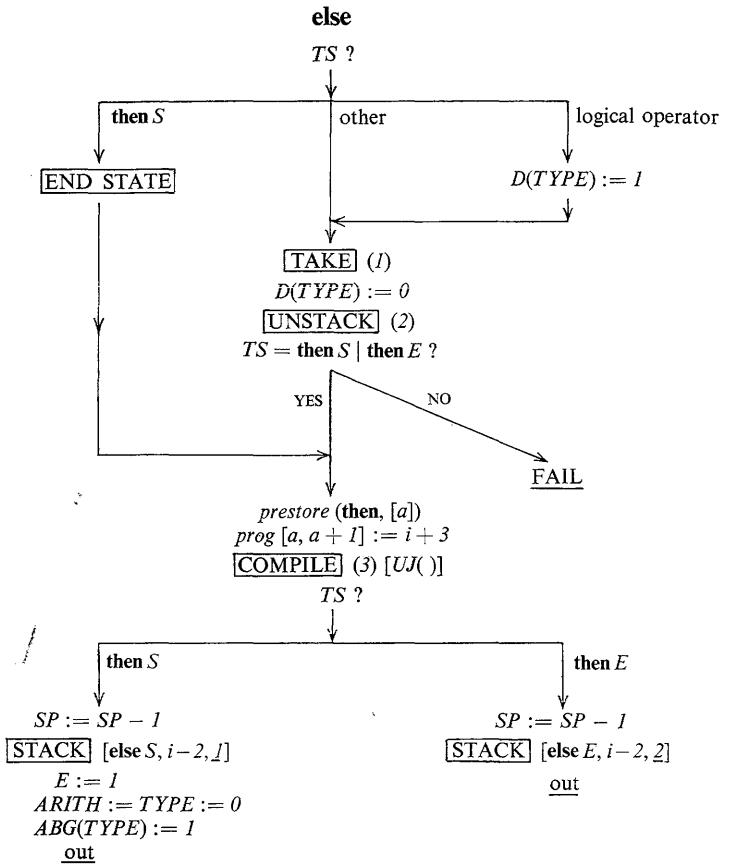
This routine completes the object program of the statement or expression preceding the current delimiter, and then checks that the top of the stack contains '**then** *S*' or '**then** *E*'. (The special case of a procedure identifier with zero parameters between the **then** and **else**, shown by the occurrence of '**then** *S*' at the top of the stack when this routine is entered, is dealt with by the subroutine END STATE.) The incomplete *IFJ* operation is completed, using the local variable *a*, and an incomplete *UJ* operation is added to the program. Finally '**else** *S*' or '**else** *E*' is stacked, with an obligation to complete the *UJ* operation when the end of the statement or expression following the current delimiter is reached.

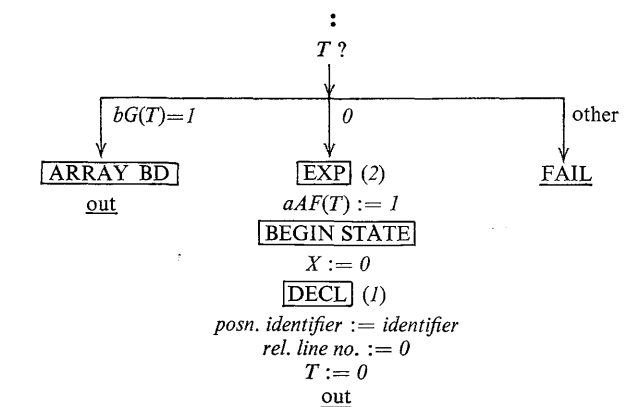
:

The state variable *T* is used to decide whether the current delimiter follows a label or a lower bound in an array declaration.

In the former case EXP is used to check the validity of the use of the label, which is then declared using the subroutine DECL, after BEGIN STATE has checked whether the label precedes the first statement of a block or compound statement.

In the latter case the current ALGOL section is processed by the subroutine ARRAY BD.


else
TS ?
then S
other
logical operator
END STATE
D(TYPE) := 1
TAKE (1)
D(TYPE) := 0
UNSTACK (2)
TS = then S | then E ?
YES
NO
FAIL
prestore (then, [a])
prog [a, a + 1] := i + 3
COMPILE (3) [UJ( )]
TS ?
then S
then E
SP := SP − 1
STACK [else S, i−2, 1]
E := 1
ARITH := TYPE := 0
ABG(TYPE) := 1
out
SP := SP − 1
STACK [else E, i−2, 2]
out


:
T ?
bG(T)=1
0
other
ARRAY BD
out
EXP (2)
aAF(T) := 1
BEGIN STATE
X := 0
DECL (1)
posn. identifier := identifier
rel. line no. := 0
T := 0
out
FAIL

step, until, while

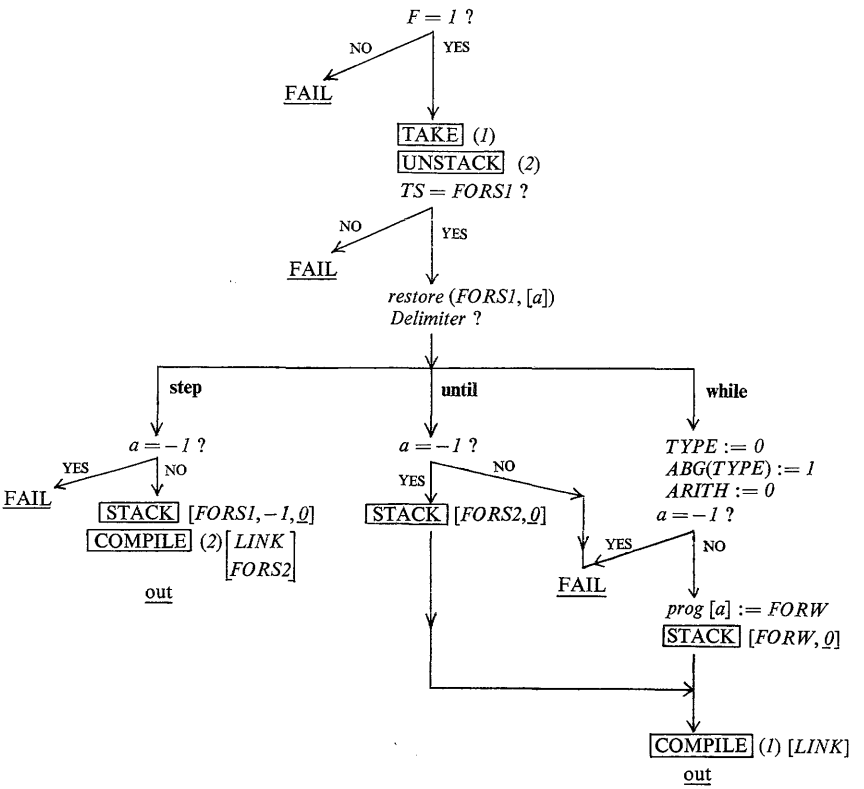
After checking that these delimiters are used only in a for clause the arithmetic expression preceding the current delimiter is dealt with by the subroutines TAKE and UNSTACK. The top of the stack should then be *FORS1*. The local variable *a* is set up with the address stacked with the item *FORS1*. This is used to change the operation *FORS1* in the object program to *FORW* if necessary.

In the case of the delimiter **step**, *FORS1* is restacked with '-1' as the 'address' to indicate that the delimiter **until** is required. In the case of the delimiter **while** the state variables *TYPE* and *ARITH* are set up for the algebraic expression following.

for

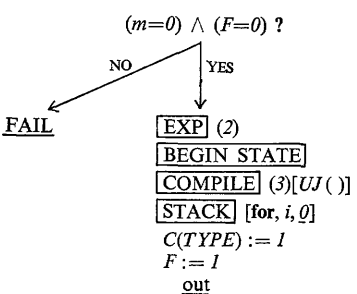
After using the state variables *m* and *F* and the subroutine EXP to check that the delimiter is used at the start of a statement, the subroutine BEGIN STATE is used to check whether this is the first statement of a block or compound statement. An incomplete *UJ* operation is generated, and the item **for** is stacked with a reminder to complete the *UJ* operation at the delimiter '$:=$'. The state variable *TYPE* is set to 'arithmetic' for the simple or subscripted controlled variable following the current delimiter, and the for clause marker *F* is set.

step, until, while



---

for



do

The subroutine FOR ',' completes the object program for the preceding for list element, and a block is set up for the controlled statement. The local variable *x* is used to complete the parameter of the *CFZ* operation generated at the delimiter ':=', and the local variable *a* is used to contain the address of the first of the sequence of one or more operations generated from the controlled variable. The for clause marker is cleared, the state variable *L* set to four to allow for the two accumulators required at run time, and the state variables *E*, *ARITH* and *TYPE* are set up as for the start of a block.

'

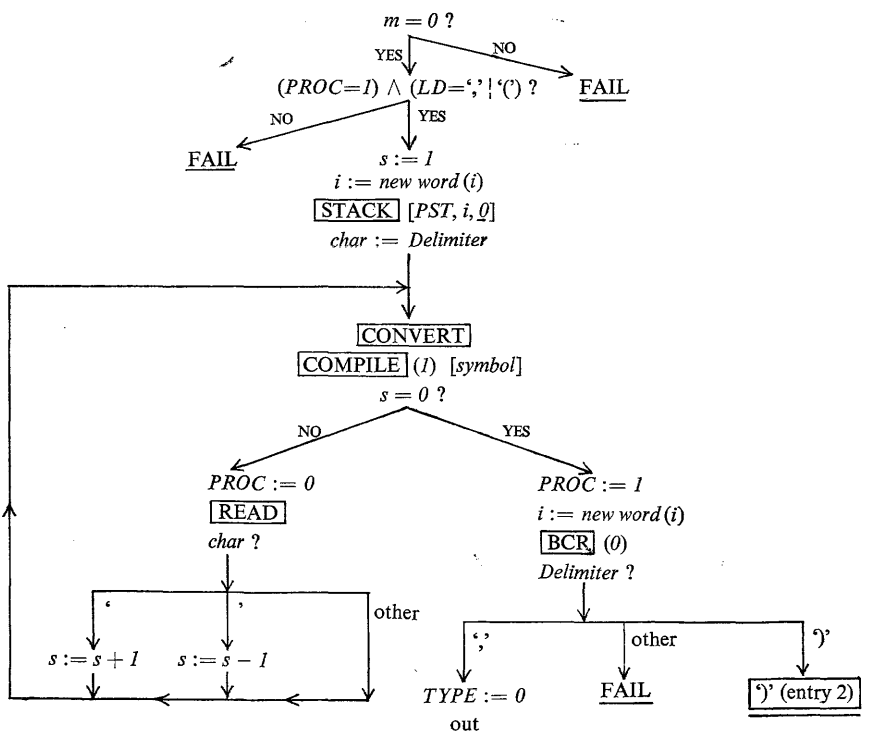
After checking that this delimiter appears as the first symbol of an actual parameter the local variable *s* is set to one, and the object program counter is advanced by a sufficient number of syllables to allow the first character of the string to be stored in the first syllable of a new word, using the function designator *new word*. The 'actual operation' *PST* is stacked, and the symbols of the string are changed into the standard 8-bit representation, using the subroutine CONVERT, and added to the object program. (An 8-bit representation of ALGOL basic symbols which is common to both KDF9 ALGOL compilers, is used, so that code procedures using strings can be used with either compiler).

The characters of the string are fetched using the subroutine READ, the local variable *s* being used to allow for nested string quotes. The state variable *PROC* is cleared whilst the characters are being fetched in order that the symbols in a parameter delimiter are not ignored inside the subroutine READ.

After the final string quote has been converted and added to the object program *PROC* is reset, and the object program counter is increased in order to point at the next word. Finally the next delimiter is fetched to ensure that the closing string quote is the last symbol of the actual parameter.

do

FOR ','
n := *n* + *1*
restore (**for**, [a], [*x*])
COMPILE (*9*) [*LINK*, *FSE* ( ), *FBE* ( ), (*a*)]
prog [*x*, *x*+*1*] := *i*−*5*
STACK [*word* (*V*, *L*, *NL*), **for begin**, *i*−*4*, *0*]
E := *1*
L := *4*
ARITH := *F* := *0*
TYPE := *0*
ABG(*TYPE*) := *1*
NL := *NLP*
NLP := *NLP* + *1*
item [*NL*] := *0*
out



,

This routine deals with various uses of the delimiter ','.

(*i*) If the state variable *PROC* is set the subroutine ACT OP is used to process the current ALGOL section.

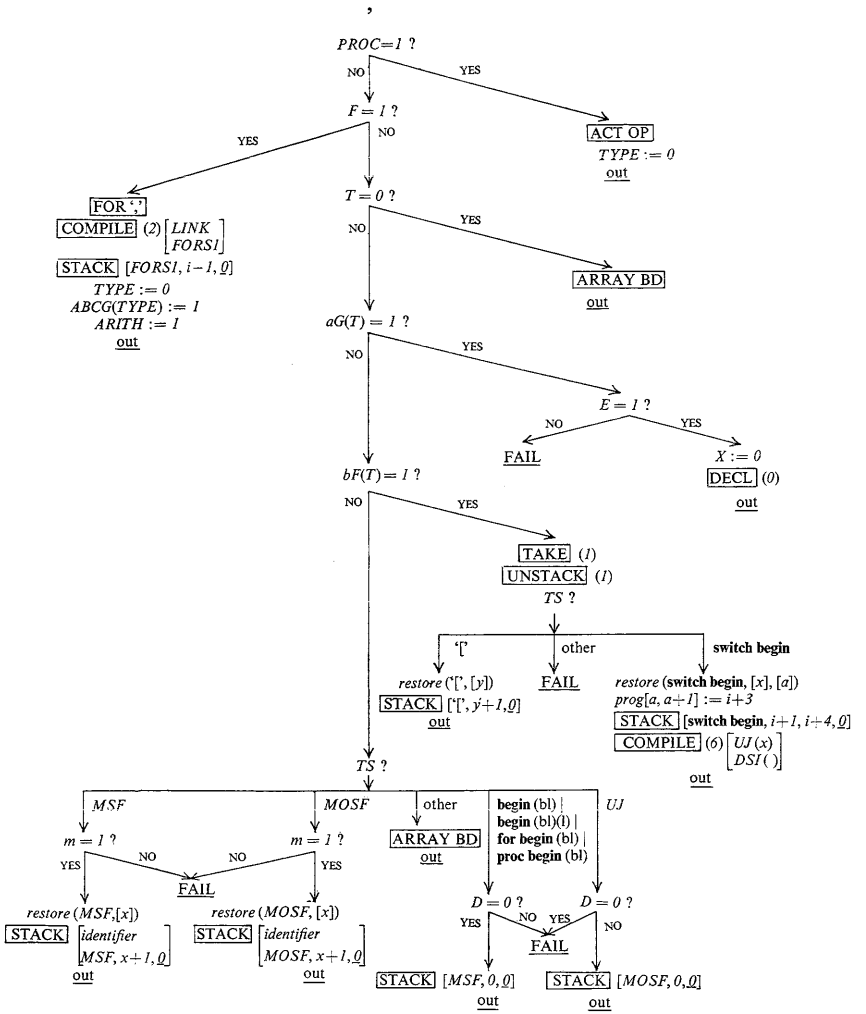
(*ii*) If the for clause marker is set, the subroutine FOR',' is used to distinguish between the use of comma between for list elements or between subscript expressions of a variable used in a for list element. In the former case the operation *LINK* is generated to complete the preceding for list element, *FORS1* is generated and stacked, and the state variables *TYPE* and *ARITH* set up for the for list element following the current delimiter. In the latter case the current ALGOL section is dealt with completely by the subroutine FOR',' which returns directly to the central loop of the Translator.

(*iii*) If the state variable *T* is zero the current delimiter is being used between subscript expressions, and is processed by the subroutine ARRAY BD.

(*iv*) If *T* has been set by a scalar declaration the comma is being used between identifiers, and the subroutine DECL is used to 'declare' the preceding identifier.

(*v*) If *T* shows that a switch element is being translated the subroutines TAKE and UNSTACK are used to complete the processing of the expression preceding the current delimiter, whereupon the top of the stack indicates whether the comma is being used between subscript expressions of a variable or between elements of the switch list. In the former case the dimension counter stacked with the item '[' is increased by one, using the local variable *y*. In the latter case the local variable *a* is used to complete the *DSI* operation generated for the preceding switch list element, and the corresponding address stacked with the item **'switch begin'** is updated to point to the incomplete *DSI* operation generated for the next switch list element. An incomplete *UJ* operation is generated and is chained to the previous *UJ* operation, if any, using the local variable *x*.

(*vi*) Finally, the top of the stack is used to differentiate between various uses of the comma in an array declaration. If the top of the stack contains *MSF* or *MOSF* the comma is being used between identifiers in the array list, and after checking that the current ALGOL section contains an identifier, this identifier is stacked under the item *MSF* or *MOSF*, whose counter is increased by one (using the local variable *x*). If the top of the stack contains *UJ* or a **begin** item the comma is being used between array segments, and after checking the own marker *D* the appropriate item (*MSF* or *MOSF*) is stacked. Otherwise the comma is being used between subscript expressions and the subroutine ARRAY BD is used to process the current ALGOL section.


,
PROC=1 ?
NO
YES
ACT OP
TYPE := 0
out
F = 1 ?
YES
NO
FOR ','
COMPILE (2) [LINK / FORS1]
STACK [FORS1, i−1, 0]
TYPE := 0
ABCG(TYPE) := 1
ARITH := 1
out
T = 0 ?
NO
YES
ARRAY BD
out
aG(T) = 1 ?
NO
YES
E = 1 ?
NO
YES
FAIL
X := 0
DECL (0)
out
bF(T) = 1 ?
NO
YES
TAKE (1)
UNSTACK (1)
TS ?
'['
other
switch begin
restore ('[', [y])
STACK ['[', y+1, 0]
out
FAIL
restore (switch begin, [x], [a])
prog[a, a+1] := i+3
STACK [switch begin, i+1, i+4, 0]
COMPILE (6) [UJ (x) / DSI ( )]
out
TS ?
MSF
MOSF
other
UJ
m = 1 ?
m = 1 ?
NO
NO
YES
YES
FAIL
ARRAY BD
out
begin (bl) |
begin (bl)(l) |
for begin (bl) |
proc begin (bl)
restore (MSF, [x])
STACK [identifier / MSF, x+1, 0]
out
restore (MOSF, [x])
STACK [identifier / MOSF, x+1, 0]
out
D = 0 ?
D = 0 ?
YES
NO
YES
NO
FAIL
STACK [MSF, 0, 0]
out
STACK [MOSF, 0, 0]
out

N*

;

This routine deals with various uses of the delimiter ';'.

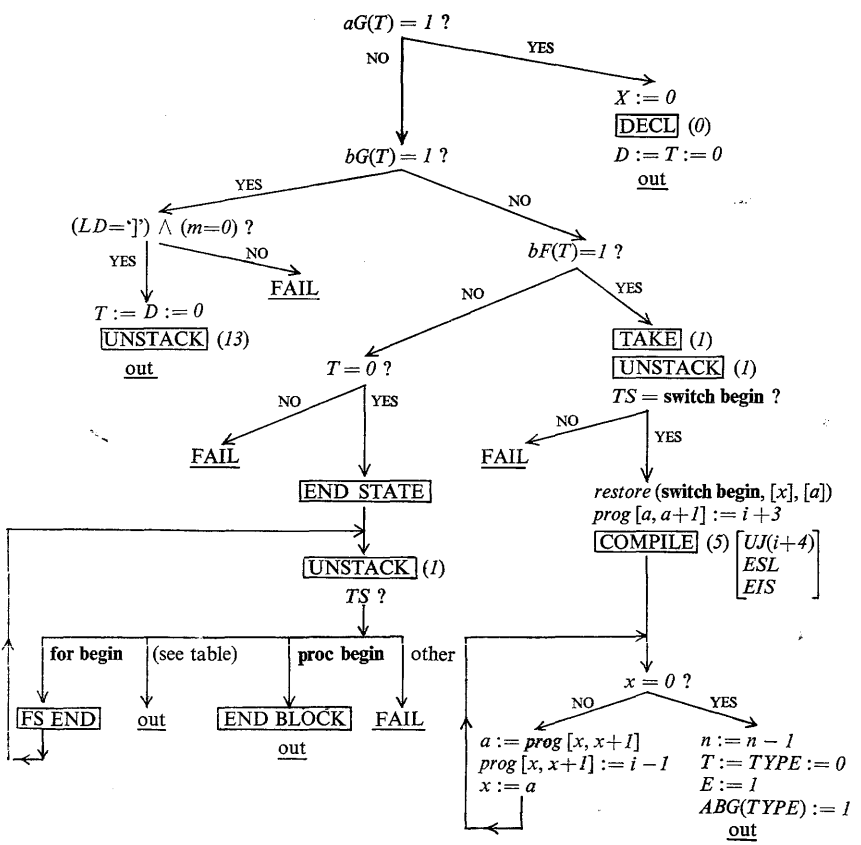
(*i*) If the state variable *T* is set to indicate a scalar declaration the subroutine DECL is used to declare the current identifier, and the state variables *T* and *D* are cleared.

(*ii*) If *T* indicates an array declaration a check is made that the current delimiter is immediately preceded by ']', the state variables *T* and *D* are cleared, and UNSTACK is used to complete the *AOA* operation generated at the start of an own array declaration.

(*iii*) If *T* indicates a switch declaration the subroutines TAKE and UNSTACK are used to complete the processing of the final designational expression, whereupon the top of the stack should be the item **'switch begin'**. The addresses stored with this item are preserved in the local variables *x* and *a*; these are then used to complete the last *DSI* operation to the operation *ESL* and to unchain and complete the *UJ* operations to the operation *EIS*. Finally, the block level *n* is reduced by one and the state variables *T*, *TYPE* and *E* are set for the start of the next declaration or statement.

(*iv*) Finally, if *T* is clear, the delimiter is being used to complete a statement. The subroutines END STATE and UNSTACK are used to complete the processing of the constituents of the statement. The top of the stack is then inspected to determine whether the subroutine FS END must be used to complete one or more for statements, or whether this delimiter completes a procedure body, in which case the subroutine END BLOCK is used. Otherwise the top of the stack should contain one or other of the items in the following table

begin
begin (bl)
begin (l)
begin (bl)(l)
for begin (bl)
proc begin (bl)


aG(T) = 1 ?
NO
YES
X := 0
DECL (0)
D := T := 0
out
bG(T) = 1 ?
YES
NO
(LD='])' ∧ (m=0) ?
YES
NO
FAIL
T := D := 0
UNSTACK (13)
out
bF(T)=1 ?
NO
YES
TAKE (1)
UNSTACK (1)
TS = switch begin ?
T = 0 ?
NO
YES
FAIL
END STATE
NO
YES
FAIL
restore (switch begin, [x], [a])
prog [a, a+1] := i +3
COMPILE (5)
UJ(i+4)
ESL
EIS
UNSTACK (1)
TS ?
for begin
(see table)
proc begin
other
FS END
out
END BLOCK
out
FAIL
x = 0 ?
NO
YES
a := prog [x, x+1]
prog [x, x+1] := i -1
x := a
n := n - 1
T := TYPE := 0
E := 1
ABG(TYPE) := 1
out

end

A check is made that this delimiter does not complete a declaration and then the subroutines END STATE and UNSTACK are used to complete the processing of the constituents of the preceding statement. If the top item of the stack is **'for begin'**, FS END is used to complete one or more for statements. The second entry to this routine (used by SCAN) is at the point where the **begin** item at the top of the stack is inspected to determine the course of action to be taken.

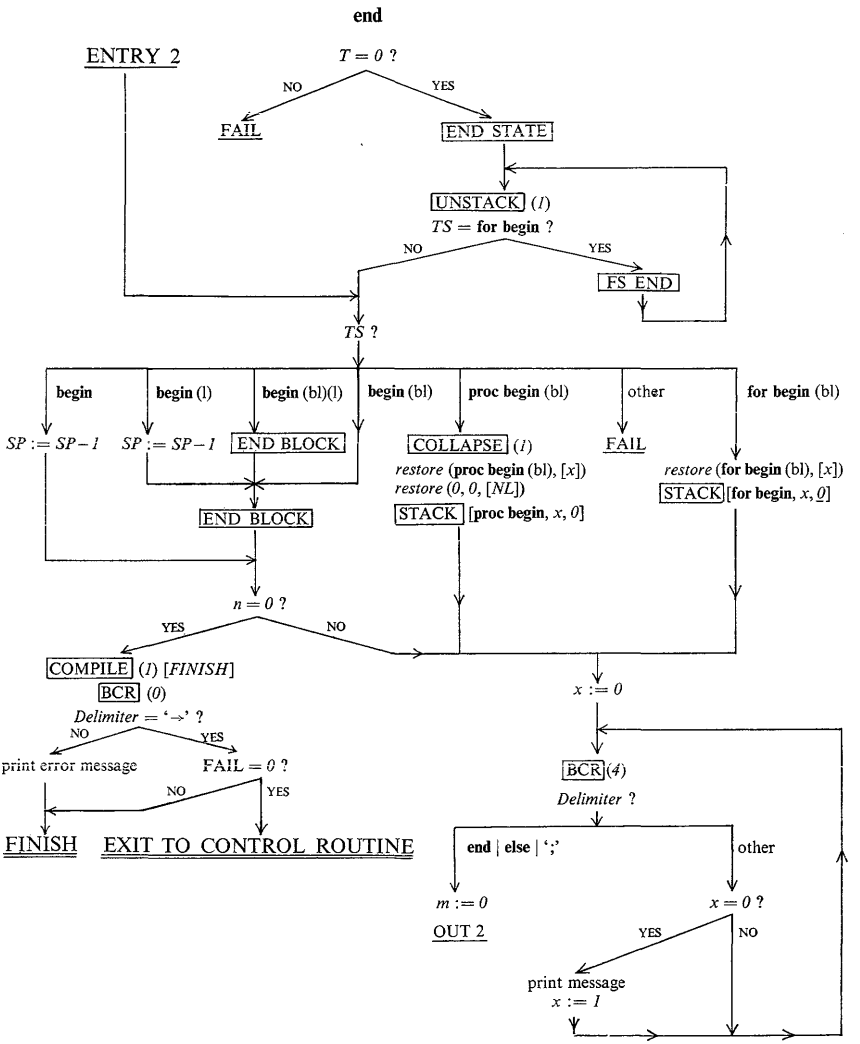
(*i*) If the top of the stack is **begin**, indicating a compound statement, it is discarded, then *n* is inspected to decide whether the current delimiter marks the end of the program, in which case the operation *FINISH* is generated; the next delimiter is checked to be the end message symbol '→', and if *FAIL* indicates that no errors have been detected, the Control Routine is entered. Otherwise the local variable *x* is used, whilst allowing for a possible comment after the current delimiter, to print a warning message if the comment contains a delimiter.

(*ii*) If the top of the stack contains **'begin** (l)' or **'begin** (bl)' the subroutine END BLOCK is used (after unstacking the **'begin** (l)' if necessary). The block level *n* is then inspected as in (*i*) above.

(*iii*) If the top of the stack is **'begin** (bl)(l)' the subroutine END BLOCK is used twice to complete the two blocks set up for this program, which is a labelled block.

(*iv*) If the top stack item is **'for begin** (bl)' the (bl) marker is removed from this item, using the local variable *x*.

(*v*) If the top stack item is **'proc begin** (bl)' the subroutine COLLAPSE is used to collapse the name list for the procedure body, the (bl) marker is removed using the local variable *x*, and the state variable *NL* is reset from the stacked value.


end
ENTRY 2
T = 0 ?
NO
YES
FAIL
END STATE
UNSTACK (1)
TS = for begin ?
NO
YES
FS END
TS ?
begin
begin (l)
begin (bl)(l)
begin (bl)
proc begin (bl)
other
for begin (bl)
SP := SP − 1
SP := SP − 1
END BLOCK
COLLAPSE (1)
restore (proc begin (bl), [x])
restore (0, 0, [NL])
FAIL
restore (for begin (bl), [x])
STACK [for begin, x, 0]
END BLOCK
STACK [proc begin, x, 0]
n = 0 ?
YES
NO
COMPILE (1) [FINISH]
BCR (0)
Delimiter = '→' ?
NO
YES
print error message
FAIL = 0 ?
NO
YES
FINISH
EXIT TO CONTROL ROUTINE
x := 0
BCR (4)
Delimiter ?
end | else | ';'
other
m := 0
OUT 2
x = 0 ?
YES
NO
print message
x := 1

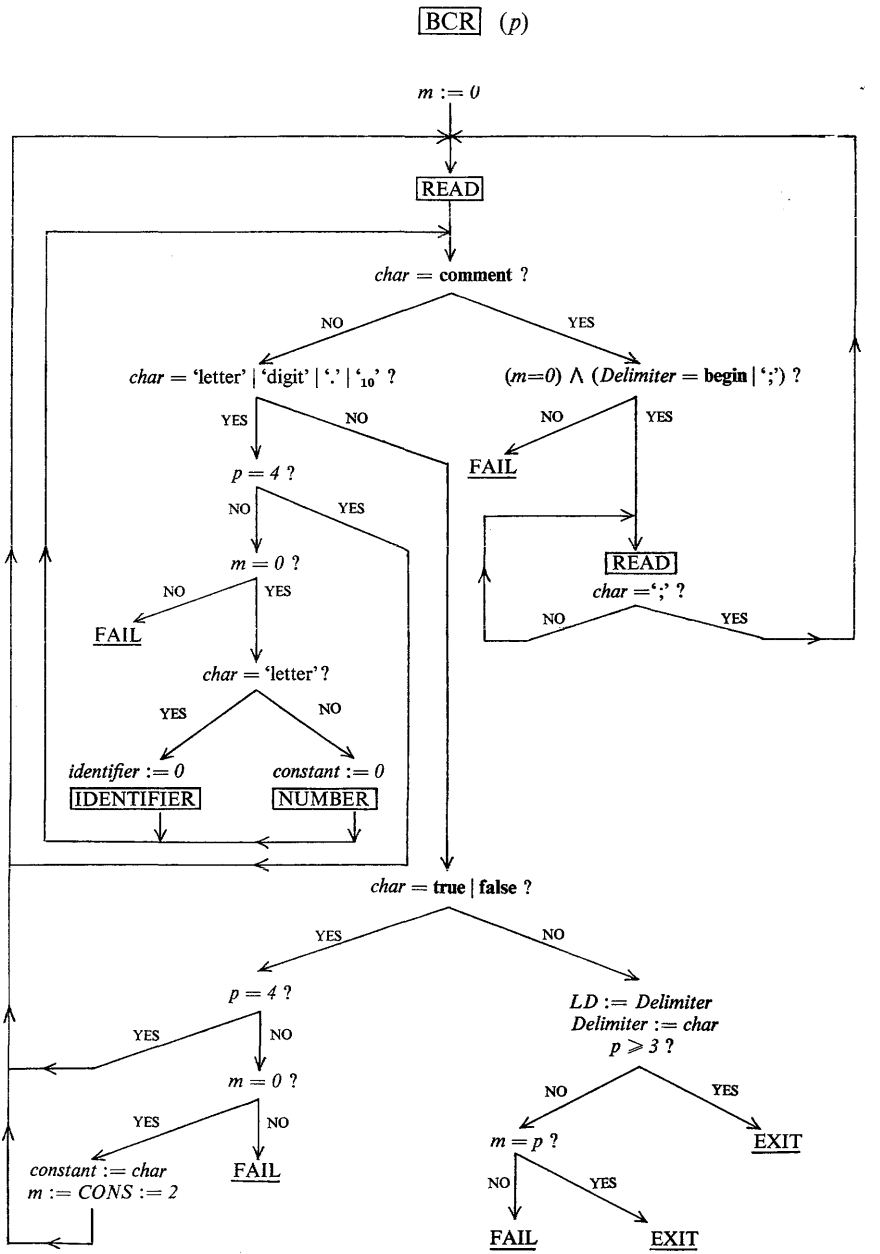
BCR

This subroutine is used to fetch the next section of ALGOL text, and to allow for comments after the delimiters **begin** and ';'.

The subroutine BCR in turn uses the subroutine READ to fetch the next ALGOL symbol, and the subroutines IDENTIFIER and NUMBER to process the constituents of identifiers and numbers.

The function of BCR is controlled by its parameter p as follows:

If $p = 0$, 1 or 2, a certain type of ALGOL section is expected, and this is checked by comparing the final value of the state variable m with the value of p; if $p = 3$ no check is made on the type of ALGOL section; finally, if $p = 4$, the constituents of identifiers and numbers need not be processed, as the subroutine BCR is being used to allow for an **end** comment, or during scanning after an error.


BCR (p)
m := 0
READ
char = comment ?
NO
YES
char = 'letter' | 'digit' | '.' | '10' ?
(m=0) ∧ (Delimiter = begin | ';') ?
YES
NO
NO
YES
p = 4 ?
FAIL
NO
YES
m = 0 ?
READ
char =';' ?
NO
YES
NO
YES
FAIL
char = 'letter' ?
YES
NO
identifier := 0
IDENTIFIER
constant := 0
NUMBER
char = true | false ?
YES
NO
p = 4 ?
LD := Delimiter
Delimiter := char
p ⩾ 3 ?
YES
NO
YES
NO
m = 0 ?
NO
YES
YES
EXIT
m = p ?
constant := char
m := CONS := 2
FAIL
NO
YES
FAIL
EXIT

READ

This subroutine uses the subroutine UPDATE BUFFER, which produces an ALGOL symbol, or a constituent of an ALGOL symbol, to form the next ALGOL symbol. The subroutine DICT is used if the constituent is an underlined character. If the state variable *PROC* is set, an extended parameter delimiter is replaced by a comma. In order to fill the code buffer at the start of translation the subroutine UPDATE BUFFER is entered repeatedly until a non-blank character is produced.

In the case of the composite ALGOL symbol ':=' or a parameter comment the second item of the code buffer (A_2) is inspected.

COMPARE

This subroutine checks the compatibility of the two bit patterns given as its parameters, and is used to check that successive occurrences of an identifier are consistent with regard to the information known or deduced about its type. The bit patterns which are used to give type information (given at the start of this Appendix) are designed such that two bit patterns are compatible if they are identical or if one can be formed from the other simply by adding in extra bits.

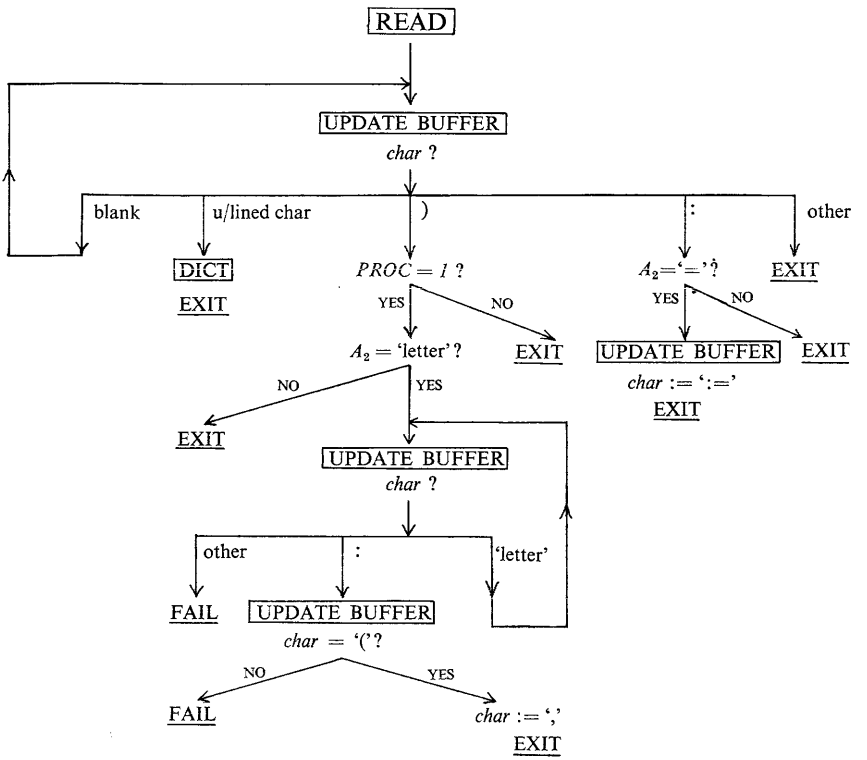
The function designator '*or* (*x*, *y*)' gives the bit pattern containing digits where either *x* or *y* has digits. The function designator '*dec* (*x*)' has the value **true** if *x* is a declared entry in the name list.

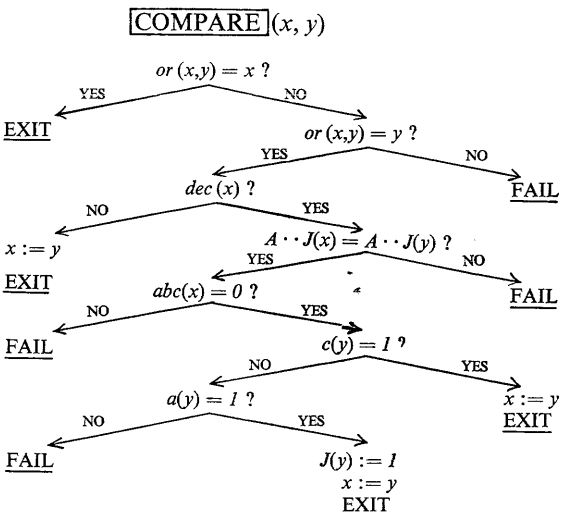
In general, the subroutine COMPARE updates the first bit pattern if it contains less information than the second bit pattern. However, if the first bit pattern is the *type* column of a declared entry in the name list a failure is indicated for any of the following reasons.

(*i*) The two bit patterns differ in columns *A*, *B*, . . , *J*. (e.g. if an identifier is declared to be real and is used as an integer.)

(*ii*) If updating would cause columns *a*, *b* and *c* to contain more than one bit.

(*iii*) If column *b* needs to be updated. (e.g. if a formal parameter specified to be a procedure (which has columns *a*, *b* and *c* left blank until it is known whether it is used with any parameters) is used as an array or switch.)


READ
UPDATE BUFFER
char ?
blank
u/lined char
)
:
other
DICT
EXIT
PROC = 1 ?
YES
NO
EXIT
A2 = 'letter' ?
NO
YES
EXIT
UPDATE BUFFER
char ?
other
:
'letter'
FAIL
UPDATE BUFFER
char = '(' ?
NO
YES
FAIL
char := ','
EXIT
A2='=' ?
YES
NO
EXIT
UPDATE BUFFER
EXIT
char := ':='
EXIT


COMPARE (x, y)
or (x,y) = x ?
YES
NO
EXIT
or (x,y) = y ?
YES
NO
FAIL
dec (x) ?
NO
YES
x := y
EXIT
A · · J(x) = A · · J(y) ?
YES
NO
FAIL
abc(x) = 0 ?
NO
YES
FAIL
c(y) = 1 ?
NO
YES
x := y
EXIT
a(y) = 1 ?
NO
YES
FAIL
J(y) := 1
x := y
EXIT

DICT

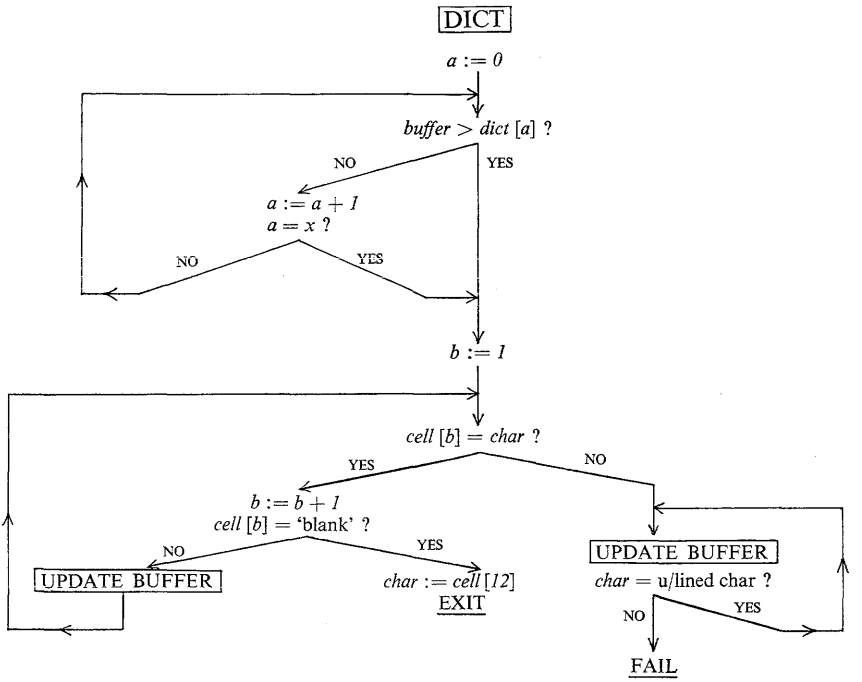
This subroutine is used to compare the underlined characters in the code buffer with a table (*dict*) of x basic symbols formed from underlined characters, and their constituent characters.

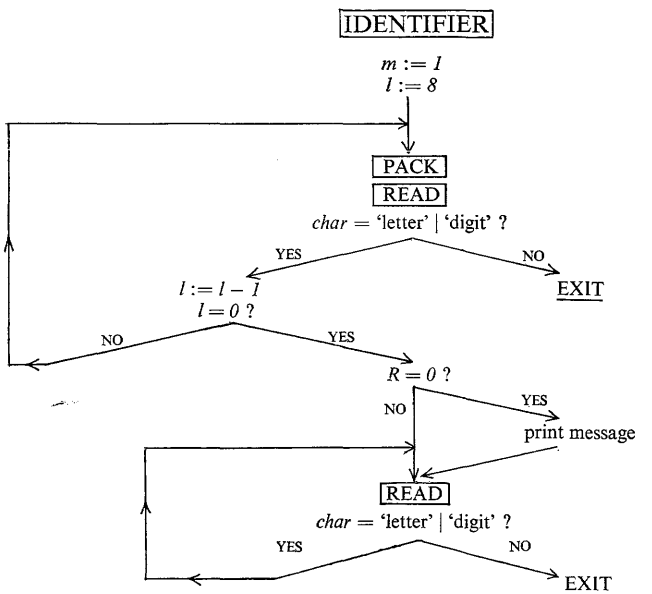
The local variable a is used to refer to successive items in the table. Having selected an item from the table the individual constituents of the basic symbol are compared in turn with the characters produced by the subroutine UPDATE BUFFER, using the local variable b, and the notation '*cell* [*b*]' to indicate the various constituents. If it is found that the item from the table agrees with a set of characters produced from the code buffer the representation of the corresponding basic symbol is obtained from the last cell of the item. Otherwise a failure is indicated, after using UPDATE BUFFER repeatedly until the end of the sequence of underlined characters is reached.

IDENTIFIER

This subroutine uses the subroutine PACK (whose flow diagram is not given) to form an identifier from its constituent letters and digits.

The local variable l is used to check the length of the identifier; a warning message is printed if the identifier contains more than eight characters, and the remaining characters are read and ignored. During the scanning of the ALGOL text after an error has been detected the printing of the warning message is inhibited by the global variable R.


DICT
a := 0
buffer > dict [a] ?
NO
YES
a := a + 1
a = x ?
NO
YES
b := 1
cell [b] = char ?
YES
NO
b := b + 1
cell [b] = 'blank' ?
NO
YES
UPDATE BUFFER
char := cell [12]
EXIT
UPDATE BUFFER
char = u/lined char ?
NO
YES
FAIL

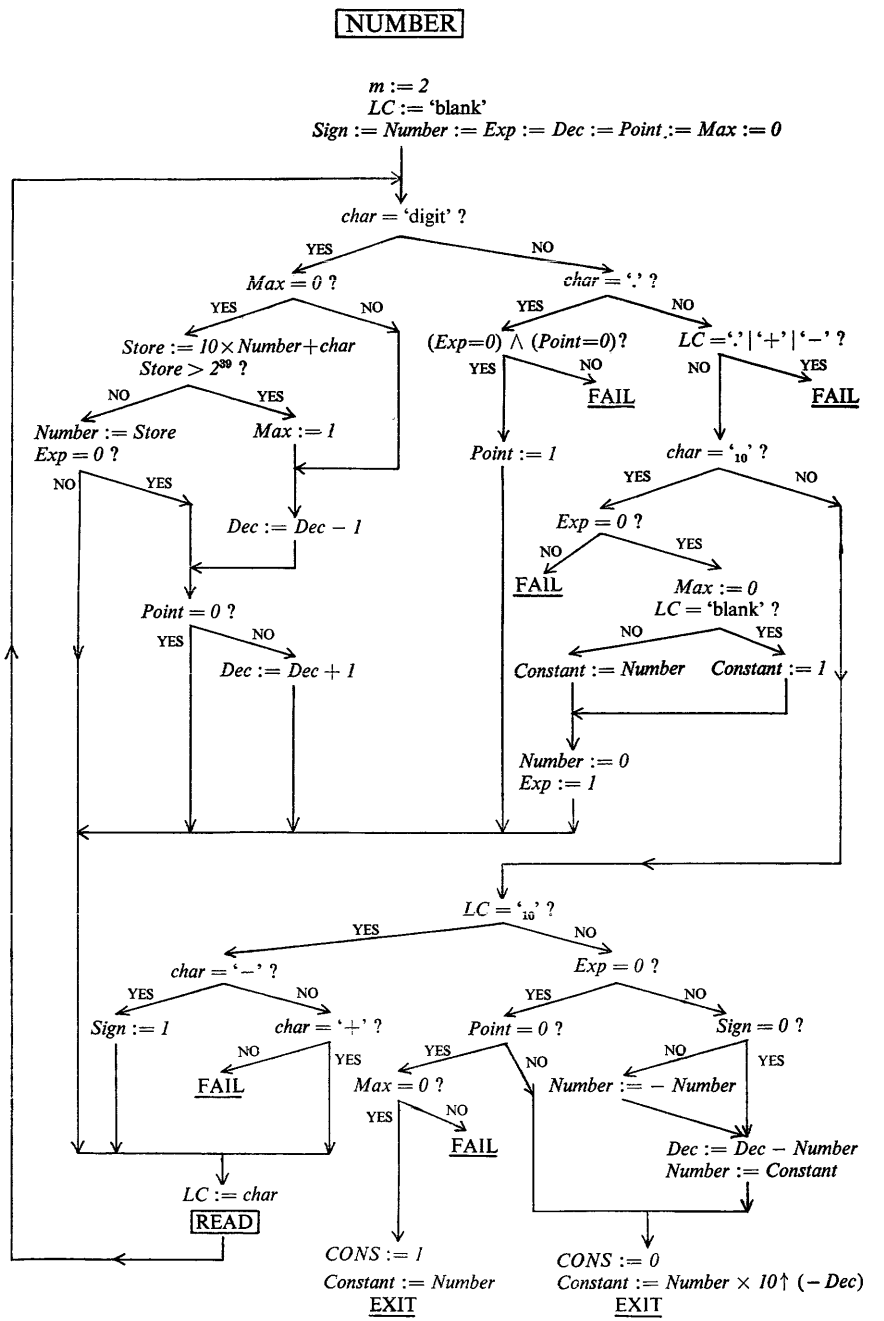

IDENTIFIER
m := 1
l := 8
PACK
READ
char = 'letter' | 'digit' ?
YES
NO
EXIT
l := l - 1
l = 0 ?
NO
YES
R = 0 ?
NO
YES
print message
READ
char = 'letter' | 'digit' ?
YES
NO
EXIT

NUMBER

This subroutine (by C. W. Nott of N.P.L.) uses the subroutine READ and processes the constituents of a number.

Quantities local to the subroutine are

LC	last character read.
Sign	set to one if the exponent part is negative
Number	used for the partially computed number
Store	used during the computation of the number
Exp	set to one at the character '$_{10}$'
Point	set to one at the character '.'
Dec	used to count number of decimal places and for any exponent
Max	set to one if the integer or exponent part of the number exceeds capacity


NUMBER
m := 2
LC := 'blank'
Sign := Number := Exp := Dec := Point := Max := 0
char = 'digit' ?
YES
NO
Max = 0 ?
char = '.' ?
Store := 10×Number+char
Store > 2^39 ?
(Exp=0) ∧ (Point=0)?
LC = '.' | '+' | '−' ?
FAIL
Number := Store
Exp = 0 ?
Max := 1
Point := 1
char = '10' ?
Dec := Dec − 1
Exp = 0 ?
Point = 0 ?
Dec := Dec + 1
Max := 0
LC = 'blank' ?
Constant := Number
Constant := 1
Number := 0
Exp := 1
LC = '10' ?
char = '−' ?
Exp = 0 ?
Sign := 1
char = '+' ?
Point = 0 ?
Sign = 0 ?
Max = 0 ?
Number := − Number
Dec := Dec − Number
Number := Constant
LC := char
READ
CONS := 1
Constant := Number
EXIT
CONS := 0
Constant := Number × 10↑ (− Dec)
EXIT

UPDATE BUFFER

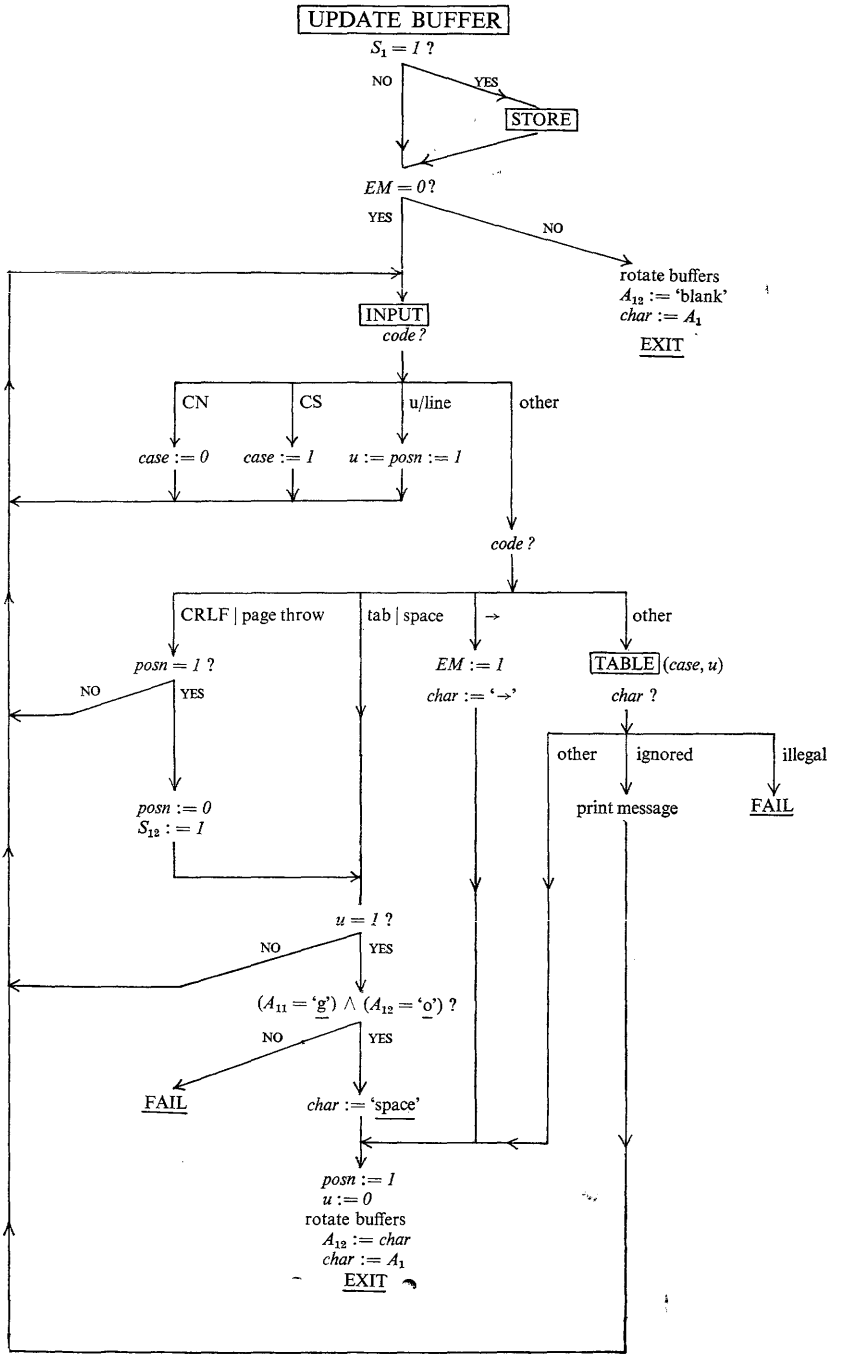
This subroutine produces an 8-bit character, having allowed for case shifts, underlining, etc. The subroutine INPUT, whose flow diagram is not given, produces the next 6-bit tape input character, which it assigns to the local variable *code*.

The subroutine uses a buffer of twelve characters ($A_1, \ldots, A_{12}$), and after rotating the buffer to remove the character in A_1, stores the newly-formed character in A_{12}, and produces as output in *char* the contents of A_1.

The variable *case* is modified appropriately by Case Shift (CS) and Case Normal (CN) characters. The variable *u* is set at an underline, and is cleared by any character which moves the typewriter carriage. The variable *posn* is set by any printed character and cleared at the end of a line. The variables *line no.* and *rel. line no.* are increased by one for each line which contains printed characters, by the subroutine STORE. Using the line buffer ($S_1, \ldots, S_{12}$), STORE, whose flow diagram is not given, stores entries in a table of object program counter against *line no.*, for use by the Control Routine at a failure.

Provision is made for a single underlined space occurring between **go** and **to**. The end of message character '→' causes the variable *EM* to be set, in order to avoid the subroutine INPUT, whereupon further uses of the subroutine UPDATE BUFFER simply deliver the characters already in the buffer.

The subroutine TABLE, whose flow diagram is not given, performs a table look up using the current code and the values of *case* and *u* in order to produce a corresponding 8-bit internal representation.

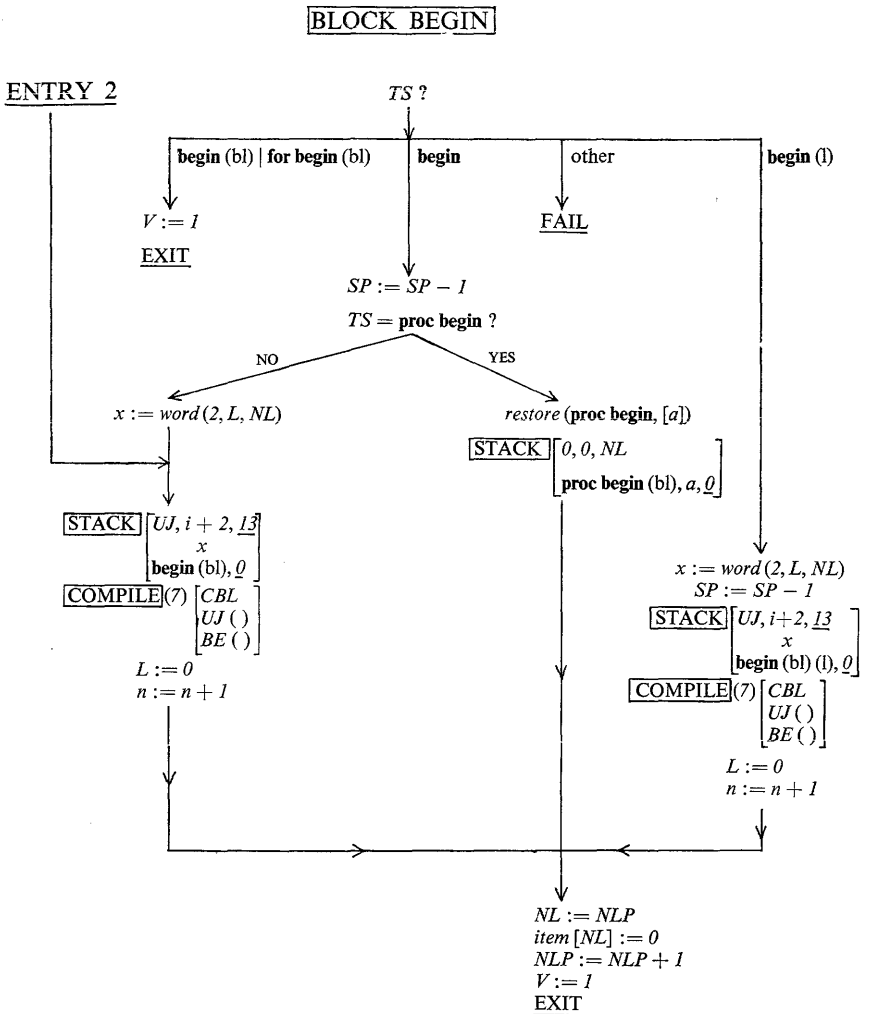

UPDATE BUFFER
$S_1 = 1$?
NO
YES
STORE
$EM = 0$?
YES
NO
rotate buffers
A_{12} := 'blank'
$char := A_1$
EXIT
INPUT
code?
CN
CS
u/line
other
$case := 0$
$case := 1$
$u := posn := 1$
code?
CRLF | page throw
tab | space
→
other
$posn = 1$?
NO
YES
$EM := 1$
$char :=$ '→'
TABLE (case, u)
char ?
other
ignored
illegal
print message
FAIL
$posn := 0$
$S_{12} := 1$
$u = 1$?
NO
YES
$(A_{11} =$ 'g') $\wedge$ $(A_{12} =$ 'o') ?
NO
YES
FAIL
$char :=$ 'space'
$posn := 1$
$u := 0$
rotate buffers
$A_{12} := char$
$char := A_1$
EXIT

BLOCK BEGIN

This subroutine is used to set up a block in the object program, stack and name list.

If the top of the stack is **begin** this item is unstacked; the top of the stack is again inspected and if it is '**proc begin**' the value of *NL* is stacked, a (bl) marker is added to the item '**proc begin**', using the local variable *a*, and a new block is set up in the name list. Otherwise, a block is set up in the normal way, using the local variable *x*. (In order to set up a program block, the second entry to the subroutine is used, *x* being set up outside the subroutine.)

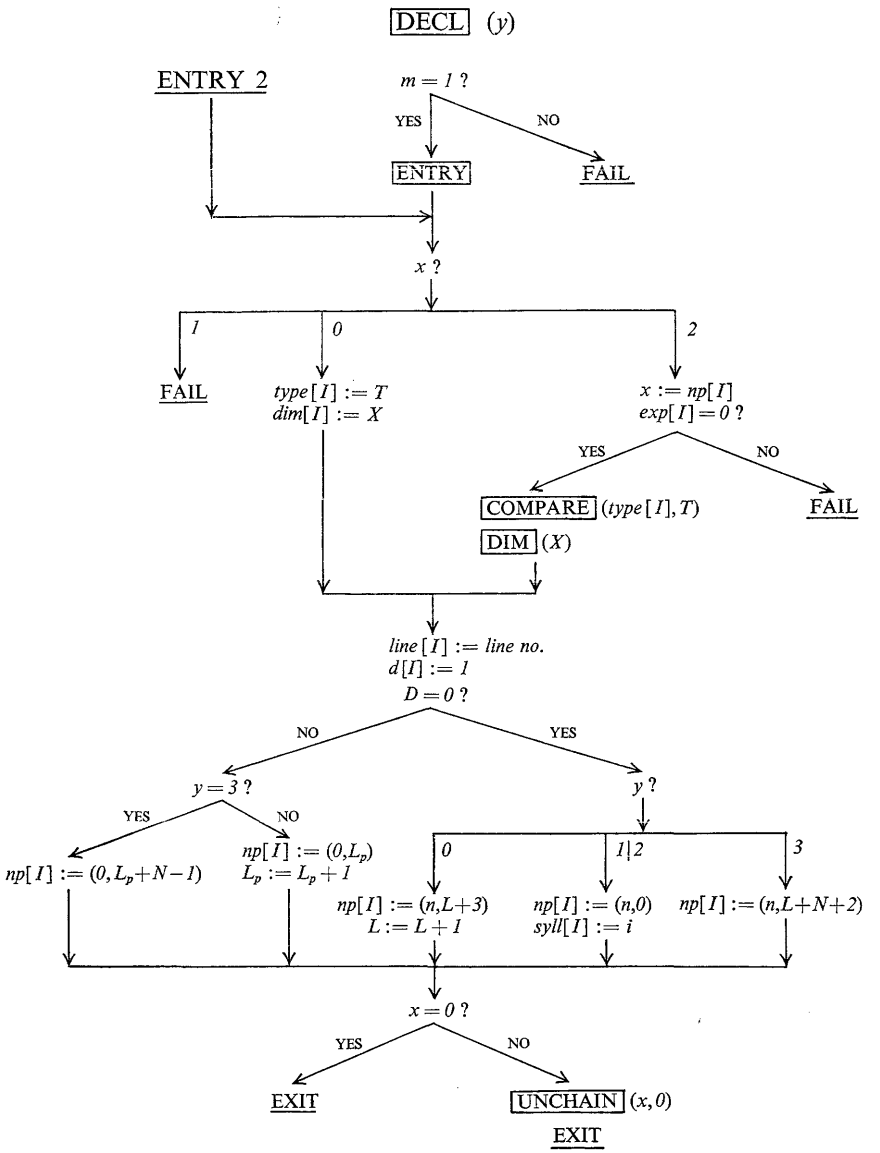
If the top of the stack is '**begin** (l)' an extra block is set up for a program which is a labelled block.


BLOCK BEGIN
ENTRY 2
TS ?
begin (bl) | for begin (bl)
begin
other
begin (l)
V := 1
EXIT
FAIL
SP := SP − 1
TS = proc begin ?
NO
YES
x := word (2, L, NL)
restore (proc begin, [a])
STACK
0, 0, NL
proc begin (bl), a, 0
STACK
UJ, i + 2, 13
x
begin (bl), 0
COMPILE (7)
CBL
UJ ( )
BE ( )
L := 0
n := n + 1
x := word (2, L, NL)
SP := SP − 1
STACK
UJ, i+2, 13
x
begin (bl) (l), 0
COMPILE (7)
CBL
UJ ( )
BE ( )
L := 0
n := n + 1
NL := NLP
item [NL] := 0
NLP := NLP + 1
V := 1
EXIT

DECL

This subroutine is used to 'declare' the current identifier, by adding *type* and *dim* information into its name list entry (the position of which is found by the subroutine ENTRY). The parameter *y* of the subroutine controls the setting up of the *np* and *syll* columns.

The subroutine ENTRY sets up the variable *x* to indicate the type of name list entry for the current declaration (*x* has the value *0* if a new entry has been created, *1* if a declared entry was found, *2* if a used entry was found). If a new entry has been created, the *type* and *dim* columns are set up using the state variables *T* and *X*. The *line* and *d* columns are set up and then *y* and the own marker *D* are used to control the setting up of the *np* and *syll* columns. If a used entry was found, the contents of the *np* column are stored in the variable *x*, and a failure is indicated if the *exp* column has been set, since this identifier, now proved to be local, has previously been used in an array declaration. The subroutines COMPARE and DIM are used to check the compatibility of the previous uses of this identifier with the current declaration. Then, after setting up the *line*, *d*, *np* and *syll* columns of the name list entry, the skeleton operations generated for previous uses of the identifier are completed, using the subroutine UNCHAIN.


DECL (y)
ENTRY 2
m = 1 ?
YES
NO
ENTRY
FAIL
x ?
1
0
2
FAIL
type[I] := T
dim[I] := X
x := np[I]
exp[I] = 0 ?
YES
NO
COMPARE (type[I], T)
FAIL
DIM (X)
line[I] := line no.
d[I] := 1
D = 0 ?
NO
YES
y = 3 ?
y ?
YES
NO
np[I] := (0, Lp+N−1)
np[I] := (0,Lp)
Lp := Lp + 1
0
1|2
3
np[I] := (n,L+3)
L := L + 1
np[I] := (n,0)
syll[I] := i
np[I] := (n,L+N+2)
x = 0 ?
YES
NO
EXIT
UNCHAIN (x, 0)
EXIT

CHECK OP

This subroutine is used by the subroutine PROC HEADING to compile the appropriate parameter list operations. The current part of the name list consists of a duplicate entry for the procedure identifier and entries for the formal parameters. A failure is indicated if a parameter has not been specified, or if a parameter specified to be a switch, a string or a procedure, has been called by value. In the case of a label called by value, the *f* column is reset to one, in order that the operation *TFL* will be generated at subsequent uses of the parameter. The *type* column is used to control the generation of the parameter list operations, according to the following tables.

Table 1

(call by name)

aABCG	*CA*
aABCEG	*CA*
aABDG	*CB*
bABCG	*CAR*
bABCEG	*CAI*
bABDG	*CAB*
aAF	*CL*
bAF	*CSW*
H	*CST*
G	*CPR*
ABCG	*CFR*
ABCEG	*CFI*
ABDG	*CFB*

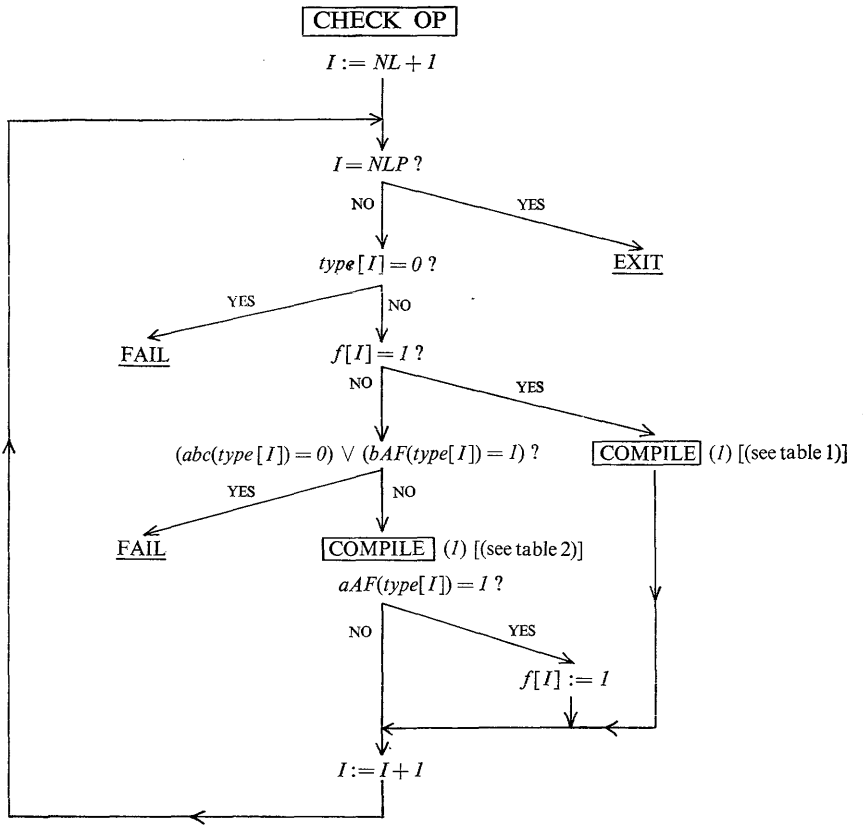
Table 2

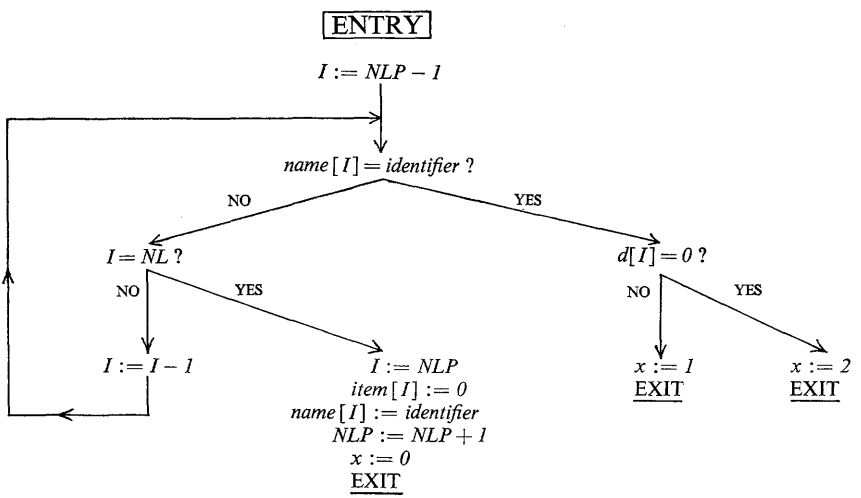
(call by value)

aABCG	*CSR*
aABCEG	*CSI*
aABDG	*CSB*
bABCG	*CRFA*
bABCEG	*CIFA*
bABDG	*CBFA*
aAF	*CSL*

ENTRY

This subroutine is used to search the current part of the name list for an entry corresponding to the current identifier. If no entry is found, a new entry is created, and *NLP* is increased by one. The variable *x* is set to *0* if a new entry has been created, to *1* if a declared entry has been found, or to *2* if a used entry has been found.

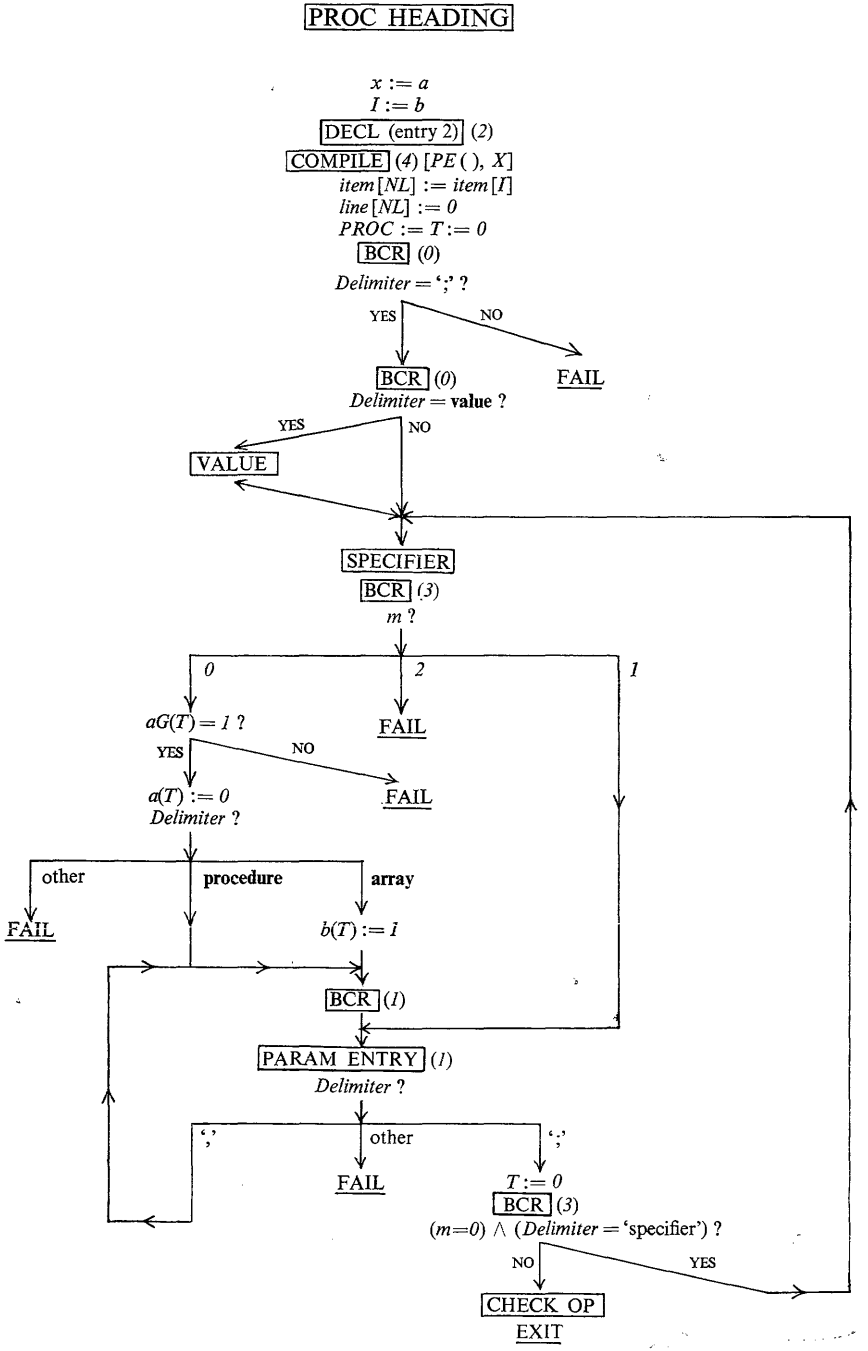

CHECK OP
I := NL + 1
I = NLP ?
NO
YES
EXIT
type[I] = 0 ?
YES
NO
FAIL
f[I] = 1 ?
NO
YES
(abc(type[I]) = 0) ∨ (bAF(type[I]) = 1) ?
COMPILE (I) [(see table 1)]
YES
NO
FAIL
COMPILE (I) [(see table 2)]
aAF(type[I]) = 1 ?
NO
YES
f[I] := 1
I := I + 1


ENTRY
I := NLP - 1
name[I] = identifier ?
NO
YES
I = NL ?
NO
YES
I := I - 1
I := NLP
item[I] := 0
name[I] := identifier
NLP := NLP + 1
x := 0
EXIT
d[I] = 0 ?
NO
YES
x := 1
EXIT
x := 2
EXIT

PROC HEADING

This subroutine is used by the routine **procedure** to declare the procedure identifier and to process the value and the specification parts of the procedure heading.

The value of x and I for the name list entry of the procedure identifier are reset from the local variables a and b, respectively, and the procedure identifier is declared using DECL (the second entry is used since the name list has already been searched for an entry corresponding to the procedure identifier). The entry for the procedure identifier is duplicated as the first entry of the current section of the name list, and the *line* column of this entry is cleared. After checking that the closing bracket of the formal parameter part is followed by ';', the next delimiter is fetched. If it is **value** the subroutine VALUE is used to process the value part. Otherwise the subroutine SPECIFIER is used to check that the delimiter can start a specification part, and to set up the state variable T accordingly. The next ALGOL section is fetched. If this contains a constant a failure is indicated. If it contains an identifier the subroutine PARAM ENTRY is used with a parameter of one, to add the specification details to the name list entry for the identifier. Otherwise, if the section contains only a delimiter this is checked to ensure that it is either **procedure** or **array**, following **real**, **integer** or **Boolean**, and the next ALGOL section is fetched (using BCR with a parameter of one, since the section must contain an identifier). This process is continued until a semicolon is reached, whereupon the state variable T is cleared. The next ALGOL section is fetched and inspected to decide whether a further specification part is to be processed or whether the end of the procedure heading has been reached. Finally the subroutine CHECK OP is used to generate the parameter list operations.


PROC HEADING
x := a
I := b
DECL (entry 2) (2)
COMPILE (4) [PE( ), X]
item[NL] := item[I]
line[NL] := 0
PROC := T := 0
BCR (0)
Delimiter = ';' ?
YES
NO
BCR (0)
FAIL
Delimiter = value ?
YES
NO
VALUE
SPECIFIER
BCR (3)
m ?
0
2
1
aG(T) = 1 ?
FAIL
YES
NO
a(T) := 0
FAIL
Delimiter ?
other
procedure
array
FAIL
b(T) := 1
BCR (1)
PARAM ENTRY (1)
Delimiter ?
','
other
';'
FAIL
T := 0
BCR (3)
(m=0) ∧ (Delimiter = 'specifier') ?
NO
YES
CHECK OP
EXIT

VALUE

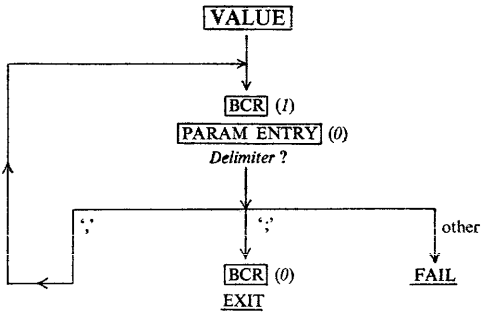
This subroutine is used by the subroutine PROC HEADING to process the value part of a procedure heading. The subroutine PARAM ENTRY is used to clear the *f* column and set the *v* column of the name list entries for the identifiers appearing in the value list. At the delimiter ';' the next ALGOL section is fetched, using BCR with a parameter of zero, since this section should contain only a delimiter.

PARAM ENTRY

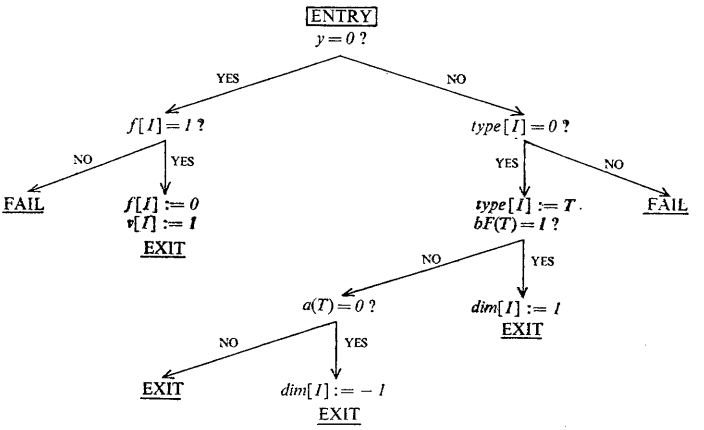
This subroutine is used by the subroutine VALUE, with a parameter of zero, as described above (a failure is indicated if an identifier appears more than once in a value list), and by the subroutine PROC HEADING, with a parameter of one, to add in the specification details to the name list entry for a formal parameter (a failure is indicated if an identifier appears more than once in the specification part). In the case of a parameter specified to be a switch, the *dim* column is set to one, and in the case of a parameter specified to be an array or procedure, it is set to '-1' (since the number of dimensions or parameters is not known).

SPECIFIER

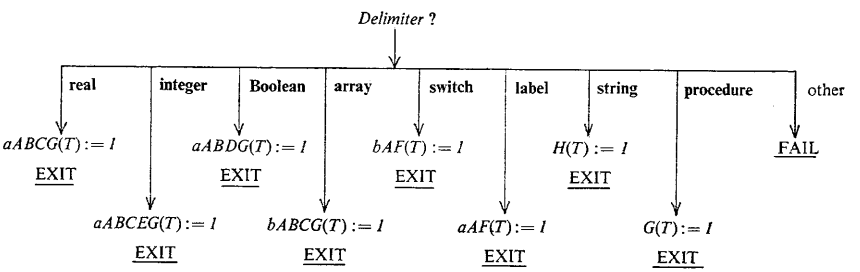
This subroutine is used by the subroutine PROC HEADING to check that the current delimiter can start a specification, and then to set the state variable T accordingly.



PARAM ENTRY (y)



SPECIFIER

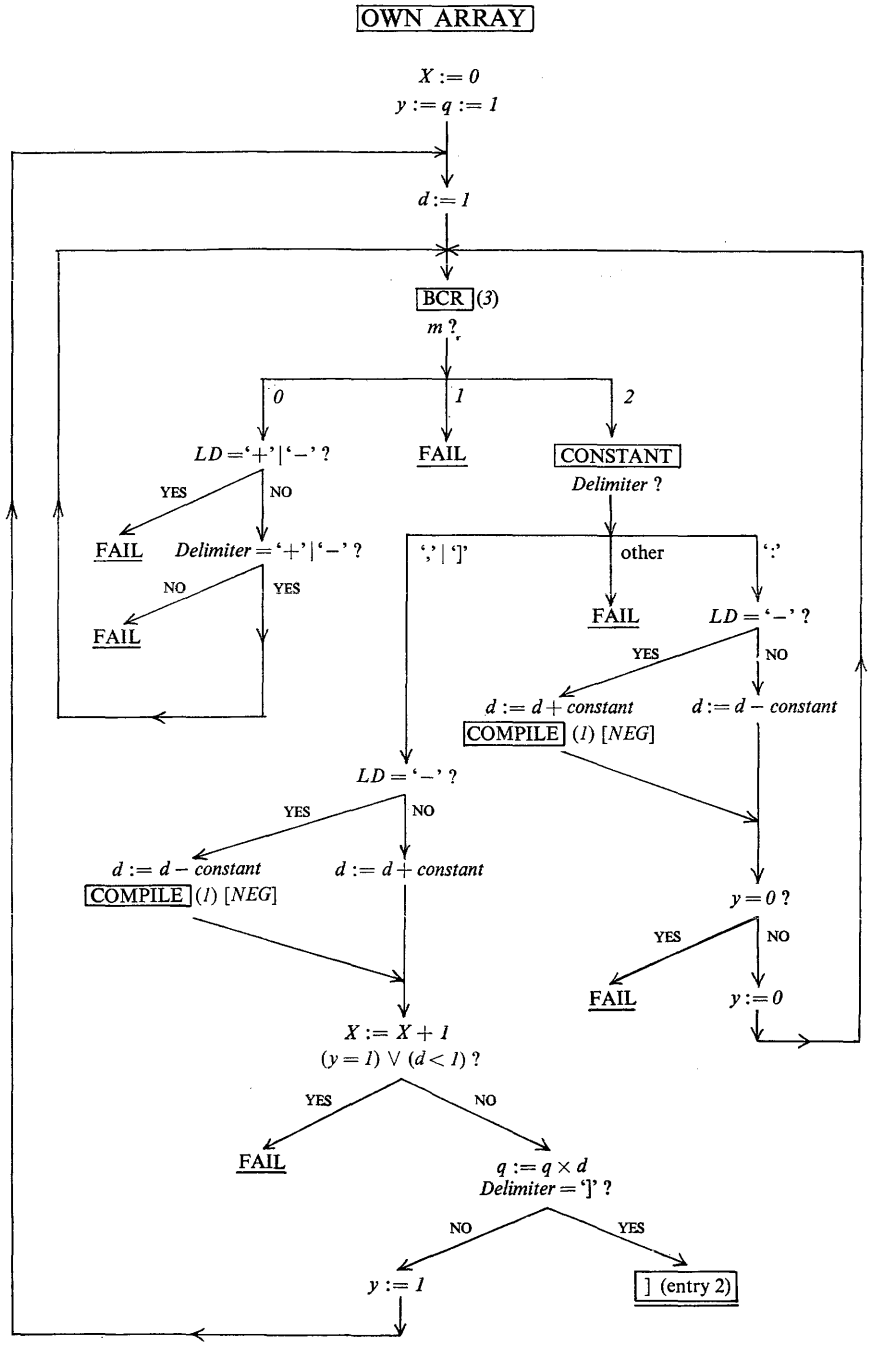


O

OWN ARRAY

This subroutine is used to process the bound pair list of an own array declaration. (The upper and lower bounds are limited to being signed or unsigned integers.)

The local variable *y* is used as a marker to ensure that the delimiter ':' is used alternately with the delimiters ',' or ']'. The local variables *d* and *q* are used in the calculation of the size of the array. The global variable *X* is set up with the number of dimensions of the array. Finally the routine ']' is entered, by the second entry point, instead of returning to the routine '[' from which this subroutine was called.


OWN ARRAY
X := 0
y := q := 1
d := 1
BCR (3)
m ?
0
1
2
LD = '+' | '−' ?
FAIL
CONSTANT
Delimiter ?
YES
NO
FAIL
Delimiter = '+' | '−' ?
NO
YES
FAIL
',' | ']'
other
':'
FAIL
LD = '−' ?
YES
NO
d := d + constant
COMPILE (1) [NEG]
d := d − constant
LD = '−' ?
YES
NO
d := d − constant
COMPILE (1) [NEG]
d := d + constant
y = 0 ?
YES
NO
FAIL
y := 0
X := X + 1
(y = 1) ∨ (d < 1) ?
YES
NO
FAIL
q := q × d
Delimiter = ']' ?
NO
YES
y := 1
] (entry 2)
O*

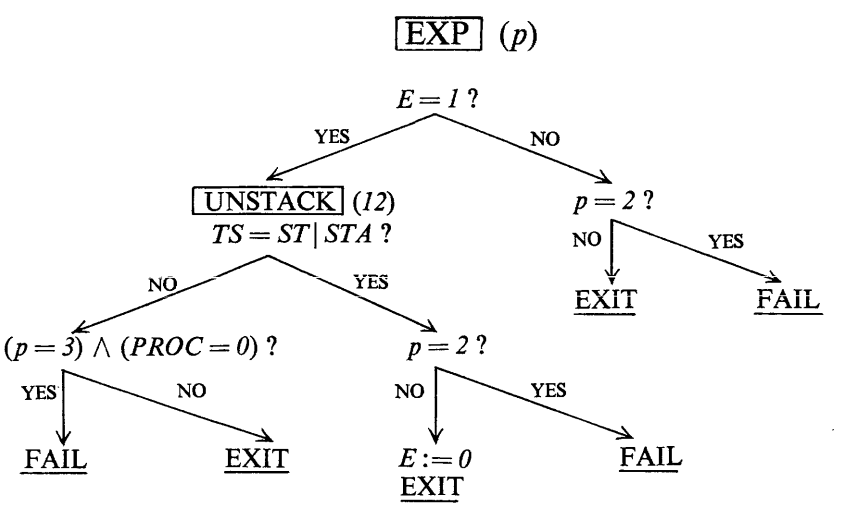
EXP

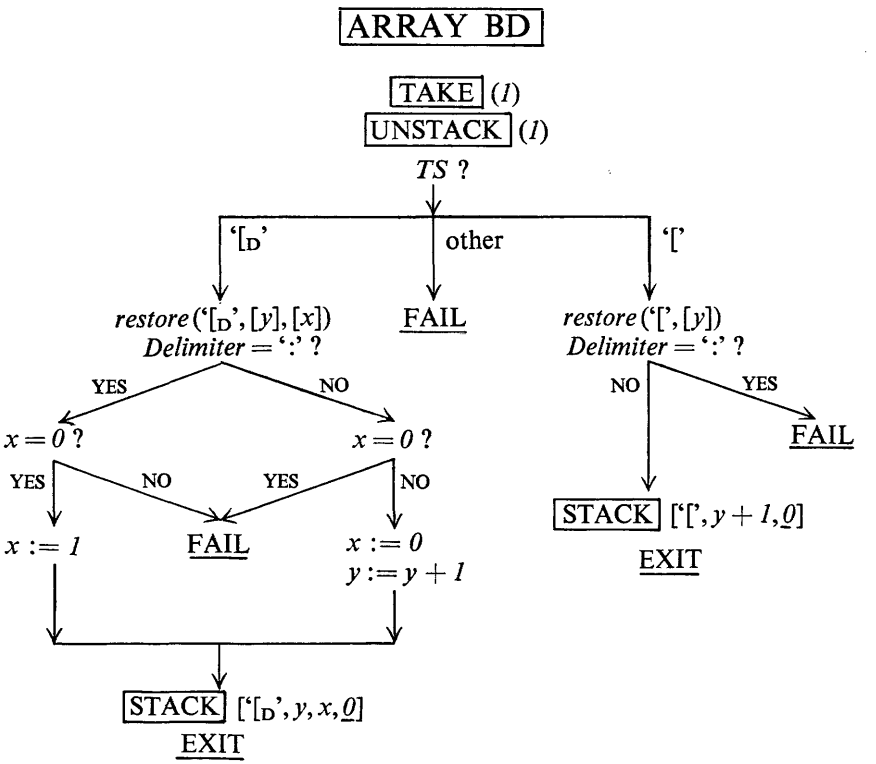
This subroutine is used to check the validity of use of the current delimiter, and to change the state variable E from statement to expression level, if necessary.

The action of the subroutine is controlled by its parameter p; if $p = 1$ the delimiter can only be used at expression level; if $p = 2$ the delimiter can only be used at statement level; if $p = 3$ the delimiter can be used at either statement or expression level. The subroutine UNSTACK is used to unstack the item *IND* if necessary (the parameter p being used by UNSTACK to decide whether *INDA* or *INDR* is to be generated).

ARRAY BD

This subroutine is used by the routines for the delimiters ':' and ',' to check the validity of their use (using the local variable x), and to complete the processing of the preceding subscript bound expression (using the subroutines TAKE and UNSTACK). Finally the dimension counter stacked with the item '[' or '$[_D$' is increased by one using the local variable y.


EXP (p)
E = 1 ?
YES
NO
UNSTACK (12)
TS = ST | STA ?
p = 2 ?
NO
YES
EXIT
FAIL
NO
YES
(p = 3) ∧ (PROC = 0) ?
p = 2 ?
YES
NO
NO
YES
FAIL
EXIT
E := 0
EXIT
FAIL


ARRAY BD
TAKE (1)
UNSTACK (1)
TS ?
'[D'
other
'['
restore('[D', [y], [x])
Delimiter = ':' ?
FAIL
restore('[', [y])
Delimiter = ':' ?
YES
NO
NO
YES
x = 0 ?
x = 0 ?
FAIL
YES
NO
YES
NO
STACK ['[', y + 1, 0]
EXIT
x := 1
FAIL
x := 0
y := y + 1
STACK ['[D', y, x, 0]
EXIT

BEGIN STATE

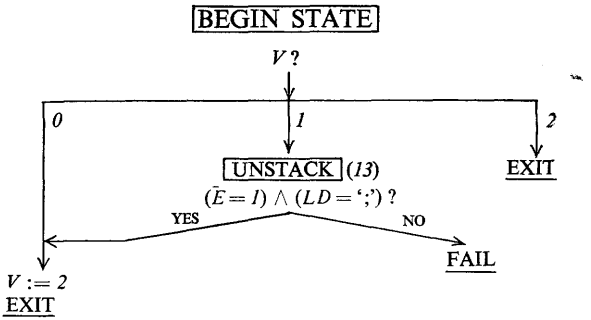
This subroutine is used at a delimiter which could start a statement, to check whether this is the first statement of a block or compound statement. If the state variable *V* is zero this is the first statement of a compound statement and *V* is set to two. If *V* is one this is the first statement of a block; the subroutine UNSTACK is used to complete, if necessary, an *UJ* operation around the last declaration or set of declarations, and a check is made that the last declaration ended with a semi-colon.

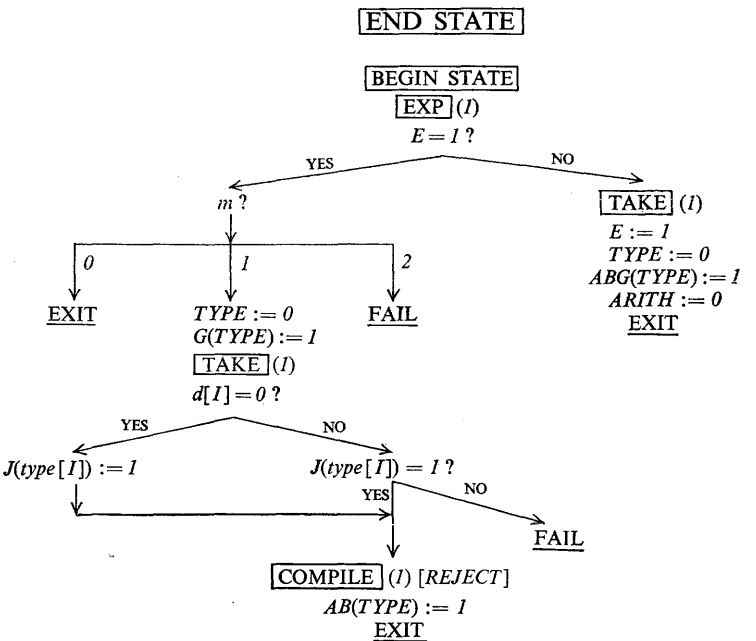
END STATE

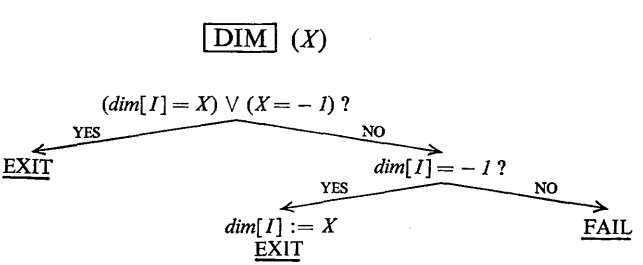
This subroutine is used to complete the processing of a statement. The subroutine BEGIN STATE is used for the case of a statement which consists of a single identifier (a procedure call with no parameters), and EXP is used to change the state variable *E* to expression level if necessary. If *E* is then zero the subroutine TAKE is used to generate the appropriate 'Take Result' operation, and the state variables *E*, *TYPE* and *ARITH* are set up for the processing of the next statement. Otherwise, if the current section contains an identifier, which must be a procedure identifier, TAKE is used to generate the appropriate *CFZ* or *CFFZ* operation. After checking the validity of this use of the identifier the operation *REJECT* is generated and the state variable *TYPE* is set up for the processing of the next statement.

DIM

The subroutine DIM is used to compare the *dim* column of the name list entry of the current identifier with the global variable *X*, and to set up the *dim* column with the value of *X* if it was previously '-1'.


BEGIN STATE
V ?
0
1
2
UNSTACK (13)
(E = 1) ∧ (LD = ';') ?
YES
NO
EXIT
FAIL
V := 2
EXIT


END STATE
BEGIN STATE
EXP (1)
E = 1 ?
YES
NO
m ?
TAKE (1)
E := 1
TYPE := 0
ABG(TYPE) := 1
ARITH := 0
EXIT
0
1
2
EXIT
TYPE := 0
G(TYPE) := 1
TAKE (1)
FAIL
d[1] = 0 ?
YES
NO
J(type[1]) := 1
J(type[1]) = 1 ?
YES
NO
FAIL
COMPILE (1) [REJECT]
AB(TYPE) := 1
EXIT


DIM (X)
(dim[1] = X) ∨ (X = − 1) ?
YES
NO
EXIT
dim[1] = − 1 ?
YES
NO
dim[1] := X
EXIT
FAIL

UNSTACK

This subroutine is used to unstack items from the top of the stack, until an item is reached whose stack priority is less than the value given by the parameter *x*, or until the stack is empty. In general, items are unstacked directly into the object program, with the following exceptions

(*i*) If the item at the top of the stack has a priority of twelve, the variable *p* (set up by the subroutine TAKE or EXP) determines whether the operation *INDA* or *INDR* is to be generated.

(*ii*) If the item at the top of the stack has a priority of eight, and the parameter *x* has the value eight a failure is indicated, since relational operators have been used incorrectly (e.g. '$x < y + z < 3$').

(*iii*) If the top stack item is **else** or has a priority of thirteen, the address stored with this item is used to complete an *UJ* operation, unchaining and completing any other *UJ* operations if necessary, using the local variables *a* and *b*.

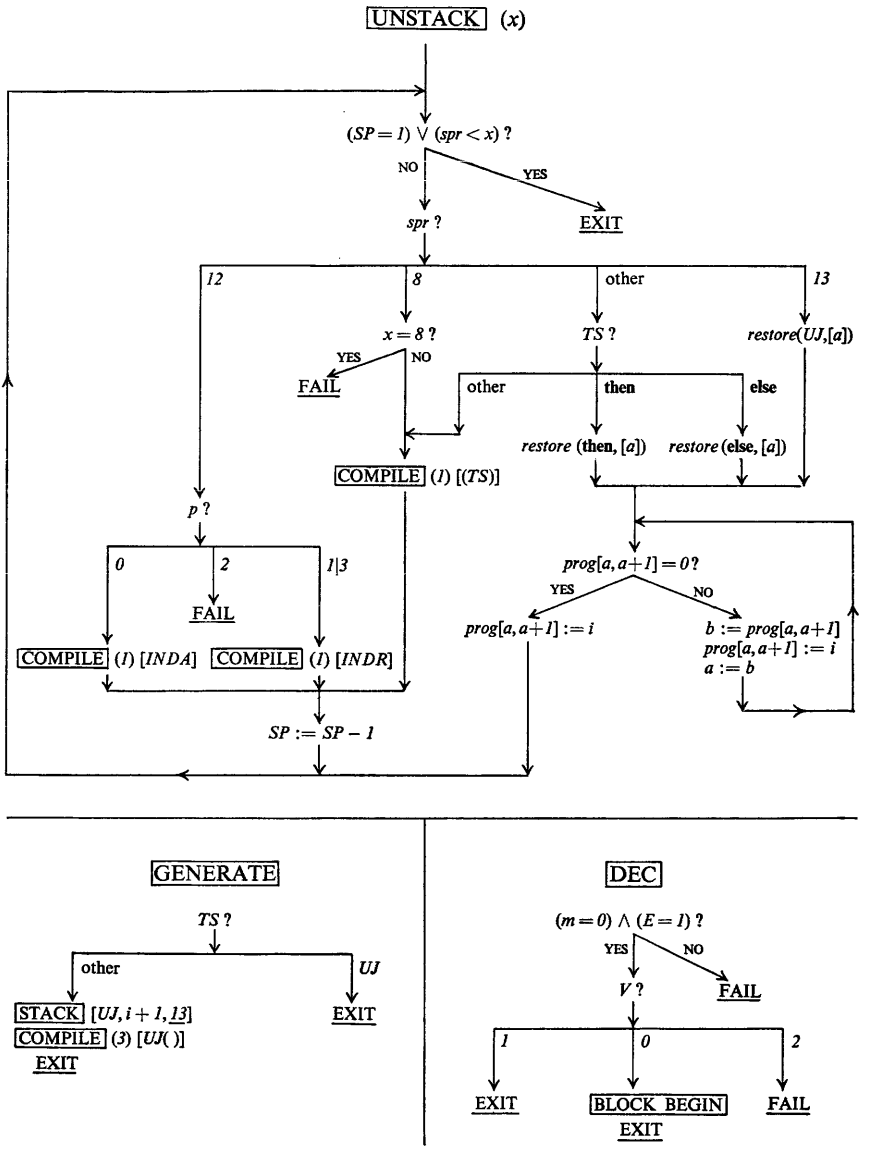
(*iv*) If the top stack item is **then**, the address stored with this item is used to complete an *IFJ* operation.

GENERATE

This subroutine is used to generate an *UJ* operation around a procedure or switch declaration. If the top stack item is *UJ* on entry to the subroutine no action is taken, thus combining the *UJ* operations around successive procedure or switch declarations into a single operation.

DEC

This subroutine is used by a delimiter which starts a declaration, to check the validity of its use, and to set up a block if this is the first declaration after a **begin**.


UNSTACK (x)
(SP = 1) ∨ (spr < x) ?
NO
YES
EXIT
spr ?
12
8
other
13
x = 8 ?
TS ?
restore(UJ,[a])
YES
NO
FAIL
other
then
else
restore (then, [a])
restore (else, [a])
COMPILE (1) [(TS)]
p ?
0
2
1|3
FAIL
prog[a, a+1] = 0?
YES
NO
prog[a, a+1] := i
b := prog[a, a+1]
prog[a, a+1] := i
a := b
COMPILE (1) [INDA]
COMPILE (1) [INDR]
SP := SP − 1
GENERATE
TS ?
other
UJ
STACK [UJ, i + 1, 13]
COMPILE (3) [UJ( )]
EXIT
EXIT
DEC
(m = 0) ∧ (E = 1) ?
YES
NO
V ?
FAIL
1
0
2
EXIT
BLOCK BEGIN
EXIT
FAIL

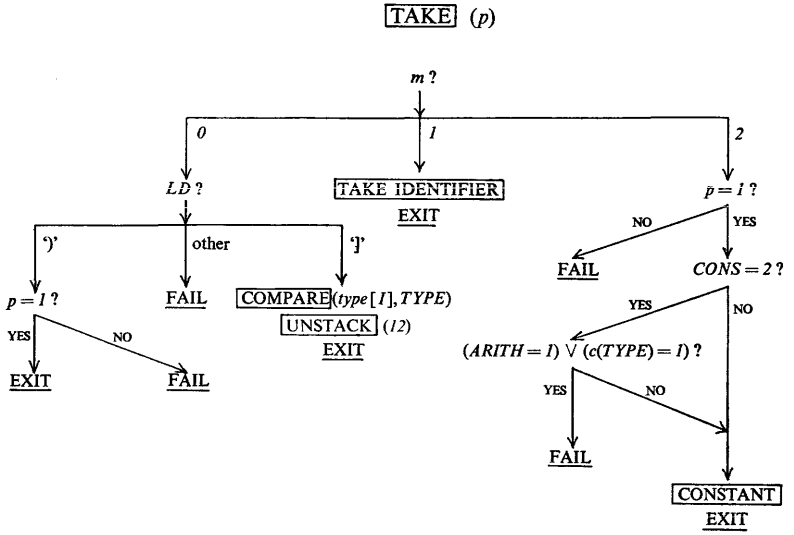
TAKE

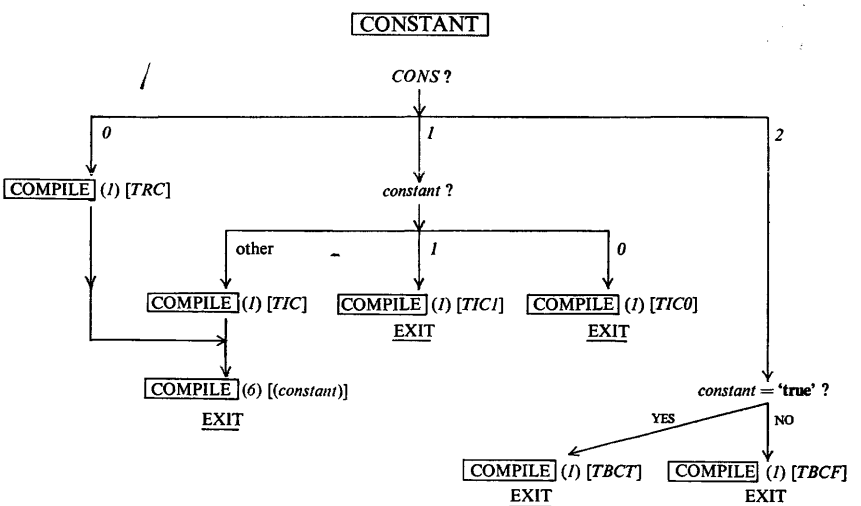
This subroutine is used to process an identifier or constant that is used in a statement or an expression. However, if the current ALGOL section does not contain an identifier or a constant, TAKE is used to check the validity of the section and, if the last delimiter is ']', to check the type of the array or switch identifier, and to generate *INDA* or *INDR*.

If the current ALGOL section contains an identifier it is dealt with by the subroutine TAKE IDENTIFIER. If the section contains a constant, checks are made on the type of the constant, and that it is not being used where a 'Take Address' operation would have been generated. The appropriate object program operation is then generated by the subroutine CONSTANT.

CONSTANT

The state variable *CONS* is used to determine the type of the current constant, and the appropriate 'Take Constant' operation is generated.


TAKE (p)
m ?
0
1
2
LD ?
TAKE IDENTIFIER
EXIT
p = 1 ?
NO
YES
')'
other
'['
FAIL
CONS = 2 ?
p = 1 ?
FAIL
COMPARE (type [1], TYPE)
UNSTACK (12)
EXIT
YES
NO
(ARITH = 1) ∨ (c(TYPE) = 1) ?
EXIT
FAIL
YES
NO
YES
NO
FAIL
CONSTANT
EXIT


CONSTANT
CONS ?
0
1
2
COMPILE (1) [TRC]
constant ?
other
1
0
COMPILE (1) [TIC]
COMPILE (1) [TIC1]
EXIT
COMPILE (1) [TIC0]
EXIT
COMPILE (6) [(constant)]
EXIT
constant = 'true' ?
YES
NO
COMPILE (1) [TBCT]
EXIT
COMPILE (1) [TBCF]
EXIT

TAKE IDENTIFIER

This subroutine is used by the subroutine TAKE to process the identifier in the current ALGOL section. It uses the subroutine ENTRY to search the current part of the name list for an entry corresponding to the identifier.

(*i*) If a new entry was created ($x = 0$) the *u*, *syll*, *line*, *type* and *np* columns of the entry are set up. If the parameter *p* (handed over from the subroutine TAKE) is zero, indicating that a 'Take Address' operation is required, the *dim* column is set to '-1', since the identifier cannot be proved to be scalar. If the identifier is being used in an array declaration the *exp* column is set. The appropriate skeleton 'Address' or 'Result Operation' is then generated. If the skeleton 'Result Operation' could later be replaced by a 'Take Label' operation the operation *DUMMY* is generated.

(*ii*) If a used entry is found ($x = 2$) the value of the *np* column is preserved in *x*, to be given as a parameter to the skeleton operation which is generated. The *type* column is checked using COMPARE, and if $p = 1$, the *dim* column is checked using the subroutine DIM.

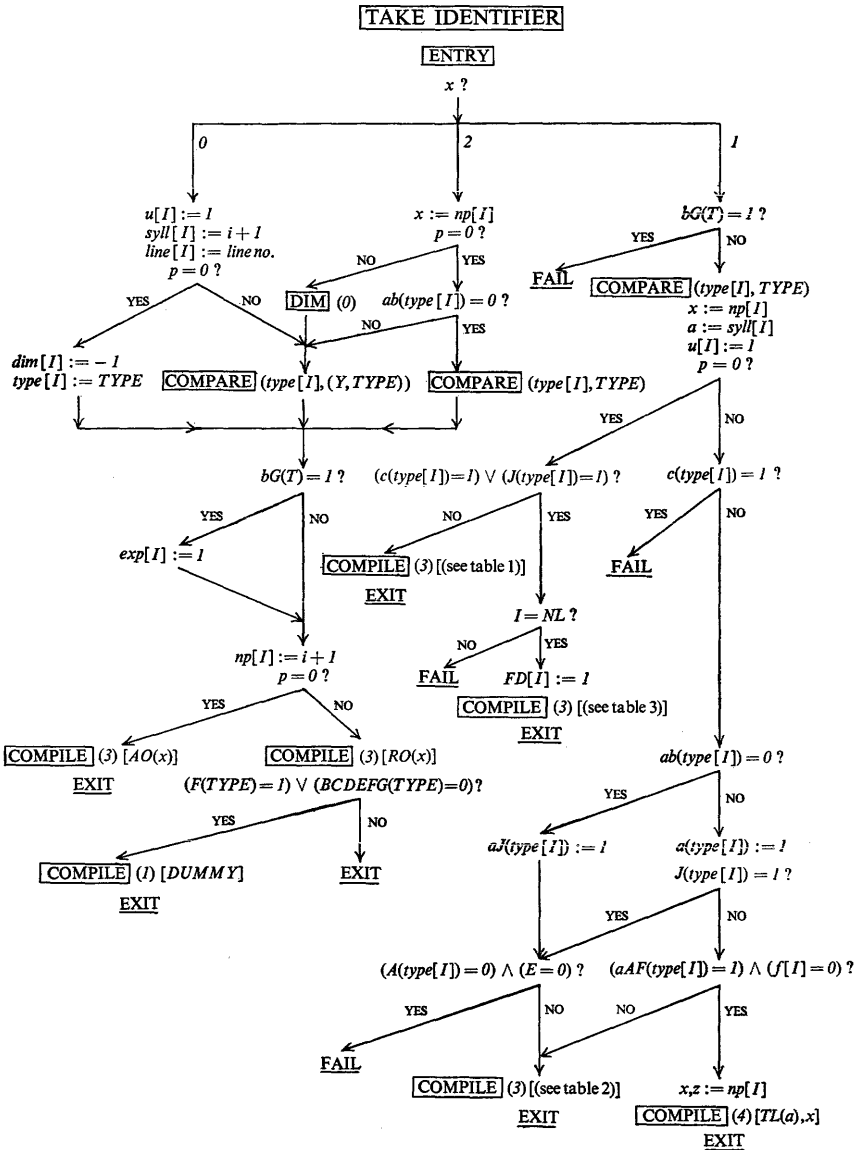
(*iii*) If a declared entry is found ($x = 1$) a check is made that the identifier is not being used in an array declaration. The *type* column of the entry is checked, and the *u* column is set. If $p = 0$ the *type* column is inspected to decide whether an assignment is being made to a procedure identifier. If this is the case the name list entry should be the duplicate entry for the procedure identifier (the first entry of the current part of the name list); the *FD* column is set and the appropriate 'Take Address' operation is generated, using Table 3 (see page 398). Otherwise Table 1 is used to decide which 'Take Address' operation should be generated. If $p = 1$ a check is made that the identifier has not been declared to be a procedure with parameters, or a procedure which cannot be used by a function designator. Then either a 'Take Label' operation, or a 'Take Result' operation (from Table 2) is generated, using the local variables *a* and *z*.

Table 1

type column	*f* column	operation
aABCG	*1*	*TFAR* (*x*)
aABCG	*0*	*TRA* (*x*)
aABCEG	*1*	*TFAI* (*x*)
aABCEG	*0*	*TIA* (*x*)
aABDG	*1*	*TFA* (*x*)
aABDG	*0*	*TBA* (*x*)
bABCG	*1*	*TFA* (*x*)
bABCG	*0*	*TRA* (*x*)
bABCEG	*1*	*TFA* (*x*)
bABCEG	*0*	*TIA* (*x*)
bABDG	*1*	*TFA* (*x*)
bABDG	*0*	*TBA* (*x*)
bAF	*1*	*TFA* (*x*)
bAF	*0*	*TSA* (*a*)

Table 2

type column	*f* column	operation
aABCG	*1*	*TFR* (*x*)
aABCG	*0*	*TRR* (*x*)
aABCEG	*1*	*TFI* (*x*)
aABCEG	*0*	*TIR* (*x*)
aABDG	*1*	*TFB* (*x*)
aABDG	*0*	*TBR* (*x*)
aAF	*1*	*TFL* (*x*)
aGJ	*1*	*CFFZ* (*x*)
aGJ	*0*	*CFZ* (*a*)
aABCGJ	*1*	*CFFZ* (*x*)
aABCGJ	*0*	*CFZ* (*a*)
aABCEGJ	*1*	*CFFZ* (*x*)
aABCEGJ	*0*	*CFZ* (*a*)
aABDGJ	*1*	*CFFZ* (*x*)
aABDGJ	*0*	*CFZ* (*a*)


TAKE IDENTIFIER
ENTRY
x ?
0
2
1
u[I] := 1
syll[I] := i + 1
line[I] := line no.
p = 0 ?
x := np[I]
p = 0 ?
bG(T) = 1 ?
YES
NO
FAIL
COMPARE (type[I], TYPE)
x := np[I]
a := syll[I]
u[I] := 1
p = 0 ?
DIM (0)
ab(type[I]) = 0 ?
dim[I] := -1
type[I] := TYPE
COMPARE (type[I], (Y, TYPE))
COMPARE (type[I], TYPE)
bG(T) = 1 ?
(c(type[I]) = 1) ∨ (J(type[I]) = 1) ?
c(type[I]) = 1 ?
exp[I] := 1
COMPILE (3) [(see table 1)]
EXIT
FAIL
I = NL ?
np[I] := i + 1
p = 0 ?
FAIL
FD[I] := 1
COMPILE (3) [(see table 3)]
EXIT
COMPILE (3) [AO(x)]
EXIT
COMPILE (3) [RO(x)]
(F(TYPE) = 1) ∨ (BCDEFG(TYPE) = 0) ?
ab(type[I]) = 0 ?
COMPILE (1) [DUMMY]
EXIT
EXIT
aJ(type[I]) := 1
a(type[I]) := 1
J(type[I]) = 1 ?
(A(type[I]) = 0) ∧ (E = 0) ?
(aAF(type[I]) = 1) ∧ (f[I] = 0) ?
FAIL
COMPILE (3) [(see table 2)]
EXIT
x,z := np[I]
COMPILE (4) [TL(a), x]
EXIT

TAKE IDENTIFIER (continued from p. 396)

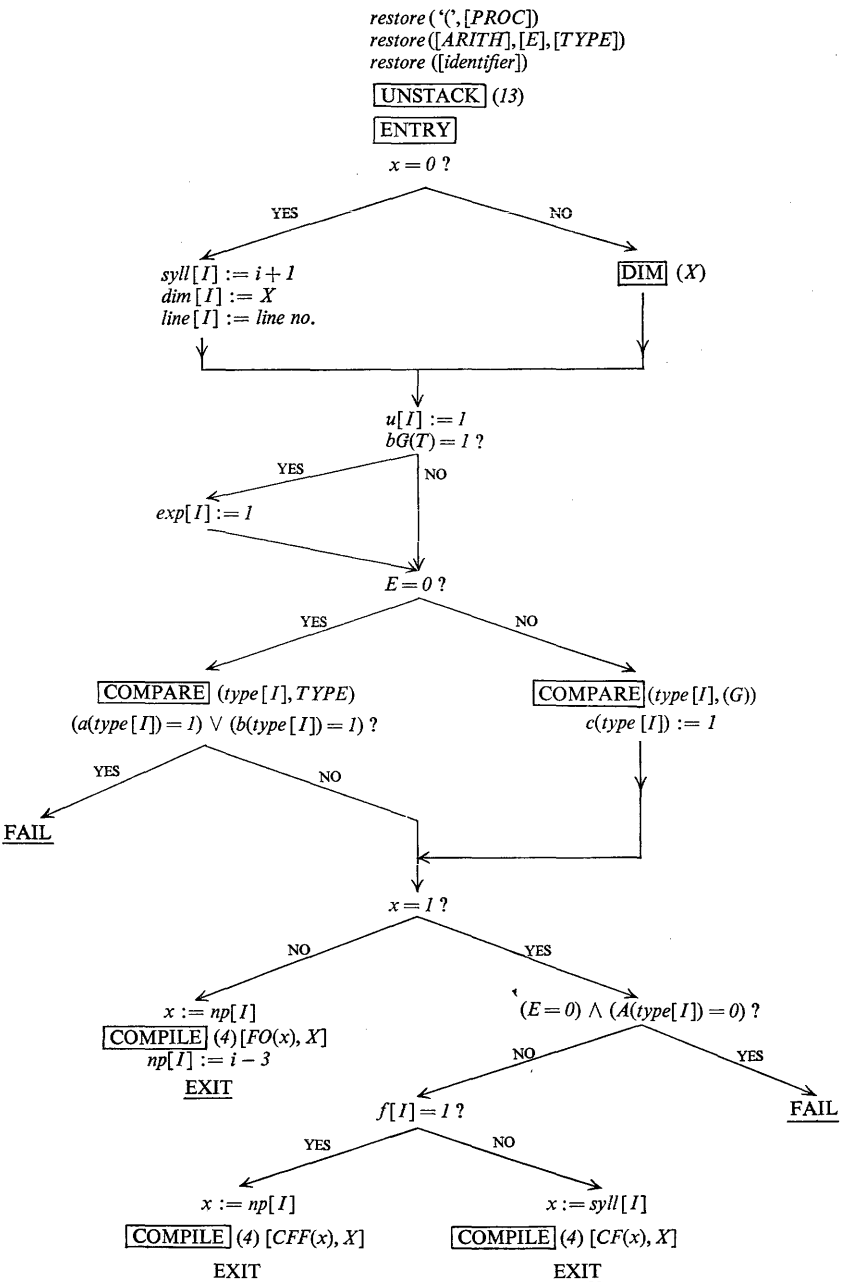
Table 3

type column	operation
cABCG	*TRA* (*x*)
aABCGJ	*TRA* (*x*)
cABCEG	*TIA* (*x*)
aABCEGJ	*TIA* (*x*)
cABDG	*TBA* (*x*)
aABDGJ	*TBA* (*x*)

PROC CALL

This subroutine is used by the routine for the delimiter ')'. The corresponding opening bracket is unstacked, and the state variables *PROC*, *ARITH*, *E* and *TYPE* are reset from the stacked values. The procedure identifier is also unstacked and the subroutine UNSTACK is used to complete the *UJ* operation to the 'Call Function' operation. The subroutine ENTRY is used to search the current part of the name list for an entry corresponding to the procedure identifier. If a new entry has been created its *syll*, *dim* and *line* columns are set up, otherwise its *dim* column is checked. Then the *u* column is set and, if the procedure call occurs in an array declaration, the *exp* column is also set. The *type* column is checked and the appropriate 'Call Function' operation is generated. A failure is indicated if a procedure which cannot be used by means of a function designator occurs when the state variable *E* is set to expression level.

PROC CALL


restore ('(', [PROC])
restore ([ARITH], [E], [TYPE])
restore ([identifier])
UNSTACK (13)
ENTRY
x = 0 ?
YES
NO
syll[I] := i + 1
dim[I] := X
line[I] := line no.
DIM (X)
u[I] := 1
bG(T) = 1 ?
YES
NO
exp[I] := 1
E = 0 ?
YES
NO
COMPARE (type[I], TYPE)
(a(type[I]) = 1) ∨ (b(type[I]) = 1) ?
COMPARE (type[I], (G))
c(type[I]) := 1
YES
NO
FAIL
x = 1 ?
NO
YES
x := np[I]
COMPILE (4) [FO(x), X]
np[I] := i - 3
EXIT
(E = 0) ∧ (A(type[I]) = 0) ?
NO
YES
f[I] = 1 ?
FAIL
YES
NO
x := np[I]
COMPILE (4) [CFF(x), X]
EXIT
x := syll[I]
COMPILE (4) [CF(x), X]
EXIT

ACT OP

This subroutine is used to generate the appropriate 'actual operation' if an actual parameter is a constant or an identifier, or to complete the processing of an implicit subroutine.

If the state variable *E* is set to expression level, the subroutines TAKE and UNSTACK are used, after setting the state variable *TYPE* if necessary, to complete the implicit subroutine. Otherwise the last delimiter is inspected; this should be ']' (in which case the actual parameter is a subscripted variable), comma used as a parameter delimiter or '('. If the current section contains a constant the appropriate 'actual operation' is generated. If the current section contains an identifier the subroutine ENTRY is used to search the current part of the name list for an entry corresponding to this identifier.

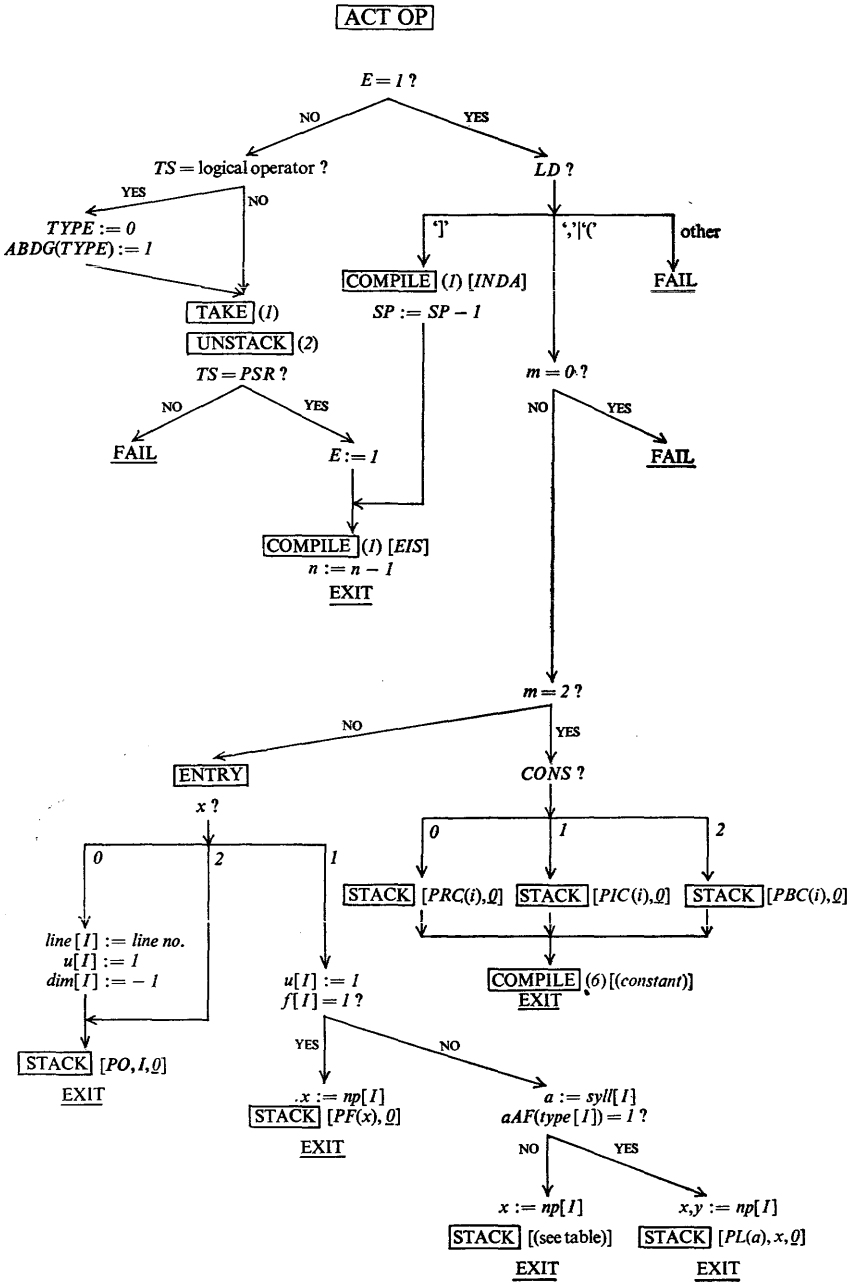
(*i*) If a new entry is created the *line* and *u* columns of the entry are set, and '-1' placed in the *dim* column. The skeleton operation *PO* is stacked together with *I*, the number of the name list entry for this identifier.

(*ii*) If a used entry is found the skeleton operation *PO* is stacked as in (*i*) above.

(*iii*) If a declared entry is found the *u* column is set and the appropriate 'actual operation' is generated, using the local variables *a* and *y*.

Table

type column	item to be stacked
aABCG	*PR* (*x*), 0
aABCEG	*PI* (*x*), 0
aABDG	*PB* (*x*), 0
bAF	*PSW* (*a*), 0
bABCG	*PRA* (*x*), 0
bABCEG	*PIA* (*x*), 0
bABDG	*PBA* (*x*), 0
G	*PPR* (*a*), 0
aGJ	*PPR* (*a*), 0
cG	*PPR* (*a*), 0
ABCG	*PFR* (*a*), 0
aABCGJ	*PFR* (*a*), 0
cABCG	*PFR* (*a*), 0
ABCEG	*PFI* (*a*), 0
aABCEGJ	*PFI* (*a*), 0
cABCEG	*PFI* (*a*), 0
ABDG	*PFB* (*a*), 0
aABDGJ	*PFB* (*a*), 0
cABDG	*PFB* (*a*), 0


ACT OP
E = 1 ?
NO
YES
TS = logical operator ?
LD ?
YES
NO
TYPE := 0
ABDG(TYPE) := 1
']'
';'|'('
other
COMPILE (I) [INDA]
FAIL
TAKE (I)
SP := SP − 1
UNSTACK (2)
TS = PSR ?
m = 0 ?
NO
YES
FAIL
E := 1
FAIL
COMPILE (I) [EIS]
n := n − 1
EXIT
m = 2 ?
NO
YES
ENTRY
CONS ?
x ?
0
1
2
STACK [PRC(i),0]
STACK [PIC(i),0]
STACK [PBC(i),0]
0
2
1
line [I] := line no.
u[I] := 1
dim[I] := − 1
u[I] := 1
f[I] = 1 ?
COMPILE (6) [(constant)]
EXIT
STACK [PO,I,0]
EXIT
YES
NO
.x := np[I]
STACK [PF(x),0]
EXIT
a := syll[I]
aAF(type[I]) = 1 ?
NO
YES
x := np[I]
STACK [(see table)]
EXIT
x,y := np[I]
STACK [PL(a),x,0]
EXIT

FOR ','

This subroutine uses the subroutines TAKE and UNSTACK to complete the processing of either the preceding for list element or of a subscript expression in a subscripted variable used in a for list element. The top of the stack is then inspected.

(*i*) If the top stack item is '[' the current delimiter is checked and the dimension counter stacked with the item '[' is increased by one, using the local variable *y*. A return is then made direct to the central loop of the Translator.

(*ii*) If the top stack item is *FORW* or *FORS2*, it is unstacked; the top of the stack should then contain the item **for**.

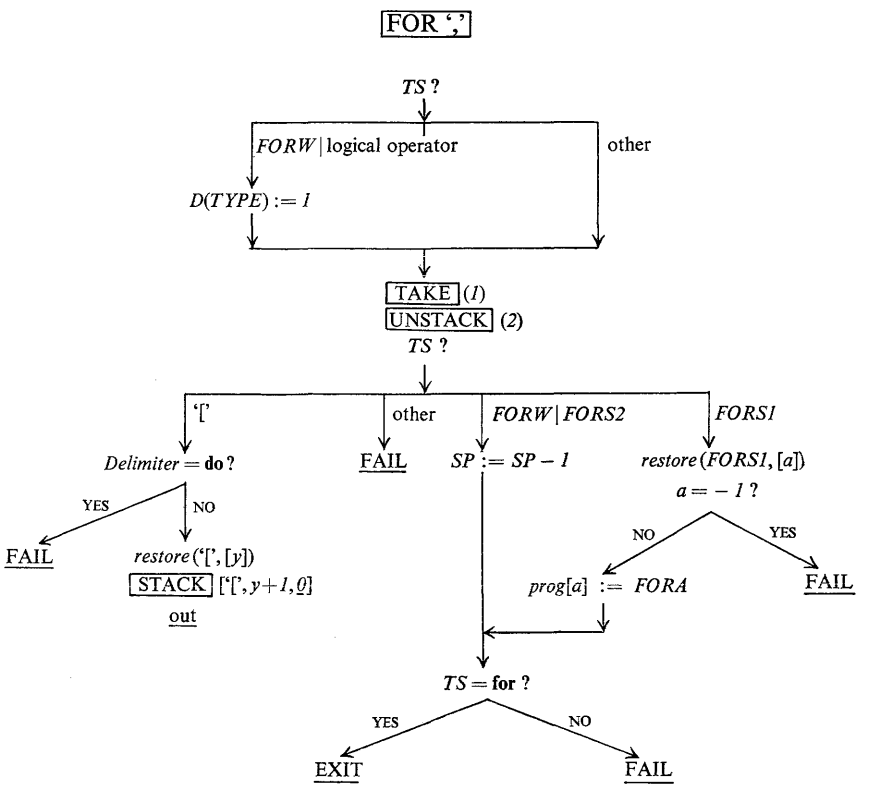
(*iii*) If the top stack item is *FORS1* the value stored with it is inspected, using the local variable *a*. If it is '—*1*' a failure is indicated since the delimiter **step** has not been followed by the delimiter **until**. Otherwise the object program operation at the address *a* is changed from *FORS1* to *FORA*.

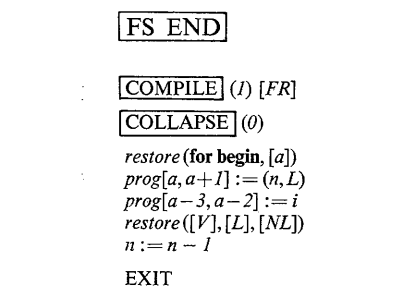
FS END

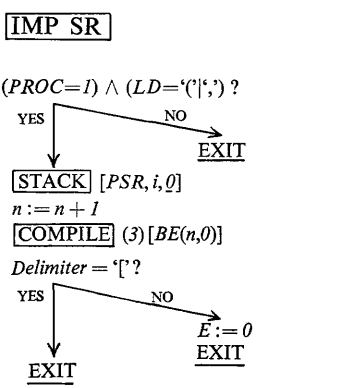
This subroutine is used to complete the processing of a for block, and uses COLLAPSE to deal with the name list entries for this block. The address stored with the item **'for begin'** at the top of the stack is used to complete the parameters of the *FBE* and *FSE* operations, using the local variable *a*.

IMP SR

This subroutine is used at any delimiter which could start an implicit subroutine. If the state variables *LD* and *PROC* show that the delimiter does start an implicit subroutine the item *PSR* is stacked and a block is set up. Unless the current delimiter is '[' the state variable *E* is set to expression level.


FOR ';'
TS ?
FORW | logical operator
other
D(TYPE) := 1
TAKE (1)
UNSTACK (2)
TS ?
'['
other
FORW | FORS2
FORS1
Delimiter = do ?
FAIL
SP := SP - 1
restore (FORS1, [a])
a = - 1 ?
YES
NO
FAIL
restore ('[', [y])
STACK ['[', y+1, 0]
out
NO
YES
prog[a] := FORA
FAIL
TS = for ?
YES
NO
EXIT
FAIL


FS END
COMPILE (1) [FR]
COLLAPSE (0)
restore (for begin, [a])
prog[a, a+1] := (n, L)
prog[a-3, a-2] := i
restore ([V], [L], [NL])
n := n - 1
EXIT


IMP SR
(PROC=1) ∧ (LD='('|',') ?
YES
NO
EXIT
STACK [PSR, i, 0]
n := n + 1
COMPILE (3) [BE(n,0)]
Delimiter = '[' ?
YES
NO
E := 0
EXIT
EXIT

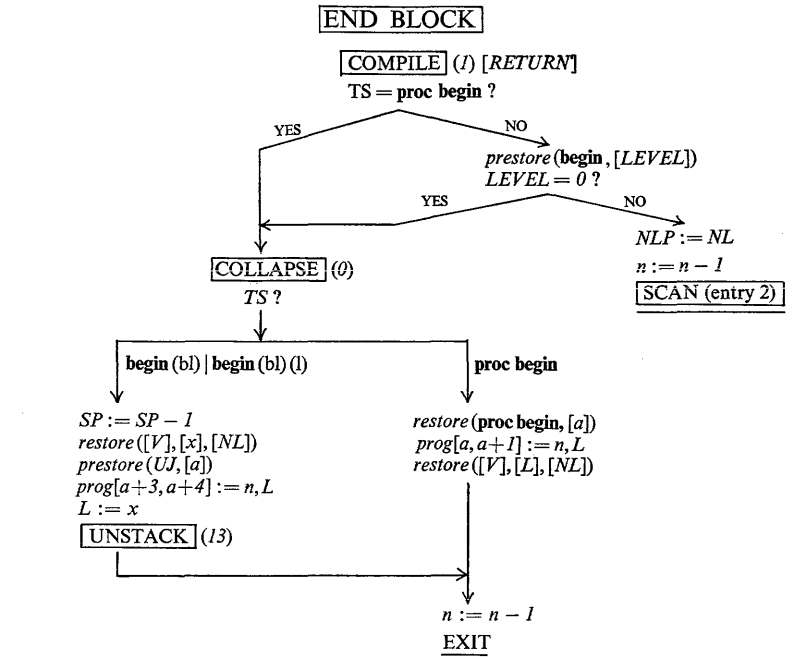
END BLOCK

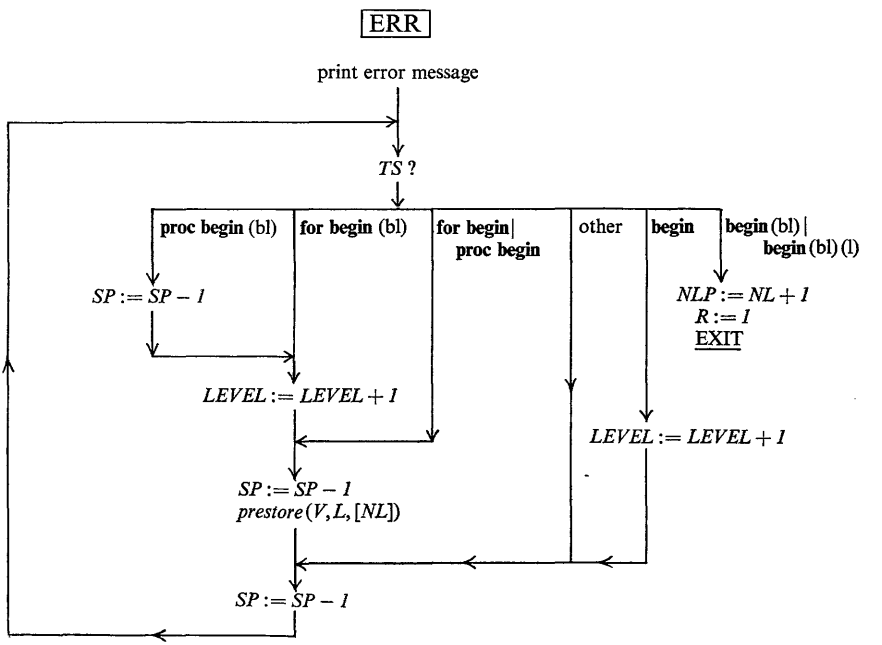
This subroutine is used to complete the translation of a block. If the top stack item is a **begin** which does not have a non-zero value of *LEVEL* stacked with it the name list entries for the current block are collapsed. Then the top stack item indicates whether a procedure block or an ordinary block is being completed. In the former case the address stored with **'proc begin'** is used to complete the parameter of the *PE* operation; in the latter case the top item is unstacked and the next two items are used to restore *V*, *L* and *NL*, and to complete the parameter of the *BE* operation, using the local variables *x* and *a*. The subroutine UNSTACK is used to complete the *UJ* operation around the block.

If, however, the item at the top of the stack has a non-zero value of *LEVEL* stacked with it the name list entries for the current block are discarded, as a return is being made to a block in which an error has been detected. In this case SCAN is entered, instead of returning to the routine from which this subroutine was called.

ERR

This subroutine is used by SCAN to print an error message and to unstack items until a **'begin** (bl)' item is reached, taking account of any values of *NL* stacked with **'proc begin'** or **'for begin'** items, in order to discard the appropriate name list entries. If the item **begin, 'proc begin** (bl)' or **'for begin** (bl)' is found, *LEVEL* is increased by one to allow for the corresponding delimiter **end.** Finally *R* is set to one so that no further error messages are printed whilst the current block is being scanned.


END BLOCK
COMPILE (1) [RETURN]
TS = proc begin ?
YES
NO
prestore (begin, [LEVEL])
LEVEL = 0 ?
YES
NO
NLP := NL
n := n − 1
SCAN (entry 2)
COLLAPSE (0)
TS ?
begin (bl) | begin (bl) (l)
proc begin
SP := SP − 1
restore ([V], [x], [NL])
prestore (UJ, [a])
prog[a+3, a+4] := n, L
L := x
UNSTACK (13)
restore (proc begin, [a])
prog[a, a+1] := n, L
restore ([V], [L], [NL])
n := n − 1
EXIT


ERR
print error message
TS ?
proc begin (bl)
for begin (bl)
for begin | proc begin
other
begin
begin (bl) | begin (bl) (l)
SP := SP − 1
NLP := NL + 1
R := 1
EXIT
LEVEL := LEVEL + 1
LEVEL := LEVEL + 1
SP := SP − 1
prestore (V, L, [NL])
SP := SP − 1

COLLAPSE

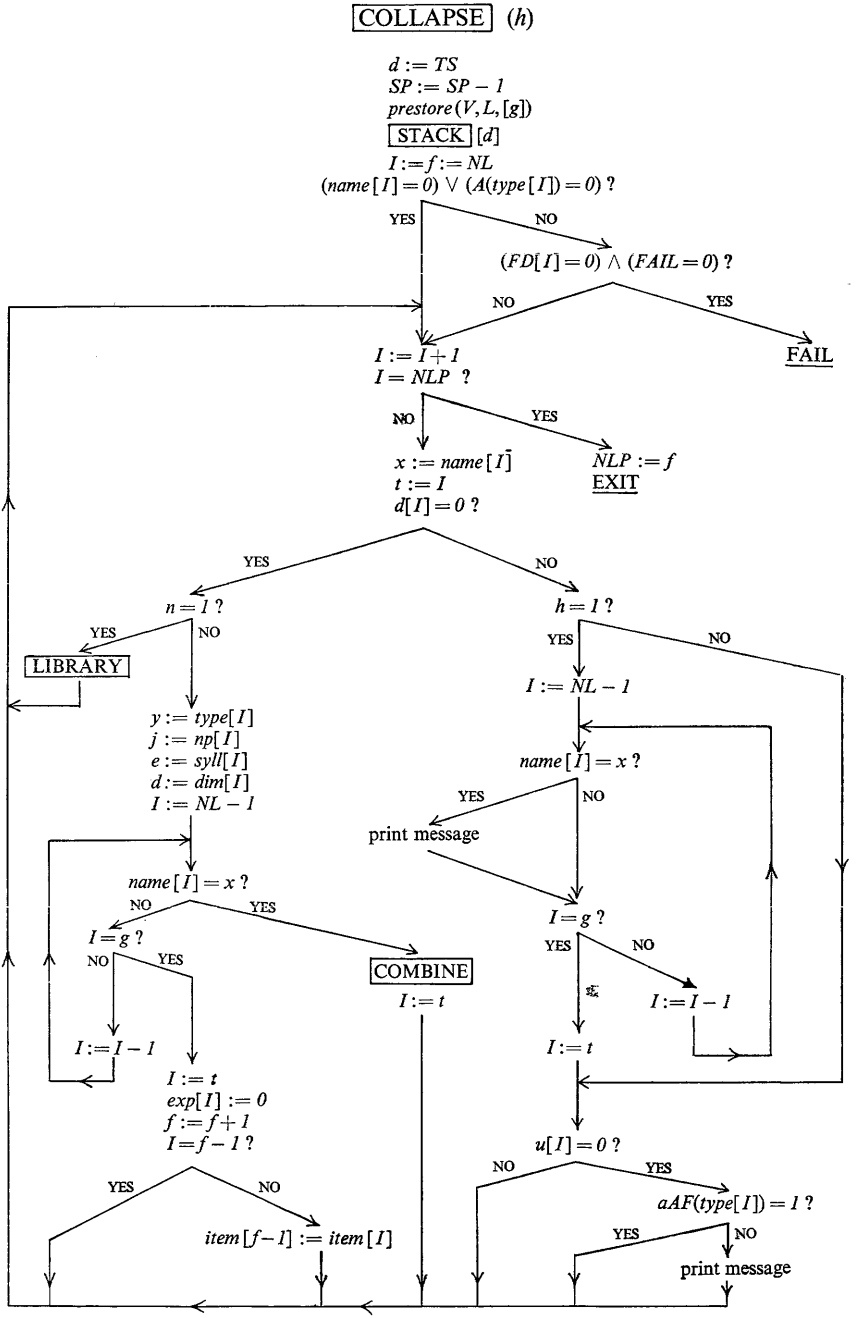
This subroutine is used at the end of a block to deal with its name list entries.

The entries in the current part of the name list are examined in turn, starting with the first entry, which can either be a blank, which is ignored, or the duplicate entry for a procedure identifier, which is ignored after checking that its *FD* column has been set in the case of a type procedure (this check is avoided if an error has been found earlier in the ALGOL program). Thereafter any declared entries are discarded, but warning messages are printed if an identifier (other than a label) has not been used, or if a formal parameter has been rendered inaccessible. This latter check is carried out, by searching the name list for the containing block, whenever the parameter *h* of the subroutine has the value one, indicating that the containing block is a procedure block.

When a used entry is found, the block level counter *n* is checked, and if this indicates that the program block is being collapsed, the subroutine LIBRARY (for which no flow diagram is given) is used to check that the identifier is one of the standard functions. Otherwise the name list for the containing block is searched for an entry corresponding to the same identifier; if an entry is found the subroutine COMBINE is used, otherwise the used entry is added to the list of entries for the containing block.

The local variables used in this subroutine are

d	used to preserve the top stack item, and then to contain the *dim* column of each used entry in the current block, for use by the subroutine COMBINE
g	set up with the value of *NL* for the containing block
f	used to indicate the extent of the name list of the containing block, and is increased by one for each entry transferred from the current to the containing block
t, *x*, *y*, *j*, *e*	set up with the entry number, and the contents of the *name*, *type*, *np* and *syll* columns, respectively, of each used entry in the current block, for use by COMBINE.


COLLAPSE (h)
d := TS
SP := SP − 1
prestore(V, L, [g])
STACK [d]
I := f := NL
(name[I] = 0) ∨ (A(type[I]) = 0) ?
YES
NO
(FD[I] = 0) ∧ (FAIL = 0) ?
NO
YES
FAIL
I := I + 1
I = NLP ?
NO
YES
x := name[I]
t := I
d[I] = 0 ?
NLP := f
EXIT
YES
NO
n = 1 ?
YES
NO
LIBRARY
y := type[I]
j := np[I]
e := syll[I]
d := dim[I]
I := NL − 1
name[I] = x ?
NO
YES
I = g ?
NO
YES
I := I − 1
I := t
exp[I] := 0
f := f + 1
I = f − 1 ?
YES
NO
item[f − 1] := item[I]
COMBINE
I := t
h = 1 ?
YES
NO
I := NL − 1
name[I] = x ?
YES
NO
print message
I = g ?
YES
NO
I := I − 1
I := t
u[I] = 0 ?
NO
YES
aAF(type[I]) = 1 ?
YES
NO
print message

COMBINE

This subroutine, used by COLLAPSE, compares the *type* and *dim* columns of the two corresponding name list entries in the current and containing block, and then inspects the *d* column of the entry for the containing block.

(*i*) If the *d* column shows that the entry is a declared entry, its *u* column is set, and the subroutine UNCHAIN is used to unchain the skeleton operations corresponding to the used entry in the current block.

(*ii*) If the entry in the containing block is a used entry the chains of skeleton operations associated with each used entry are combined into a single chain.

UNCHAIN

This subroutine traces through the chain of skeleton operations associated with a used entry in the name list, and calls the appropriate subroutine to process each skeleton operation. The parameter *x* (which is used in each of the inner subroutines) contains the address of the first skeleton operation in the chain; the local variable *e* is used to store the addresses of any further skeleton operations. The parameter *y* indicates whether UNCHAIN has been called by the subroutine DECL or COMBINE.

ADDRESS

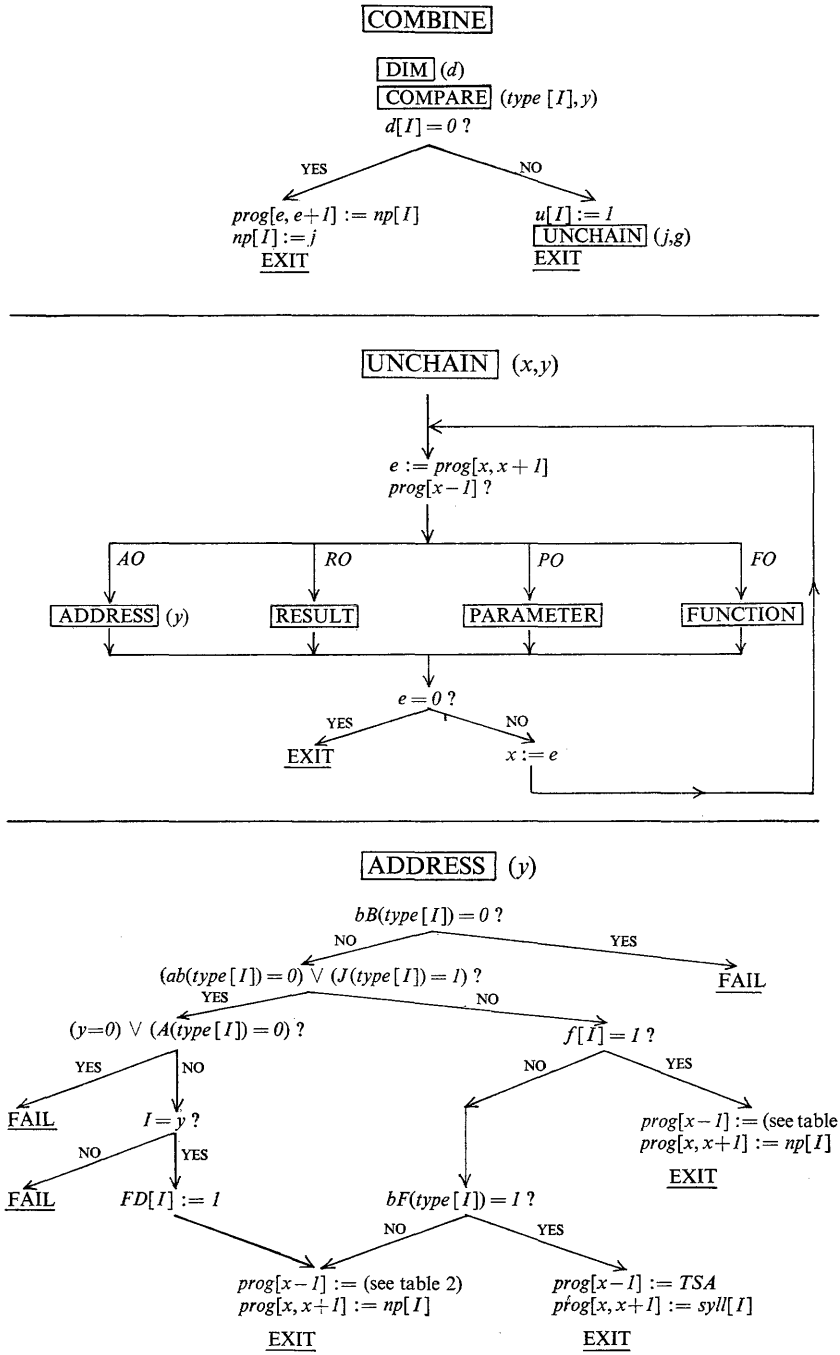
This subroutine is used to replace a skeleton 'Address Operation'. Checks are made that a 'Take Address' operation which corresponds to an assignment to a procedure identifier only occurs within the procedure body of a procedure which is declared to be a type procedure. Table 1 is used to select the appropriate operation if the *f* column of the corresponding name list entry is set, otherwise either *TSA* is generated or Table 2 is used.

Table 1

type column	operation
aABCG	*TFAR*
aABCEG	*TFAI*
aABDG	*TFA*
bABCG	*TFA*
bABCEG	*TFA*
bABDG	*TFA*
bAF	*TFA*

Table 2

type column	operation
aABCG	*TRA*
aABCEG	*TIA*
aABDG	*TBA*
bABCG	*TRA*
bABCEG	*TIA*
bABDG	*TBA*
cABCG	*TRA*
cABCEG	*TIA*
cABDG	*TBA*
aABCGJ	*TRA*
aABCEGJ	*TIA*
aABDGJ	*TBA*


COMBINE
DIM (d)
COMPARE (type [I], y)
d[I] = 0 ?
YES
NO
prog[e, e+1] := np[I]
np[I] := j
EXIT
u[I] := 1
UNCHAIN (j,g)
EXIT
UNCHAIN (x,y)
e := prog[x, x + 1]
prog[x − 1] ?
AO
RO
PO
FO
ADDRESS (y)
RESULT
PARAMETER
FUNCTION
e = 0 ?
YES
NO
EXIT
x := e
ADDRESS (y)
bB(type [I]) = 0 ?
NO
YES
FAIL
(ab(type [I]) = 0) ∨ (J(type [I]) = 1) ?
YES
NO
(y=0) ∨ (A(type [I]) = 0) ?
f[I] = 1 ?
YES
NO
NO
YES
FAIL
I = y ?
prog[x − 1] := (see table 1)
prog[x, x+1] := np[I]
EXIT
NO
YES
FAIL
FD[I] := 1
bF(type [I]) = 1 ?
NO
YES
prog[x − 1] := (see table 2)
prog[x, x+1] := np[I]
EXIT
prog[x − 1] := TSA
prog[x, x+1] := syll[I]
EXIT

RESULT

This subroutine replaces a skeleton 'Result Operation' by the appropriate 'Take Result' or 'Call Function Zero' operation. For an identifier which is declared to be a procedure with no parameters a check is made, using the operation following the skeleton operation, that a non-type procedure is called by a procedure statement, rather than a function designator; then the skeleton operation is replaced by a *CFZ* or *CFFZ* operation. Otherwise the *type* and *f* columns are used to decide whether the operation *TL* is to be generated (using the local variables *a* and *z*), or whether Table 1 or Table 2 is to be used.

Table 1

type column	operation
aABCG	*TFR*
aABCEG	*TFI*
aABDG	*TFB*
aAF	*TFL*

Table 2

type column	operation
aABCG	*TRR*
aABCEG	*TIR*
aABDG	*TBR*

PARAMETER

This subroutine replaces a skeleton 'Parameter Operation' by the appropriate 'actual operation'. The *type* and *f* columns of the corresponding name list entry are used to decide whether a *PF* or *PL* operation is to be generated (using the local variables *a* and *z*) or whether the following table is to be used.

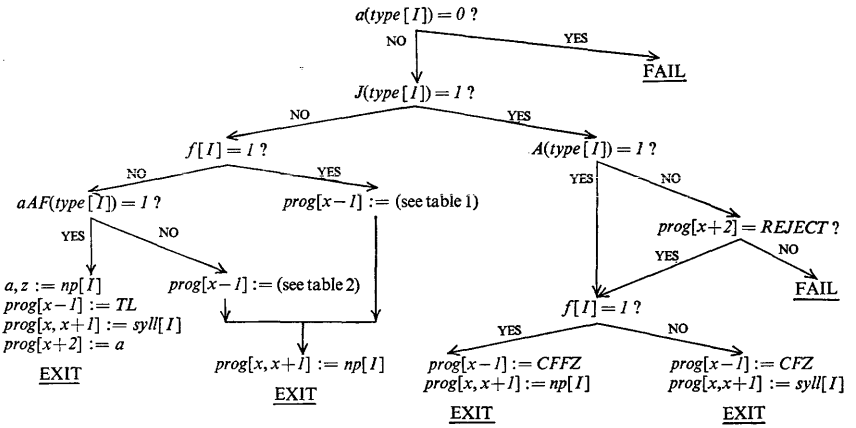
Table 3

type column	operation	*type* column	operation
aABCG	*PR*	*ABDG*	*PFB*
aABCEG	*PI*	*cABDG*	*PFB*
aABDG	*PB*	*G*	*PPR*
bABCG	*PRA*	*cG*	*PPR*
bABCEG	*PIA*	*bAF*	*PSW*
bABDG	*PBA*	*aABCGJ*	*PFR*
ABCG	*PFR*	*aABCEGJ*	*PFI*
cABCG	*PFR*	*aABDGJ*	*PFB*
ABCEG	*PFI*	*aGJ*	*PPR*
cABCEG	*PFI*		

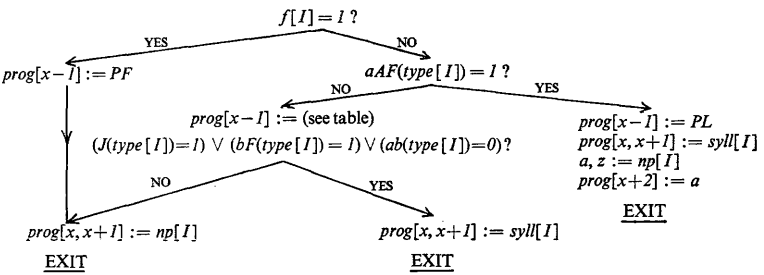
FUNCTION

This subroutine is used to replace a skeleton 'Function Operation' by either a *CF* or a *CFF* operation. A check is made that a non-type procedure is called by a procedure statement and not by a function designator.

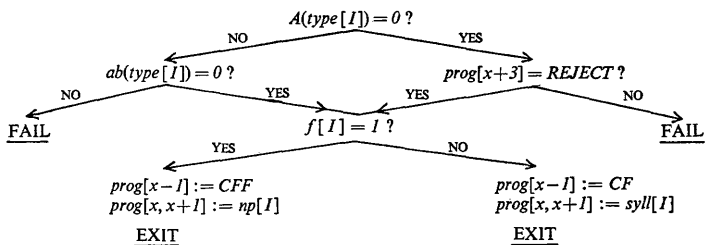
RESULT



PARAMETER



FUNCTION



SCAN

The normal entry to this subroutine is after an error has been found in the ALGOL text. The global variable *FAIL* is set and the subroutine ERR is used to print an error message and to discard stack and name list entries for the current block. The ALGOL text is then scanned until the end of the block or the start of an inner block is reached; the delimiters **begin** and **end** appearing within string quotes are ignored, using the local variable *s*.

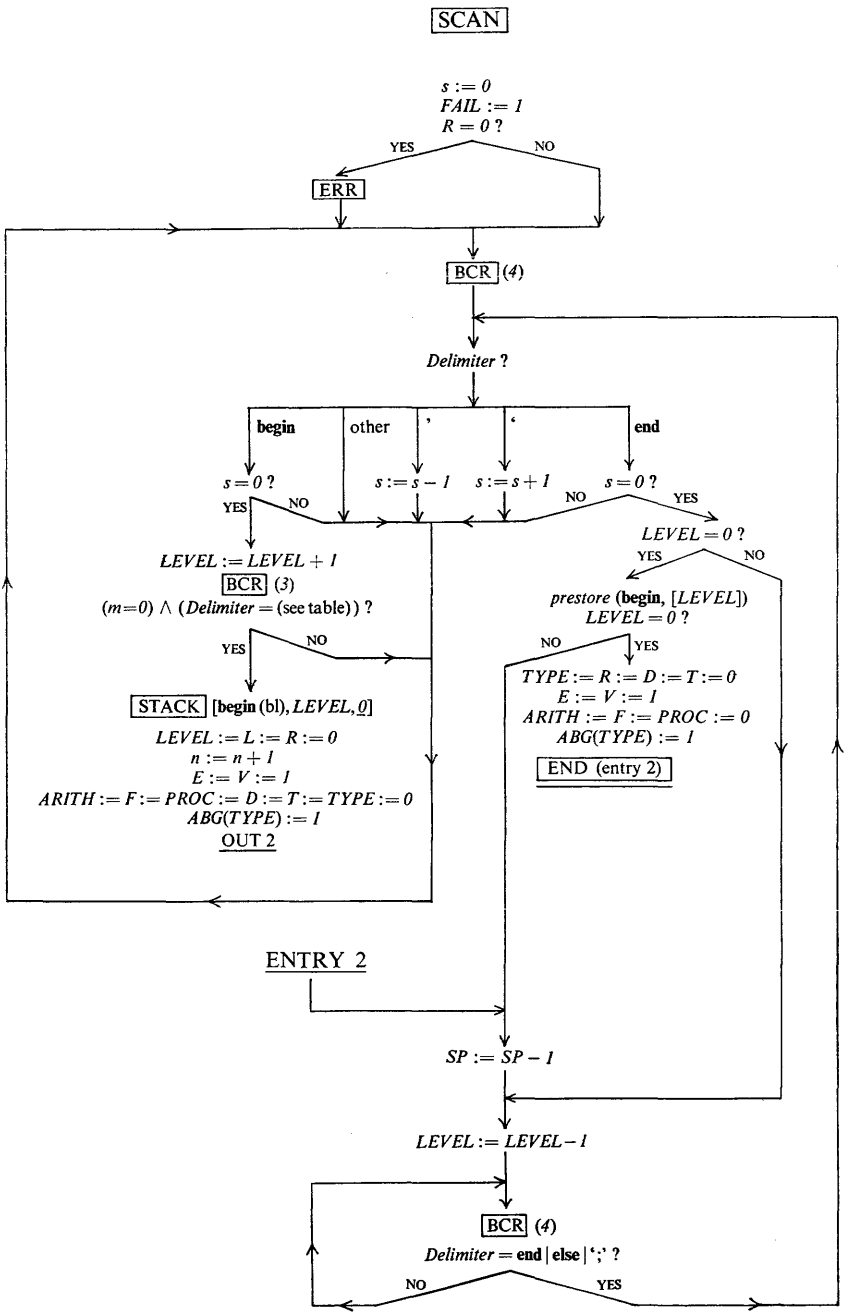
When a **begin** which starts a block or compound statement is reached, the counter *LEVEL* is increased by one and the next delimiter is checked. If this delimiter indicates that a compound statement has been reached the scan continues. Otherwise the item '**begin** (bl)' is stacked, together with the value of *LEVEL*, and, after setting up the state variables as for the start of a block, the block is processed in the normal way.

When the delimiter **end** is reached the value of *LEVEL* is checked. If it is non-zero it is decreased by one and scanning continues, after allowing for a possible **end** comment. Otherwise the end of the block containing the failure has been reached and the **begin** item is unstacked from the top of the stack and *LEVEL* is reset from the value stacked with the **begin**. If *LEVEL* is now non-zero, the containing block has also been found to contain an error, and so *LEVEL* is decreased by one and scanning continues; otherwise the state variables are set up as for the start of a block, and the **end** routine is entered (the second entry is used because there is no need to complete the processing of a preceding statement).

After completing the processing of any inner blocks a return is made to SCAN via its second entry, to continue the scanning of the block which contains an error. If an error is found by the BCR subroutine whilst scanning the ALGOL text, the subroutine ERR is avoided since the global variable *R* is non-zero.

Table

own
real
integer
Boolean
array
procedure
switch


SCAN
s := 0
FAIL := 1
R = 0 ?
YES
NO
ERR
BCR (4)
Delimiter ?
begin
other
'
'
end
s = 0 ?
YES
NO
s := s − 1
s := s + 1
NO
s = 0 ?
YES
LEVEL := LEVEL + 1
BCR (3)
(m=0) ∧ (Delimiter = (see table)) ?
YES
NO
STACK [begin (bl), LEVEL, 0]
LEVEL := L := R := 0
n := n + 1
E := V := 1
ARITH := F := PROC := D := T := TYPE := 0
ABG(TYPE) := 1
OUT 2
LEVEL = 0 ?
YES
NO
prestore (begin, [LEVEL])
LEVEL = 0 ?
NO
YES
TYPE := R := D := T := 0
E := V := 1
ARITH := F := PROC := 0
ABG(TYPE) := 1
END (entry 2)
ENTRY 2
SP := SP − 1
LEVEL := LEVEL − 1
BCR (4)
Delimiter = end | else | ';' ?
NO
YES

Index