

LOUGHBOROUGH  
UNIVERSITY OF TECHNOLOGY  
LIBRARY

AUTHOR/FILING TITLE

WILLIAMS, S

ACCESSION/COPY NO.

152080/01

VOL. NO.

CLASS MARK

ARCHIVES  
COPY

FOR REFERENCE ONLY



APPROACHES TO THE DETERMINATION OF PARALLELISM  
IN COMPUTER PROGRAMS

by

Shirley Ann Williams (*née Smith*)

A Doctoral Thesis

Submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

August, 1978.

Supervisor: Professor D.J. Evans, Ph.D., D.Sc.  
Department of Computer Studies.

Loughborough University  
of Technology Library

Date Oct. 78

Class

Acc. No. 152060/01

## ACKNOWLEDGEMENTS

The author would like to express her appreciation and thanks to Professor D.J. Evans for providing motivation and assistance throughout her academic training; to Dr. D.J. Cooke for his help in understanding the problems of correctness; to Miss J.M. Briers for her dexterous typing and preparation of diagrams for this thesis; and to the Science Research Council for the provision of a Research Studentship.

Finally, many thanks must go to Roy, the author's husband, and her parents for their help and support during the 'ups and downs' of this work.

## DECLARATION

I declare that the following thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

S.A. WILLIAMS.

# CONTENTS

	<u>PAGE</u>
<u>CHAPTER 1:</u> INTRODUCTION TO PARALLELISM	
1.1 General Description .. .. .	1
1.2 Architecture .. .. .	3
1.3 Levels of Parallelism .. .. .	9
1.4 Explicit and Implicit Parallelism .. ..	10
<u>CHAPTER 2:</u> SOFTWARE CONCEPTS AMENABLE TO PARALLEL PROCESSING	
2.1 Systems Software .. .. .	14
2.2 Compilation Processes .. .. .	15
2.3 Parsing an Expression .. .. .	18
2.3.1 Reverse Polish Notation .. .. .	18
2.3.2 Tree Representation .. .. .	19
2.4 Algol-Type Programming Languages .. .. .	26
2.5 Usage of Language Constructs .. .. .	28
<u>CHAPTER 3:</u> DETECTION OF POTENTIAL PARALLELISM AT THE INSTRUCTION LEVEL	
3.1 Tree Representations of Expressions .. ..	30
3.2 A Survey of Techniques for Recognising Expression Parallelism .. .. .	33
3.2.1 Squire's Algorithm .. .. .	33
3.2.2 Hellerman's Algorithm .. .. .	38
3.2.3 Stone's Algorithm .. .. .	38
3.2.4 Baer and Bovet's Algorithm .. ..	39
3.2.5 Other Methods for Recognising Parallelism Within Expressions ..	48
3.3 Formation of a Balanced Binary Tree .. ..	52
3.4 A New Algorithm .. .. .	57
3.4.1 The Basic Algorithm .. .. .	57
3.4.2 Extensions to the Basic Algorithm	60
3.5 A Comparison of Algorithms for Recognising Expression Parallelism ..	66





CHAPTER 6: OPTIMISATION OF PARALLEL PROGRAMS

6.1	Optimisation Techniques .. . . . .	133
6.2	Optimisation Techniques Readily Amenable to Parallel Processing .. . .	135
6.2.1	Procedure Integration .. . . . .	135
6.2.2	Constant Folding and Dead Code Elimination .. . . . .	135
6.2.3	Peephole Transformations .. . . . .	136
6.3	Optimisation Techniques That Detract from Potential Parallelism .. . . . .	138
6.3.1	Common Subexpression Elimination ..	138
6.3.2	Strength Reduction and Linear Function Test Replacement .. . . . .	138
6.4	Loop Transformations .. . . . .	141
6.4.1	Loop Unrolling .. . . . .	141
6.4.2	Loop Unfolding .. . . . .	141
6.4.3	Loop Folding.. . . . .	142
6.4.4	Combinations of Loop Transformations .. . . . .	142

CHAPTER 7: CORRECTNESS OF PARALLEL PROGRAMS

7.1	Introduction to Program Correctness .. . .	146
7.2	Symbolic Execution of Programs .. . . . .	148
7.2.1	Symbolic Execution of Sequential Program Statements .. . . . .	149
7.2.2	Symbolic Execution of a Conditional .. . . . .	149
7.2.3	Parallel Symbolic Execution .. . . . .	150
7.3	Tests to Determine the Consequents of Parallel Programs .. . . . .	158
7.3.1	Contemporary.. . . . .	158
7.3.2	Commutative .. . . . .	159
7.3.3	Prerequisite.. . . . .	160
7.3.4	Conservative.. . . . .	160
7.3.5	Consecutive .. . . . .	161
7.3.6	Synchronous .. . . . .	161
7.3.7	Inclusive.. . . . .	162

7.4	Example of Proving Correctness Between Two Stanzas .. .. .	164
7.4.1	Contemporary .. .. .	165
7.4.2	Commutative.. .. .	166
7.4.3	Prerequisite .. .. .	167
7.4.4	Conservative .. .. .	167
7.4.5	Consecutive .. .. .	168
7.4.6	Synchronous.. .. .	168
7.4.7	Inclusive .. .. .	169
7.5	Proving Correctness Between a Number of Stanzas .. .. .	179

## CHAPTER 8: CONCLUSIONS

8.1	Summary .. .. .	180
8.2	Detection of Implicit Parallelism and Correctness of Parallel Programs .. ..	181
8.3	Other Applications of Parallel Processing .. .. .	183
8.4	Areas for Further Research .. .. .	184
8.4.1	Automatic Stanza Formation .. ..	184
8.4.2	Detection of Parallelism Between Stanzas .. .. .	184
8.4.3	Expression Parallelism .. .. .	185
8.4.4	Termination of a Correct Parallel Program .. .. .	185
8.4.5	Explicit Parallelism .. .. .	185

REFERENCES	187
------------	-----

<u>APPENDIX 1:</u> Algorithm for Constructing a Balanced Binary Tree	196
<u>APPENDIX 2:</u> Analyser	199
<u>APPENDIX 3:</u> Detector	217
<u>APPENDIX 4:</u> Sample Program	241

CHAPTER 1

INTRODUCTION TO PARALLELISM

## 1.1 GENERAL DESCRIPTION

The elapsed time taken to execute a given set of programs, or the speed of throughput on a particular computer, will be dependent on three factors. One is the hardware of the computer (or as it is often called, the machine) being considered, this will depend on such things as switching times and distances between the components (Stone, 1975) and there are physical limitations on these. The other two are the organisation of the machine's logic and the organisation of the programs under consideration. These two areas are where improved speed of throughput must be sought, given the physical constraints on 'speeding-up' the hardware.

For serial computers (i.e. ones with only one main processing unit) there has been a great deal of work carried out concerning the compilation of programs (e.g. Hopgood, 1969) so as to decrease the elapsed time taken for the execution of a program. If however, machines are available which logically have more than one processor, then by sharing parts of a set of programs between these processors it may be possible to further decrease the elapsed time taken for the execution of these programs.

The term 'parallel processing' will be used to indicate the execution of several 'tasks' at the same time on different processors or processing units, see Figure 1.1. A 'task' is some part of a program ranging from within a micro-instruction to whole programs. Depending upon the type of processors available, the part of program to be considered will vary. For example, a program may be best divided such that parts of each arithmetic statement may be assigned to separate arithmetic processors.

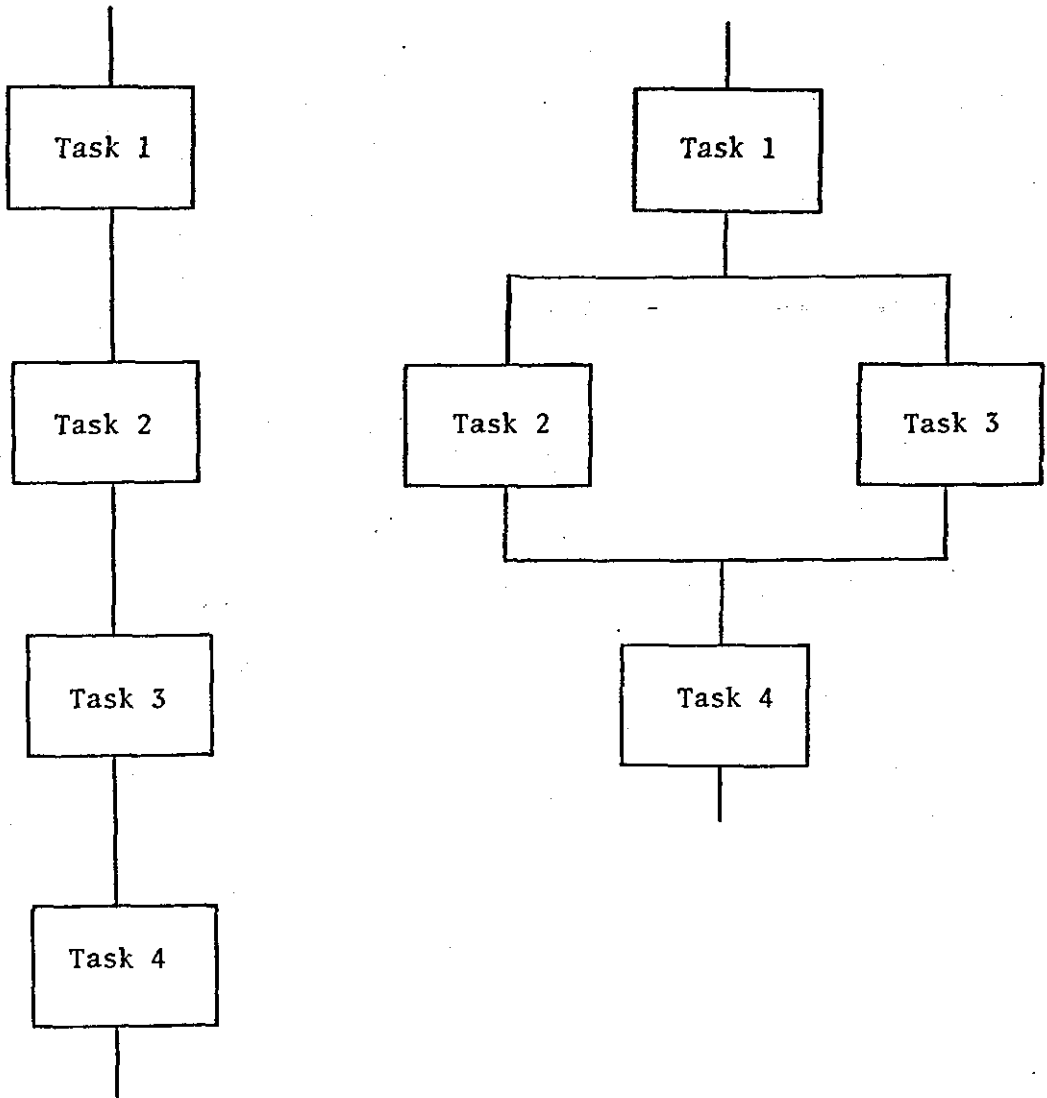


Figure 1.1

SEQUENTIAL AND PARALLEL EXECUTION OF TWO TASKS  $T_2$  AND  $T_3$

## 1.2 ARCHITECTURE

Stone (1975) describes in detail the four classes that Flynn (1966) defined computer systems will fall into. These are:-

- (i) 'Single Instruction Stream-Single Data Stream' (SISD) computer.
- (ii) 'Single Instruction Stream-Multiple Data Stream' (SIMD) computer.
- (iii) 'Multiple Instruction Stream-Single Data Stream' (MISD) computer.
- (iv) 'Multiple Instruction Stream-Multiple Data Stream' (MIMD) computer.

The SISD computer is the serial computer mentioned above, where there is at the most only one instruction in execution, at any one time, and this affects at the most one item of data, see Figure 1.2. Most existing software is written to run on this type of computer.

The SIMD computer is one where each instruction can operate on a data vector which is supplied by means of a multiple data stream, see Figure 1.3. This type of computer (which is also known as a vector processor) is very useful when problems using a large proportion of array operations are computed such as are found in weather forecasting and numerical analysis (e.g. Roberts, 1977).

The MISD computer is a machine for which each item of data is operated on simultaneously by several different instructions, see Figure 1.4. At the present time there does not appear to be a viable worthwhile computer of this type, an artificial example of this may be a line printer where a line of information is considered to be a piece of data, and each print a separate operation.

The MIMD computer can be viewed as several interconnected individual computers, as each processor, at any one time, may be carrying out different instructions on different items of data, see Figure 1.5. Thus each processor may be working on a separate part of program.

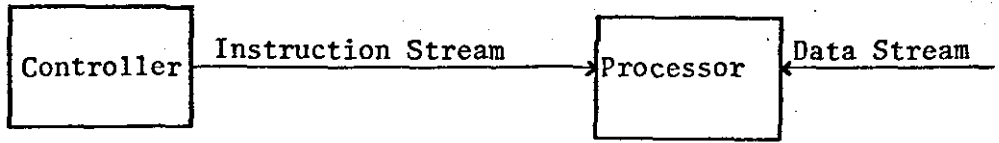


Figure 1.2

MODEL OF A SERIAL OR SISD COMPUTER

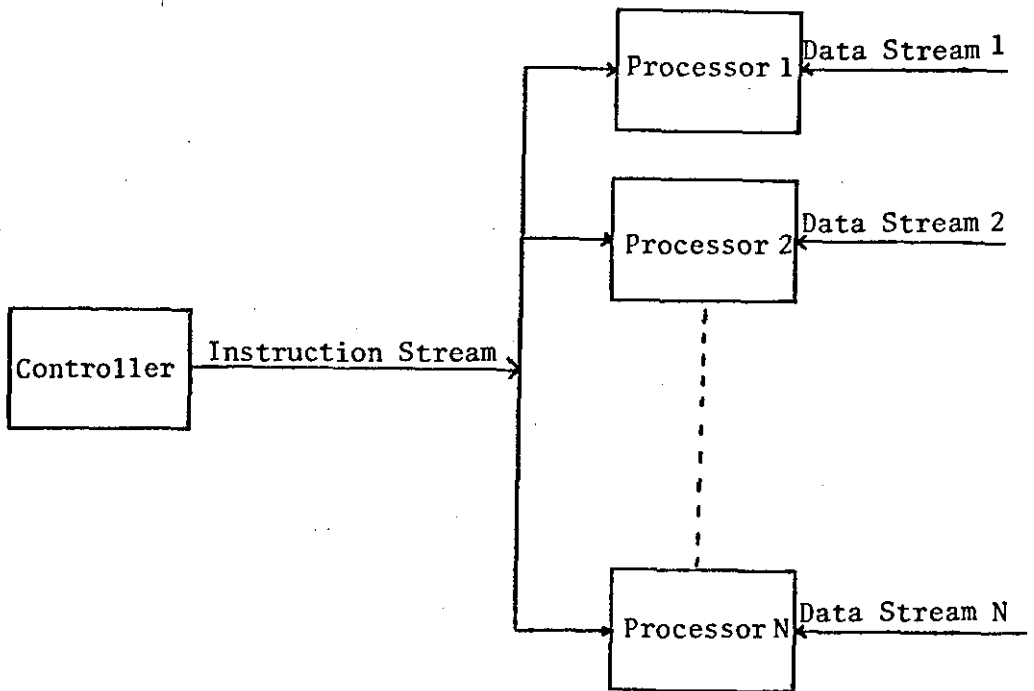


Figure 1.3

MODEL OF A VECTOR OR SIMD COMPUTER

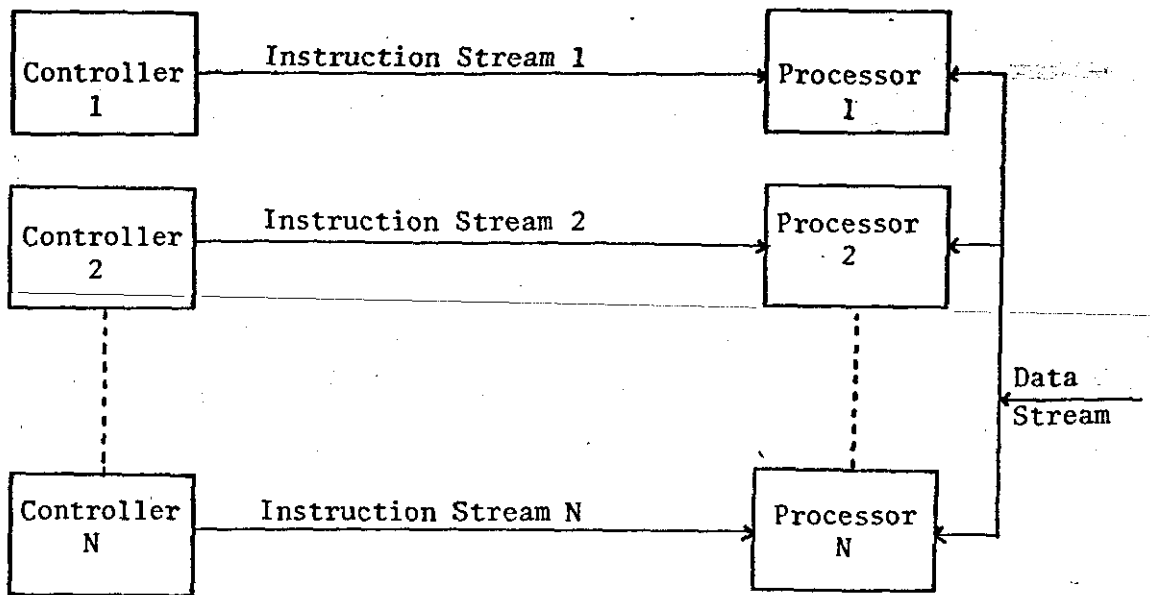


Figure 1.4

MODEL OF A MISD COMPUTER

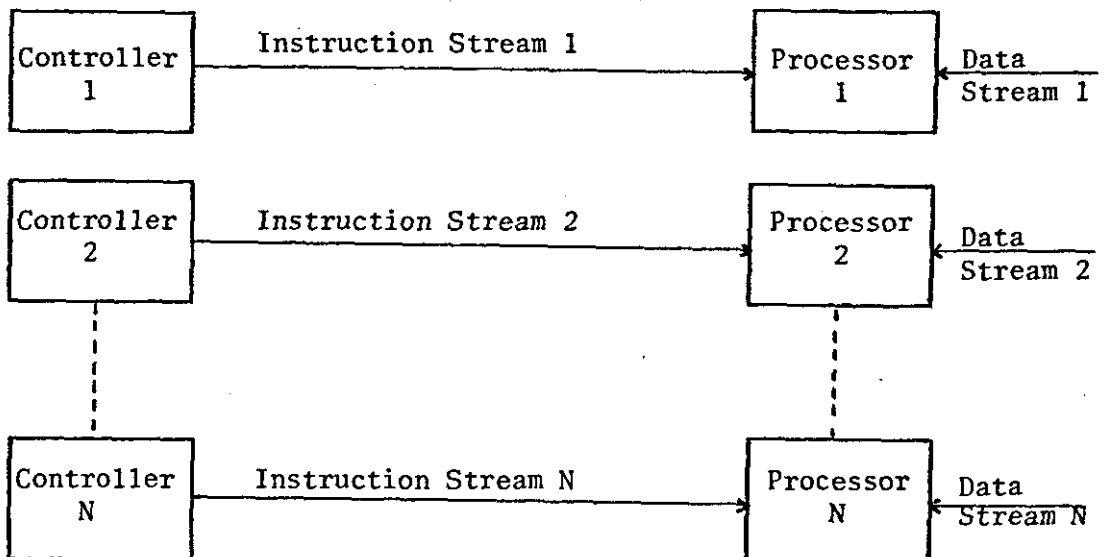


Figure 1.5

MODEL OF A MIMD COMPUTER



An architecture technique that can be applied to all types of computer systems is pipelining (Chen, 1975). Essentially a new task can be initiated before the previous task has completed and the speed of throughput will depend on the rate at which tasks can be initiated rather than on the time for individual operations. Figure 1.6 shows how  $M$  tasks (where  $M$  is a positive integer) may pass through a four segment pipeline, the four operations may be 'Fetch', 'Decode', 'Execute' and 'Store'; for this example each operation is considered to be distinct and this, in general, is the case for pipeline machines. Indeed, each process or operation is performed by a specially designed unit, which is where the pipeline computer differs from the basic computer systems defined by Flynn (1966).

A computer system with a pipeline will take the same amount of elapsed time to execute a task as a similar system without the pipeline. However, for  $M$  tasks the time taken for the pipeline system will, at best, approach the time taken by the other system divided by the number of operations (in the above example that would be four). If an operation involving a jump is executed, the other tasks in the pipeline will not be required and the tasks at the point jumped to will need to be calculated. In the worst case when every task involves a jump, the elapsed time taken by both systems will be the same.

Pipelines are used at the present time with computers that fit the SISD description. The result is a high performance machine such as the CDC 7600 (Chen, 1975). The technique may also be applied to SIMD, MISD and MIMD computers without affecting the Flynn (1966) definitions of any of these systems.

The SIMD and MIMD computers are generally known as parallel processors or parallel computers. Sometimes pipeline computers are

also called parallel computers but that terminology is not used here. The MIMD computer can always work in the same manner as the SIMD computer. However there may be extra overheads involved in having to have a copy of the instruction for each processor. Conversely, the SIMD has only one instruction and cannot always work in the same manner as the MIMD computer. Throughout this thesis the type of computer being considered, unless stated otherwise, will be an MIMD computer without a pipeline.

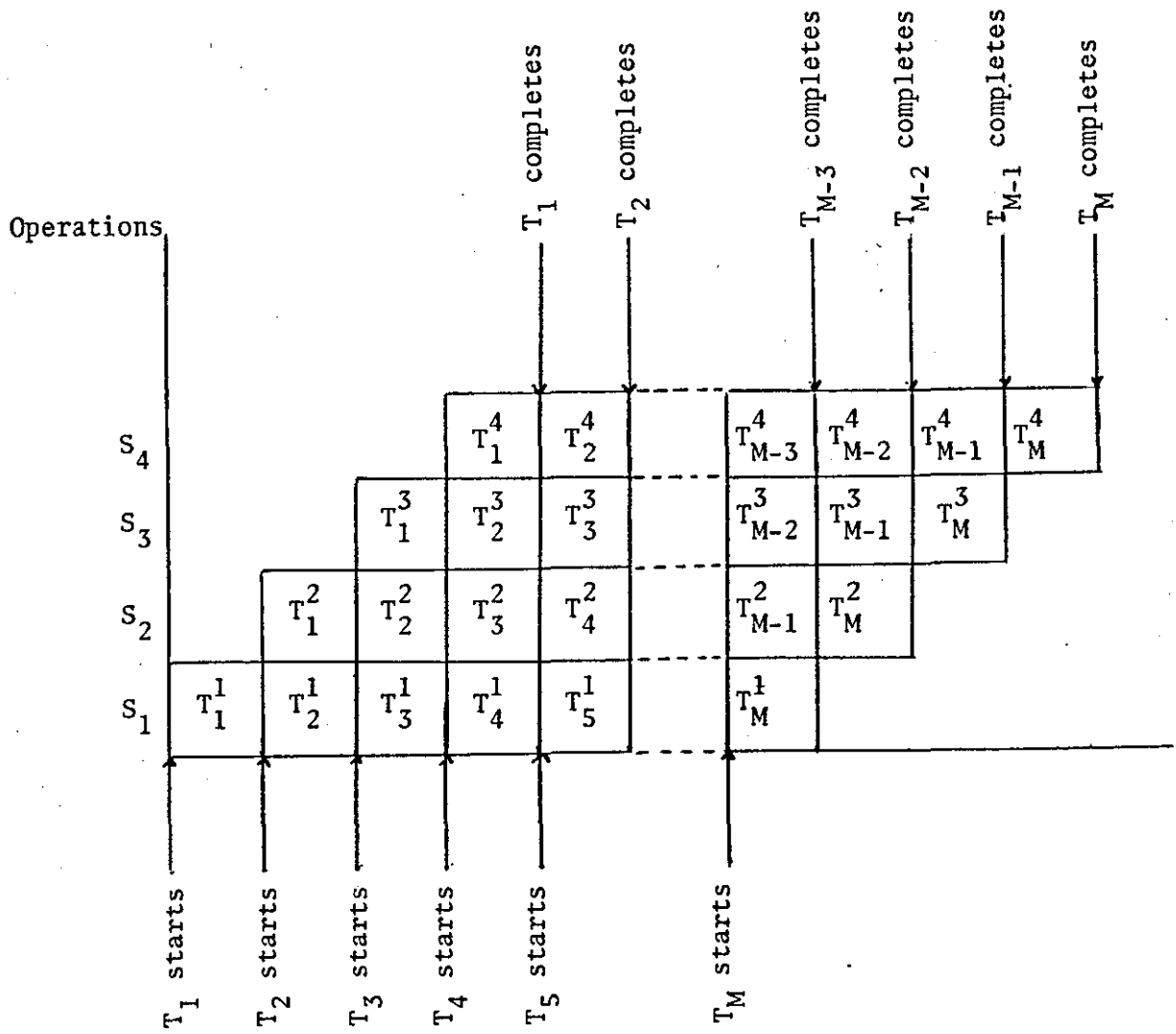


Figure 1.6

M TASKS PASSING THROUGH A FOUR SEGMENT PIPELINE

### 1.3 LEVELS OF PARALLELISM

All possible tasks in a parallel processing environment may be considered as being at one of four levels:-

(i) Machine Level

- e.g. (a) within micro-instructions,
- (b) between micro-instructions.

(ii) Instruction Level

- e.g. (a) within expressions,
- (b) between individual statements.

(iii) Block Level

- e.g. (a) between groups of statements,
- (b) between and within program constructs (such as loops).

(iv) Program Level

- e.g. (a) between individual programs,
- (b) between groups of programs.

Obviously the boundaries between these levels are not always clearly defined. In poorly defined cases a particular construct may be considered to belong at the most suitable level.

The first level is very machine oriented (Freeman, 1975) and to keep this work applicable to a general MIMD computer, machine level parallelism will not be further considered. The program level is also known as inter-program parallelism, or, more commonly, multiprocessing and has been discussed in detail in Enslow (1977). This thesis will discuss inter-program parallelism occurring in the second and third levels.

#### 1.4 EXPLICIT AND IMPLICIT PARALLELISM

Methods have been formulated and implemented by which a programmer may indicate, by means of special statements, where parts of his program may be executed by different processors at the same time (i.e. in parallel). This is called explicit parallelism.

Anderson (1965) introduces statements for parallel processing to be used in Algol 60. These include FORK which initiates parallel tasks, JOIN which waits for parallel tasks to finish (this is the complement of FORK) and statements for synchronising parallel tasks. Figure 1.7 shows an example of a program written using Anderson's FORK and JOIN. The synchronising mechanism allows one of a number of parallel processes to have exclusive use of a particular set of variables during part of its execution. This can be used, for example, to prevent two processes simultaneously trying to change a location.

Gosden (1966) gives the premise that there is a large potential for parallel activity in loops. A good example of such a loop is the Algol 60 FOR statement. A parallel loop construct, PARALLEL FOR, is introduced where each iteration of a loop may be executed in parallel. An example of a matrix sum, an inherently parallel process is given in Figure 1.8.

Variations on these constructs exist for instance in Algol 68 (van Wijngaarden, 1976) a parallel clause, PAR, is defined such that PAR (task 2, task 3) would mean that task 2 could be executed at the same time as task 3. This can be used in conjunction with semaphores of mode SEMA, to provide any necessary synchronisation between task 2 and task 3.

The converse of explicit parallelism is implicit parallelism where the possibilities of parallel processing are automatically detected.

For instance, a sequential program (i.e. one written to run on a serial computer) may be divided as part of its compilation process into parts of code. Detection of the relationships between these parts allows the program to be run on a parallel computer.

There are both advantages and disadvantages in the use of explicit and implicit parallelism. The main advantages of explicit parallelism are, firstly that the programmer is not bound to translate an inherently parallel problem into serial form for computation; and secondly, should a particular algorithm not be suitable for parallel processing it may be changed for another method. On the other hand, implicit parallelism removes the onus from the programmer to detect and express all possible parallelism in his program. Another advantage of implicit parallelism is that a sequential program need not be rewritten to run efficiently on a parallel computer.

In this thesis methods of detecting implicit parallelism within computer programs will be proposed. Techniques for handling programs in which the parallelism is explicitly declared, will also be described and discussed.

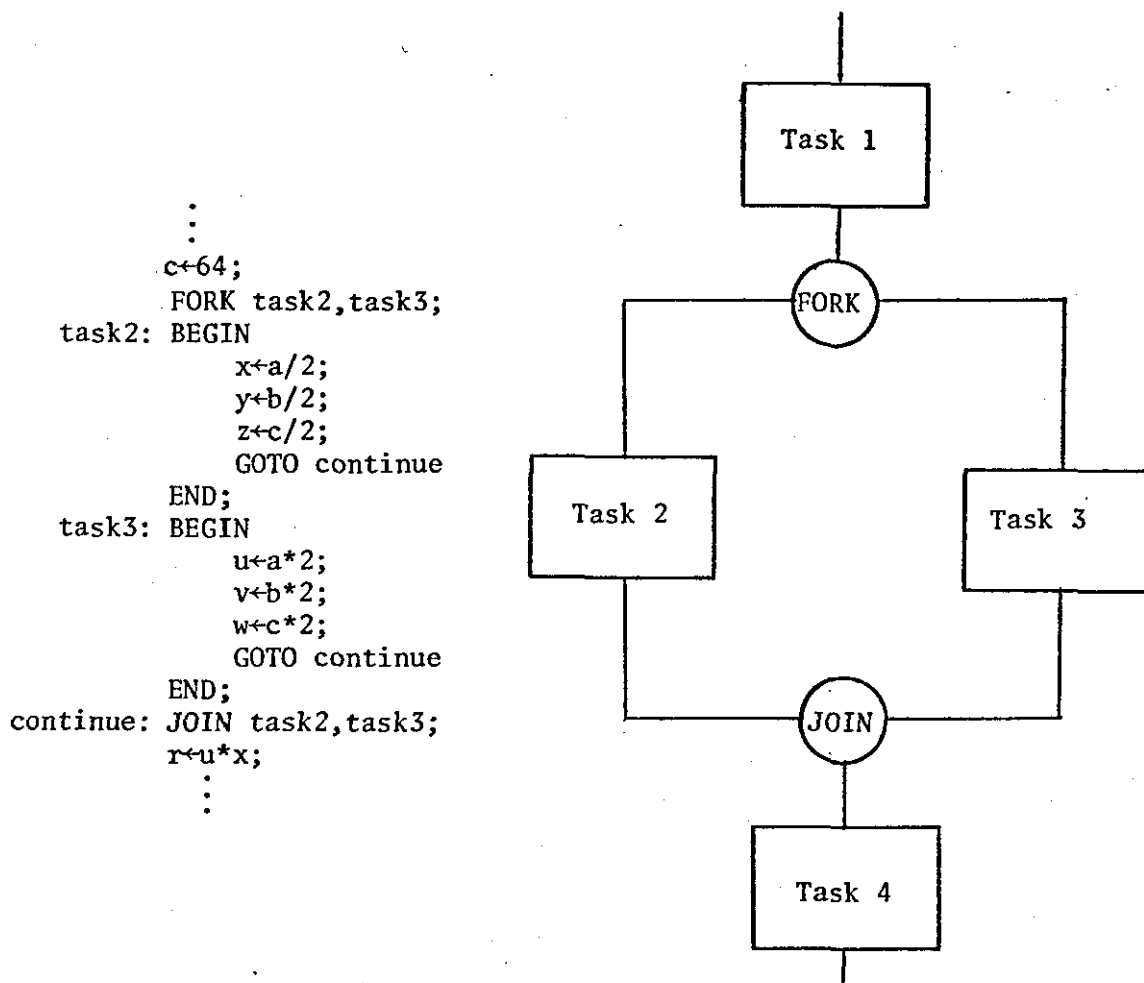


Figure 1.7

SAMPLE PROGRAM USING ANDERSON'S FORK AND JOIN

```
FOR i←1 STEP 1 UNTIL 60 DO
FOR j←1 STEP 1 UNTIL 50 DO
  m[i,j]←m1[i,j]+m2[i,j];
```

Algol instructions for the addition of two matrices

```
PARALLEL FOR i←1 STEP 1 UNTIL 60 DO
PARALLEL FOR j←1 STEP 1 UNTIL 50 DO
  m[i,j]←m1[i,j]+m2[i,j];
```

Figure 1.8

PARALLEL VERSION OF THE ADDITION OF TWO MATRICES  
USING GOSDEN'S NOTATION



CHAPTER 2

SOFTWARE CONCEPTS AMENABLE TO PARALLEL PROCESSING

## 2.1 SYSTEMS SOFTWARE

The term 'systems software' is used to describe the interface between the hardware of a computer and a program being run on it. In a parallel processing environment there will be a need to alter some of the systems software from that used with a serial computer.

The collection of programs that has responsibility for all resources is called the operating system. When more than one processor is available the operating system will have the ultimate responsibility for allocating work to each of the processors. Operating systems for parallel processing computers are discussed in Enslow (1977).

A compiler can be considered to be a computer program. The compiler takes as data the program to be compiled and produces for its results computer-oriented code, that can be run on a particular set of computers. In a parallel processing environment the computer-oriented code produced should indicate possible parallel paths. Compilers and compiling techniques are discussed in more detail in the following two sections.

The programming language used as a media for transmitting problems to a computer may also be considered as part of the systems software. Indeed careful choice of programming language can facilitate the programming of a problem (Barron, 1977). In this thesis parallelism in Algol-type programming languages are primarily considered.

## 2.2 COMPILATION PROCESSES

A compiler is used to produce computer-oriented code from a program. There are many types of compilers to allow for different languages, machines and aims of the implementors. Hopgood (1969), Lee (1974), Rohl (1975) and Wulf et al (1975) along with many other authors discuss types of compilers and compiling techniques.

The time taken to compile a given program on a particular machine depends to a large extent on the number of times the program (as source text) or a version of it has to be scanned (this is called a pass) and so two types of compilers can be considered:-

(i) One-Pass Compiler

The program is only scanned once (i.e. after a statement has been scanned it may not be returned to). This type of compiler is fast but tends to produce inefficient code.

(ii) Multi-Pass Compiler

The program is scanned in several stages (for example see Figure 2.1) after each stage a code is produced to be passed on to the next stage until the final computer-oriented code is produced. Although this is slower than the one-pass compiler, generally more efficient code is produced.

Thus one-pass compilers are useful for short jobs which are only run once. Whereas multi-pass compilers are more beneficial for long jobs that may be run frequently and stored in a compiled state between runs. Since parallel processing is being used primarily to increase the overall speed of throughput efficient code will be preferential to a short compile time. Thus, in general, multi-pass compilers will be considered to be used in a parallel processing environment.

Aho and Ullman (1977) give details of what may happen at possible stages of a multi-pass compiler. For the example compiler in Figure 2.1 four passes are given plus two stages that are available throughout compilation. The Lexical and Syntax Analyses will be used to identify the names and uses of identifiers, to parse expressions (section 2.3) and to find simple errors (may be such things as simple typographical mistakes). The Table Management section is used to keep track of identifiers, usage of variables and links. Whilst the Error Handling routines after the capabilities of recovering from some errors, depending on the language and type of compiler; and in other cases causes the compilation to abort. The Intermediate Code Generation produces, from the information provided by the Lexical and Syntax Analyses, a coded copy of the program which the Code Optimisation stage can optimise, from which stage the final code can be obtained via the Code Generation stage.

A compiler used with programs where the parallelism is explicit will only need to handle the extra language constructs used for expressing parallelism and to test parallel paths for legality and ambiguities; and perhaps carry out some special optimisations. Whereas with implicit parallelism it will be necessary to analyse the program to see how it can be divided into tasks. The compiler could then detect parallel relationships between tasks as well as carrying out the normal compiling work for a program to be run on a serial computer.

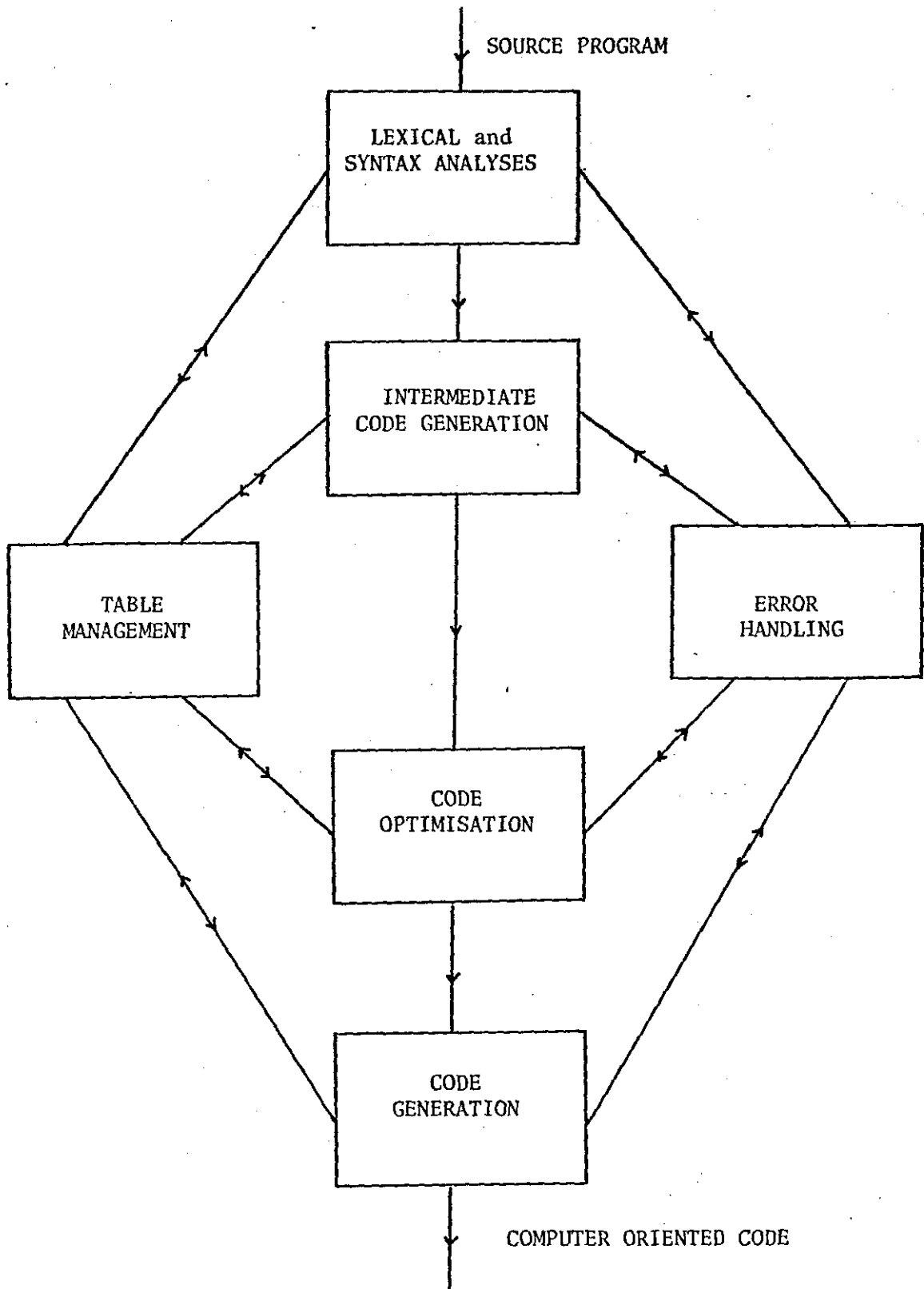


Figure 2.1

POSSIBLE PHASES OF A MULTI-PASS COMPILER

## 2.3 PARSING AN EXPRESSION

The usual order of execution of an arithmetic expression, written in a programming language, is dictated by the rules of mathematics. Thus an expression is calculated by performing operations in descending order of precedence. As in mathematics brackets have the highest precedence and addition the lowest precedence.

A process called 'parsing' is used to translate an expression into a form from which the order of execution is obtainable (Hopgood, 1969). Parsing usually is part of the Syntax Analysis stage of a multi-pass compiler.

### 2.3.1 Reverse Polish Notation

Reverse Polish is the name given to a technique of parsing expressions, and is used extensively in compilers for serial computers. This method provides an unambiguous means of representing an expression (usually arithmetic) without the use of brackets (parentheses). The process can be viewed as the translation of an input string to an output string via a stack. A stack is a means of storing data such that the last item stored on a stack will be the first item removed; similarly, the first item stored on the stack will be the last item removed (Barron, 1968). The translation takes place by passing operands directly from the front of the input string to the rear of the output string. Operators at the front of the input string cause operators on the stack to be moved to the rear of the output string, until the top item on the stack is one with lower precedence number (see Table 2.1) than the one at the front of the input string. The operator at the front of the input string is then moved to the top of the stack. Matching brackets are an exception as they are discarded when

they occupy the top two positions of the stack. Figure 2.2 gives an example of parsing using reverse Polish techniques and illustrates the discarding of matching brackets.

Having obtained a reverse Polish form of an expression it is necessary to produce some type of machine instructions to ensure that the correct pairs of operands are manipulated by the appropriate operator. The reverse Polish string obtained by parsing becomes to a new stage, in which it is analysed from right to left. A recursive procedure CALC (Figure 2.3) can be used to obtain triples indicating the operator and two operands. Here the operands may be temporary results representing triples.

### 2.3.2 Tree Representations

A tree structure may be drawn to represent many relationships (Knuth, 1968;1973) including those of arithmetic and Boolean expressions. A tree structure may be considered to represent a 'branching' relationship between 'nodes', in a similar way to the nodes of trees in nature have branches connecting them. Figure 2.4 gives an example of how a tree may be drawn, and the names given to its constituent parts. Continuing with the use of popular terms, the point from which the whole tree originates is called the root node. Similarly any node from which no branch emanates is called a leaf. All nodes horizontally adjacent are said to be at the same level.

Relationships between nodes of a tree may be considered similar to those in a family tree, containing only male relatives. In Figure 2.4, for example, the following relationships can be considered to exist, node D is the son of node B, nodes D,E and F are brothers and node A is the great-grandfather of node G. A subtree can be considered

to be a node and all its direct descendants. It is the convention to draw trees 'upside-down' so the root node appears at the top of the tree, and all father nodes above their respective sons. Trees for which each node has at the most two sons are called binary trees. When a node has two sons they are usually referred to as the left and right hand sons. Throughout this thesis the trees referred to will be binary trees.

Binary trees are used to represent expressions such that each of the leaves represent either a variable or a constant. All other nodes represent operations to be carried out on their son, if there is only one, or between their sons. Figure 2.5 shows two examples of how a tree may be drawn to represent an expression.

A reverse Polish expression may be converted in to a binary tree structure by using a procedure similar to CALC which was defined in Figure 2.3. The reverse Polish form of the expression is scanned from right to left. A recursive procedure TREE, Figure 2.6, is called when the first operator is detected. The final result will be returned to the calling routine as a binary tree



Symbol or Operator	Precedence Number
)	1
(	2
+ -	6
* /	7
↑	8

Table 2.1

PRIORITIES OF OPERATORS

Input String	Stack	Output String
(A+B)*C	empty	empty
A+B)*C	(	empty
+B)*C	(	A
B)*C	+ (	A
) *C	+ (	AB
) *C	(	AB+
*C	empty	AB+
C	*	AB+
empty	*	AB+C
empty	empty	AB+C*

Figure 2.2

THE DERIVATION OF THE REVERSE POLISH FORM OF AN EXPRESSION

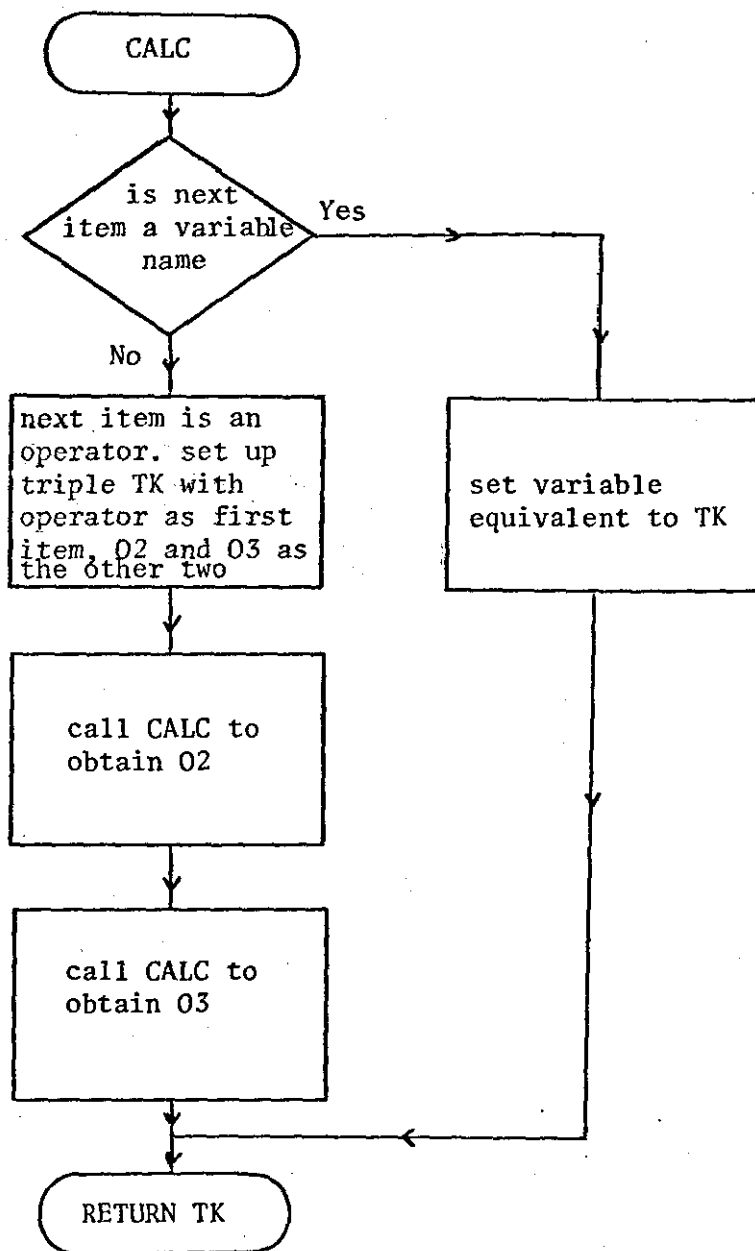


Figure 2.3

RECURSIVE PROCEDURE CALC

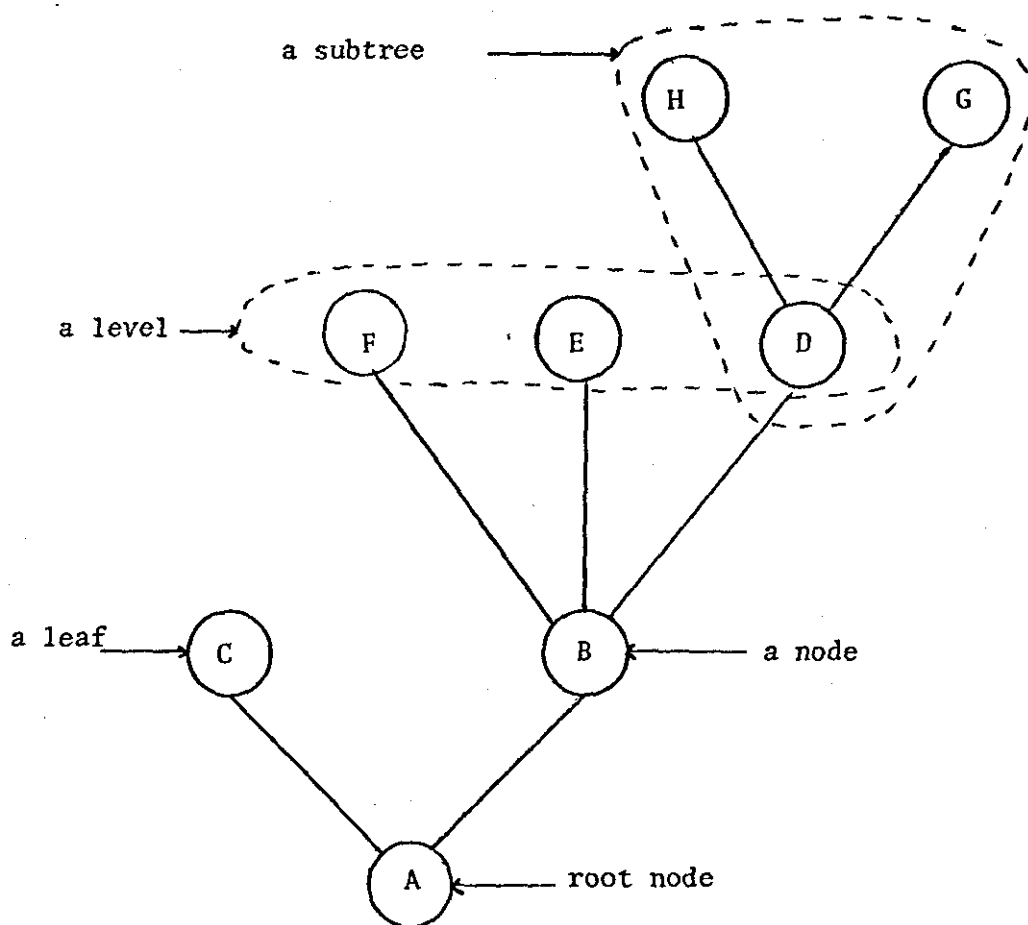


Figure 2.4

A TREE STRUCTURE

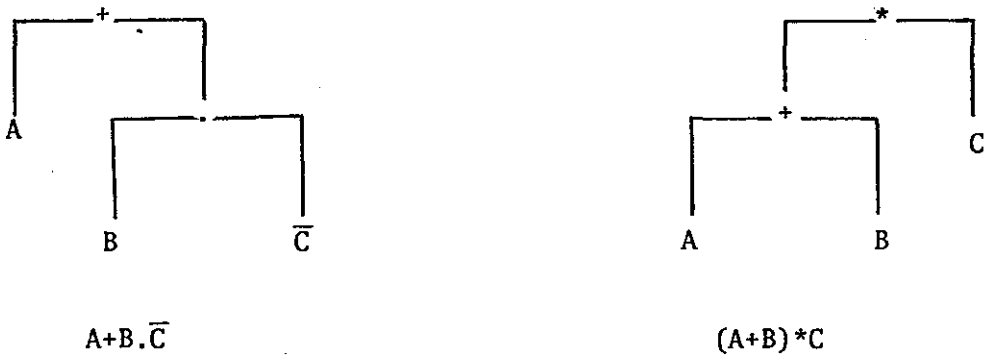


Figure 2.5

TREE REPRESENTATIONS OF A BOOLEAN AND ARITHMETIC EXPRESSION

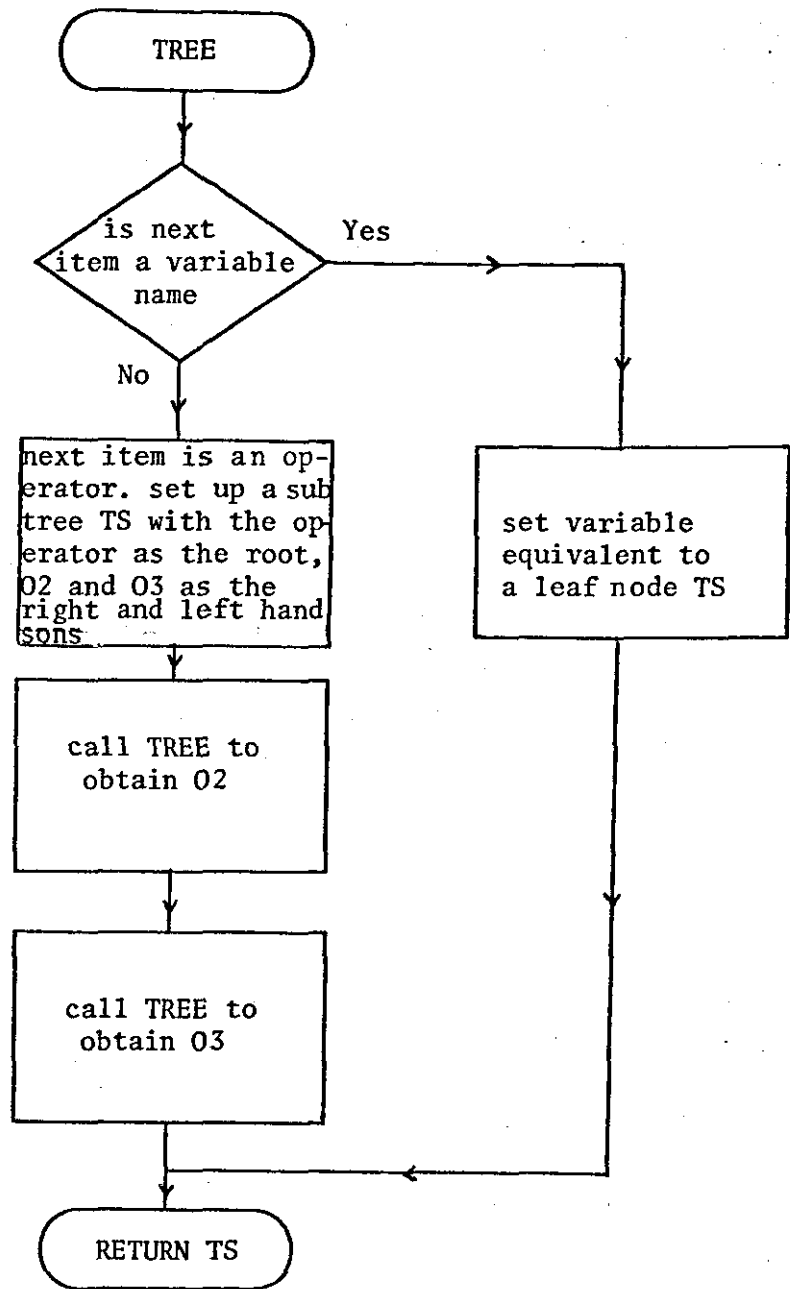


Figure 2.6

RECURSIVE PROCEDURE TREE

## 2.4 ALGOL-TYPE PROGRAMMING LANGUAGES

Higman (1977) defines an Algol-like programming language to be one that has the following properties:

- "a) Use of CBNF to define syntax, with semantics in English.
- b) Acceptance of as much of current mathematical notation as could be proved workable, with elimination of all arbitrary restrictions whose origins lie in Compiler Design.
- c) A clear distinction in symbolism between the imperative (assignment) equals and the predicative (relational) equals.
- d) Use of English words in a distinct font (e.g. black type or underlined) to supply such new symbols as it requires.
- e) Page lay-out completely at the service of legibility to human readers."

Here rather simpler and looser conditions will be given for a language to be considered Algol-type.

### Definition 2.1

An Algol-type programming language is one which has a block-structure and whose design is based on Algol 60.

A block-structured language being one which uses blocks as defined below.

### Definition 2.2

A block is a segment of program delimited by a bracketing structure (e.g. BEGIN and END). Names may be declared to be known only inside a block (i.e. local names) and blocks may be nested inside other blocks.

Some languages which conform to Definition 2.1 would not be classified as Algol-like by Higman's ideals. Examples of languages that may be considered to be Algol-type according to Definition 2.2 are given in Table 2.2, along with references to published specifications.

ALGOL 60	Naur (1962)
ALGOL 68	van Wijngaarden (1976)
ALGOL 68-R	Woodward and Bond (1972)
CORAL 66	Woodward et al (1970)
PASCAL	Jensen and Wirth (1976)
RTL/2	Barnes (1976)

Table 2.2

EXAMPLES OF ALGOL-TYPE LANGUAGES

## 2.5 USAGE OF LANGUAGE CONSTRUCTS

Programs have been analysed for the use of various programming constructs in both static and dynamic program states. Knuth (1971) and Robinson and Torsun (1976a) have carried out empirical studies on Fortran programs, Wichmann (1970;1973) and Robinson and Torsun (1976b) have carried out studies of Algol programs. In all of these studies, both Fortran and Algol assignment was the most frequently used programming construct; loop, conditional and call to routine were also seen to be frequently used. In Robinson and Torsun (1976b) samples over 85% of the static construct used were accounted for by: assignment statements, 'FOR-loops', 'IF-conditionals' and procedure calls. Unconditional jumps (GOTO's) were used more frequently in the Fortran samples than in Algol samples (less than 5%) and with the increase in using 'structured programming techniques' (Kernighan and Plauser, 1976 and Barron, 1977) unconditional jumps should only account for less than 1% of program constructs used for programs written in the future.

When a program is being run it may occupy the majority of its time executing only a few statements. For example Knuth (1971) mentions a 140 line program that spends more than half of its time executing 5 lines which create a loop. This point is also observed by Bingham and Reigel (1968) who state that "the major part of the execution time on single processor machines is spent within loops".

An 'IF-conditional' can be considered in three parts:-

- (i) A condition that is tested.
- (ii) Code that is executed if the condition is true.
- (iii) Code that is executed if the condition is false.

Only one of (ii) and (iii) will be executed for a given pass of the 'IF-conditional', whereas (i) will be executed everytime. These facts



will need to be taken into account when detecting parallelism.

Similarly, when a procedure call is made more code is executed than appears in the static form of the program and this too must be accounted for when detecting parallelism.

For the analysis of either explicit or implicit parallelism the division of a program into tasks will be considerably easier in a structured programming environment. One of the underlying criteria of structured programming is that programs are written in modules (Kernighan and Plauger, 1976) and these modules may be equated to tasks in a parallel processing environment.

CHAPTER 3

DETECTION OF POTENTIAL PARALLELISM

AT THE INSTRUCTION LEVEL

### 3.1 TREE REPRESENTATIONS OF EXPRESSIONS

In this chapter the parsing of expressions to be executed on a parallel computer with a number of arithmetic units or processors, will be studied. Expressions that are used in Algol-type programming languages indicate the type of operations (e.g. addition) to be carried out on a set of operands. The order in which these operations should be executed is also inferred. As mentioned in the previous chapter this is dictated by the usual rules of mathematics. Because of the similarities between arithmetic and Boolean expressions both may be handled using the same techniques. So here attention will be focused upon arithmetic expressions.

A machine with a number ( $N$ ) of arithmetic units or processors will be considered. Where each arithmetic unit or processor can perform any arithmetic operation in unit time. The time taken for an arithmetic expression to be calculated on a parallel computer can be estimated to be proportional to the number of levels in the tree representation of the expression. Suppose  $N$  (the number of arithmetic units) is sufficiently large to perform all possible operations at a given level. If there are  $M$  operations to be performed at a given level, the time taken to execute that level will be proportional to  $\lceil M/N \rceil$ . (NB  $\lceil M/N \rceil$  is the integer that satisfies  $M/N \leq \lceil M/N \rceil < (M/N) + 1$ ). For a serial computer the time taken to calculate an expression can be estimated to be proportional to the number of operations needed to be performed.

Provided that there are sufficient arithmetic units or processors available the following definition applies:

#### Definition 3.1

Any operations that appear at the same level, in a tree representation of an expression, may be executed in parallel on separate processors.

Throughout this work it will be assumed there are sufficient arithmetic units or processors available to perform any given set of operations, unless otherwise stated.

From Figure 3.1(a) it can be seen that it will take 7 units of time to calculate the expression

$$A + B + C + D + E + F + G + H .$$

Whereas in Figure 3.1(b) the same calculation only takes 3 units of time. The tree representations of the expression having seven and three levels respectively. In the latter case, however, four processors are required at level 1, two at level 2 and one at level 3. Whereas in the former case only one processor is required throughout.

The tree representation of the expression given in Figure 3.1(b) shows there is more potential parallelism than in the representation given in Figure 3.1(a). In general, in a parallel processing environment the amount of potential parallelism for the execution of an expression is inversely proportional to the number of levels (or height) of the tree representation of the expression. Thus, when the tree representation of an expression is being formed it will be beneficial to form a tree of the least possible number of levels. The class of operations that will form such trees are called 'balancing' operations. Balancing will usually take place as part of the parsing operation. The execution of an expression from a balanced tree representation should produce identical results to those of from any other tree representation of that expression.

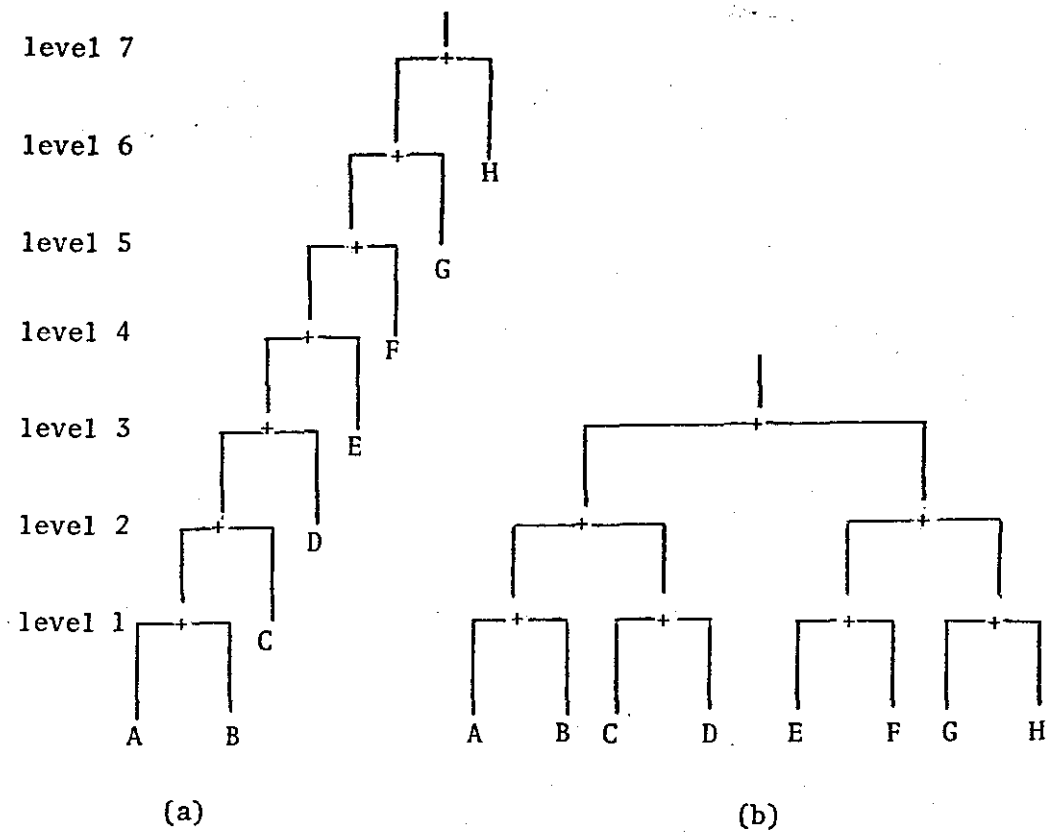


Figure 3.1

POSSIBLE BINARY TREE REPRESENTATIONS OF

$$A + B + C + D + E + F + G + H$$

### 3.2 A SURVEY OF TECHNIQUES FOR RECOGNISING EXPRESSION PARALLELISM

Various methods have previously been proposed for recognising parallelism at the expression level.

These methods determine which parts of the expressions or statements are most suitable for execution in parallel. Descriptions of such algorithms are given in the following sub-sections. Two simple arithmetic expressions will be used, where necessary to illustrate the working of these algorithms. The expressions are:

$$A + B + C + D + E + F + G + H$$

and  $A + B * C + D * E * F * G + H + I .$

#### 3.2.1 Squire's Algorithm

The algorithm proposed by Squire (1963) is based on information relating to operands, operators and the height (or level) on a tree representation at which an operation may be performed. Such information is held in quintuples of the form:-

(operand A, operator, operand B, start height, end height).

All variables are considered to be at the bottom level (i.e. level zero) of a tree and so their end heights will be zero.

The algorithm involves using both right to left scans and left to right scans thus becoming very involved. Here a brief description will be given as to how the method analyses an expression. Figure 3.2 gives an example of Squire's algorithm as applied to an expression of the form

$$A + B * C + D * E * F * G + H + I .$$

An expression is analysed by scanning it from right to left, stacking all the operands and operators on a type of stack called LIST. The stacking procedure is halted when the precedence of an operator scanned is less than that of the last one placed on LIST (Table 3.1

contains a list of the precedences of operators). The precedence of the last operator placed on LIST can be represented by the symbol K. A left to right scan of LIST is then performed (i.e. the stack is scanned from the top downwards). The scan finishes when an operator with a priority different to K is detected. During this scan the two operands (say, A and B) with the lowest height are chosen. A quintuple is then formed consisting of the following information:-

- (a) The operand A.
- (b) The operator immediately to the left of B.
- (c) The operand B.
- (d) The maximum end heights of A and B.
- (e) The end height of this quintuple (i.e. (d) plus 1).

This quintuple then replaces the operand A in LIST, whilst the operand B and the operator are removed from LIST. Then, the left to right scan is repeated from the left most end of LIST until the precedence of the first operator in LIST is different to K. The right to left scan is then continued from the point where it was halted. When the right to left scan has placed all operands and operators on LIST the left to right scan is reinitiated until only one quintuple remains. This quintuple will correspond to the calculation to be executed at the root node of the tree.

Figures 3.3(a) and (b) show the tree representations that would be obtained for the expressions

$$A + B + C + D + E + F + G + H \quad \text{and} \quad A + B * C + D * E * F * G + H + I.$$

It is suggested that subtraction and division may be handled by using the inverse operations and that function calls may have a special quintuple. Similarly, bracketed expressions may be treated as a special case by giving opening and closing brackets a precedence of 1.

Original Expression  
(parsed right to left)

$\equiv \{ - A + B * C + D * E * F * G + H + I - \}$

LIST (Parsed left to right)

Quintuples Formed

$D * E * F * G + H + I - \}$

$Q1 \equiv D, *, E, 0, 1$

$Q1 * F * G + H + I - \}$

$Q2 \equiv F, *, G, 0, 1$

$Q1 * Q2 + H + I - \}$

$Q3 \equiv Q1, *, Q2, 1, 2$

$Q3 + H + I - \}$

$B * C + Q3 + H + I - \}$

$Q4 \equiv B, *, C, 0, 1$

$Q4 + Q3 + H + I - \}$

$A + Q4 + Q3 + H + I - \}$

$Q5 \equiv A, +, H, 0, 1$

$Q5 + Q4 + Q3 + I - \}$

$Q6 \equiv Q5, +, I, 1, 2$

$Q6 + Q4 + Q3 - \}$

$Q7 \equiv Q6, +, Q4, 2, 3$

$Q7 + Q3 - \}$

$Q8 \equiv Q7, +, Q3, 3, 4$

$Q8 - \}$

$\{ - Q8 - \}$

Figure 3.2

THE PARSING OF AN EXPRESSION BY SQUIRE'S  
ALGORITHM



Operator or Symbol	Precedence
-  (start and end)	0
+ -	3
* /	4

Table 3.1

PRIORITIES ASSIGNED TO OPERATORS (by Squire)

level

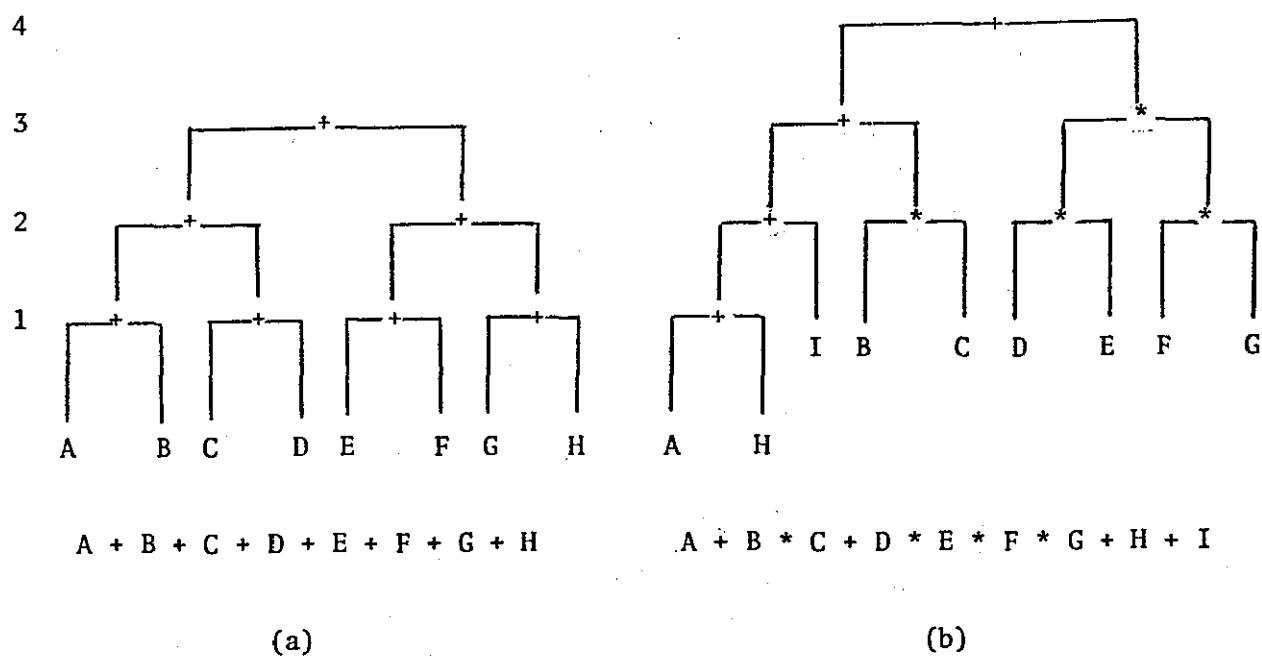


Figure 3.3

TREE REPRESENTATIONS OBTAINED BY SQUIRE'S ALGORITHM

### 3.2.2 Hellerman's Algorithm

Hellerman (1966) proposed a strategy in which he considers the optimum way in which an arithmetic expression, presented in reverse Polish form, may be computed on a parallel processing machine. This algorithm does not involve any balancing operations.

The reverse Polish form of an expression can be found by the method described in section 2.3.1. Hellerman points out that by studying the tree form of the reverse Polish expression it can be seen there are one or more critical paths (Mitchell, 1972) from leaves to the root node. On non-critical paths it may be possible to adjust the level at which a temporary result is formed, thus optimising the number of processors used.

Figure 3.4 shows the binary tree representations that would be formed using Hellerman's algorithm. It can be seen that the shorter expression (a) takes one more level to compute than (b). This is because (a) only uses one processor throughout its calculation whereas as (b) uses two processors at levels 2 and 3 and one processor at the remaining levels..

### 3.2.3 Stone's Algorithm

The aim of the algorithm proposed by Stone (1967) is to generate, in one pass of an arithmetic expression, a type of reverse Polish expression. The tree representation of the expression will have the maximum number of operations at a given level.

A grammar is defined in B.N.F. (for an explanation of B.N.F. see Barron, 1968) and gives a detailed set of Algol 60 highly recursive procedures which will produce a reverse Polish type of string. Basically the algorithm attempts where possible to join two subtrees of

the same number of levels, say  $i$ , to form a new subtree of level  $i+1$ .

Figure 3.5 shows how the expression

$$A + B + C + D + E + F + G + H$$

is translated in to a 3 level tree, represented by the reverse Polish type expression

$$AB + CD ++ EF + GH +++ .$$

The standard reverse Polish form of this expression is

$$AB + C + D + E + F + G + H + .$$

Figure 3.6(a) and (b) show the binary tree representations of the expressions

$A + B + C + D + E + F + G + H$  and  $A + B * C + D * E * F * G + H + I$  obtained by using Stone's algorithm.

Subtraction and division are handled by using inversions, although unary minus itself is not catered for. Exponentiation, because it is not associative, has to be treated separately, as are bracketed expressions.

Using this method it is possible to produce a full binary tree of  $i$  levels when there are  $2^i$  variables linked by one type of associative operator. The tree given in Figure 3.5(a) is an example of this.

#### 3.2.4 Baer and Bovet's Algorithm

The algorithm proposed by Baer and Bovet (1968) was designed to satisfy specific aims. These aims are as follows:-

- "(a) To obtain a minimum number of levels in the syntactic tree.
- (b) To use a left to right scan so that the same symbol is not scanned more than once during a given pass.
- (c) To produce a simple intermediate language with temporary results already sorted by levels".

Temporary results are stored as triples of the form:-

(operand, operator, operand) .

Such triples are given a means of identification so they can be referred to as operands during subsequent passes. Each pass performed by the algorithm corresponds to a level in the tree representation. Each triple formed in a particular pass may be calculated at the corresponding level in the tree.

The algorithm uses two stacks for storage, one for operands and the other for operators. At a given stage in a scan three parts of an expression are under consideration. An operand, ITEM, the operators to the left and right of ITEM, LSCOP and SCOP respectively. Usually LSCOP will have the initial value of "plus". Depending on the relative precedences of LSCOP and SCOP (see Table 3.2) various actions are taken. Basically these are:-

- (1) If the precedence of SCOP is greater than that of LSCOP, or the stacks are empty, then ITEM and SCOP are put on top of the respective stacks.
- (2) If the precedence of SCOP is not greater than that of LSCOP then two subcases are considered:
  - (a) The operator at the top of the operator stack is of precedence equal to that of LSCOP. If this is the case then a triple,  $T_K$ , is formed consisting of:-  
(top of operand stack, top of operator stack, ITEM).  
 $T_K$  and SCOP are then added to the output string.
  - (b) In the other case only ITEM and SCOP are added to the output string.

A scan will end after the terminator (a semi-colon) has been processed as the operator SCOP. The overall process is repeated until the output string contains only one item, which will be the triple

representing the calculation at the root node of the tree representation.

Figure 3.7 shows how the expression

$$A + B * C + D * E * F * G + H + I$$

is parsed in the manner described above.

Figure 3.8(a) and (b) show the binary tree representations of the expressions

$$A + B + C + D + E + F + G + H \text{ and } A + B * C + D * E * F * G + H + I$$

obtained by Baer and Bover's algorithm.

Extra stages are necessary to handle subtraction and division which are dealt with in conjunction with addition and multiplication respectively.

For example consider the expression

$$A / B / C / D .$$

The first scan using Baer and Bover's algorithm will give two temporary results

$$T_1 \equiv A / B \quad \text{and} \quad T_2 \equiv C * D$$

The next pass will give the temporary result

$$T_3 \equiv T_1 / T_2 .$$

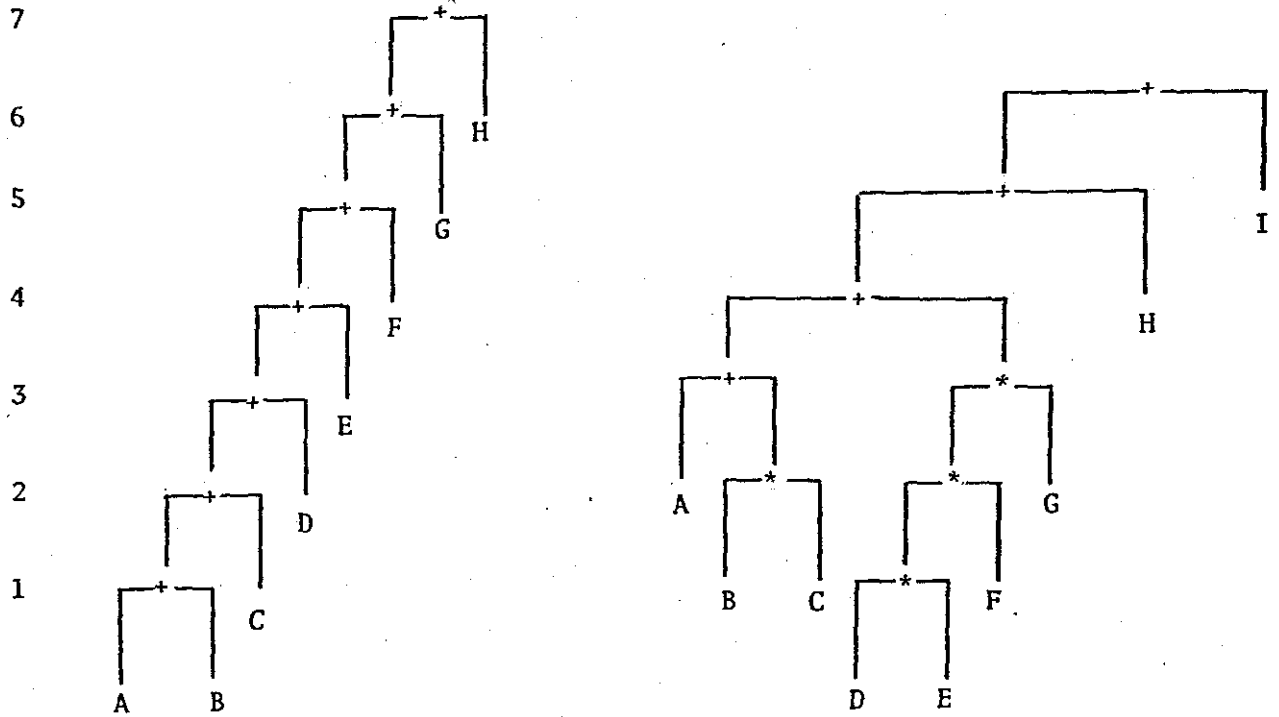
Thus the expression has effectively been converted into a more convenient form of

$$A / B / (C * D) .$$

Unary minus is dealt with by means of a switch. Sometimes it is possible to avoid generation of a unary minus by changing the sign of an operator. However, if a unary minus must be generated it is left until the last possible level.

Brackets have the same precedence as the terminator (see Table 3.2) and so bracketed expressions are calculated as independent entities. When in the output string, an opening bracket and a closing bracket are detected to be only separated by a single operand then the two brackets are deleted.

level



$$A + B + C + D + E + G + H$$

(a)

$$A + B * C + D * E * F * G + H + I$$

(b)

Figure 3.4

TREE REPRESENTATIONS OBTAINED USING HELLERMAN'S ALGORITHM

<u>Input String</u>	<u>Stack</u>	<u>Output String</u>
A+B+C+D+E+F+G+H	empty	empty
+B+C+D+E+F+G+H	empty	A
B+C+D+E+F+G+H	+	A
+C+D+E+F+G+H	+	AB
+C+D+E+F+G+H	empty	AB+
C+D+E+F+G+H	+	AB+
+D+E+F+G+H	+	AB+C
D+E+F+G+H	++	AB+C
+E+F+G+H	++	AB+CD
+E+F+G+H	+	AB+CD+
+E+F+G+H	empty	AB+CD++
E+F+G+H	+	AB+CD++
+F+G+H	+	AB+CD++E
F+G+H	++	AB+CD++E
+G+H	++	AB+CD++EF
+G+H	+	AB+CD++EF+
G+H	++	AB+CD++EF+
+H	++	AB+CD++EF+G
H	+++	AB+CD++EF+G
empty	+++	AB+CD++EF+GH
empty	empty	AB+CD++EF+GH+++

Figure 3.5

THE DERIVATION OF A REVERSE POLISH TYPE OF EXPRESSION  
USING STONE'S TECHNIQUES



level

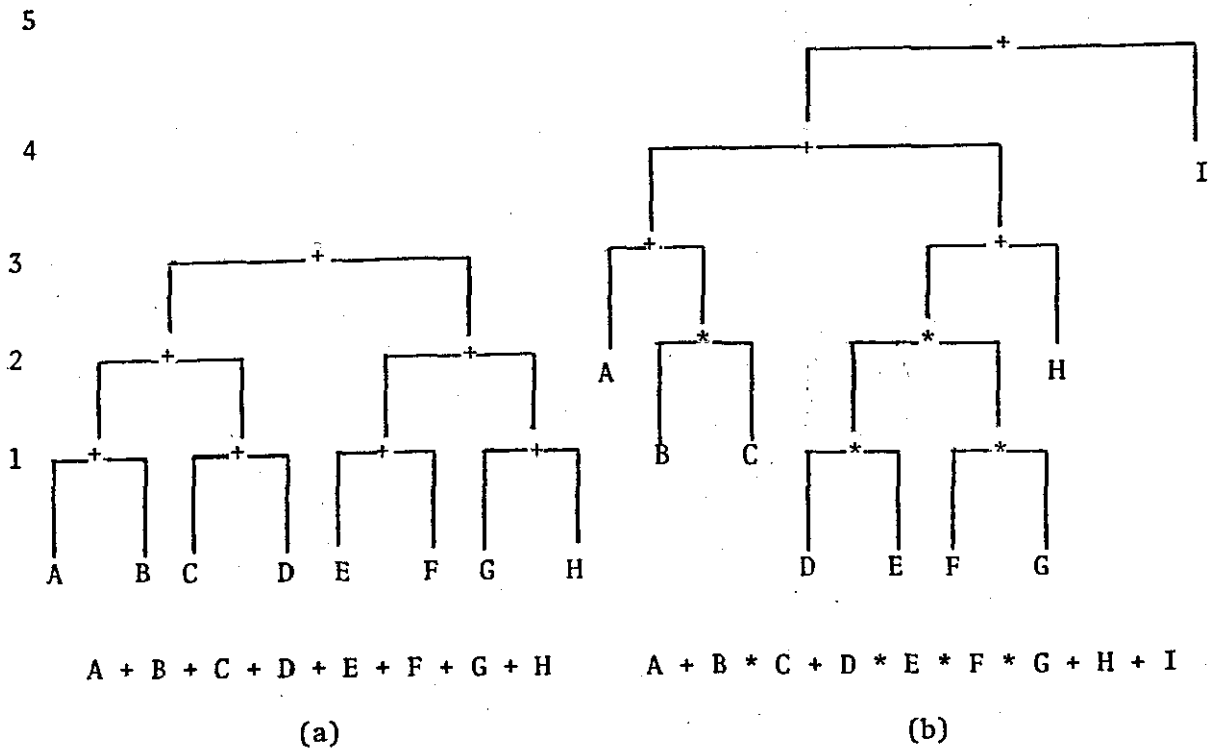


Figure 3.6

TREE REPRESENTATIONS OBTAINED FROM STONE'S ALGORITHM

Operator	Precedence
( )	0
; (terminator)	0
+ -	1
* /	2
†	3

Table 3.2

PRIORITIES ASSIGNED TO OPERATORS BY BAER AND BOVET

$$A + B * C + D * E * F * G + H + I;$$

$$T_1 \equiv B * C$$

$$T_2 \equiv D * E$$

$$T_3 \equiv A + H$$

$$T_1 + T_2 * F * G + T_3 + I;$$

$$T_4 \equiv T_2 * F$$

$$T_5 \equiv T_1 + T_3$$

$$T_4 * G + T_5 + I;$$

$$T_6 \equiv T_4 * G$$

$$T_7 \equiv T_5 + I$$

$$T_6 + T_7;$$

$$T_8 \equiv T_6 + T_7$$

$$T_8$$

Figure 3.7

THE PARSING OF AN EXPRESSION USING  
BAER AND BOVET'S ALGORITHM

level

4

3

2

1

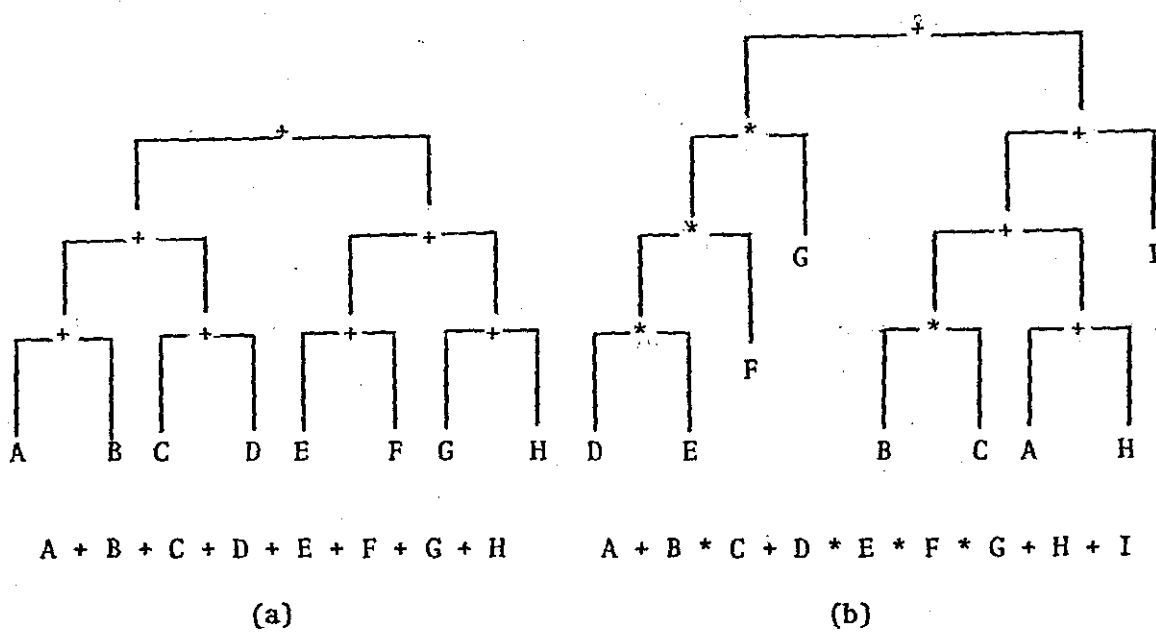


Figure 3.8

TREE REPRESENTATIONS OBTAINED FROM BAER AND BOVET'S ALGORITHM

### 3.2.5 Other Methods For Recognising Parallelism Within Expressions

In the previous four sub-sections methods have been described for the recognition of parallelism within expressions. Possible extensions and variations of these algorithms have been studied by various authors.

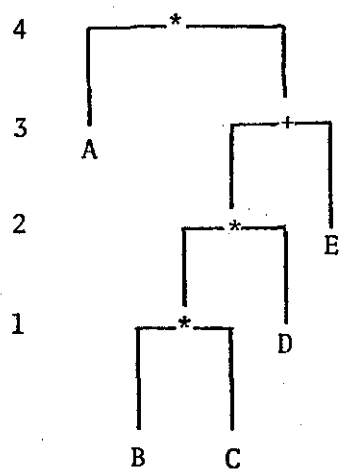
Ramamoorthy and Gonzalez (1969) and Ramamoorthy et al (1973) have proposed two similar approaches that involve weighting reverse Polish expressions. Parts of such expressions may be swapped around, according to their weights, creating a new expression. The new expression will be equivalent to the original, except that the tree representation has a minimum number of levels. These methods work readily for short expressions but become unwieldy for long ones.

Kuck et al (1972) and Kuck (1977) examine the usage of re-distribution over expressions such that a tree representation is of minimum height. This may involve performing extra operations such as shown in Figure 3.9. The distributed form (b) requires five operations whereas in the normal form (a) only four operations are performed. However (b) is completed in three levels whilst (a) takes four levels. Associativity and commutativity are handled in a similar manner to that described in Baer and Bovet (1968). Expressions for which an optimal form is obtainable by just using associativity and commutativity are not distributed. The removal of brackets may cause problems with certain classes of numeric problems.

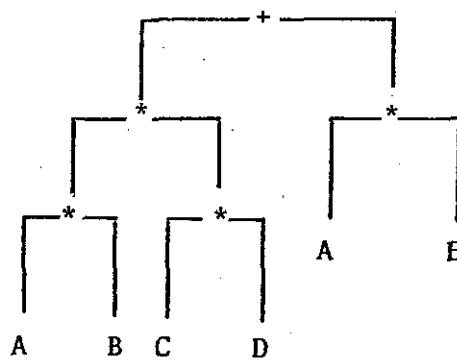
Ward (1974) proposed a method of creating a tree representation of several assignment statements. The approach is based on the work of Baer and Bovet (1968). The algorithm can be explained by considering M assignment statements that appear adjacently. If none of the M statements use the same variable then all the individual tree structures for each statement may be executed in parallel. However, if

one statement fetches a variable that has been previously assigned to, it must be ensured that the new value is fetched. Similarly, if a statement fetches a variable that will be subsequently assigned to, it must be ensured that the old value is fetched. If each statement is considered separately, a tree structure similar to the one shown in Figure 3.10(a) will be obtained, where one statement is completed before the next is commenced. Ward's algorithm allows a variable delay to be associated with a variable that is assigned to, and subsequently fetched. Figure 3.10(b) shows how this technique may decrease the number of levels in a tree representation of two statements.

level



(a)



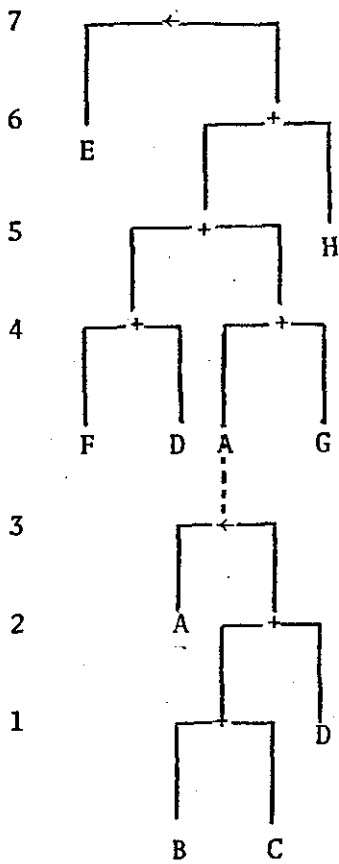
(b)

Figure 3.9

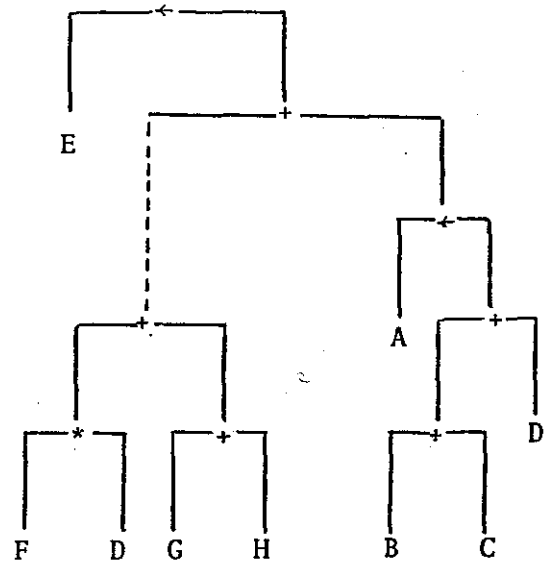
POSSIBLE TREE REPRESENTATIONS OF

$$A * (B * C * D + E)$$

level



(a)



(b)

Figure 3.10

POSSIBLE TREE REPRESENTATIONS OF

$$A \leftarrow B + C + D;$$

$$E \leftarrow F * D + A + G + H$$



### 3.3 FORMATION OF A BALANCED BINARY TREE

Evans and Smith (1977) consider how a binary tree of minimum number of levels (a balanced binary tree or a balanced tree) is systematically constructed from single element components (see Figure 3.11). Assume that the first element is attached to the null node. A second element can be added by forming a new node whose left hand son is the original element and whose right hand son is the new element. This called 'inserting one place above' because the join (i.e. the position of insertion) is immediately above the new position of the previous element inserted. In the case of adding a second element this is the first insertion and the previous element is the original node. Similarly, a third element can be added by inserting the new element two places above the last element inserted (i.e. the join is two levels above the new position of the previous element inserted). A fourth element can be added by inserting a new node one place above the third element (i.e. the last one inserted). Whilst a fifth element can be added by inserting a new node three places above the fourth element.

The tree construction process given can be enumerated by using a numeric code, which can be generated in the following manner. At any point in the tree the number 1 is used to indicate that the next element should be inserted one place above the previous entry in the tree. The number 2 is used to represent the fact that the next insertion should be two places above the previous entry in the tree. In general, the number  $K$  (where  $K$  is a positive integer) will indicate that the next insertion should be  $K$  positions above the last element inserted. So the four insertions shown in Figure 3.11 can be represented by the code 1,2,1,3.

The process can be extended by recognising the symmetry of binary trees. The three insertions following those given in Figure 3.11 will be in the same manner as the first three (i.e. 1,2,1). The eighth insertion (that is  $2^3$ ) will be inserting the ninth element at the fourth ( $3+1$ ) level above all other nodes. Generalising, the  $2^i$  insertion will be at the  $i+1$  level above all other nodes. The insertion of the  $(2^i+1)$  to  $(2^{i+1}-1)$  elements will be in the same manner as the insertion of the first  $(2^i-1)$  elements.

The process described above allows single elements to be added into a binary tree structure. In some cases it will be necessary to add subtrees to existing tree structures. Using the following criteria subtrees may be added into binary trees without unnecessarily increasing the height of the tree, whilst retaining the structure of the subtree.

#### Criteria for Inserting Subtrees

- (1) Any increase in the overall height of the tree caused by the insertion process should be kept to an absolute minimum. This is so that the number of levels in the tree will continue to be minimised.
- (2) An insertion at the top of the tree is preferable to extending the tree below the lowest existing level. This provides for possible future extensions to the tree. If the tree is extended below the lowest existing level then the next insertion must also extend the height of the tree. Whereas, if the tree is extended above all existing levels, the next insertion may not extend the overall height of the tree.
- (3) A subtree should be placed in the first available position in the tree, provided the previous conditions are met. This, again, is done to allow further extensions to the tree, so that the maximum number of vacant nodes are available for successive insertions.

Figure 3.12 gives examples of how subtrees may be added into trees.

When a single element had been added to the tree the next available position in the tree, for an insertion, was defined by that element. However, when a subtree has been inserted according to the above criteria, the next available position in the tree will not be immediately obvious. A dummy pointer can be used to indicate the next position in the tree where a single element may be inserted. The value of the dummy pointer will depend both on the subtree and the tree into which it is being inserted. If the subtree is shorter than the tree into which it is being inserted, then it is assumed that the maximum number of elements that could be held in a subtree of that size has been added. The dummy pointer is obtained from the last item theoretically added in that subtree. A different approach is necessary when the subtree's height is greater than or equal to that of the tree into which it, the subtree, is being inserted. The next insertion (after the subtree) will need to be above the join of the tree and subtree (criteria 2). The dummy pointer will then be obtained from the last element theoretically inserted in a full tree of one level greater than the subtree actually inserted.

After several subtrees have been inserted into a tree, the tree may no longer be of optimal form. This is because insertions are always at the next available position in the tree. Any suitable positions available earlier in the tree are not accessible. However, this situation is in line with the tree being formed systematically from components.

An algorithm that will create balanced binary trees is given in Appendix 1 as an Algol 68-R program.

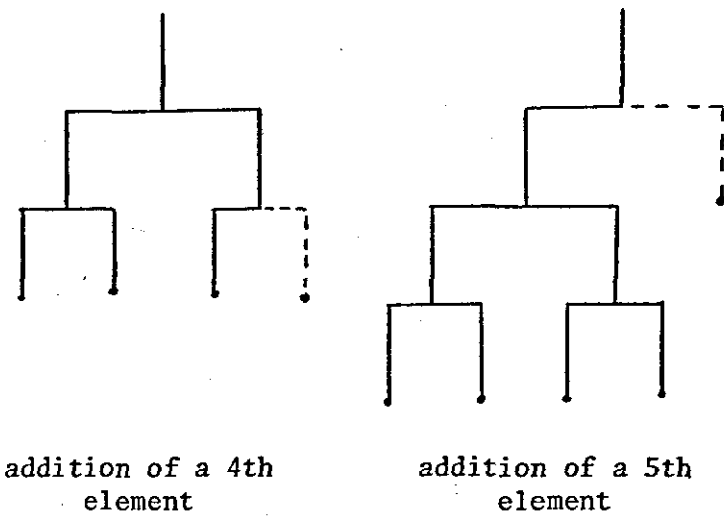
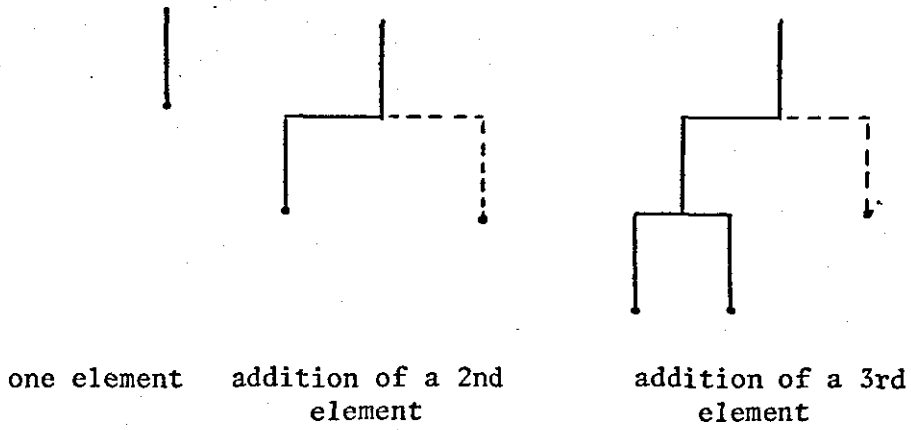


Figure 3.11

SYSTEMATIC CONSTRUCTION OF A BALANCED BINARY TREE

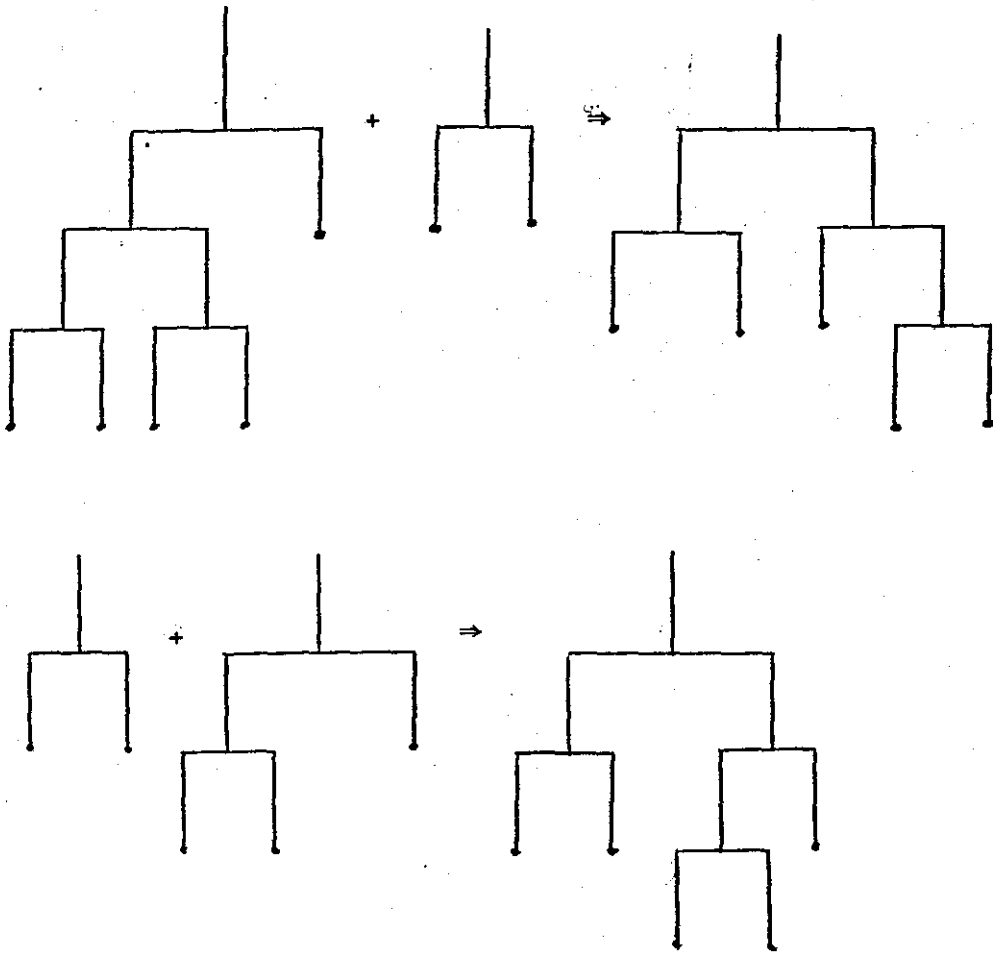


Figure 3.12

ADDITION OF SUBTREES TO TREES

### 3.4 A NEW ALGORITHM

In this section a new technique for producing a binary tree representation of an arithmetic expression will be introduced. The method will use the technique for forming a balanced binary tree described in section 3.3. The following constraints will be applied to the algorithm:

- (1) The priority of brackets will be observed.
- (2) Expressions are not to be reordered.
- (3) The tree representation should be of minimum possible height.

The first two constraints should ensure that results from sensitive numeric equations are not effected by this technique. This is particularly important as one of the main areas in which parallel processing will be useful is the solving of large numeric equations. The precedence of operators are arbitrarily assigned the values given in Table 3.3.

The balancing technique described in the previous section is used to form balanced binary tree representations of expressions and statements. The leaves of the tree will correspond to variables and constants, whilst all other nodes will represent operators. As long as operands connected by operators of the same precedence are being considered (other than exponentiation) the formation of a balanced binary tree is carried out as explained. For the operation exponentiation the next item must be inserted at the top of the tree because exponentiation is not associative. The balancing technique will also provide information about the level at which an insertion is performed. This information may be stored as part of a tree structure. The whole of this process is referred to as the Balancing Method.

#### 3.4.1 The Basic Algorithm

Two stacks will be used during the execution of this algorithm for storing symbols already scanned. Operators are stored on OPSTACK, operands

and any temporary results in the form of subtrees are stored in RANDSTACK. The first item on OPSTACK will be a fictitious operator with precedence -1.

The symbols of an expression are scanned one at a time, from left to right. Depending on what the symbols are various actions are taken.

These being:

(a) Operand.

When a symbol is recognised as not being an operator it is treated as an operand and stacked on RANDSTACK.

(b) Operator.

If the operator is a minus (or divide) the corresponding operand is marked to be negated (or reciprocated) and the operator becomes a plus (or multiply). Two possible cases are then considered:

- (1) When the precedence of the operator just scanned is greater than or equal to the operator at the top of the stack. The new operator is stacked on top of OPSTACK and the next symbol scanned.
- (2) Otherwise the precedence of the operator just scanned is less than that of the operator at the top of OPSTACK. Then the two operands from the top of RANDSTACK are joined into a subtree by the operator from the top OPSTACK using the technique described earlier as the Balancing Method. The two operands and the operator are then removed from the top of the respective stacks. There are then three possible situations:
  - (i) The precedence of the operator now at the top of OPSTACK is the same as the one just removed. In which case the operand from the top of RANDSTACK is joined into the subtree being formed using the Balancing Method. The top items from each stack are removed, and the three possibilities are reconsidered.

(ii) The precedence of the operator now at the top of OPSTACK is greater than that of the operator just stacked but less than that of the operator just removed from the top of OPSTACK. In which case the operand from the top of RANDSTACK is joined in to the subtree being formed at the top. The three possibilities are then reconsidered.

(iii) The precedence of the operator now at the top of OPSTACK is less than or equal to that of the operator just scanned. Then the subtree being formed is stored at the top of RANDSTACK and the operator just scanned is put on the top of OPSTACK. The next symbol is then scanned.

(c) Blanks.

Blank characters are ignored.

(d) Brackets.

When an opening bracket is encountered it is placed on the top of OPSTACK and the scanning continues in the ordinary manner, until the matching closing bracket is scanned. Then for all the operators on OPSTACK, from the top one until the one above the opening bracket, and the corresponding operands on RANDSTACK a subtree is formed. The formation of the subtree is done in the manner described in (b). The resulting subtree is placed on top of RANDSTACK and the brackets are discarded.

(e) Semicolon.

A semicolon is used to indicate the end of an expression has been reached. So the final tree must be formed, this is done by considering the remaining items on the two stacks as described in (b).

Figure 3.13 shows how an expression is parsed using the new algorithm. The tree is effectively built backwards, as it is always a left hand son



that is added. Figure 3.14(a) and (b) show the tree representations obtained by the new algorithm for the two expressions:

$$A + B + C + D + E + F + G + H \quad \text{and} \quad A + B * C + D * E * F * G + H + I .$$

### 3.4.2 Extension to the Basic Algorithm

In the previous section a new algorithm was described that would deal with the fundamental arithmetic operations. Here, extensions to the algorithm will be described, which will increase the potency of the algorithm.

It is possible to handle unary minus by using additional techniques when the expression is scanned. A unary minus is recognised when two operators are read in succession and the second is a minus or when an operator followed by an opening bracket is followed by a minus. In either case instead of stacking a minus sign a non-standard sign, say '?' is stacked. On unstacking when a unary minus is detected the corresponding operand is marked to be negated. The unary minus is then removed from the operator stack and the process continued.

Simple assignment statements can be catered for by defining each statement to consist of a variable name, followed by an assignment symbol, then an arithmetic expression. Thus when the first operand is detected by (a) in the previous section it is set aside and stored in LHS. The next operator scanned must then be an assignment, which is then discarded. The remainder of the expression is then scanned in the normal manner, assume that this gives a tree of  $i$  levels. A node is then inserted at the  $(i+1)^{\text{th}}$  level with an operator assignment, LHS as its left hand son and the expression as its right hand son. Thus, it may be said that LHS took  $(i+1)$  levels to compute or LHS is available at level  $(i+1)$  assuming there are sufficient processors.

When several assignment statements, which are executed one after another, are being considered, it is possible that one statement may use a variable that is assigned to by another statement. This is similar to the problem described in section 3.2.5. Using the information obtained from forming a tree for a single assignment statement it is possible to say at what level  $(i+1)$  a variable will be available. So when this variable is inserted in a tree representation of a subsequent expression the variable will be known not to be available to level  $i+1$ . The insertion in to the tree will thus be the same as for inserting a subtree of level  $(i+1)$ . Figure 3.15 shows how the algorithm forms trees for a set of assignment statements.

Another possible extension to the algorithm would be to allow for various operations to have different execution time. Multiplication may be considered to take four times as long as addition. Thus when a subtree consisting of  $j$  levels, with all operations being multiplication, is being inserted in to a tree formed from additions the subtree would be treated as though it has  $4*j$  levels.

Operator	Precedence
;	0
space	0
(	1
)	2
+ -	6
* /	7
↑	8

Table 3.3

PRIORITIES ASSIGNED TO OPERATORS BY THE NEW ALGORITHM

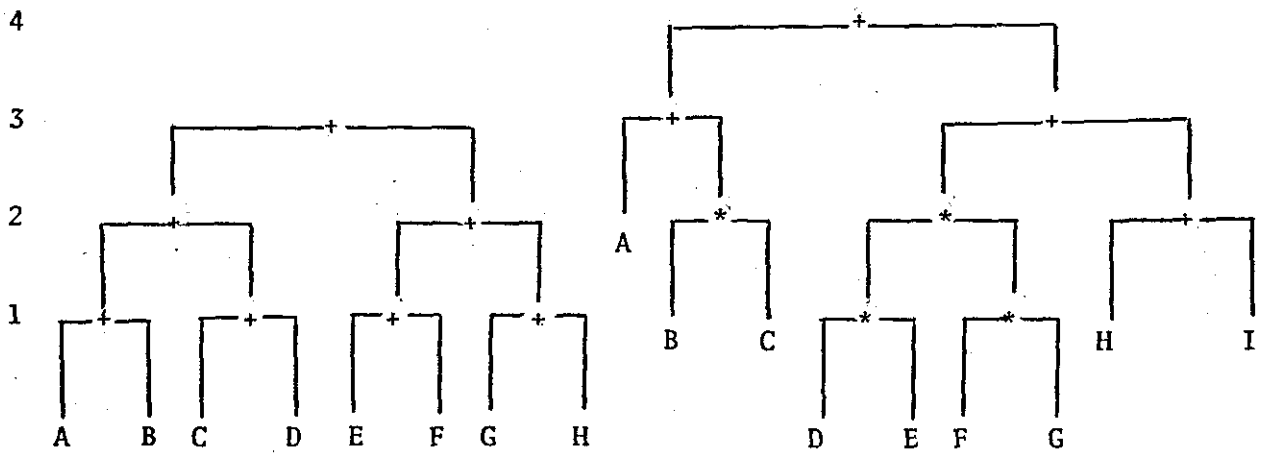
<u>Input String</u>	<u>RANDSTACK</u>	<u>OPSTACK</u>
A+B*C+D*E*F*G+H+I;	empty	empty
+B*C+D*E*F*G+H+I;	A	empty
B*C+D*E*F*G+H+I;	A	+
*C+D*E*F*G+H+I;	BA	+
C+D*E*F*G+H+I;	BA	**
+D*E*F*G+H+I;	CBA	**
+D*E*F*G+H+I;	{B*C}A	+
D*E*F*G+H+I;	{B*C}A	++
*E*F*G+H+I;	D{B*C}A	++
E*F*G+H+I;	D{B*C}A	+++
*F*G+H+I;	ED{B*C}A	+++
F*G+H+I;	ED{B*C}A	++++
*G+H+I;	FED{B*C}A	++++
G+H+I;	FED{B*C}A	+++++
+H+I;	GFED{B*C}A	+++++
+H+I;	{F*G}ED{B*C}A	++++
+H+I;	{E*{F*G}}D{B*C}A	+++
+H+I;	{{D*E}*{F*G}}{B*C}A	++
H+I;	{{D*E}*{F*G}}{B*C}A	+++
+I;	H{{D*E}*{F*G}}{B*C}A	+++
I;	H{{D*E}*{F*G}}{B*C}A	+++
;	IH{{D*E}*{F*G}}{B*C}A	++++
;	{H+I}{{D*E}*{F*G}}{B*C}A	+++
;	{{{D*E}*{F*G}}+{H+I}}{B*C}A	++
;	{{B*C}+{{{D*E}*{F*G}}+{H+I}}}A	+
;	{{A+{B*C}}+{{{D*E}*{F*G}}+{H+I}}}	empty

Figure 3.13

## DERIVATION OF AN EXPRESSION BY THE NEW ALGORITHM

*N.B. Curly brackets are used to enclose a subtree.*

level



$$A + B + C + D + E + F + G + H$$

(a)

$$A + B * C + D * E * F * G + H + I$$

(b)

Figure 3.14

TREE REPRESENTATIONS OBTAINED FROM THE NEW ALGORITHM

$T \leftarrow A * B * C + 2;$   
 $U \leftarrow D - 1 - E;$   
 $V \leftarrow T + U;$   
 $W \leftarrow T - U;$

5

4

3

2

1

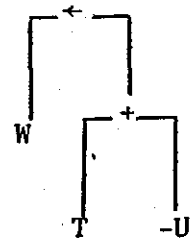
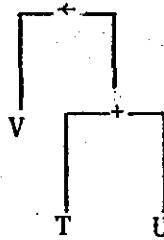
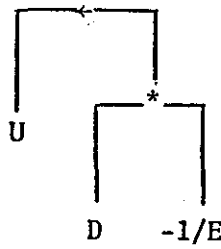
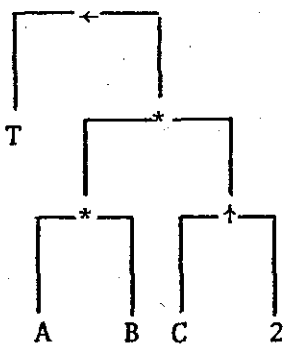


Figure 3.15

TREE REPRESENTATIONS OF ASSIGNMENT STATEMENTS

### 3.5 A COMPARISON OF ALGORITHMS FOR RECOGNISING EXPRESSION PARALLELISM

All the algorithms described will handle addition, multiplication and some other operations as described for each algorithm. The methods proposed by Square (1963), Hellerman (1966), Stone (1967), Baer and Bovet (1968) and the new algorithm propounded in the previous section will be compared. The algorithms of Kuck et al (1972), Kuck (1977) and Ward (1972) are excluded as they can be considered to have the same properties as Baer and Bovet's (1968) algorithm. The methods suggested by Ramamoorthy and Gonzalez (1969) and Ramamoorthy et al (1973) are also excluded, because of the complexities that arise when they are used (see section 3.2.5).

The algorithms produce the results of parsing an expression in various formats. The algorithms of Stone and Hellerman present their results in a reverse Polish type of notation. Whereas the methods of Squire and Baer and Bovet present their results in the form of 'temporaries' which are linked together by a final temporary result. The new algorithm's results are available in the form of a tree structure. Where the results of a parse are only available in a reverse Polish type of notation extra work will be necessary to determine at what level operations may be performed.

In all the methods considered, except Baer and Bovet's, minus and divide are handled by negating and reciprocating. Baer and Bovet handle subtraction and division by using their associative properties. Thus, hopefully avoiding their generation or, at least, not performing these operations until the latest possible level. There are potential problems with this, for instance if a different pair of numbers are divided to those initially intended, then overflow or underflow problems may occur. Unary minus is not handled in the algorithm proposed by Stone. Hellerman's algorithm handle's unary minus in the

standard reverse Polish manner. A special quintuple is formed when Squire's method is used. Whereas Baer and Bovet introduce a switch that is used to indicate unary minus, but as with subtraction the forming of such results is avoided where possible. The new algorithm negates the corresponding operand or subtree when a unary minus is detected.

All these methods treat bracketed expressions as entities. This avoids the need to introduce 'special inviolable parentheses', (Kuck et al, 1972) to protect delicate numerical calculations.

Actual run-time comparisons of the algorithms are difficult to make. The physical time taken to parse an expression is short for each algorithm. Because of the different ways expressions are handled by each method, a given algorithm cannot be expected to parse every expression in a time proportional to that taken by another algorithm. Three tests were carried out on a single version of each algorithm, written in Algol 68-R (Woodward et al, 1974). The tests involved were:-

- (i) Calculating the theoretical times for the operations executed in the program versions of each algorithm (Wichmann, 1973).
- (ii) Running the algorithms within a program loop on the ICL 1904A at Loughborough University.
- (iii) Running the algorithms within a program loop on the ICL 1906A at Nottingham University.

Table 3.4 gives an example of typical figures. The units of measurement are only significant down the columns. All the algorithms, except Hellerman's, produce from the expression:

$$A + B + C + D + E + F + G + H$$

a representation of the tree given in Figure 3.1(b). Hellerman's algorithm, which performs no balancing produces a representation of



the tree given in Figure 3.1(a). It must also be noted that extra calculations are necessary to decide at which level operations may be executed, when the results are presented in a reverse Polish notation.

The other expression considered,

$$A + B * C + D * E * F * G + H + I ,$$

created five different trees for each of the five algorithms (see Figures 3.3(b), 3.4(b), 3.6(b), 3.8(b) and 3.14(b)). Table 3.5 shows the number of levels in each of the tree representations. Both of the trees formed by Squire and Baer and Bovet have used commutativity such that 'A' and 'H' are added together. Stone's algorithm fails to detect that 'I' need not be at the top of the tree. Hellerman's algorithm produces the tallest tree, but still offers some scope for parallelism. The new algorithm has not moved any parts of the expression around, but nevertheless for the expression considered forms a tree of minimum height.

Of all the methods suggested, Hellerman's or Stone's will probably provide the fastest means of finding some parallelism within an expression. The algorithm suggested by Baer and Bovet provides a thorough analysis of an expression. However, this algorithm and the one suggested by Squire may create problems with sensitive expressions that would not occur otherwise. The new algorithm presents its results in a form suitable for determining the maximum amount of parallelism without unnecessarily affecting sensitive numeric equations.

ALGORITHM \ TEST	THEORETICAL	1904A	1906A
Squire	7786	9	8
Hellerman	4576	10	4
Stone	5162	6	6
Baer and Bovet	13048	12	10
New Algorithm	7391	11	7

Table 3.4

TIMES TAKEN TO ANALYSE

$A + B + C + D + E + F + G + H$

Algorithm	No. of Levels
Squire	4
Hellerman	6
Stone	5
Baer and Bovet	4
New Algorithm	4
Minimum No. of Levels	4

Table 3.5

NUMBER OF LEVELS IN THE TREE REPRESENTATION

$A + B * C + D * E * F * G + H + I$

CHAPTER 4

ANALYSIS OF GROUPS OF STANZAS

WITH A VIEW TO DETECTING PARALLELISM

#### 4.1 INTERDEPENDENCIES BETWEEN PARTS OF PROGRAM

Consider a parallel processing system where each processor is capable of executing several operations without independent action having to be taken. Independent parts of a program being executed may, in that case, be allocated to separate processors. If however interdependent parts of a program are assigned to different processors, anomalies may occur. For instance, after the parallel execution of Process 1 and 2 (see Figure 4.1) the variable 'x' may be equal to 'v1' or 'v2' or some undefined value. The undefined value would arise if both processes simultaneously assign to the variable 'x'. Brinch Hansen (1973) discusses the possibilities of what may happen in such situations.

Given a program designed to run on a serial computer, control is assumed to pass from one statement to the one immediately beneath, except where a jump (e.g. a loop) dictates otherwise (i.e. the Von Neumann concept). However each statement, or group of statements, are not necessarily dependent on their predecessors. By finding parts of a program which are independent it will be possible to advantageously use a parallel processing system of the type mentioned above. Thus, any approach to determine parallelism, at this level, will have to study dependencies between one or more program areas. In this context six main areas may be considered, these being:-

- (i) Individual statements.
- (ii) Groups of assignment statements.
- (iii) Blocks of Algol-type code.
- (iv) Iterations of a loop.
- (v) Conditional statements.
- (vi) Execution of procedures (or similar) after calls.

A suitable term for referring to these areas would be 'block' but because of the possible ambiguities when considering Algol-type

programming languages another term should be used. So a new term 'stanza' will be introduced to represent any of these six categories.

A stanza can be defined as follows:

Definition 4.1

A stanza is either a single program statement or a group of statements appearing adjacently in a computer program and intended to be executed one after the other.

The existing approaches to determining parallelism between stanzas may be divided into two classes. The first are the methods which use graph theory as part of their detection process and the second are all other methods. Since both methods detect independencies there will be some overlap in the techniques used in both approaches.

Process 1

a←b+c;

x←v1;

g←h-i;

Process 2

d←e+f;

x←v2;

j←k-l;

Figure 4.1

TWO PARALLEL PROCESSES

#### 4.2 USAGE OF PRIVATE AND SHARED MEMORIES

Wilkes (1965) introduced the idea of using a slave memory of fast core to save on the time spent fetching data items from main memory. Although such time delays are now of less significance it is possible to apply this concept to a parallel processing environment.

Within a parallel processing environment all processors should be permitted to access a main memory so that more than one processor may work on a set of inter-related stanzas. Thus the main memory can be considered to be 'shared' by all processors. In addition it is possible to allow each processor to have a private memory that can be used in the same manner as a slave memory. Thus all variables once used in a stanza would be stored in the processor's private memory until the stanza has been completed, when they may be transferred to the main (or shared) memory.

Thus, there are two types of memory structures that can reasonably be used in a parallel processing environment. Either all processors just use the main memory or each processor has attached to it a private memory in which information that is currently being processed can be temporarily stored. Figure 4.2(a) illustrates the former case where only shared memory is available while the latter memory structure is shown in Figure 4.2(b).

The difference between the two memory structures can be emphasised by considering two processors P1 and P2 operating in parallel. Both use a set of locations L which P1 alters and P2 fetches. Then, if P1 and P2 have private memories then P2 will fetch the original values of L. Whereas if P1 and P2 only have access to a shared main memory then there are three possible values of L that P2 may, theoretically, fetch. The values of L may be the original values, the values assigned in P1 or some undefined values which would indicate P2 was fetching L during the time P1 was changing it.

Bernstein (1966; see also the following section) shows that the conditions necessary to execute two stanzas in parallel are much weaker when processors with private memory are available. This is mainly due to the avoidance of problems similar to the one described above where the values that will be fetched were not defined, since both processors were using the same memory. Hoogendoorn (1975) has suggested how the 'mechanics' of providing each processor with a private memory may be implemented. It is possible that control can be exercised over the order in which private memory restores to main memory. This facility will be considered to be available throughout this work whenever machines with private memories are discussed.

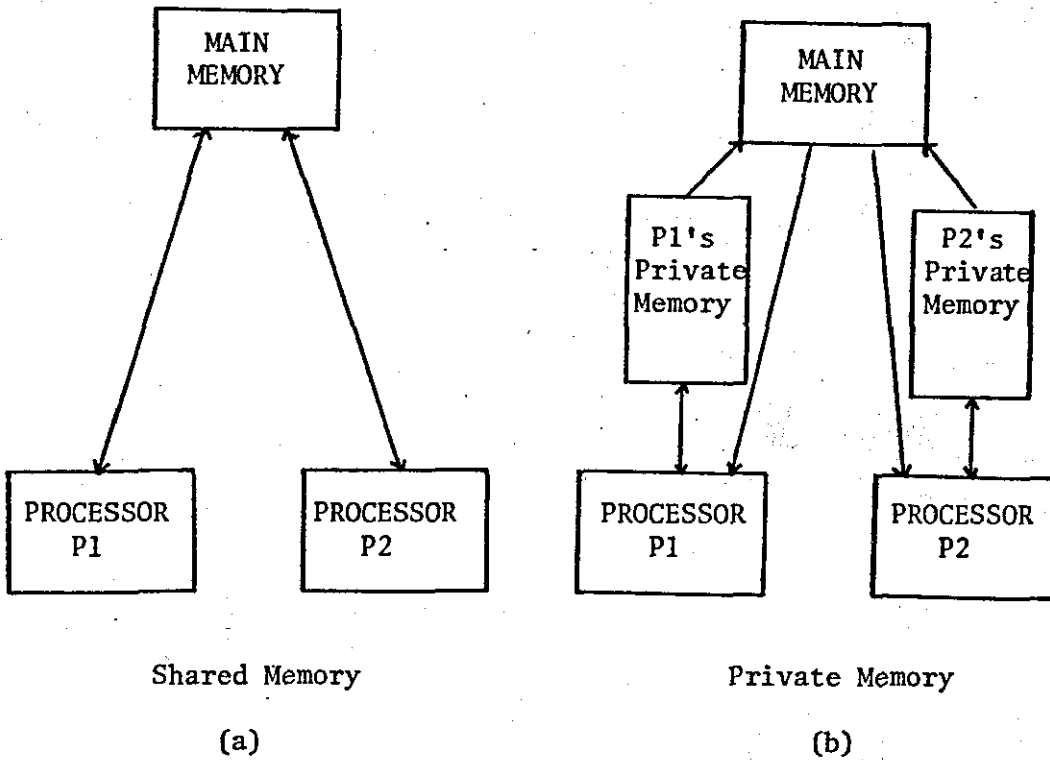


Figure 4.2

MEMORY STRUCTURES FOR PARALLEL PROCESSORS



#### 4.3 EXISTING TECHNIQUES FOR RECOGNISING PARALLELISM BETWEEN STANZAS

The methods already in existence for determining parallelism at the statement level can be considered in two categories. Those that use a great deal of graph theory in their determination of parallelism and all other methods. The other methods are usually based on aspects of the structure of the program.

##### 4.3.1 Graph Based Methods

Kuck (1975) credits Estrin and his students at U.C.L.A. of being the first to study program graphs in an attempt to locate parallelism (e.g. Martin and Estrin, 1967 and Baer and Russell, 1970). Kuck and his co-workers (e.g. Kuck et al, 1972; Kuck, 1975 and Towle, 1976) have continued this work, using data dependence graphs. Ramamoorthy and his co-workers (e.g. Ramamoorthy and Gonzalez, 1969; Gonzalez and Ramamoorthy, 1970 and 1971 and Ward, 1974) have taken a more formal approach based on a connectivity matrix of a graph representation of a program.

The work of Kuck is based on Fortran-like programming languages. Basically each statement is considered and its dependency on other statements is calculated. Loops formed by the 'DO' statement may be divided such that the dependencies between successive iterations can be found. The problems of using arrays indexed by a control variable of a loop are examined. Similarly, conditionals formed by the 'IF' statement are examined for indeterminism which may exist at execution time as well as compile time. Various types of conditionals are described which may be considered to be in two classes. Those for which the path that will be taken can be predicted and others. Combinations of loops and conditionals are also examined. Figure 4.3 shows a part of a Fortran program and the dependence graph Kuck (1977)

formed for the program. It can be seen, or found by partitioning, that  $S_2$  and  $S_3$  are completely independent of  $S_4$  and  $S_5$ . Hence  $S_2$  and  $S_3$  may be executed in parallel with  $S_4$  and  $S_5$ . It is also possible to determine that different instances of the arrays 'A' and 'H' are referred to in one iteration of the outer loop. Leasure (1976) gives a description of a compiler that will detect parallelism in serial programs in the manner described.

The work of Ramamoorthy is also based on the Fortran programming language. A program graph is derived which identifies the order in which tasks must be performed in a program written to be executed on a serial computer. In their work, tasks are treated as a single program statement and hence are a subset of the stanza defined in Definition 4.1. The program graph of  $N$  tasks is translated into an  $N \times N$  connectivity matrix. In the matrix a one will be used in position  $i, j$  to represent a directed edge between nodes  $i$  and  $j$ . Where  $i$  and  $j$  are integers in the range 1 to  $N$ . A zero will be inserted where there is not such connection between  $i$  and  $j$ . All tasks that create a strongly connected subgraph are treated as one task (or a stanza). So a reduced graph can be formed, for which there are no strongly connected subgraphs. A sufficiency condition is defined as given below.

#### Definition 4.2

Two tasks can be executed in parallel if the input set of one task does not depend on the output set of the other and vice versa.

This condition along with scheduling information is used to decide which tasks (stanzas) may be executed in parallel.

```

DO S5 I=1,N
S1: A(I)=B(I)*C(I)
DO S3 J=1,N
S2: D(J)=A(I-3)+E(J-1)
S3: E(J)=D(J-1)+F
DO S4 K=1,N
S4: G(K)=H(I-5)+1
S5: H(I)=SQRT(A(I-2))

```

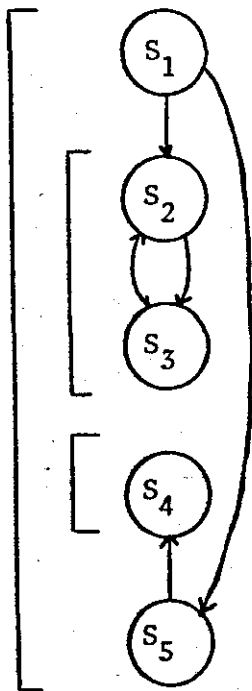


Figure 4.3

FORTAN PROGRAM AND DEPENDENCE GRAPH

#### 4.3.2 Methods Based on the Structure of the Program

In 1966 the Burroughs Corporation initiated research into the desirability and feasibility of automatically recognising parallelism within computer programs. The results of this work were discussed in a series of reports and papers (e.g. Bingham et al, 1967; Bingham and Reigel, 1968 and Reigel, 1970). Bernstein (1966) presented a different method based on set theory. More recently Firestone (1971) outlined a method of locating parallelism based on data flow analysis.

The Burroughs work was based on a subset of their B5500 Extended Algol programming language. An algorithm was developed that could detect implicit parallelism between various program structures such as loops and conditionals (but not blocks). Simulations of the workings of the algorithm have been written and are described in Bingham et al (1969). Bingham and Reigel (1969) stressed that they considered explicit exposure of parallelism (see Chapter 1) is necessary to detect parallelism between groups of statements.

Bernstein's work is based on the four ways in which a memory location may be used by a set of instructions or sub-program (or stanza)  $P_i$ . These are:-

- (1) The location is only fetched during the execution of  $P_i$ .
- (2) The location is only stored during the execution of  $P_i$ .
- (3) The first operation involving this location is a fetch.

One of the succeeding operations of  $P_i$  stores in this location.

- (4) The first operation involving this location is a store.

One of the succeeding operations of  $P_i$  fetches from this locations.

The set of all variables in  $P_i$  that fall into these categories are called  $W_i, X_i, Y_i$  and  $Z_i$  respectively. For two stanzas  $P_1$  and  $P_2$  to be capable of being executed in parallel, the following three conditions must hold:-

(1) The inputs of  $P_1$  must not coincide with the outputs of  $P_2$ ,

$$\text{i.e. } (W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset.$$

(2) The inputs of  $P_2$  must not coincide with the outputs of  $P_1$ ,

$$\text{i.e. } (X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \emptyset.$$

(3) Any location changed in both  $P_1$  and  $P_2$  must be reset before being reused,

$$\text{i.e. } (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap (W_3 \cup Y_3) = \emptyset.$$

Where  $(W_3 \cup Y_3)$  is the set of all variables subsequently fetched without being reset.

Table 4.1 gives the meanings of the notation used in set theory.

If each processor is allowed private memory three weaker conditions replace the above. These are:-

$$(1a) \quad (W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset.$$

$$(2a) \quad (X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2) = \emptyset.$$

$$(3a) \quad (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap (W_3 \cup Y_3) = \emptyset.$$

The weaker conditions apply because temporary results being formed by  $P_1$  can no longer be effected by  $P_2$  and vice versa.

Firestone (1971) developed a method of detecting implicit parallelism based on dependency. He uses the data flow analysis techniques of Kennedy (1971) to find independent parts of a program. Code that takes a 'long' time to execute is examined more thoroughly than code which has only a 'short' execution time. This method most closely resembles those based on graph theory. So Firestone's method could have been described in the previous subsection.

Symbol	Meaning
$\cup$	Union
$\cap$	Intersection
$A$	The set A
$A \cup B$	The set of all elements in A and B
$A \cap B$	The set of all elements in both A and B
$\emptyset$	The null or empty set.

Table 4.1

SYMBOLS USED IN SET THEORY

#### 4.4 CLASSIFICATION OF RELATIONSHIPS BETWEEN STANZAS

When using a parallel processing machine it is usually assumed that a stanza may be either executed sequentially after another stanza or simultaneously (Ramamoorthy and Gonzalez, 1969). Bernstein (1966) has suggested that two stanzas may be commutative. That is although they may not be executed in parallel either may be executed first. Towle (1976) stated that there are inter-relationships between data dependencies and control dependencies.

Here all possible relationships that may exist between stanzas are defined.

Initially only two stanzas will be studied but this will later be generalised to any number of stanzas. Consider two stanzas that would be executed one after the other in a serial program. The stanza that would have been executed first is called  $S_i$  and the other is  $S_{i+1}$ , for  $i$  such that  $1 \leq i \leq N$ , where  $N$  is the total number of stanzas in the program.

The five possible relationships that may exist between two such adjacent stanzas are now named and the conditions that must exist are defined:-

Definition 4.3: Contemporary -  $CT(S_i, S_{i+1})$

Stanzas  $S_i$  and  $S_{i+1}$  can be executed at the same time and the locations used in any order.

Definition 4.4: Commutative -  $CM(S_i, S_{i+1})$

Stanza  $S_i$  may be executed before or after  $S_{i+1}$  but not at the same time.

Definition 4.5: Prerequisite -  $PR(S_i, S_{i+1})$

Stanza  $S_i$  must fetch what it requires before  $S_{i+1}$  stores its results.

Definition 4.6: Conservative -  $CV(S_i, S_{i+1})$

Stanza  $S_i$  must store its results before  $S_{i+1}$  does.

Definition 4.7: Consecutive -  $CC(S_i, S_{i+1})$ 

Stanza  $S_i$  must store its results before  $S_{i+1}$  fetches what it requires.

For completeness two more relationships will be defined, which cannot sensibly exist within a serial program.

Definition 4.8: Synchronous -  $SN(S_i, S_{i+1})$ 

Stanzas  $S_i$  and  $S_{i+1}$  must both have the same inputs, i.e.  $S_i$  cannot store its results until  $S_{i+1}$  has fetched its input and vice versa.

Definition 4.9: Inclusive -  $IN(S_i, S_{i+1})$ 

Stanza  $S_{i+1}$  must store its results after  $S_i$  has fetched what it requires but before  $S_i$  stores its results.

It is possible to extend the relationships already defined for two stanzas to cover  $M$  stanzas  $\{S_1, S_2, \dots, S_M\}$ . Where  $S_k$  would be executed in the serial program immediately before  $S_{k+1}$ , for all  $k$  such that  $1 \leq k < M$ . The new definitions are:-

Definition 4.10: Contemporary -  $CT(S_1, S_2, \dots, S_M)$ 

Stanzas  $\{S_1, S_2, \dots, S_M\}$  can be executed at the same time, the ordering of fetching and storing being of no consequence.

Definition 4.11: Commutative -  $CM(S_1, S_2, \dots, S_M)$ 

The set of stanzas  $\{S_{i_1}, S_{i_2}, \dots, S_{i_M}\}$  may be executed in any possible order of the set  $\{i_1, i_2, \dots, i_M\}$  which is any permutation of the set  $\{1, 2, \dots, M\}$ , providing  $S_{i_k}$  is completed before  $S_{i_{k+1}}$  commences, for all  $k$  such that  $1 \leq k < M$ .

Definition 4.12: Prerequisite -  $PR(S_1, S_2, \dots, S_M)$ 

Stanza  $S_k$  must fetch what it requires before  $S_{k+1}$  stores its results, for all  $k$  such that  $1 \leq k < M$ .



Definition 4.13: Conservative -  $CV(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must store its results before  $S_{k+1}$  does, for all  $k$  such that  $1 \leq k < M$ .

Definition 4.14: Consecutive -  $CC(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must be completed before  $S_{k+1}$  commences, for all  $k$  such that  $1 \leq k < M$ .

Again, for completeness two more relationships will be defined, which, however, cannot sensibly exist in a serial program.

Definition 4.15: Synchronous -  $SN(S_1, S_2, \dots, S_M)$

Stanzas  $S_1, S_2, \dots, S_M$  must all receive the same input sets.

Definition 4.16: Inclusive -  $IN(S_1, S_2, \dots, S_M)$

Stanza  $S_{k+1}$  must store its results after  $S_k$  has fetched what it requires but before  $S_k$  has stored its results, for all  $k$  such that  $1 \leq k < M$ .

#### 4.5 FORMATION OF A STANZA

Using the terminology of Bernstein (1966) it is possible to define four sets for a given stanza  $S_i$ :-

- (1)  $W_i$  - represents the set of all locations that are only fetched during the execution of  $S_i$ .
- (2)  $X_i$  - represents the set of all locations that are only stored during the execution of  $S_i$ .
- (3)  $Y_i$  - represents the set of all locations for which the first operation is a fetch and one of the succeeding operations of  $S_i$  is a store.
- (4)  $Z_i$  - represents the set of all locations for which the first operation is a store and one of the succeeding operations of  $S_i$  is a fetch.

In Appendix 2 an Algol 68-R program Analyser is given that will divide a given serial Algol-type program into stanzas. Some of the work performed by Analyser (e.g. recognition of statements) is already performed by compilers and so could be removed from Analyser when it is integrated into a compiler. Analyser arbitrarily limits a stanza to be a specific program construct (e.g. a loop) or a collection of statements not using more than fifteen different variables. The variables used within a stanza  $S_i$  are classified as belonging to the sets  $W_i, X_i, Y_i$  and  $Z_i$  depending on their usage. Figure 4.4 illustrates how these sets are formed for a stanza consisting of three assignment statements.

A new set,  $V_i$ , is now introduced to represent all locations that may be fetched without being reset after the execution of the stanza  $S_i$ . The calculation of a particular  $V$  will, in general, be a non-trivial matter, in which case the  $V$  may be considered to be the set of all

variables used in the program.

Thus the input set of a stanza  $S_i$  is

$$W_i \cup Y_i$$

whereas the set of all variables fetched by  $S_i$  is

$$W_i \cup Y_i \cup Z_i$$

The output set of  $S_i$  is the same as the set of all variables stored in  $S_i$  and is

$$X_i \cup Y_i \cup Z_i$$

The set of all variables that on a serial machine will be fetched without being reset after the execution of  $S_i$  are represented by  $V_i$ .

<u>Stanza <math>S_i</math></u>	<u><math>W_i</math></u>	<u><math>X_i</math></u>	<u><math>Y_i</math></u>	<u><math>Z_i</math></u>
$a1 \leftarrow b1 * b2;$	b1,b2	a1	-	-
$a2 \leftarrow a1 * b1;$	b1,b2	a2	-	a1
$c1 \leftarrow a1 + c1;$	b1,b2	a2	c1	a1

Figure 4.4

FORMATION OF THE W,X,Y AND Z SETS

CHAPTER 5

DETECTION OF PARALLELISM BETWEEN STANZAS

## 5.1 TESTS TO EXPOSE THE RELATIONSHIPS BETWEEN TWO STANZAS

Two stanzas  $S_i$  and  $S_{i+1}$  which would have been executed one after the other in a serial program will be considered. The sets of usage of variables described in Section 4.5 will be used to form tests to determine which of the relationships defined in Section 4.4 exist between two stanzas.

The differences between parallel machines with private memories and those without have been mentioned previously. To allow for these, separate tests will be developed for both machines and these will be detailed in the following subsections.

### 5.1.1. Private Memories Available

The tests for the five possible relations defined in Definitions 4.3 to 4.7 will be developed individually. The fact that each stanza's temporary results will be stored in private memory will be taken into account where necessary when a relationship is considered.

#### (1) *Contemporary* - $CT(S_i, S_{i+1})$

This relationship implies that stanzas  $S_i$  and  $S_{i+1}$  may be executed simultaneously. Thus, there must not be any dependencies between the locations fetched by  $S_i$  and those changed by  $S_{i+1}$  and vice versa. As private memories are available any temporary results formed by one stanza cannot be altered by the other. Thus there must not be any dependencies between the inputs and outputs of  $S_i$  and  $S_{i+1}$ . In terms of sets that is:

$$(W_i \cup Y_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = \emptyset \quad (5.1)$$

$$\text{and} \quad (X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) = \emptyset \quad (5.2)$$

Locations that are modified by both stanzas  $S_i$  and  $S_{i+1}$  must not be used elsewhere without being reset first, since the values of such locations are undefined. That is:

$$(X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap V_{i+1} = \emptyset . \quad (5.3)$$

Thus the conditions for two stanzas  $S_i$  and  $S_{i+1}$  to be considered *contemporary* are (5.1), (5.2) and (5.3).

(2) *Commutative* -  $CM(S_i, S_{i+1})$

Stanza  $S_i$  may be executed before or after stanza  $S_{i+1}$ . Thus none of the inputs of  $S_i$  (or  $S_{i+1}$ ) must not coincide with any of the outputs of  $S_{i+1}$  (or  $S_i$ ). That is:

$$(W_i \cup Y_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = \emptyset \quad (5.4)$$

and  $(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) = \emptyset . \quad (5.5)$

Locations that are modified by both  $S_i$  and  $S_{i+1}$  must not be used elsewhere without first being reset. Since the value of such locations are undefined. That is:

$$(X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap V_{i+1} = \emptyset . \quad (5.6)$$

Thus the conditions for two stanzas to be considered *commutative* are (5.4), (5.5) and (5.6). It can be seen that by using private memory that the condition for two stanzas to be *commutative* are identical to those for them to be *contemporary*.

(3) *Prerequisite* -  $PR(S_i, S_{i+1})$

Stanza  $S_i$  must fetch what it requires before  $S_{i+1}$  stores its results. This implies that at least one of  $S_i$ 's inputs corresponds to an output of  $S_{i+1}$ , that is:

$$(W_i \cup Y_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \neq \emptyset . \quad (5.7)$$

Stanza  $S_{i+1}$  must not require information computed in  $S_i$ , since  $S_i$  will not necessarily be completed, that is:

$$(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) = \emptyset . \quad (5.8)$$

Locations that are modified in both  $S_i$  and  $S_{i+1}$  must not be used elsewhere without being reset first, since the values of such locations are undefined, that is:

$$(X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap W_{i+1} = \emptyset . \quad (5.9)$$

Thus, (5.8) and (5.9) are the conditions to be satisfied for  $S_i$  and  $S_{i+1}$  to be *prerequisite*. If (5.7) is also true then it can be seen that the relationship is neither *contemporary* or *commutative*.

(4) *Conservative* -  $CV(S_i, S_{i+1})$

Stanza  $S_i$  must store its results before  $S_{i+1}$  does. This implies that at least one location is changed by both  $S_i$  and  $S_{i+1}$  and subsequently fetched without being reset, that is:

$$(X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap W_{i+1} = \emptyset . \quad (5.10)$$

Stanza  $S_{i+1}$  must not require information computed in  $S_i$ , since  $S_i$  will not necessarily be completed, that is:

$$(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) = \emptyset . \quad (5.11)$$

This is the only condition necessary to be satisfied for  $S_i$  and  $S_{i+1}$  to be *conservative*. If (5.10) is also satisfied it can be seen that the relationship is not *prerequisite*.

(5) *Consecutive* -  $CC(S_i, S_{i+1})$

Stanza  $S_i$  must store its results before  $S_{i+1}$  fetches what it requires. This implies that at least one location changed by  $S_i$  is fetched by  $S_{i+1}$ , that is

$$(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) \neq \emptyset . \quad (5.12)$$

Thus any two stanzas  $S_i$  and  $S_{i+1}$  may be considered to be *consecutive*. If (5.12) is satisfied it can be seen that the relationship is not *conservative*.

### 5.1.2 Only Shared Memory Available

The tests for the five possible relationships defined in Definition 4.3 to 4.7 will be developed individually. The effects of a stanza's temporary results being available to the other stanza

will be taken in to account where necessary.

(1) *Contemporary* - CT( $S_i, S_{i+1}$ )

Stanzas  $S_i$  and  $S_{i+1}$  can be executed at the same time. Thus, there must be no dependencies between the set of locations that are fetched during the execution of  $S_i$  and those that are stored during the execution of  $S_{i+1}$  and vice versa, that is:

$$(W_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = \emptyset \quad (5.13)$$

and  $(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = \emptyset$  . (5.14)

Locations that are modified by both  $S_i$  and  $S_{i+1}$  must not be used elsewhere without being reset first, that is:

$$(X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap V_{i+1} = \emptyset \quad (5.15)$$

Thus the conditions for two stanzas to be considered to be *contemporary*, when only shared memory is available, are (5.13), (5.14) and (5.15).

(2) *Commutative* - CM( $S_i, S_{i+1}$ )

Stanza  $S_i$  may be executed before or after stanza  $S_{i+1}$ . Thus none of the inputs of  $S_i$  (or  $S_{i+1}$ ) must not coincide with any of the outputs of  $S_{i+1}$  (or  $S_i$ ), that is:

$$(W_i \cup Y_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = \emptyset \quad (5.16)$$

and  $(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) = \emptyset$  . (5.17)

Locations that are modified by both  $S_i$  and  $S_{i+1}$  must not be used elsewhere without first being reset, since the value of such locations are undefined, that is:

$$(X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap V_{i+1} = \emptyset \quad (5.18)$$

Thus the conditions for two stanzas to be considered *commutative* are (5.16), (5.17) and (5.18). It can be seen that these conditions are weaker than those for  $S_i$  and  $S_{i+1}$  to the *contemporary* stanzas.



(3) *Prerequisite* -  $PR(S_i, S_{i+1})$ 

Stanza  $S_i$  must fetch what it requires before  $S_{i+1}$  stores its results. As both stanzas are using the same memory it will be necessary to consider that there will be fetches and stores using main memory throughout the execution of both the stanzas. The very last fetch of  $S_i$  must be completed before the first store of  $S_{i+1}$ , thus the relationship can be considered to degenerate into a *consecutive* one.

(4) *Conservative* -  $CV(S_i, S_{i+1})$ 

Stanza  $S_i$  must store its results before  $S_{i+1}$  does. As both stanzas are using the same memory it will be necessary to consider that stores using main memory are occurring throughout the execution of both stanzas. The very last store of  $S_i$  must be completed before the first store of  $S_{i+1}$ , thus the relationship can be considered to degenerate into a *consecutive* one.

(5) *Consecutive* -  $CC(S_i, S_{i+1})$ 

Stanza  $S_i$  must store its results before  $S_{i+1}$  fetches what it requires. This implies that:

$$(X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) \neq \emptyset. \quad (5.19)$$

Thus any two stanzas  $S_i$  and  $S_{i+1}$  may be considered to be *consecutive*.

For a machine for which only shared memory is available, any two stanzas  $S_i$  and  $S_{i+1}$  for which (5.16), (5.17) and (5.18) are not true must be executed in a *consecutive* manner.

Table 5.1 is a summary of the conditions necessary for a given relationship to exist between two stanzas  $S_i$  and  $S_{i+1}$ , which would be executed one after the other in a serial program. The conditions for a particular relationship to exist between two stanzas are 'weaker' for those at the bottom of the table. The 'strongest' conditions being

those for two stanzas to be *contemporary* in a shared memory environment.

It is possible to simplify some of the tests given. For example, consider the *contemporary* relationships  $CT(S_i, S_{i+1})$ , in a private memory environment. From equations (5.1) and (5.2) it is possible to simplify (5.3) to:

$$(X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1}) \cap W_{i+1} = \emptyset \quad (5.3a)$$

However, this detracts from the clarity of the method and so is not used here.

Relationship	Conditions	
	Private Memories	Shared Memory
<i>Contemporary</i> CT( $S_i, S_{i+1}$ )	$(W_{i,i} \cup Y_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (W_{i+1,i+1} \cup Y_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) \cap V_{i+1} = \emptyset$	$(W_{i,i} \cup Y_i \cup Z_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (W_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) \cap V_{i+1} = \emptyset$
<i>Commutative</i> CM( $S_i, S_{i+1}$ )	as <i>Contemporary</i>	$(W_{i,i} \cup Y_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (W_{i+1,i+1} \cup Y_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) \cap V_{i+1} = \emptyset$
<i>Prerequisite</i> PR( $S_i, S_{i+1}$ )	$(X_{i,i} \cup Y_i \cup Z_i) \cap (W_{i+1,i+1} \cup Y_{i+1,i+1}) = \emptyset$ $(X_{i,i} \cup Y_i \cup Z_i) \cap (X_{i+1,i+1} \cup Y_{i+1,i+1} \cup Z_{i+1,i+1}) \cap V_{i+1} = \emptyset$	as <i>Consecutive</i>
<i>Conservative</i> CV( $S_i, S_{i+1}$ )	$(X_{i,i} \cup Y_i \cup Z_i) \cap (W_{i+1,i+1} \cup Y_{i+1,i+1}) = \emptyset$	as <i>Consecutive</i>
<i>Consecutive</i> CC( $S_i, S_{i+1}$ )	No conditions necessary as this implies $(X_{i,i} \cup Y_i \cup Z_i) \cap (W_{i+1,i+1} \cup Y_{i+1,i+1}) = \emptyset$	

Table 5.1

CONDITIONS NECESSARY FOR A GIVEN RELATIONSHIP TO EXIST BETWEEN TWO STANZAS

## 5.2 TESTS TO EXPOSE A SINGLE RELATIONSHIP BETWEEN A NUMBER OF STANZAS

A number,  $M$ , of stanzas  $\{S_1, S_2, \dots, S_M\}$  which would have been executed one after the other in a serial program will be considered. The sets of usage of variables described in Section 4.5 will be used to determine if a single relationship as defined in Section 4.4 exists between these  $M$  stanzas. It is possible that more than one relationship may exist within a group of stanzas in which case the tests will reveal the relationship which exists between all of the stanzas. Alternatively the group may be subdivided such that only one relationship exists within each of the new groups.

The difference between parallel machines with private memories and those without have been discussed previously. To allow for these, separate tests will be developed for both types of machine and will be detailed in the following subsections.

### 5.2.1 Private Memories Available

The tests for the five possible relationships defined in Definitions 4.10 to 4.14 will be developed individually. Within this subsection it is assumed that any processor used has its own private memory.

#### (1) *Contemporary* - $CT(S_1, S_2, \dots, S_M)$

This relationship implies that all of the stanzas  $\{S_1, S_2, \dots, S_M\}$  may be executed simultaneously. As private memories are available any temporary results formed by one stanza cannot be altered by any other stanza. Thus there must not be any dependencies between the inputs of one stanza and the outputs of all other stanzas. In terms of set theory that is:

$$(W_k \cup Y_k) \cap (X_\ell \cup Y_\ell \cup Z_\ell) = \emptyset$$

for all  $k$  such that  $1 \leq k \leq M$  and

for all  $\ell$  such that  $1 \leq \ell \leq M$  and  $\ell \neq k$ .

(5.20)

Locations that are modified by more than one stanza must not be fetched elsewhere without being reset first, since the value of such locations are undefined. That is:

$$(X_k UY_k UZ_k) \cap ((X_{k+1} UY_{k+1} UZ_{k+1}) \cap \dots \cap (X_M UY_M UZ_M)) \cap V_M = \emptyset \quad (5.21)$$

for all  $k$  such that  $1 \leq k < M$ .

Thus the conditions for  $M$  stanzas  $\{S_1, S_2, \dots, S_M\}$  to be considered *contemporary* are (5.20) and (5.21).

(2) *Commutative* -  $CM(S_1, S_2, \dots, S_M)$

The set of stanzas  $\{S_1, S_2, \dots, S_M\}$  may be executed in any possible order, providing that only one stanza is being executed at a given time. Thus the inputs of any one stanza must not coincide with any of the outputs of all other stanzas. That is:

$$(W_k UY_k) \cap (X_\ell UY_\ell UZ_\ell) = \emptyset$$

for all  $k$  such that  $1 \leq k \leq M$  and  $\ell \neq k$  . (5.22)

Locations that are modified by more than one stanza must not be fetched without first being reset, since the value of such locations are undefined. That is:

$$(X_k UY_k UZ_k) \cap ((X_{k+1} UY_{k+1} UZ_{k+1}) \cap \dots \cap (X_M UY_M UZ_M)) \cap V_M = \emptyset \quad (5.23)$$

for all  $k$  such that  $1 \leq k < M$ .

Thus the conditions for  $M$  stanzas  $\{S_1, S_2, \dots, S_M\}$  to be considered as *commutative* are (5.22) and (5.23). It can be seen that by using private memories the conditions for a given number of stanzas to be *commutative* are identical to those for them to be *contemporary*.

(3) *Prerequisite* -  $PR(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must fetch what it requires before  $S_{k+1}$  stores its results, for all values of  $k$  such that  $1 \leq k < M$ . This implies for all values of  $k$  at least one input of  $S_k$  corresponds to an output of  $S_{k+1}$ . That is:

$$(W_k UY_k) \cap (X_{k+1} UY_{k+1} UZ_{k+1}) \neq \emptyset \quad (5.24)$$

for all k such that  $1 \leq k < M$ .

Stanzas  $\{S_{k+1}, \dots, S_M\}$  must not require information computed in  $S_k$  since  $S_k$  will not necessarily be completed. That is:

$$(X_k UY_k UZ_k) \cap ((W_{k+1} UY_{k+1}) \cap \dots \cap (W_M UY_M)) = \emptyset \quad (5.25)$$

for all k such that  $1 \leq k < M$ .

Locations that are modified by more than one stanza must not be fetched without being reset first, since the value of such locations are undefined. That is:

$$(X_k UY_k UZ_k) \cap ((X_{k+1} UY_{k+1} UZ_{k+1}) \cap \dots \cap (X_M UY_M UZ_M)) \cap V_M = \emptyset \quad (5.26)$$

for all k such that  $1 \leq k < M$ .

Thus (5.25) and (5.26) are the conditions that must be satisfied for M stanzas  $\{S_1, S_2, \dots, S_M\}$  to be *prerequisite*.

#### (4) *Conservative* - $CV(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must store its results before  $S_{k+1}$  does, for all k such that  $1 \leq k < M$ . This implies for all values of k (from 1 to M-1) at least one location is changed by both  $S_k$  and  $S_{k+1}$  which is subsequently fetched without first being reset. That is:

$$(X_k UY_k UZ_k) \cap (X_{k+1} UY_{k+1} UZ_{k+1}) \cap V_M \neq \emptyset \quad (5.27)$$

for all k such that  $1 \leq k < M$ .

Stanzas  $\{S_{k+1}, \dots, S_M\}$  must not require information computed in  $S_k$  since  $S_k$  will not necessarily be completed. That is:

$$(X_k UY_k UZ_k) \cap ((W_{k+1} UY_{k+1}) \cap \dots \cap (W_M UY_M)) = \emptyset \quad (5.28)$$

for all k such that  $1 \leq k < M$ .

The conditions given in (5.28) are the only ones necessary for M stanzas  $\{S_1, S_2, \dots, S_M\}$  to be considered *conservative*.

#### (5) *Consecutive* - $CC(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must store its results before  $S_{k+1}$  fetches what it requires. This implies that for all values of k (between 1 and M-1)

at least one location is changed by  $S_k$  and fetched by  $S_{k+1}$ . That is:

$$\begin{aligned} (X_k \cup Y_k \cup Z_k) \cap (W_{k+1} \cup Y_{k+1}) \neq \emptyset \\ \text{for all } k \text{ such that } 1 \leq k < M. \end{aligned} \quad (5.29)$$

Thus any  $M$  stanzas  $\{S_1, S_2, \dots, S_M\}$  may be considered to be *consecutive*.

### 5.2.2 Only Shared Memory Available

The tests for the five possible relationships defined in Definitions 4.10 to 4.14 will be developed individually. The effects of a stanza's temporary results possibly being available to all other stanzas will be taken into account where necessary.

#### (1) *Contemporary* - $CT(S_1, S_2, \dots, S_M)$

This relationship implies that all the stanzas  $\{S_1, S_2, \dots, S_M\}$  may be executed simultaneously. There must be no dependencies between the set of locations that are fetched during the execution of any stanza and those that are stored during the execution of all the other stanzas. That is:

$$\begin{aligned} (W_k \cup Y_k \cup Z_k) \cap (X_l \cup Y_l \cup Z_l) = \emptyset \\ \text{for all } k \text{ such that } 1 \leq k \leq M \text{ and} \\ \text{for all } l \text{ such that } 1 \leq l \leq M \text{ and } l \neq k. \end{aligned} \quad (5.30)$$

Locations that are modified by more than one stanza must not be fetched elsewhere without first being reset, since the value of such locations are undefined. That is:

$$\begin{aligned} (X_k \cup Y_k \cup Z_k) \cap ((X_{k+1} \cup Y_{k+1} \cup Z_{k+1}) \cap \dots \cap (X_M \cup Y_M \cup Z_M)) \cap W_M = \emptyset \\ \text{for all } k \text{ such that } 1 \leq k < M. \end{aligned} \quad (5.31)$$

Thus the conditions for  $M$  stanzas to be considered *contemporary* are (5.30) and (5.31).

#### (2) *Commutative* - $CM(S_1, S_2, \dots, S_M)$

The set of stanzas  $\{S_1, S_2, \dots, S_M\}$  may be executed in any possible order, without more than one stanza being in execution at

a given time. Thus the inputs of any one stanza must not coincide with any of the outputs of all other stanzas. That is:

$$\begin{aligned} (W_k \cup Y_k) \cap (X_l \cup Y_l \cup Z_l) &= \emptyset \\ \text{for all } k \text{ such that } 1 \leq k < M \text{ and} & \\ \text{for all } l \text{ such that } 1 \leq l < M \text{ and } l \neq k. & \end{aligned} \quad (5.32)$$

Locations that are modified by more than one stanza must not be fetched elsewhere without first being reset, since the values of such locations are undefined. That is:

$$\begin{aligned} (X_k \cup Y_k \cup Z_k) \cap (X_{k+1} \cup Y_{k+1} \cup Z_{k+1}) \cap \dots \cap (X_M \cup Y_M \cup Z_M) \cap W_M &= \emptyset \\ \text{for all } k \text{ such that } 1 \leq k < M. & \end{aligned} \quad (5.33)$$

Thus the conditions for  $M$  stanzas  $\{S_1, S_2, \dots, S_M\}$  to be considered *commutative* are (5.32) and (5.33).

(3) *Prerequisite* -  $PR(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must fetch what it requires before  $S_{k+1}$  stores its results, for all values of  $k$  between 1 and  $M-1$ . As all stanzas are using the same memory it will be necessary to consider that there will be fetches and stores using the main memory throughout the execution of all stanzas. Thus the very last fetch of  $S_k$  must be completed before the first store of  $S_{k+1}$  so the relationship can be considered to degenerate into a *consecutive* one.

(4) *Conservative* -  $CV(S_1, S_2, \dots, S_M)$

Stanza  $S_k$  must store its results before  $S_{k+1}$  does, for all values of  $k$  between 1 and  $M-1$ . As all stanzas are using the same memory, it will be necessary to consider that stores using the main memory are occurring throughout the execution of all the stanzas. Thus, the very last store of  $S_k$  must occur before the first store of  $S_{k+1}$ . So, again, the relationship can be considered to degenerate into a *consecutive* one.



(5) *Consecutive* -  $CC(S_1, S_2, \dots, S_M)$ 

Stanza  $S_k$  must store its results before  $S_{k+1}$  fetches what it requires. This implies that for all values of  $k$  (between 1 and  $M-1$ ) at least one location is changed by  $S_k$  and fetched by  $S_{k+1}$ . That is:

$$(X_k \cup Y_k \cup Z_k) \cap (W_{k+1} \cup Y_{k+1}) \neq \emptyset \quad (5.34)$$

for all  $k$  such that  $1 \leq k < M$ .

Thus any  $M$  stanzas  $\{S_1, S_2, \dots, S_M\}$  may be considered to be *consecutive*.

Table 5.2 is a summary of the conditions necessary for a given relationship to exist between  $M$  stanzas, which would be executed one after the other in a serial program. Again, the 'weaker' conditions for a particular relationship to exist are at the bottom of the table. The 'strongest' conditions being those for  $M$  stanzas to be *contemporary* in a shared memory environment.

Again, simplifications are not applied to any of the conditions to maintain the clarity of the method.

Relationship	Conditions	
	Private Memories	Shared Memory
Contemporary CT(S <sub>1</sub> , S <sub>2</sub> , ..., S <sub>M</sub> )	$(W_k U Y_k) \cap (X_l U Y_l U Z_l) = \emptyset^{\dagger}$ $(X_k U Y_k U Z_k) \cap ((X_{k+1} U Y_{k+1} U Z_{k+1}) \cup \dots \cup (X_M U Y_M U Z_M)) \cap V_M = \emptyset^{\ddagger}$	$(W_k U Y_k U Z_k) \cap (X_l U Y_l U Z_l) = \emptyset^{\dagger}$ $(X_k U Y_k U Z_k) \cap ((X_{k+1} U Y_{k+1} U Z_{k+1}) \cup \dots \cup (X_M U Y_M U Z_M)) \cap V_M = \emptyset^{\ddagger}$
Commutative CM(S <sub>1</sub> , S <sub>2</sub> , ..., S <sub>M</sub> )	as Contemporary	$(W_k U Y_k) \cap (X U Y U Z) = \emptyset^{\dagger}$ $(X_k U Y_k U Z_k) \cap ((X_{k+1} U Y_{k+1} U Z_{k+1}) \cup \dots \cup (X_M U Y_M U Z_M)) \cap V_M = \emptyset^{\ddagger}$
Prerequisite PR(S <sub>1</sub> , S <sub>2</sub> , ..., S <sub>M</sub> )	$(X_k U Y_k U Z_k) \cap ((W_{k+1} U Y_{k+1}) \cup \dots \cup (W_M U Y_M)) = \emptyset^{\ddagger}$ $(X_k U Y_k U Z_k) \cap ((X_{k+1} U Y_{k+1} U Z_{k+1}) \cup \dots \cup (X_M U Y_M U Z_M)) \cap V_M = \emptyset^{\ddagger}$	as Consecutive
Conservative CV(S <sub>1</sub> , S <sub>2</sub> , ..., S <sub>M</sub> )	$(X_k U Y_k U Z_k) \cap ((X_{k+1} U Y_{k+1} U Z_{k+1}) \cup \dots \cup (X_M U Y_M U Z_M)) \cap V_M = \emptyset^{\ddagger}$	as Consecutive
Consecutive CC(S <sub>1</sub> , S <sub>2</sub> , ..., S <sub>M</sub> )	No conditions necessary as this implies $(X_k U Y_k U Z_k) \cap (W_{k+1} U Y_{k+1}) \neq \emptyset^{\ddagger}$	

<sup>†</sup> for all k such that 1 ≤ k ≤ M and for all l such that 1 ≤ l ≤ M and l ≠ k

<sup>‡</sup> for all k such that 1 ≤ k < M

TABLE 5.2

CONDITIONS NECESSARY FOR A SINGLE GIVEN RELATIONSHIP TO EXIST BETWEEN M STANZAS

### 5.3 ASSIGNMENT STANZAS

A stanza which only contains assignment statements can be called an Assignment stanza or an As-stanza. The relationships that exist between As-stanzas can be readily found by testing the conditions given in Table 5.1 or 5.2. An example of how these tests are carried out will now be given.

Figures 5.1(a) and (b) give two examples of assignment stanzas  $S_i$  and  $S_{i+1}$ . The sets of usage of variables, described in Section 4.5, are given for each stanza in Figures 5.2(a) and (b) respectively. Assume that in the original program  $S_i$  was written to be executed immediately before  $S_{i+1}$ . As nothing is known about any subsequent statements used in the program  $V_{i+1}$  will be considered to be the full set. The conditions given in Table 5.1 will be used to derive the relationship that can exist between  $S_i$  and  $S_{i+1}$ . Figures 5.3 and 5.4 show how the tests are carried out for machines with and without private memories. It can be seen from these that if private memories are available the relationship between  $S_i$  and  $S_{i+1}$  may be considered to be *prerequisite*. Otherwise the relationship must be considered to be *consecutive*.

BEGIN	BEGIN
$a1 \leftarrow b1 + c1;$	$a3 \leftarrow b1 + b2;$
$a2 \leftarrow a1 * b1;$	$b2 \leftarrow b1 / d1;$
$c1 \leftarrow b1 + b2$	$d2 \leftarrow a3 - d1$
END	END
(a) Stanza $S_i$	(b) Stanza $S_{i+1}$

Figure 5.1

TWO ASSIGNMENT STANZAS

$W_i$	$b1, b2$	$W_{i+1}$	$b1, d1$
$X_i$	$a2$	$X_{i+1}$	$d2$
$Y_i$	$c1$	$Y_{i+1}$	$b2$
$Z_i$	$a1$	$Z_{i+1}$	$a3$
(a)		(b)	

Figure 5.2

SETS OF USAGE OF VARIABLES FOR STANZAS  $S_i$  AND  $S_{i+1}$

$$\begin{aligned} & (X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) \\ & (a \cup c \cup a1) \cap ((b1, d1) \cup b2) = \emptyset \end{aligned}$$

∴ The relationship is at least *Conservative*

$$\begin{aligned} & (X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap V_{i+1} \\ & (a \cup c \cup a1) \cap (d \cup b \cup a3) = \emptyset \end{aligned}$$

∴ The relationship is at least *Prerequisite*

$$\begin{aligned} & (W_{i+1} \cup Y_{i+1}) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \\ & ((b1, b2) \cup c1) \cap (d \cup b \cup a3) = b2 \neq \emptyset \end{aligned}$$

∴ The relationship is not *Contemporary*

Figure 5.3

RELATIONSHIPS BETWEEN  $S_i$  AND  $S_{i+1}$  USING PRIVATE MEMORIES

$$\begin{aligned} & (W_i \cup Y_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \\ & ((b1, b2) \cup c1) \cap (d \cup b \cup a3) = b2 \neq \emptyset \end{aligned}$$

$$\begin{aligned} & (X_i \cup Y_i \cup Z_i) \cap (W_{i+1} \cup Y_{i+1}) \\ & (a \cup c \cup a1) \cap ((b1, d1) \cup b2) = \emptyset \end{aligned}$$

$$\begin{aligned} & (X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) \cap V_{i+1} \\ & (a \cup c \cup a1) \cap (d \cup b \cup a3) = \emptyset \end{aligned}$$

∴ The relationship is not *Commutative*

Figure 5.4

RELATIONSHIPS BETWEEN  $S_i$  AND  $S_{i+1}$  WITHOUT PRIVATE MEMORIES

## 5.4 PARALLELISM WITHIN LOOPS

### 5.4.1 Simple Loops

In this section a stanza that forms the body of a loop (i.e. a Do-stanza) will be considered. This stanza will be executed a number of times (the exact number depending on various control mechanisms such as the value of a control variable). A separate stanza may be formed for each possible iteration of a loop. Then by forming the sets described in Section 4.5 the relationships that exist between iterations may be found. Here a limited sub-set of loops will be considered and methods will be proposed to readily determine the relationships between iterations of a loop.

Initially only loops that obey the following constraints will be considered:

- (i) Only one variable (the control variable) is used to limit the number of iterations a loop performed.
- (ii) The amount by which the control variable is altered for each iteration (i.e. the step size) should be constant.
- (iii) The loop may not be exited on a condition.
- (iv) Each iteration only varies in locations accessed via the control variable plus or minus a constant.
- (v) Any location accessed via the control variable is not capable of being accessed in any other manner.

Some theoretical assertions about loops will now be made, which will be shown to be correct for the subset of loops being considered.

#### Theorem 5.1: Total Independence

When all assignments within a loop are to be members of arrays indexed via the control variable and any element of such an array, other than the one assigned to, is not used elsewhere in the Do-stanza,

then each iteration of the loop is completely independent of all others.

Proof

Consider a loop to be iterated  $N$  times and an iteration of the loop to be represented by  $S_k$  where  $1 \leq k \leq N$ . Then the conditions of the theorem give:

$$\begin{aligned} (W_k \cup X_k \cup Y_k \cup Z_k) \cap (X_\ell \cup Y_\ell \cup Z_\ell) &= \emptyset \\ \text{for all } k \text{ such that } 1 \leq k \leq N \text{ and} & \\ \text{for all } \ell \text{ such that } 1 \leq \ell \leq N \text{ and } \ell \neq k. & \end{aligned} \quad (5.35)$$

The conditions for a group of stanzas to be *contemporary* are given in (5.20) and (5.21) when private memories are available and (5.30) and (5.31) otherwise. From (5.35) the following three equations can be derived:

$$\begin{aligned} (W_k \cup Y_k) \cap (X_\ell \cup Y_\ell \cup Z_\ell) &= \emptyset \\ \text{for all } k \text{ such that } 1 \leq k \leq N \text{ and} & \\ \text{for all } \ell \text{ such that } 1 \leq \ell \leq N \text{ and } \ell \neq k. & \end{aligned} \quad (5.36)$$

$$\begin{aligned} (W_k \cup Y_k \cup Z_k) \cap (X_\ell \cup Y_\ell \cup Z_\ell) &= \emptyset \\ \text{for all } k \text{ such that } 1 \leq k \leq N \text{ and} & \\ \text{for all } \ell \text{ such that } 1 \leq \ell \leq N \text{ and } \ell \neq k. & \end{aligned} \quad (5.37)$$

$$\begin{aligned} (X_k \cup Y_k \cup Z_k) \cap (X_\ell \cup Y_\ell \cup Z_\ell) &= \emptyset \\ \text{for all } k \text{ such that } 1 \leq k \leq N \text{ and} & \\ \text{for all } \ell \text{ such that } 1 \leq \ell \leq N \text{ and } \ell \neq k. & \end{aligned} \quad (5.38)$$

It can be seen that (5.20) is the same as (5.36), (5.30) is the same as (5.37); and (5.21) and (5.31) are the same as (5.38) when  $V$  is taken to be the full set (the strongest condition). Hence all iterations of the loop may be executed simultaneously. Thus the theorem is proved.

Theorem 5.2: Repeated Relationships

The relationship between the  $j^{\text{th}}$  iteration of a loop and the  $(k+j)^{\text{th}}$  is the same as that between the  $i^{\text{th}}$  iteration and the  $(k+i)^{\text{th}}$ , where  $(k+i)$  and  $(k+j)$  are less than or equal to the number of iterations ( $N$ ) in the loop.

Proof

The sets of usage of variables used in  $S_i$  (i.e.  $W_i, X_i, Y_i$  and  $Z_i$ ) may be divided into two subsets i.e., those that are accessed via the control variable and all others that are independent of it. That is:

$$\begin{aligned} &cv_i^{W_i}, cv_i^{X_i}, cv_i^{Y_i} \text{ and } cv_i^{Z_i} \text{ represented by } cv_i^{S_i} \text{ and} \\ &o_i^{W_i}, o_i^{X_i}, o_i^{Y_i} \text{ and } o_i^{Z_i} \text{ represented by } o_i^{S_i}. \end{aligned}$$

Owing to the constraints given at the beginning of this subsection, that is:

$$\begin{aligned} &(o_k^{W_i} \cup o_k^{X_i} \cup o_k^{Y_i} \cup o_k^{Z_i}) \cap (cv_\ell^{W_i} \cup cv_\ell^{X_i} \cup cv_\ell^{Y_i} \cup cv_\ell^{Z_i}) = \emptyset \\ &\text{for all } k \text{ such that } 1 \leq k \leq N \text{ and} \\ &\text{for all } \ell \text{ such that } 1 \leq \ell \leq N. \end{aligned}$$

The tests for a given relationship (see Sections 5.1 and 5.2) may be considered in two parts.

The sets of  $o_j^{S_i}$  will be identical to those of  $o_i^{S_j}$  for all values of  $i$  and  $j$  such that  $1 \leq i, j \leq N$ . Hence the relationship ( $o$ R) between  $o_j^{S_i}$  and  $o_{k+j}^{S_i}$  will be the same as those between  $o_i^{S_j}$  and  $o_{k+i}^{S_j}$  (for  $(k+i) \leq N$ ).

Within the constraints given all members of  $cv^S$  are indexed by the control variable plus or minus a constant value. So all variables in  $cv_i^{S_i}$  will be off-set in their respective arrays by the same amount from those in  $cv_j^{S_j}$  for all values of  $i$  and  $j$  such that  $1 \leq i, j \leq N$ . Hence the relationship ( $cv$ R) between  $cv_j^{S_j}$  and  $cv_{k+j}^{S_j}$  will be the same as those between  $cv_i^{S_i}$  and  $cv_{k+i}^{S_i}$ .

The overall relationship between  $S_j$  and  $S_{k+j}$  will be the weaker of the two relationships  $o$ R and  $cv$ R. Similarly the overall relationship between  $S_i$  and  $S_{k+i}$  will be the weaker of  $o$ R and  $cv$ R. Hence the relationship between  $S_j$  and  $S_{k+j}$  will be the same as that between  $S_i$  and  $S_{k+i}$  and so the theorem is proved.



Corollary 5.1

The relationship between the first iteration of a loop and the  $(k+1)^{\text{th}}$  is the same as that between the  $i^{\text{th}}$  iteration and the  $(k+i)^{\text{th}}$  where  $(k+i)$  is less than or equal to the number of iterations in the loop.

Corollary 5.2: Pattern Recurrence

Within the constraints given earlier all the relationships between the  $m$  iterations starting at the  $j^{\text{th}}$  iteration are the same as those between the  $m$  iterations starting at the  $i^{\text{th}}$  iteration where  $(j+m)$  and  $(j+i)$  are both less than or equal to the number of iterations in the loop.

Corollary 5.3

The maximum number of relationships that need to be tested to establish all relationships within a loop is  $N-1$ , where  $N$  is the number of iterations performed for that loop.

Corollary 5.4: Total Dependence

If the relationship between the first and second iterations of a loop is *consecutive* then all iterations of that loop must be executed sequentially.

Now for a loop that complies with the constraints given earlier it can be readily found whether each iteration of the loop may be executed simultaneously or must be executed sequentially. It will only be necessary to determine the relationship between the first iteration and some other iterations as this will provide information about all other relationships, by applying the above theorems and corollaries.

5.4.2 Nested Loops

Nested loops will now be considered. A nested loop is a Do-stanza which is enclosed by more than one loop. The tests given previously

for a single loop, can be expanded to allow for nested loops. One more constraint will be introduced to those given at the beginning of subsection 5.4.2:

- (vi) Any array that is indexed by a control variable plus or minus a constant value is not to be used elsewhere in the Do-stanza indexed by the same control variable plus or minus a constant value in a different subscript position.

Consider  $\ell$  nested loops to be represented by  $\{L_1, L_2, \dots, L_\ell\}$  where  $L_\ell$  is the inner-most loop and  $L_1$  is the outer-most loop. The extensions to the tests will now be derived for these  $\ell$  nested loops.

### 1. Total Independence

Consider all assignments within a Do-stanza are to be arrays indexed by all the control variables of the loops  $\{L_1, L_2, \dots, L_\ell\}$  and none of these arrays are used anywhere else in the Do-stanza. Then each iteration of every loop may be executed simultaneously.

Otherwise for each loop  $\{L_1, L_2, \dots, L_\ell\}$ , for which the total independence test holds, every iteration may be executed simultaneously.

At this point it may be remarked that an N dimension array may be considered to consist of a number of independent N-1 dimension arrays. For example a three dimensional array  $A[1:x, 1:y, 1:z]$  can be considered to consist of x independent two dimensional arrays  $\{A[1, 1:y, 1:z], A[2, 1:y, 1:z], \dots, A[x, 1:y, 1:z]\}$ .

### 2. Total Dependence

For each loop  $\{L_1, L_2, \dots, L_\ell\}$  for which the total dependence test holds all iterations must be executed sequentially. If total dependence holds for all loops then all iterations of every loop must be executed sequentially.

### 3. Repeated Relationships

Repeated relationships need only be considered for those loops which are not totally independent or totally dependent. Each loop is then handled in the same manner as with single loops.

Figure 5.5 shows a nested loop, where  $l=3$ , that satisfies the constraints given previously. The tests described above will now be applied to this nested loop.

- (1) Form the sets of usage of variables, ignoring any subscripts

W	d
X	a[,,,],c[,,,]
Y	b[,,,]
Z	$\emptyset$

The whole of the nested loop cannot be totally independent as one array (b) is fetched and subsequently stored.

The loops {L1,L2,L3} will now be considered individually starting with the inner-most loop.

- (2) Form the sets of usage of variables for L3 including all the subscripts

W	d,b[i1+3,i2,i3+3],b[i1,i2,i3+2]
X	a[i1,i2,i3],c[i1,i2,i3],b[i1,i2,i3]
Y	$\emptyset$
Z	$\emptyset$

Since the array  $b[i1,i2, ]$  appears in both W and X the loop L3 is not totally independent. The repeated relationships are now examined

Iteration 1 of L3 -  $i3=1$

$W_1$	d,b[i1+3,i2,4],b[i1,i2,3]
$X_1$	a[i1,i2,1],b[i1,i2,1],c[i1,i2,1]
$Y_1$	$\emptyset$
$Z_1$	$\emptyset$

Iteration 2 of L3 -  $i3=2$

$W_2$	d,b[i1+3,i2,5],b[i1,i2,4]
$X_2$	a[i1,i2,2],b[i1,i2,2],c[i1,i2,2]
$Y_2$	$\emptyset$
$Z_2$	$\emptyset$

Carrying out the tests described in Section 5.1 reveals that these two iterations are *contemporary*.

Iteration 3 of L3 -  $i3+3$

$W_3$	$d, b[i1+3, i2, 6], b[i1, i2, 5]$
$X_3$	$a[i1, i2, 3], b[i1, i2, 3], c[i1, i2, 3]$
$Y_3$	$\emptyset$
$Z_3$	$\emptyset$

Again, carrying out the relationship tests shows that the first and third iterations are *consecutive*.

So for the whole of the loop L3 the iterations can be carried out in pairs that are *contemporary* and each set of pairs must be *consecutive*.

Thus an execution order may be:

$$CC(CT(L3_1, L3_2), CT(L3_3, L3_4), \dots, CT(L3_9, L3_{10})) ,$$

where  $L3_N$  is the  $N^{\text{th}}$  iteration of L3.

- (3) Form the sets of usage of variables for L2, including all subscripts except for those used in inner loops (i.e.  $i3$ )

$W$	$d, b[i1+3, i2]$
$X$	$a[i1, i2], c[i1, i2]$
$Y$	$b[i1, i2]$
$Z$	$\emptyset$

Since all arrays in the X, Y and Z sets are indexed by  $i2$  and each array only appears once the loop L2 is totally independent (N.B.  $b[i1, \cdot]$  is a different array to  $b[i1+3, \cdot]$ ).

Thus the execution order may be:

$$CT(L2_1, L2_2, \dots, L2_{10}).$$

- (4) Form the sets of usage of variables for L1, excluding all subscripts used in inner loops, (i.e.  $i2$  and  $i3$ )

$W$	$d, b[i+3]$
$X$	$a[i1], c[i1]$
$Y$	$b[i1]$
$Z$	$\emptyset$

Since the array  $b[ ]$  appears in W and Y, L1 is not totally independent.

The repeated relationships are now examined.

Iteration 1 of L1 -  $i1+1$

$W_1$      $d, b[4]$   
 $X_1$      $a[1], c[1]$   
 $Y_1$      $b[1]$   
 $Z_1$      $\emptyset$

Iteration 2 of L1 -  $i1+2$

$W_2$      $d, b[5]$   
 $X_2$      $a[2], c[2]$   
 $Y_2$      $b[2]$   
 $Z_2$      $\emptyset$

Carrying out the relationship tests as before shows that these two iterations are *contemporary*.

Iteration 3 of L1 -  $i1+3$

$W_3$      $d, b[5]$   
 $X_3$      $a[3], c[3]$   
 $Y_3$      $b[3]$   
 $Z_3$      $\emptyset$

The relationship between the first and third iteration can also be found to be *contemporary*.

Iteration 4 of L1 -  $i1+4$

$W_4$      $d, b[6]$   
 $X_4$      $a[4], c[4]$   
 $Y_4$      $b[4]$   
 $Z_4$      $\emptyset$

The relationship between the first and fourth iterations is *consecutive*. Thus an execution order of L1 may be:

$CC(CT(L1_1, L1_2, L1_3), CT(L1_4, L1_5, L1_6), CT(L1_7, L1_8, L1_9), L1_{10})$

So assuming the availability of 60 processing units the 1000 iterations can be executed in the time taken to execute 20 iterations of the loop sequentially (see Figure 5.6).

If an array within a nested loop is indexed by a given control variable in one subscript position and is later indexed by the same variable in a different position, it becomes difficult to predict the usage of a particular element of an array. This is why constraint (vi) was introduced for nested loops. However for certain loops it is possible to detect some type of 'wave front' relationship between iterations of the loops (see Kuck, 1975). Consider the simple nested loop in Figure 5.7. It can be seen that sometimes the value of  $a[i,j]$  will be set to a value previously set in the loop and otherwise the value will be one set outside the loop. Figure 5.8 indicates which iterations of Figure 5.7 depend on the old value (O) of an element being available, which depend on a new value (N) being available and which it does not matter for (X). It can be seen that for all values of  $i_1$  and  $j_1$  such that  $j_1 < i_1$  the  $i_1, j_1^{\text{th}}$  iteration must be executed before the  $j_1, i_1^{\text{th}}$  iteration and the  $i_1, i_1^{\text{th}}$  iteration may be done at any time. Similar solutions may be obtained for more complex Do-stanzas as explained in Kuck (1975).

```
FOR i1←1 STEP 1 UNTIL 10 DO
  FOR i2←1 STEP 1 UNTIL 10 DO
    FOR i3←1 STEP 1 UNTIL 10 DO
      BEGIN
        a[i1,i2,i3]←b[i1,i2,i3+2];
        b[i1,i2,i3]←d;
        c[i1,i2,i3]←b[i1+3,i2,i3+3]
      END
```

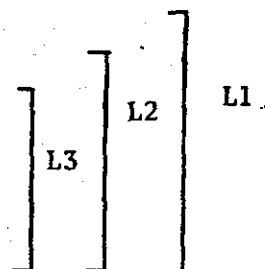


Figure 5.5

A NESTED LOOP

Processor Time	1	2	3	...	20	21	...	59	60
1	(1,1,1)	(1,1,2)	(1,2,1)		(1,10,2)	(2,1,1)		(3,10,1)	(3,10,2)
2	(1,1,3)	(1,1,4)	(1,2,3)		(1,10,4)	(2,1,3)		(3,10,3)	(3,10,4)
⋮									
5	(1,1,9)	(1,1,10)	(1,2,9)		(1,10,10)	(2,1,9)		(3,10,9)	(3,10,10)
6	(4,1,1)	(4,1,2)	(4,2,1)		(4,10,2)	(5,1,1)		(6,10,1)	(6,10,2)
⋮									
10	(4,1,9)	(4,1,10)	(4,2,9)		(4,10,10)	(5,1,9)		(6,10,1)	(6,10,2)
11	(7,1,1)	(7,1,2)	(7,2,1)		(7,10,2)	(8,1,1)		(9,10,1)	(9,10,2)
⋮									
15	(7,1,9)	(7,1,10)	(7,2,9)		(7,10,10)	(8,1,9)		(9,10,9)	(9,10,10)
16	(10,1,1)	(10,1,2)	(10,2,1)		(10,10,2)	not used			
⋮									
20	(10,1,9)	(10,1,10)	(10,2,9)		(10,10,10)				

where  $(i,j,k)$  represents the  $i^{\text{th}}$  iteration of L1,  
the  $j^{\text{th}}$  iteration of L2 and the  $k^{\text{th}}$  iteration of L3.

Figure 5.6

POSSIBLE EXECUTION ORDER OF A NESTED LOOP



```

FOR i←1 STEP 1 UNTIL 10 DO
  FOR j←1 STEP 1 UNTIL 10 DO
  BEGIN
    a[i,j]←a[j,i]
  END

```

Figure 5.7

NESTED LOOPS

i \ j	1	2	3	4	5	6	7	8	9	10
1	X	0	0	0	0	0	0	0	0	0
2	N	X	0	0	0	0	0	0	0	0
3	N	N	X	0	0	0	0	0	0	0
4	N	N	N	X	0	0	0	0	0	0
5	N	N	N	N	X	0	0	0	0	0
6	N	N	N	N	N	X	0	0	0	0
7	N	N	N	N	N	N	X	0	0	0
8	N	N	N	N	N	N	N	X	0	0
9	N	N	N	N	N	N	N	N	X	0
10	N	N	N	N	N	N	N	N	N	X

where 0≡Old value; N≡New value and X≡don't care

Figure 5.8

VALUE OF THE ELEMENTS FETCHED BY THE  $i, j^{\text{th}}$  ITERATION

### 5.5 CONDITIONAL STANZAS

Here a simple Algol-type IF statement will be considered which will be called an If-stanza. The If-stanza  $S_i$  may be considered in three parts:

- (i) The condition -  $c_{S_i}$ .
- (ii) The statements executed if the condition is true -  $T_{S_i}$ .
- (iii) The statements executed if the condition is false -  $F_{S_i}$ .

For each of these three it is possible to form the sets of usage of variables (see Section 4.5). These will be represented by:

$W_{c_i}$  - The variables tested in the condition (For a simple If-stanza, assignments will not be carried out in  $c_{S_i}$ ).

$T_{S_i}^X$   
 $T_{S_i}^Y$   
 $T_{S_i}^Z$  } The variables used when the condition was true.

$F_{S_i}^W$   
 $F_{S_i}^X$   
 $F_{S_i}^Y$   
 $F_{S_i}^Z$  } The variables used when the condition was false.

Since both of  $T_{S_i}$  and  $F_{S_i}$  cannot be executed for any value  $c_{S_i}$  the variables used in  $S_i$  will be given by

$$W_{c_i}^U W_{T_i}^U X_{T_i}^U Y_{T_i}^U Z_{T_i}^U$$

or

$$W_{c_i}^U W_{F_i}^U X_{F_i}^U Y_{F_i}^U Z_{F_i}^U .$$

Tests will be developed that will determine the relationship between an If-stanza ( $S_i$ ) and those stanzas executed immediately before it (A). Then further tests will show the relationship between the If-stanza ( $S_i$ ) and those stanzas executed immediately after it (P).

### 5.5.1 Relationships Between A and $S_i$

For clarity A may be considered as one stanza with the following sets of usage of variables:

$$\begin{array}{l} W_A \\ X_A \\ Y_A \\ Z_A \end{array}$$

By testing the relationship between A and  $S_i$  it is possible to readily detect if A and  $S_i$  must be executed as *consecutive* stanzas. This is done by testing the intersection of the output sets of A and the input sets of  $S_i$ . That is:

$$(X_A \cup Y_A \cup Z_A) \cap W_i = \emptyset \quad (5.39)$$

If the intersection is not empty A and  $S_i$  must be executed as *consecutive* stanzas. Otherwise further tests will need to be carried out to establish the relationships between A and  $S_i$ . These subsequent tests can be considered in two classes:

(1) A and  $T S_i$ :

The relationship ( $T R_i$ ) between A and  $T S_i$  is established using the tests given in Sections 5.1 or 5.2.

(2) A and  $F S_i$ :

The relationship ( $F R_i$ ) between A and  $F S_i$  is established using the tests given in Sections 5.1 or 5.2.

It can now be stated that when  $S_i$  is true the relationship between A and  $S_i$  is  $T R_i$  and otherwise it is  $F R_i$ . Figure 5.8 gives an example of an As-stanza followed by an If-stanza. The sets of usage of variables for both these stanzas are given in Figure 5.9. The tests described will not be carried out in the prescribed manner, for a machine with private memories available.

A and  $cS_1$

$$(X_A \cup Y_A \cup Z_A) \cap cW_1$$

$$((a,b) \cup c) \cap (g,h) = \emptyset$$

$\therefore$  A and  $S_1$  cannot be considered *consecutive*.

A and  $T_1S_1$

$$(X_A \cup Y_A \cup Z_A) \cap (T_1W_1 \cup T_1Y_1)$$

$$((a,b) \cup c) \cap (j,k) = \emptyset$$

$\therefore$  A and  $T_1S_1$  are at least *conservative*.

$$(X_A \cup Y_A \cup Z_A) \cap (T_1X_1 \cup T_1Y_1 \cup T_1Z_1) \cap V_1$$

$$((a,b) \cup c) \cap (i) = \emptyset$$

$\therefore$  A and  $T_1S_1$  are at least *prerequisite*.

$$(W_A \cup Y_A) \cap (T_1X_1 \cup T_1Y_1 \cup T_1Z_1)$$

$$((g,h,j,d,e,f) \cup c) \cap (i) = \emptyset$$

$\therefore$  A and  $T_1S_1$  are *contemporary*.

A and  $F_1S_1$

$$(X_A \cup Y_A \cup Z_A) \cap (F_1W_1 \cup F_1Y_1)$$

$$((a,b) \cup c) \cap ((l,m) \cup j) = \emptyset$$

$\therefore$  A and  $F_1S_1$  are at least *conservative*.

$$(X_A \cup Y_A \cup Z_A) \cap (F_1X_1 \cup F_1Y_1 \cup F_1Z_1) \cap V_1$$

$$((a,b) \cup c) \cap (j) = \emptyset$$

$\therefore$  A and  $F_1S_1$  are at least *prerequisite*.

$$(W_A \cup Y_A) \cap (F_1X_1 \cup F_1Y_1 \cup F_1Z_1)$$

$$((g,h,j,d,e,f) \cup c) \cap (j) \neq \emptyset$$

$\therefore$  A and  $F_1S_1$  are not *commutative* or *contemporary*.

Thus it can be seen that  $T_1R_1$  is *contemporary* and  $F_1R_1$  is *prerequisite*.

BEGIN			
a←g/h;			
b←j+c;			
c←d+e/f			
END;			
IF g=h THEN			
i←j+k		$S_1$	
ELSE		$T_1^{S_1}$	
j←j+1+m;		$F_1^{S_1}$	

Figure 5.8

AN AS-STANZA FOLLOWED BY AN IF-STANZA

$W_A$	g,h,j,d,e,f
$X_A$	a,b
$Y_A$	c
$Z_A$	$\emptyset$
$W_1$	g,h
$T_1^W$	j,k
$T_1^X$	i
$T_1^Y$	$\emptyset$
$T_1^Z$	$\emptyset$
$F_1^W$	ℓ,m
$F_1^X$	$\emptyset$
$F_1^Y$	j
$F_1^Z$	$\emptyset$

Figure 5.9

W,X,Y AND Z SETS FOR A AND  $S_1$

### 5.5.2 Relationship Between $S_i$ and P

For clarity P may be considered as one stanza with the following sets of usage of variables:

$$\begin{array}{l} W_P \\ X_P \\ Y_P \\ Z_P \end{array}$$

No test is available to readily detect a specific relationship between  $S_i$  and P. So it will be necessary, again, to establish relationships depending on possible values of a condition. Since nothing is known about the relationship between  $S_i$  and P it will be necessary to include the variables of  $S_i$  with those of  $T_i$  and  $F_i$  as necessary. The two classes of tests that will be carried out are:-

(i)  $T_i$  and P

The relationship ( $R_i$ ) between  $T_i$  and P is established as described in Sections 5.1 or 5.2. except  $W_i$  is included in all the input sets and sets of variables fetched for  $T_i$ . That is:

$$(W_i^U W_i^U Y_i) \text{ replaces } (T_i^U Y_i)$$

and  $(W_i^U W_i^U Y_i^U Z_i)$  replaces  $(T_i^U Y_i^U Z_i)$ .

(ii)  $F_i$  and P

The relationship ( $R_i$ ) between  $F_i$  and P is established as described in Sections 5.1 or 5.2, except that  $W_i$  is included in all the input sets and the sets of variables fetched for  $F_i$ . That is:

$$(W_i^U W_i^U Y_i) \text{ replaces } (F_i^U Y_i)$$

and  $(W_i^U W_i^U Y_i^U Z_i)$  replaces  $(F_i^U Y_i^U Z_i)$ .

It can now be stated that when  $S_i$  is true the relationship between  $S_i$  and P is  $T_i$  and otherwise it is  $F_i$ .

The tests described in subsections 5.5.1 and 5.5.2 may be extended to allow for more complicated Algol-type IF statements, where

assignments may take place in the condition  $c S_i$ . It will be necessary to form new sets of usage of variables:

$\left. \begin{array}{l} T_i^W \\ T_i^X \\ T_i^Y \\ T_i^Z \end{array} \right\}$	<p>The variables used in the condition and those used when the condition is true.</p>
$\left. \begin{array}{l} F_i^W \\ F_i^X \\ F_i^Y \\ F_i^Z \end{array} \right\}$	<p>The variables used in the condition and those used when the condition is false.</p>

These will combine all the variables used in  $S_i$ , and reflect that  $c S_i$  is always executed before  $T S_i$  or  $F S_i$ . For example a variable that appeared in both  $c X_i$  and  $T Y_i$  would be placed in  $T Z_i$ . The tests could then be carried out in the manner described previously.

When two adjacent If-stanzas are considered there are at the most four possible relationships between them (see Figure 5.10). However only one path will be taken through these stanzas, for a particular pass through this section of code. In general, the path to be taken will not be known until the stanzas are executed.

Since the number of paths through  $n$  adjacent If-stanzas is  $2^n$ , then for practical purposes it will be necessary to limit the number of adjacent If-stanzas considered at one time. However, for two adjacent If-stanzas the work is not onerous and the gains should be worthwhile.

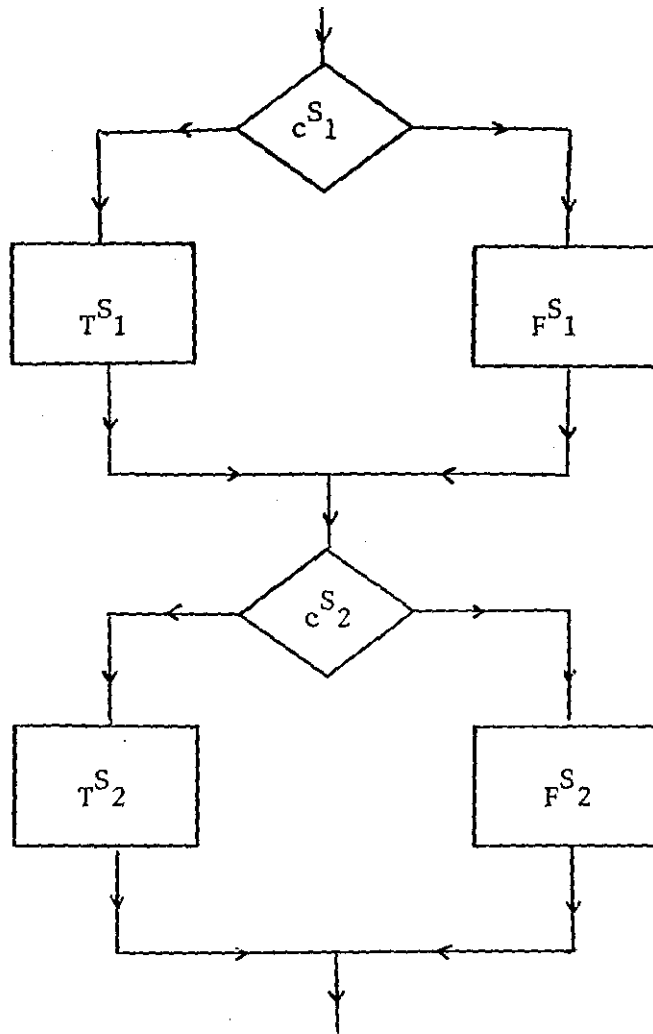


Figure 5.10

TWO ADJACENT IF-STANZAS



## 5.6 STANZAS CONTAINING LOOPS AND CONDITIONALS

A loop ( $S_L$ ) may be contained within an If-stanza ( $S_i$ ), it may appear in  $T S_i$ ,  $F S_i$  or, in the more complex If-stanzas,  $C S_i$ . The relationships between iterations of the loop ( $S_L$ ) may be calculated in the normal manner. Similarly, the relationships between the If-stanza and the surrounding program can be established in the manner described previously.

However, when an If-stanza ( $S_i$ ) appears in a loop ( $S_L$ ), there are three situations to be considered:

- (1) For any execution of  $S_L$  the same path will be taken through  $S_i$ .
- (2) For any execution of  $S_L$  one path will be taken through  $S_i$  up to a certain point after which the other path is taken through  $S_i$ .
- (3) For any execution of  $S_L$  the path taken through  $S_i$  will alter more than once.

How these situations can be detected and any potential for parallelism exploited will now be detailed.

- (1) The same path will always be taken through  $S_i$  when none of the variables set in  $S_L$  are fetched in  $C S_i$ . For certain trivial cases (such as testing if  $l=2$ ) only one path will ever be taken through  $S_i$  and so the relationships between iterations of the loop can be determined accordingly. However in general it will not be known which path will be taken until the program is executed. So it will be necessary to establish the relationships between iterations of  $S_L$  for both cases.
- (2) A 'switch-over' of this type will occur when the test is such that when one of  $T S_i$  or  $F S_i$  is taken it will always be taken. This may be because:
  - either (i) The control variable is tested for being smaller, or larger than a value (C) which is constant within  $S_L$ .

or (ii) A variable (V) which is only set within  $S_L$  in one of  $T S_i$  or  $F S_i$  is tested for being smaller or larger than a value (C) which is constant within  $S_L$ .

Again, except in trivial cases there will be insufficient information until execution to determine which path or paths will be taken. However, at execution some potential parallelism may be retained. Assume that the conditional  $C S_i$  has the value B on the first iteration of the loop  $S_L$  (where B is either true or false). Then while  $C S_i$  is equal to B the results of an iteration must be stored before those of the next iteration (i.e. the *conservative* or *consecutive* relationship). Then when the condition changes (i.e.  $C S_i$  is no longer equal to B) any iterations performed and not stored will be discarded. The remainder of the iterations can then be calculated, possibly in parallel depending upon the relationships between iterations when  $C S_i$  is not equal to B. Thus it will be necessary to calculate the relationships between iterations of the loop for when  $C S_i$  is true and when it is false.

- (3) An approach similar to the one described in (2) can be used here, where  $O S_i$  uses variables set in  $S_L$ . However since there is no way of determining which path will be taken in advance such loops will be iterated sequentially.

An example of an If-stanza within a loop is given in Figure 5.11. It can be seen that the variable tested in  $C S_i$  is set elsewhere in  $S_L$  and thus this is not the situation (1) discussed previously. However the variable ('x') tested in  $C S_i$  is only set within  $S_L$  in  $F S_i$  which is the second situation discussed above. Hence for any execution of  $S_L$  the path  $F S_i$  will be taken through  $S_i$  until the variable ('x') is greater than 5 then the path  $T S_i$  will be taken. Using the tests

described previously for loops it can be established that when  $S_{c_i}$  is true iterations of  $S_L$  are *contemporary* whereas when  $S_{c_i}$  is false the iterations are *conservative* (when private memories are available).

Assuming that

$$x=3 \quad \text{and} \quad j=1$$

then the first three iterations of the loop will be executed in a *conservative* order and the remaining seven will be executed in a *contemporary* order.

A loop which stops on a condition may be called a While-loop and the stanza ( $S_W$ ) that represents it may be considered in two parts:

- (i)  $S_{c_W}$  - the condition.
- (ii)  $S_{B_W}$  - the body of the loop.

Again except in trivial circumstances, it will not be possible to decide in advance for which iteration of the loop  $S_{c_W}$  will become false.

However if a machine with private memories is available some potential parallelism may be retained. The *conservative* relationship and a technique similar to pipelining (see Chapter 1) will be used. Whatever other conditions exists the results of the  $(i+1)^{th}$  iteration will not be stored until those of the  $i^{th}$  iteration have been stored (where  $i$  is less than the total maximum number of iterations of the loop). Suppose during the  $i^{th}$  iteration the condition  $S_{c_W}$  becomes false. Then any calculations made for subsequent iterations may be discarded as the loop is now complete.

```

FOR k←1 STEP 1 UNTIL 10 DO
BEGIN
  IF x>5 THEN
    a[k]←c
  ELSE
    x←2*j+k
END

```

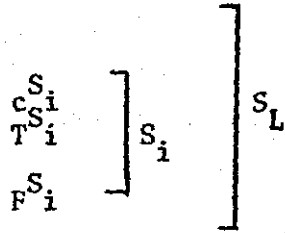


Figure 5.11

AN IF-STANZA WITHIN A DO-STANZA

## 5.7 PROCEDURE CALLS

A procedure is used in Algol-type programming languages to describe a commonly used process, which will be executed when a 'call' is made to the appropriate procedure and any necessary parameters supplied for the passing of its inputs and outputs. The description of a process to be performed on certain parameters will be known as the procedure definition. The 'body' of a procedure is the code executed each time the procedure is called.

Within the body of a procedure three types of variables can be considered to be used:

- (i) Local variables
- (ii) Global variables
- (iii) Parameters

The effects these types of variables will have on potential parallelism between a call of a procedure and the surrounding stanzas will vary. Thus:

- (i) The local variables will have no affect on parallelism since by definition they cannot be used elsewhere.
- (ii) The global variables may be affected by the external environment and so must be included in the sets of usage of variables for the stanza that represents a call to this procedure.
- (iii) The actual variables passed as parameters may vary from call to call of a procedure. However from the procedure's definition the method in which a parameter is used will be known. For example in Algol-60 a parameter may be called by 'name' or 'value'. In the former case a parameter may be considered to be used in the manner Y described in Chapter 4 whereas in the latter case it would be W.

Continuing with the Algol-60 example there will be six sets of usage of variables to be considered for the call of a procedure as stanza  $S_i$ . These six sets are:

$B_i^W, B_i^X, B_i^Y$  and  $B_i^Z$  the global variables used in the procedure's body

and  $P_i^W, P_i^Y$  the parameters passed to the procedure.

The first four sets may be formulated when the procedure is defined. The remaining two must, however, be formed for each call of the procedure. These sets will be joined together to give the sets of usage of variables for  $S_i$ . When a global variable used in  $S_i$  is also passed as a parameter care must be taken to ensure it is placed in the correct set (e.g. a variable appearing in both  $B_i^X$  and  $P_i^W$  must be considered to be used in the manner Y). The four sets of usage of variables for  $S_i$  will be:

$W_i, X_i, Y_i$  and  $Z_i$ .

Figure 5.12 shows an example of a procedure definition and its call. There is only one global variable used in the body of the procedure, thus the sets of usage of variables are:

$B_i^W$   $\emptyset$

$B_i^X$  e

$B_i^Y$   $\emptyset$

$B_i^Z$   $\emptyset$

and

$P_i^W$  e, f

$P_i^Y$  g, h

The combined sets for the call of the procedure are:

$W_i$  f

$X_i$   $\emptyset$

$Y_i$  e, g, h

$Z_i$   $\emptyset$

Having formed the sets  $W_i, X_i, Y_i$  and  $Z_i$  the relationships between a call of a procedure and the surrounding stanzas may be established by using the appropriate method described in the previous sections.

```
PROCEDURE example (a,b,c,d);  
  INTEGER a,b,c,d;  
  VALUE a,b;  
  BEGIN  
    INTEGER m;  
    IF a=b THEN m←1 ELSE m←2;  
    c←(a-b*2)*m;  
    d←(b-a*2)*m;  
    e←c+d  
  END;  
  ⋮  
  example (e,f,g,h);
```

] procedure body  
] S<sub>i</sub>

Figure 5.12

A PROCEDURE CALL

## 5.8 ADDITIONAL CONSIDERATIONS

The most frequently used programming constructs (see Chapter 2) have now been discussed. In this section possible methods of handling other programming constructs will be outlined.

### 5.8.1 Unconditional Jumps

Unconditional jumps are represented by the use of a GOTO statement and a label which indicates the position to be 'gone to'. Such situations may be recognised when the stanzas are being formed. Each time a label is recognised a new stanza will be started and when a GOTO is recognised the current stanza is closed. An example of such a stanza is given in Figure 5.13. The relationship between stanzas that could be executed one after the other can then be found in the manner described previously.

### 5.8.2 Input and Output


The only potential for parallelism between two or more input operations will be when they are from different channels. Similarly the only potential for parallelism between two or more output operations will be when they are to different channels. In all other circumstances it may be considered that an input operation is storing to the variables input and the output operation is fetching the variables it will output.

### 5.8.3 Declarations

In the samples given in Robinson and Torsun (1976b) declarations accounted for 7½% of program statements. Whether any potential parallelism between declarations can be used advantageously will depend on a particular machine's main memory's architecture.



```
      a←d-b;  
label 1: b←d+c;  
        d←c/2;  
        GOTO label 2;  
        e←f*2;
```



A Stanza

Figure 5.13

A STANZA DELIMITED BY A LABEL AND A GOTO

## 5.9 IMPLEMENTATION OF AN IMPLICIT PARALLELISM DETECTOR

At the end of the previous chapter an Analyser was mentioned that would divide a simple program into stanzas. Another program Detector is given in Appendix 3 which will take the stanzas and calculate the relationships between pairs of stanzas and between iterations of simple loops. Appendix 4 shows a simple part of an Algol-type program, the stanzas formed from it and the relationships found to exist between them.

As with the Analyser some of the work done in the Detector will normally be carried out by the usual compiler routines. Figure 5.14 shows where the routines of Analyser and Detector may be inserted in a multipass compiler.

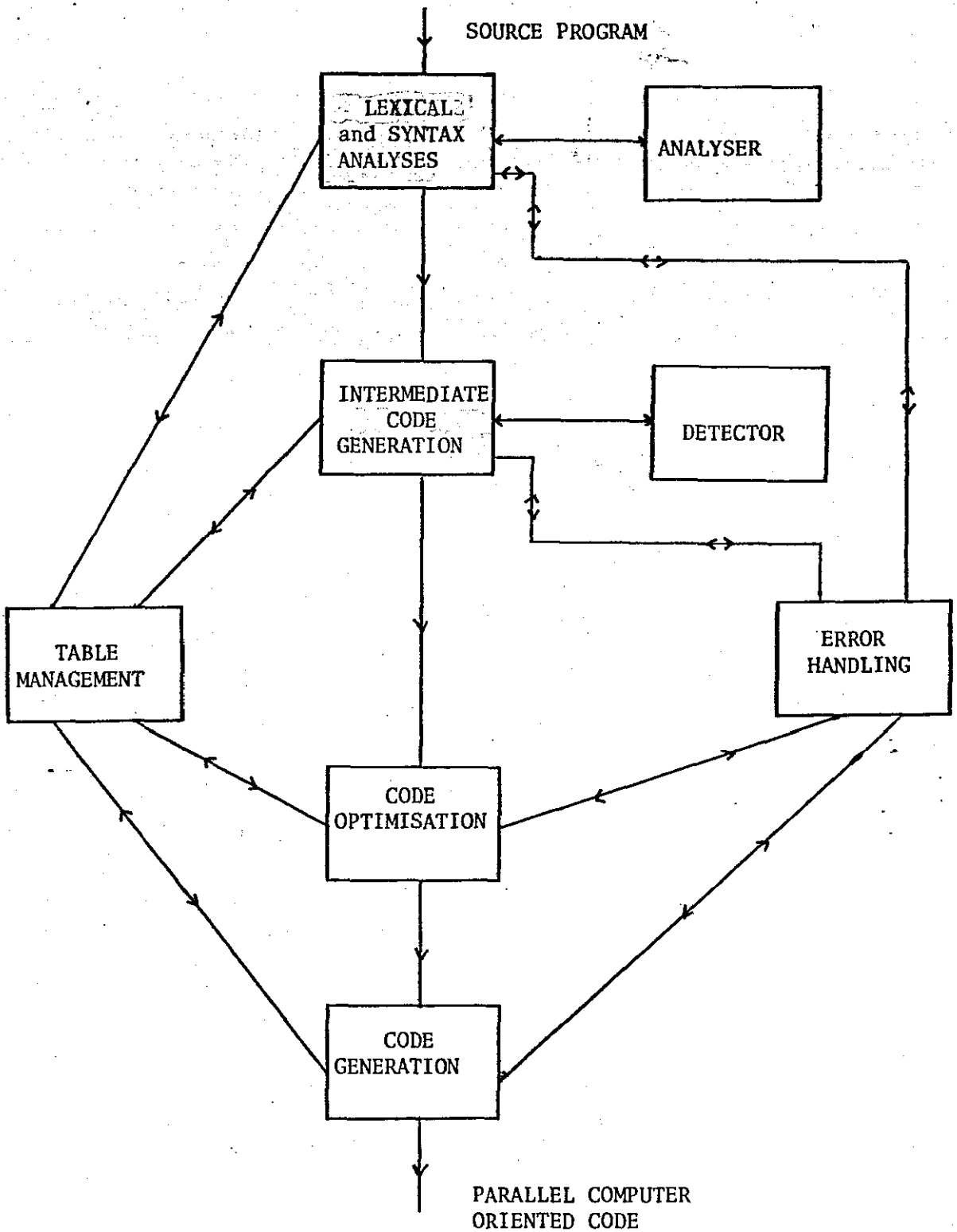


Figure 5.14

POSSIBLE PHASES OF A PARALLELISM DETECTOR AND MULTI-PASS COMPILER

CHAPTER 6

OPTIMISATION OF PARALLEL PROGRAMS

## 6.1 OPTIMISATION TECHNIQUES

A compiler which contains some means of producing an extremely efficient object code is called an 'Optimising Compiler' (Rustin, 1972 and Wulf et al, 1975). Most optimising compilers achieve such efficient code by the elimination of instructions and variables that are repetitious or redundant. However, for a program that is to be run in a parallel processing environment, optimisation will be used to produce efficient object code which contains an 'optimum' amount of potential parallelism. Where it is reasonable existing parallelism should not be removed from a program by the optimising process and, indeed, more parallelism may be introduced. Thus, one of the aims of optimising a parallel program will be to reduce dependencies within the code, even at the expense of using instructions and variables that are repetitious or redundant.

In this chapter optimisation techniques will be discussed for parallel programs which have been formed in either an implicit or explicit manner. Many types of optimising transformations have been considered for serial programs, Allen and Cocke (1972) give a catalogue of such techniques. Here it will be considered how some of the techniques they describe will effect the optimisation of parallel programs. It will be seen that some optimising techniques are equally well suited to both serial and parallel programs (e.g. *constant folding* and *peephole transformations*), whereas other techniques used for serial programs may in fact detract from potential parallelism of a parallel program (e.g. *strength reduction* and *linear function test replacement*). Some of the *loop transformation* optimising techniques are suitable for both serial and parallel programs whereas others are not and may even detract from parallelism within a program.

Many of the optimising techniques mentioned here can be applied at the 'Code Optimisation' stage of a multi-pass compiler, as described in Chapter 2. However, some optimisation may be carried out at different stages. For example *peephole transformations* may be carried out after the rest of the compilation is completed. The position in a multi-pass compiler where a particular optimisation is carried out will be the same for both serial and parallel programs.

## 6.2 OPTIMISATION TECHNIQUES READILY AMENABLE TO PARALLEL PROCESSING

In this section optimisation techniques that may be applied to both serial and parallel computer programs will be discussed. Full definitions of all the types of optimisation mentioned here for serial programs are given in Allen and Cocke (1972).

### 6.2.1 Procedure Integration

*Procedure integration* is essentially the replacing of a procedure call by what is to be executed at that point. For both serial and parallel programs, the methods by which parameters are passed will effect the possibilities of being able to integrate a large procedure. Similarly it is more complicated to integrate a large procedure than a small one. However, the advantages of the contents of a procedure being known at the point of call will be useful in the execution of both serial and parallel programs. Indeed, in the previous chapter *procedure integration* was used to determine implicit parallelism, in Algol-type programming languages, between a call to a procedure and the surrounding code.

### 6.2.2 Constant Folding and Dead Code Elimination

Sometimes a variable name is used to represent a constant value throughout a program (e.g. as a dimension of a set of arrays). When such a case is recognised the uses of that variable may be replaced by its constant value (i.e. *constant folding*). The use of *constant folding* will not have any detrimental effect on the parallelism within a computer program.

Code may become 'dead' because of *constant folding* and sometimes by other means. Code may be considered dead if it is in part of the

program that can never be reached. Figure 6.1 gives an example of a part of a program to which *dead code elimination* is applied. Since dead code will never be executed it may be removed from both serial and parallel programs.

### 6.2.3 Peephole Transformations

The final code produced from a compilation of a serial program can often be improved upon by carrying out a local scan on a sequence of instructions. Such optimisation may readily be applied to a stanza of a parallel program (whether explicit or implicit) such that the stanza itself may run optimally on one processor.



```
t←8;
IF t≠8 THEN
BEGIN
  a←b+c;
  d←e/f;
END
ELSE
BEGIN
  a←b-c;
  d←e*f
END
```

after *dead code elimination* becomes:-

```
t←8;
a←b-c;
d←e*f
```

Figure 6.1

DEAD CODE ELIMINATION

### 6.3 OPTIMISATION TECHNIQUES THAT DETRACT FROM POTENTIAL PARALLELISM

Some of the techniques used to optimise serial programs may have adverse effects on the parallelism in programs. However if a particular part of a parallel program is dictated to be run sequentially then any serial optimisation techniques may be applied to that part of the program. In this section it will be shown why, in general, some serial optimisation techniques are not suitable to be applied to parallel programs.

#### 6.3.1 Common Subexpression Elimination

*Common subexpression elimination* is used, in the optimisation of serial programs, to avoid recalculating a value that is already available. This is effected by storing the value of a subexpression in some temporary location that can be fetched when necessary. However, in a parallel program this may cause dependencies between stanzas or branches of a binary tree. Thus, elimination of common subexpressions may detract from the potential parallelism of either an explicit or implicit parallel program and so should be used with caution.

#### 6.3.2 Strength Reduction and Linear Function Test Replacement

The *strength reduction optimisation* is used to replace certain computations using recursively defined variables by recursively defined computations. A common example of this is in a loop replacing a calculation using the control variable by a variable incremented within the loop (see Figure 6.2). However, this will frequently increase dependencies within the program under consideration and so detract from potential parallelism. Thus *strength reduction* will not, in general, be applied to parallel programs.

*Linear function test replacement* is often applied after *strength reduction* and occasionally in other circumstances in the compilation of serial programs. Briefly, a test on one variable (e.g. the control variable) is replaced by a test on another recursively defined variable (e.g. a variable assigned to in the loop). As before this may increase the dependencies within a program and so detract from potential parallelism and thus, will not be usually used in the , compilation of parallel programs.

```
FOR i←1 STEP 1 UNTIL 100 DO  
BEGIN  
  a[i*5]←b+c[i]  
END
```

may become after *strength reduction*

```
INTEGER t←5;  
FOR i←1 STEP 1 UNTIL 100 DO  
BEGIN  
  a[t]←b+[i];  
  t←t+5  
END
```

Figure 6.2

STRENGTH REDUCTION TRANSFORMATION

## 6.4 LOOP TRANSFORMATIONS

Several transformations can be used to optimise the loops of a serial program. Here the three areas unrolling, unfolding and folding of loops will be considered. In the following subsections it will be indicated how such optimising transformations can be applied to a parallel program to attract parallelism between iterations of loops.

### 6.4.1 Loop Unrolling

A loop may be unrolled such that statements that would have been executed in different iterations may appear sequentially. Figure 6.3 shows two examples of how a loop may be unrolled. *Loop unrolling* may be used in the compilation of parallel programs to ensure the amount of code in each iteration of the loop is sufficient to justify any overheads of allocating independent iterations to separate processors.

### 6.4.2 Loop Unfolding

A loop can be unfolded such that statements that would have been executed in a loop are split between two or more loops. This may be used to remove dependencies between iterations of a loop as can be seen in Figure 6.4. The original loop, given in Figure 6.4, must be executed in a *consecutive* manner; after the transformation loops L1 and L2 are *consecutive* but both sets of iterations are *contemporary*. In cases where dependencies are not removed decreasing the amount of code in a loop will be unnecessary and may indeed increase the overheads of parallelism.

### 6.4.3 Loop Folding

*Loop folding* is sometimes referred to as jamming or fusion of loops. Briefly it is the joining together of two or more loops such that they are expressed by one loop (see Figure 6.5). This will have the same advantages as *loop unrolling*. However, it will be more difficult to implement as all loops within a program do not usually have the same step size and limits.

### 6.4.4 Combinations of Loop Transformations

It may be possible to combine the techniques, given in the previous three subsections, to create new loops in which there is more potential parallelism than in the original loops. Obviously if all iterations of a loop are already of a suitable size and the relationship between them all is *contemporary* there will be no need to apply any loop transformations. However, if they are not by judiciously unrolling, unfolding, and folding more potential parallelism may be introduced, assuming that folding does not recreate a loop just unfolded and vice versa. Figure 6.6 gives an example of using both the unfolding and unrolling techniques followed by more unrolling and folding to increase the amount of potential parallelism.

```
FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  a[i]←a[i+50]+b[i]
END
```

may be unrolled to give

```
FOR i←1 STEP 4 UNTIL 100 DO
BEGIN
  a[i]←a[i+50]+b[i];
  a[i+1]←a[i+51]+b[i+1];
  a[i+2]←a[i+52]+b[i+2];
  a[i+3]←a[i+53]+b[i+3]
END
```

or

```
FOR i←1 STEP 1 UNTIL 50 DO
BEGIN
  a[i]←a[i+50]+b[i];
  a[i+50]←a[i+100]+b[i+50]
END
```

Figure 6.3

LOOP UNROLLING

```

FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  a[i+1]←b[i]+c[i+1];
  c[i]←a[i]+b[i]
END

```

may be unfolded to give

```

FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  a[i+1]←b[i]+c[i+1]
END;
FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  c[i]←a[i]+b[i]
END

```

Figure 6.4

#### LOOP UNFOLDING

```

FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  a[i]←a[i]+b[i]
END;
FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  c[i]←d[i]-e[i]
END

```

may be folded to give

```

FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  a[i]←a[i]+b[i];
  c[i]←d[i]-e[i]
END

```

Figure 6.5

#### LOOP FOLDING



```

FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  a[i]←a[i+50]+b[i];
  x[i]←y[i]-z[i]
END

```

may be unfolded and unrolled to give

```

FOR i←1 STEP 1 UNTIL 50 DO
BEGIN
  a[i]←a[i+50]+b[i];
  a[i+50]←a[i+100]+b[i+50]
END;
FOR i←1 STEP 1 UNTIL 100 DO
BEGIN
  x[i]←y[i]-z[i]
END

```

may be unrolled and folded to give

```

FOR i←1 STEP 1 UNTIL 50 DO
BEGIN
  a[i]←a[i+50]+b[i];
  a[i+50]←a[i+100]+b[i+50];
  x[i]←y[i]-z[i];
  x[i+50]←y[i+50]-z[i+50]
END

```

Figure 6.6

LOOP UNROLLING, UNFOLDING AND FOLDING

CHAPTER 7

CORRECTNESS OF PARALLEL PROGRAMS

## 7.1 INTRODUCTION TO PROGRAM CORRECTNESS

The conditions expected to be true on entry to a program, or part of program, are called its 'antecedents'. Those expected to be true when the program, or part of program, exits are called its 'consequents'. Using this terminology the conditions for a program, or part of a program, to be considered correct can be defined.

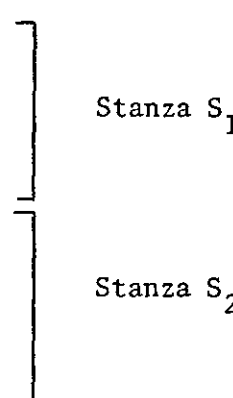
### Definition 7.1

A program or part of a program is correct if the truth of its antecedents ensures the truth of its consequents.

Elsewhere this is sometimes called 'partial correctness' since there is no guarantee that the program will terminate. However, here the termination of programs will not be considered.

Approaches to determining the correctness of parallel programs have been described in Owicki (1975), Gries (1977) and Fion and Suzuki (1977). Here the correctness of a parallel program written explicitly using the seven relationships defined in Definitions 4.3 to 4.9 and 4.10 to 4.16 will be considered. Figure 7.1 indicates how two stanzas may be explicitly shown to be *prerequisite*. The techniques of symbolic execution (Hantler and King, 1976) will be extended to indicate how the correctness of programs using these new relationships may be proved.

```
PR
BEGIN
BEGIN
  a2←c1;
  a1←b1+c1;
  b1←b2;
  a2←a1
END,
BEGIN
  b1←d1+c1;
  b2←d1;
  a2←b1+c1;
  e1←a2
END
END
```



Stanza  $S_1$

Stanza  $S_2$

Figure 7.1

TWO EXPLICIT *Prerequisite* Stanzas

## 7.2 SYMBOLIC EXECUTION OF PROGRAMS

To prove that a program is correct for all possible inputs will mean, in general, that a large or infinite number of inputs will have to be considered. This can be avoided by making statements about the properties of all inputs (antecedents) and outputs (consequents) of a program. This is achieved by a standard mathematical technique, using Greek symbols to represent arbitrary program inputs. If it can be proved that the output conditions will be met, using these symbols and any special properties they are deemed to have, then the program may be said to be correct. The process of proving a program using symbols to represent its inputs is called symbolic execution.

Here, three terms will be introduced to express conditions within the symbolic execution of a program:

### 1. Undefined Values

A variable is said to be undefined, at a particular point, if its value is not calculable in terms of program inputs and constants. Symbolically the undefined state will be represented by omega ( $\omega$ ).

### 2. Indefiniteness of Variables

The general property of a set of variables being undefined is called indefiniteness.

### 3. Propagation of Values

When in a program, a variable ( $V$ ) is assigned a value which is a function of a set of variables ( $SV$ ), any of the values of  $SV$  may be said to propagate through to  $V$ . In particular, when one of  $SV$  is undefined  $V$  will also be undefined after the assignment. This will be called the Propagation of Indefiniteness.

In the following three subsections methods of performing symbolic execution on different parts of programs will be examined.

### 7.2.1 Symbolic Execution of Sequential Program Statements

When a part of a program (e.g. a stanza  $S_1$ ) that is executed sequentially is considered, it can be seen that the consequents of the  $i^{\text{th}}$  statement is the antecedent of the  $(i+1)^{\text{th}}$  statement of  $S_1$ . Figure 7.2 contains a sample stanza with its antecedents and consequents, it can be seen that the stanza is correct since the outputs assumed for the stanza (the consequents of  $S_1$ ) agree with those derived. A simplification of the symbolic execution of  $S_1$  is given in Figure 7.3.

An example of special properties that may be associated with an input is that 'b1' and 'c1' (given in Figure 7.3) of  $S_1$  must both be positive. It can then be proved in the consequent of  $S_1$  that 'a1', 'a2' and 'c1' are all positive.

### 7.2.2 Symbolic Execution of a Conditional

A conditional is used to indicate that there is a choice of which piece of code will be executed next. A common type of conditional used in Algol-type languages takes the form:

IF *bool* THEN *stanza1* ELSE *stanza2*

where *stanza1* is executed when *bool* is true and *stanza2* is executed when it is false. The symbolic execution of such an expression will begin by replacing all the variables in the boolean (*bool*) by their symbolic values. This will give rise to three possible values of the resulting boolean expression:

- (i) true.
- (ii) false.

- (iii) some boolean expression that is true for at least one program input and false for at least one other.

For both (i) and (ii) only one path will ever be taken and this can be treated as executing code without any branches. However, with (iii) both the cases of execution of *stanza1* and *stanza2* must be examined, and this may be done by means of a symbolic execution tree. An example of a conditional is given in Figure 7.4 and the symbolic execution tree for it is given in Figure 7.5.

Looping structures may be considered to be a special form of branching for which some condition must be true for a specific set of statements to be repeated. So a symbolic execution tree may be used to represent a loop. Hantler and King (1976) give a detailed account of the symbolic execution of various conditionals including loops.

### 7.2.3 Parallel Symbolic Execution

When two or more stanzas are being executed in parallel it is possible that some of them may access the same variable simultaneously. This may lead to indefiniteness, for instance, if one stanza fetches the copy of a variable that another stanza is in the process of changing, then the value fetched is undefined (see section 4.2).

When conditionals were considered, a symbolic execution tree was introduced. Here a symbolic execution network will be introduced to allow for variables being accessed by more than one stanza simultaneously. The exact manner the network is constructed will depend on the relationship deemed to exist between the stanzas and the type of memory available. Figure 7.6 gives an example of how a symbolic execution network may be drawn for two stanzas that are executed simultaneously. Further examples of usages of symbolic execution networks can be found in section 7.4.

Here, parallel symbolic execution will only be considered for two stanzas  $S_1$  and  $S_2$ . All seven relationships in Definitions 4.3 to 4.9 will be considered in both private and shared memory environments where appropriate. The following four definitions describe what may happen to variables that are used in both stanzas  $S_1$  and  $S_2$ .

Definition 7.2

If  $S_1$  may access a variable (V) that  $S_2$  may or may not have changed or be in the process of changing, then there are two possibilities depending on the type of memory available:

- (i) Only shared memory available

Throughout  $S_1$  the variable (V) must be considered to be undefined each time it is fetched.

- (ii) Private memories available

The variable (V) will be considered to be undefined in  $S_1$  until such time it is assigned to in  $S_1$ .

Definition 7.3

When  $S_1$  and  $S_2$  are both able to change the same variable (V) such that  $S_1$  changes it before or after  $S_2$  or both changes are made simultaneously then there are two possibilities depending on the type of memory available:

- (i) Only shared memory available

Throughout  $S_1$  and  $S_2$  the variable (V) must be considered to be undefined.

- (ii) Private memories available

In  $S_1$  the variable (V) will be considered to be undefined until it is assigned to in  $S_1$ , similarly in  $S_2$ .



Definition 7.4

When it is dictated that  $S_1$  must store its results before  $S_2$ , does, then variables assigned to in both will have, on completion of  $S_1$  and  $S_2$ , the value assigned to them in  $S_2$ .

Definition 7.5

When  $S_1$  and  $S_2$  may store their results in either order (i.e.  $S_1$  first and then  $S_2$  or  $S_2$  first and then  $S_1$ ) or both may store their results simultaneously then variables assigned to, in both, will be considered undefined upon completion of  $S_1$  and  $S_2$ .

Antecedents of  $S_1$

$a_1:\alpha, a_2:\beta, b_1:\gamma, b_2:\delta, c_1:\epsilon$

BEGIN

$a_2 \leftarrow c_1;$

① Consequents of ① and Antecedents of ②  
 $a_1:\alpha, a_2:\epsilon, b_1:\gamma, b_2:\delta, c_1:\epsilon$  .

$a_1 \leftarrow b_1 + c_1;$

② Consequents of ② and Antecedents of ③  
 $a_1:\gamma+\epsilon, a_2:\epsilon, b_1:\gamma, b_2:\delta, c_1:\epsilon$  .

$b_1 \leftarrow b_2;$

③ Consequents of ③ and Antecedents of ④  
 $a_1:\gamma+\epsilon, a_2:\epsilon, b_1:\delta, b_2:\delta, c_1:\epsilon$  .

$a_2 \leftarrow a_1$

④ Consequents of ④  
 $a_1:\gamma+\epsilon, a_2:\gamma+\epsilon, b_1:\delta, b_2:\delta, c_1:\epsilon$  .

END

Consequents of  $S_1$

$a_1:\gamma+\epsilon, a_2:\gamma+\epsilon, b_1:\delta, b_2:\delta, c_1:\epsilon$

Figure 7.2

ANTECEDENTS AND CONSEQUENTS OF A STANZA  $S_1$

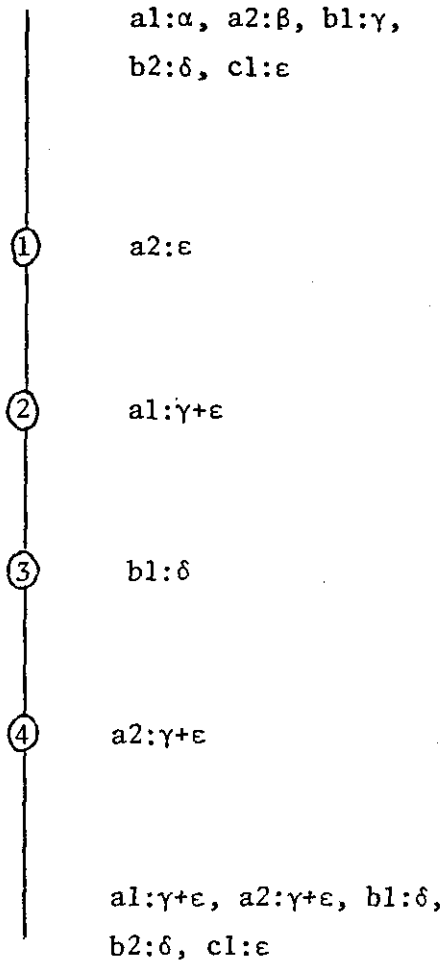


Figure 7.3

SYMBOLIC EXECUTION OF THE STANZA  $S_1$

```
IF bool THEN
BEGIN
  a2←c1;
  a1←b1+c1;
  b1←b2;
  a2←a1
END
ELSE
BEGIN
  b1←d1+c1;
  b2←d1;
  a2←b1+c1;
  e1←a2
END
```

①  
②  
③  
④ ] Stanza S<sub>1</sub>

⑤  
⑥  
⑦  
⑧ ] Stanza S<sub>2</sub>

Figure 7.4

AN ALGOL-TYPE CONDITIONAL STATEMENT

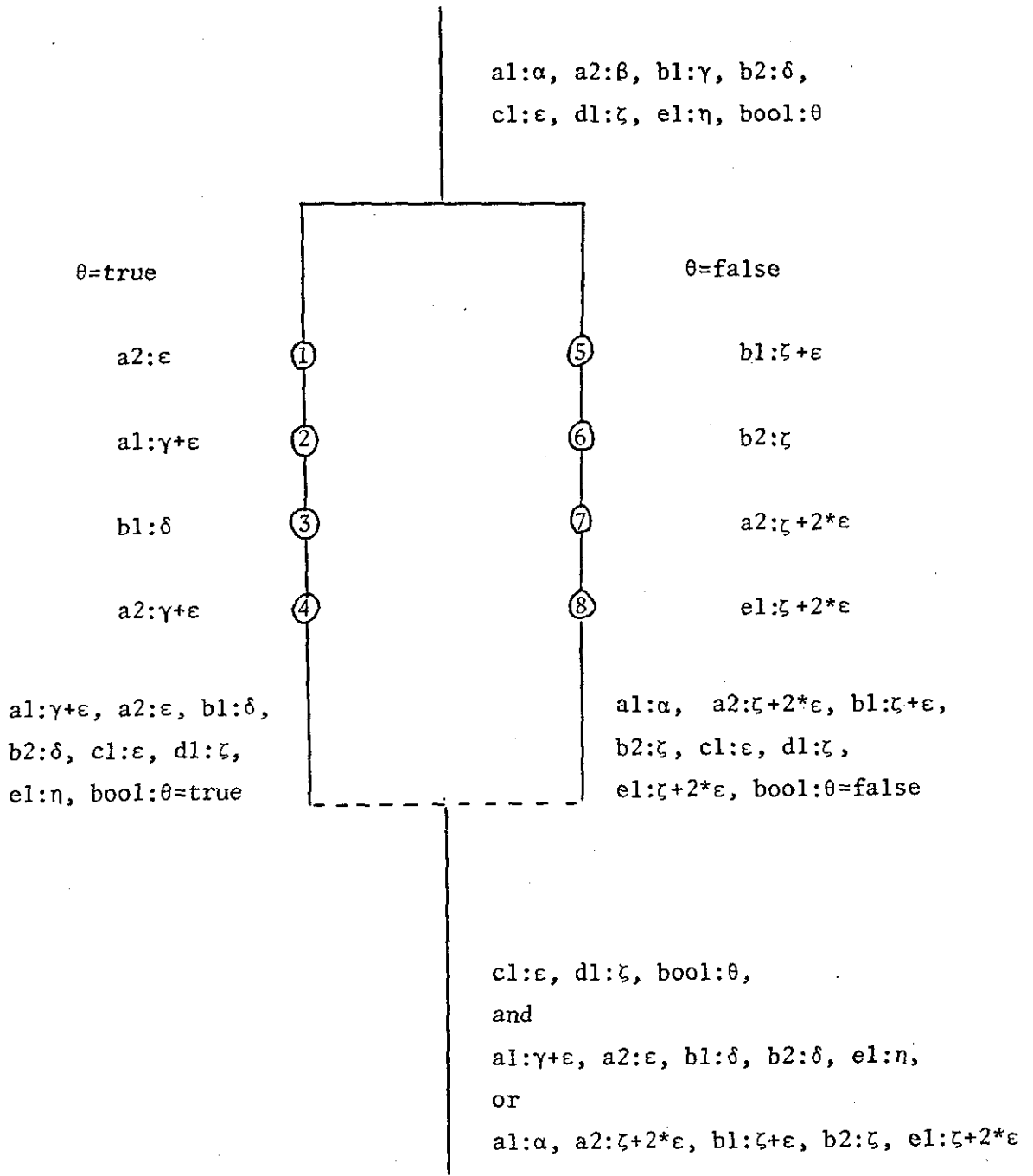


Figure 7.5

THE SYMBOLIC EXECUTION TREE OF A CONDITIONAL STATEMENT

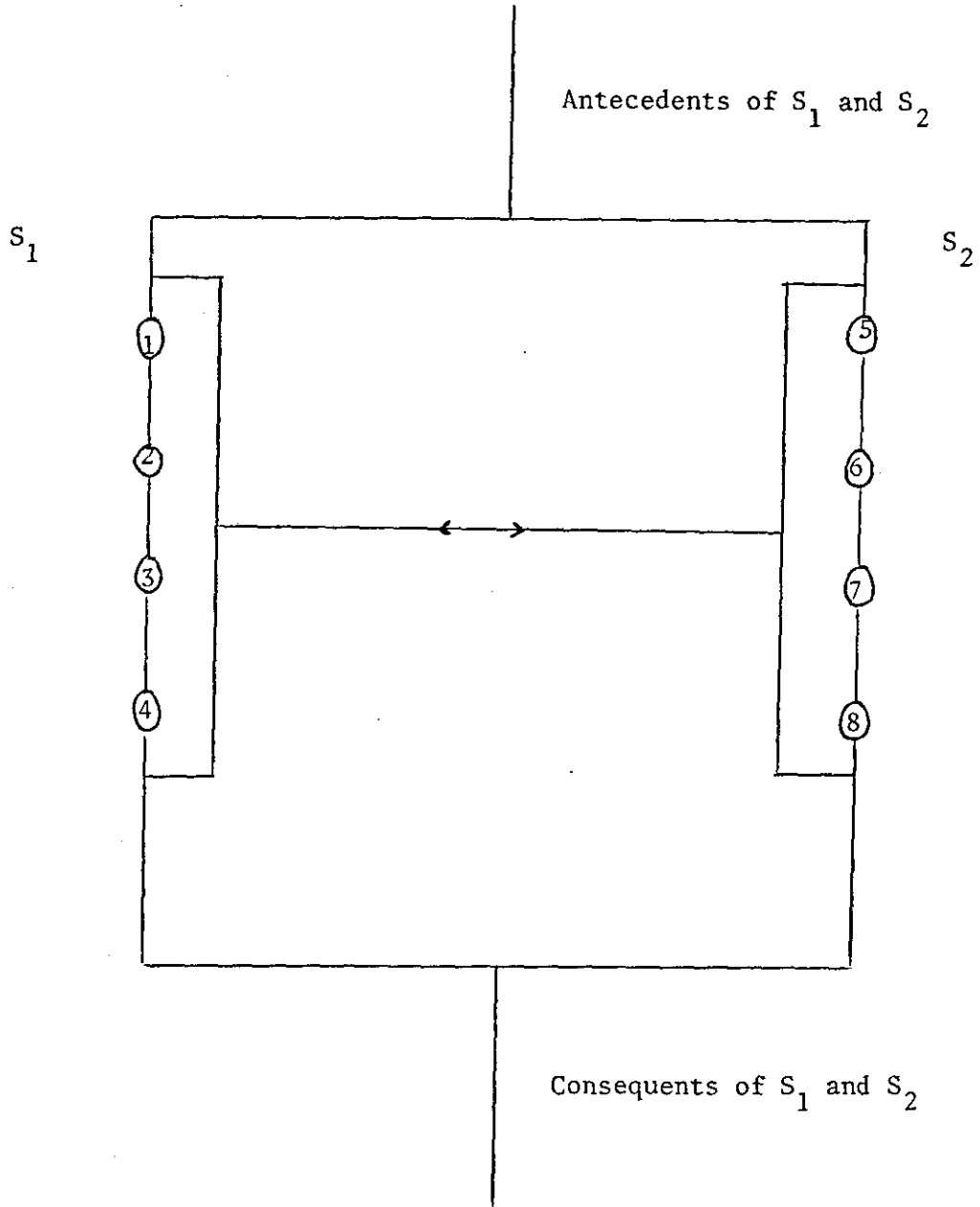


Figure 7.6

A SYMBOLIC EXECUTION NETWORK

### 7.3 TESTS TO DETERMINE THE CONSEQUENTS OF PARALLEL PROGRAMS

In this section the two stanzas  $S_1$  and  $S_2$  mentioned in subsection 7.2.3 will continue to be considered. Rules will be established to determine the consequents of  $S_1$  and  $S_2$  from their antecedents. The four sets of usage of variables ( $W, X, Y$  and  $Z$ ) described in section 4.5 will be used in establishing these rules.

#### 7.3.1 Contemporary - $CT(S_1, S_2)$

Stanzas  $S_1$  and  $S_2$  can be executed at the same time and the locations used may be accessed in any order.

Bearing in mind the definition of *contemporary*, repeated above, and the differences between parallel machines with private memories and those without the following rules can be derived:

#### Rule 7.1(a): $CT(S_1, S_2)$ with Private Memories

1. Any variable only fetched in  $S_1$  and stored in  $S_2$  (i.e. a member of the set  $(W_1 \cap (X_2 \cup Y_2 \cup Z_2))$ ) is undefined in  $S_1$  and upon completion of  $CT(S_1, S_2)$  will have the last value set in  $S_2$ .
2. Any variable only fetched in  $S_2$  and stored in  $S_1$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap W_2)$ ) is undefined in  $S_2$  and upon completion of  $CT(S_1, S_2)$  will have the last value in set  $S_1$ .
3. Any variable changed in both  $S_1$  and  $S_2$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2))$ ) will be undefined upon completion of  $CT(S_1, S_2)$  and will be undefined in  $S_1$  (or  $S_2$ ) until it is set in  $S_1$  (or  $S_2$ ).
4. In all other instances the consequents will be the same as if both  $S_1$  and  $S_2$  (including any indefiniteness introduced above) had been executed sequentially both with the same antecedents.

Rule 7.1(b): CT(S<sub>1</sub>, S<sub>2</sub>) with Shared Memory

Rule 7.1(b) only varies from Rule 7.1(a) in the third case, which is adapted to give:

3. Any variable changed in both S<sub>1</sub> and S<sub>2</sub> (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2))$ ) will be undefined throughout S<sub>1</sub> and S<sub>2</sub> and remain so upon completion of CT(S<sub>1</sub>, S<sub>2</sub>).

7.3.2 Commutative - CM(S<sub>1</sub>, S<sub>2</sub>)

Stanza S<sub>1</sub> may be executed before or after S<sub>2</sub> is executed but not at the same time.

From the definition of *commutative*, repeated above, there are two possible ways CM(S<sub>1</sub>, S<sub>2</sub>) may be executed; these being S<sub>1</sub> then S<sub>2</sub> or S<sub>2</sub> then S<sub>1</sub>. The availability of private memories will have no effect on the *commutative* relationship.

Rule 7.2: CM(S<sub>1</sub>, S<sub>2</sub>)

1. Any variable only fetched in S<sub>1</sub> and changed in S<sub>2</sub> (i.e. a member of the set  $(W_1 \cap (X_2 \cup Y_2 \cup Z_2))$ ) will have in S<sub>1</sub>:
  - either (i) the value in the antecedent - if S<sub>1</sub> is executed before S<sub>2</sub>.
  - or (ii) the final value stored to it in S<sub>2</sub> - if S<sub>1</sub> is executed after S<sub>2</sub>.
2. Any variable only fetched in S<sub>2</sub> and changed in S<sub>1</sub> (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap W_2)$ ) will have in S<sub>2</sub>:
  - either (i) the final value stored to it in S<sub>1</sub> - if S<sub>1</sub> is executed before S<sub>2</sub>.
  - or (ii) the value in the antecedent - if S<sub>1</sub> is executed after S<sub>2</sub>.



3. Any variable changed in both  $S_1$  and  $S_2$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2))$ ) will have in the consequents:
  - either (i) the final value set in  $S_2$  - if  $S_1$  is executed before  $S_2$ .
  - or (ii) the final value set in  $S_1$  - if  $S_1$  is executed after  $S_2$ .
4. In all other instances the consequents will be the same as if both  $S_1$  and  $S_2$  had been executed sequentially both with the same antecedents.

### 7.3.3 Prerequisite - $PR(S_1, S_2)$

Stanza  $S_1$  must fetch what it requires before  $S_2$  stores its results.

As mentioned in section 5.2 the *prerequisite* relationship degenerates into a *consecutive* one when private memories are not available, so here the rule will assume private memories are available.

#### Rule 7.3: $PR(S_1, S_2)$

1. Any variable only fetched in  $S_2$  and changed in  $S_1$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap W_2)$ ) is undefined in  $S_2$  and upon completion of  $PR(S_1, S_2)$  will have the last value set in  $S_1$ .
2. Any variable changed in both  $S_1$  and  $S_2$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2))$ ) will be undefined upon completion of  $PR(S_1, S_2)$  and will be undefined in  $S_2$  until such time as it is set.
3. In all other instances the consequents will be the same as if both  $S_1$  and  $S_2$  (including any indefiniteness introduced above) had been executed sequentially both with the same antecedents.

### 7.3.4 Conservative - $CV(S_1, S_2)$

Stanza  $S_1$  must store its results before  $S_2$  does.

As with *prerequisite* the *conservative* relationship, repeated above,

it will be assumed that private memories are available.

Rule 7.4:  $CV(S_1, S_2)$

1. Any variable only fetched in  $S_2$  and stored in  $S_1$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap W_2))$  is undefined in  $S_2$  and upon completion of  $CV(S_1, S_2)$  will have the last value set in  $S_1$ .
2. Any variable changed in both  $S_1$  and  $S_2$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)))$  will be undefined in  $S_2$ , until such time as it is set in  $S_2$ , and upon completion of  $CV(S_1, S_2)$  will have the last value set in  $S_2$ .
3. In all other instances the consequents will be the same as if both  $S_1$  and  $S_2$  (including any indefiniteness introduced above) had been executed sequentially both with the same antecedents.

7.3.5 Consecutive -  $CC(S_1, S_2)$

Stanza  $S_1$  must store its results before  $S_2$  fetches what it requires.

The *consecutive* relationship between two stanzas indicates that they are to be executed sequentially. Hence such stanzas are handled in the manner described for sequential program statements in section 7.2.

7.3.6 Synchronous -  $SN(S_1, S_2)$

Stanzas  $S_1$  and  $S_2$  must both have the same inputs.

The synchronous relationship can only sensibly exist for execution on a machine with private memories. So here the rule will assume that private memories are available.

Rule 7.5:  $SN(S_1, S_2)$

1. Any variable changed in both  $S_1$  and  $S_2$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)))$  will be undefined upon completion of  $SN(S_1, S_2)$ .

2. In all other instances the consequents will be the same as if both  $S_1$  and  $S_2$  had been executed sequentially both with the same antecedents.

### 7.3.7 *Inclusive* - $IN(S_1, S_2)$

Stanza  $S_2$  must store its results after  $S_1$  has fetched what it requires but before  $S_1$  stores its results.

As with the *synchronous* relationship the *inclusive* relationship can only sensibly exist for execution on a machine with private memories. So here the rule will assume that private memories are available.

#### Rule 7.6: $IN(S_1, S_2)$

1. Any variable changed in both  $S_1$  and  $S_2$  (i.e. a member of the set  $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2))$ ) will be defined throughout  $S_1$  and  $S_2$  and upon completion of  $IN(S_1, S_2)$  will have the last value set in  $S_1$ .
2. In all other instances the consequents will be the same as if both  $S_1$  and  $S_2$  had been executed sequentially with the same antecedents.

Table 7.1 contains a summary of the values a variable may take when two stanzas access it. When a symbolic execution of two parallel stanzas takes place it is possible, by using the table, to determine which variables will be undefined in one stanza because of a use in the other. Such variables will be given the symbolic value ' $\omega$ ' such that indefiniteness, along with the other values may be propagated through a stanza. The table may also be used to determine the value of a variable on completion of the stanzas. If the consequents thus obtained for the stanzas are the same as those expected to be true on their completion then those stanzas are said to be correct for the parallel relationship being considered.

Relationship between $S_1$ and $S_2$ Operations Performed Value at a given point		Contemporary $CT(S_1, S_2)$ Shared Memory	Contemporary $CT(S_1, S_2)$ Private Memories	Commutative $CM(S_1, S_2)$ Shared/ Private	Prerequisite $PR(S_1, S_2)$ Private Memories	Conversative $CV(S_1, S_2)$ Private Memories	Consecutive $CC(S_1, S_2)$ Shared/ Private	Synchronous $SN(S_1, S_2)$ Private Memories	Inclusive $IN(S_1, S_2)$ Private Memories
$S_1$ only fetches a variable that $S_2$ changes $(W_1 \cap (X_2 \cup Y_2 \cup Z_2))$	Value in $S_1$	Undefined	Undefined	Original value or value set in $S_2$	Original value	Original value	Original value	Original value	Original value
	Value after $S_1$ and $S_2$	Last value set in $S_2$							
$S_1$ changes a variable that $S_2$ only fetches $((X_1 \cup Y_1 \cup Z_1) \cap W_2)$	Value in $S_2$	Undefined	Undefined	Value set in $S_1$ or original value	Undefined	Undefined	Value set in $S_1$	Original value	Original value
	Value after $S_1$ and $S_2$	Last value set in $S_1$							
$S_1$ and $S_2$ both change the same variable $((X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2))$	Value before being set in $S_1$	Undefined	Undefined	Original value or value set in $S_2$	Original value	Original value	Original value	Original value	Original value
	Value after being set in $S_1$	Undefined	Previous value set in $S_1$						
	Value before being set in $S_2$	Undefined	Undefined	Value set in $S_1$ or original value	Undefined	Undefined	Value set in $S_1$	Original value	Original value
	Value after being set in $S_2$	Undefined	Previous value set in $S_2$						
	Value after $S_1$ and $S_2$	Undefined	Undefined	Value set in $S_2$ or value set in $S_1$	Undefined	Value set in $S_2$	Value set in $S_2$	Undefined	Value set in $S_1$

Table 7.1: VALUES OF A VARIABLE ACCESSED BY TWO STANZAS

#### 7.4 EXAMPLES OF PROVING CORRECTNESS BETWEEN TWO STANZAS

In this section the two stanzas  $S_1$  and  $S_2$ , given in Figure 7.7, will be considered. The symbolic antecedents of these will be arbitrarily assigned as:

$$a1:\alpha, a2:\beta, b1:\gamma, b2:\delta, c1:\epsilon, d1:\zeta, e1:\eta$$

Two sets of consequents, in turn, will be considered to be true on exiting from  $S_1$  and  $S_2$ . The consequents considered will be:

##### Assumption 1

$$a1:\gamma+\epsilon, a2:\zeta+2*\epsilon, b1:\zeta+\epsilon, b2:\zeta, c1:\epsilon, d1:\zeta, e1:\zeta+2*\epsilon,$$

and

##### Assumption 2

$$a1:\gamma+\epsilon, b2:\zeta, c1:\epsilon, d1:\zeta,$$

where in Assumption 2 some of the values calculated in  $S_1$  and  $S_2$  will not be required later.

For the two stanzas  $S_1$  and  $S_2$  the sets of usage of variables are:

$$\begin{array}{ll} W_1 & b2, c1 \\ X_1 & a2 \\ Y_1 & b1 \\ Z_1 & a1 \end{array}$$

and

$$\begin{array}{ll} W_2 & c1, d1 \\ X_2 & b2, e1 \\ Y_2 & \emptyset \\ Z_2 & a2, b1 \end{array}$$

Using these sets it is possible to apply the rules given in the previous section to determine indefiniteness and the values propagated through to the exit of the stanza (i.e. the consequents). If the values of these consequents agree with Assumption 1, the execution of these two stanzas can be said to be correct for Assumption 1, similarly the correctness of Assumption 2 can be tested.

#### 7.4.1 Contemporary - $CT(S_1, S_2)$

As this relationship varies depending whether private memories are available or not the two will be considered separately.

##### Private Memories Available

First it is necessary to find which variables are used in both  $S_1$  and  $S_2$  and the effect they may have, by using Rule 7.1(a).

$$1. W_1 \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (b2, c1) \cap ((b2, e1) \cup \emptyset \cup (a2, b1)) = b2$$

By Condition 1 the variable 'b2' will be undefined in  $S_1$  and upon completion will have the value set in  $S_2$ .

$$2. (X_1 \cup Y_1 \cup Z_1) \cap W_2$$

$$\text{i.e. } (a2 \cup b1 \cup a1) \cap (c1, d1) = \emptyset$$

No variables are affected by the second condition.

$$3. (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (a2 \cup b1 \cup a1) \cap ((b2, e1) \cup \emptyset \cup (a2, b1)) = (a2, b1)$$

By the third condition the variables 'a2' and 'b1' will be undefined upon completion of  $CT(S_1, S_2)$  and will be undefined in  $S_1$  (or  $S_2$ ) until they are set in  $S_1$  (or  $S_2$ ).

The symbolic execution network for  $CT(S_1, S_2)$  with private memories is given in Figure 7.8. The consequents of its execution are:

$$a1:\omega, a2:\omega, b1:\omega, b2:\zeta, c1:\epsilon, d1:\zeta, e1:\zeta+2*\epsilon.$$

Thus it can be seen that  $CT(S_1, S_2)$  with private memories is neither correct for Assumption 1 or Assumption 2 (because, for example, the value of 'a1' is not ' $\gamma+\epsilon$ ' but undefined).

##### Only Shared Memory Available

Again, it will be necessary to find which variables are used in both  $S_1$  and  $S_2$  and the effects they may have, this time by using Rule 7.1(b).

The first and second conditions are the same as those for Rule 7.1(a) and so give the same results.

$$3. (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (a2 \cup b1 \cup a1) \cap ((b2, e1) \cup (a2, b1)) = (a2, b1)$$

By the third condition the variables 'a2' and b1' will be undefined throughout  $S_1$  and  $S_2$  and upon completion of  $CT(S_1, S_2)$ .

The symbolic execution network for  $CT(S_1, S_2)$  with only shared memory available is given in Figure 7.9. The consequents of its execution are:

$$a1:\omega, a2:\omega, b1:\omega, b2:\zeta, c1:\epsilon, d1:\zeta, e1:\omega$$

Thus it can be seen that  $CT(S_1, S_2)$  without private memories is neither correct for Assumption 1 or Assumption 2. It is also of interest to note that when private memories are available the value of 'e1' is defined, whereas without them it is undefined.

#### 7.4.2 Commutative - $CM(S_1, S_2)$

The *commutative* relationship may be treated as though it was a conditional where if a fictitious condition is true,  $S_1$  is executed before  $S_2$ , and if it is false,  $S_2$  is executed before  $S_1$ , as described in Rule 7.2. Hence a symbolic execution tree can be used to represent  $CM(S_1, S_2)$ , such a tree is given in Figure 7.10. The consequents of its execution are:

$$a2:\zeta+2*\epsilon, b2:\zeta, c1:\epsilon, d1:\zeta, e1:\zeta+2*\epsilon$$

$$\text{and } a1:\gamma+\epsilon, b1:\zeta+\epsilon \text{ or } a1:\zeta+2*\epsilon, b1:\zeta$$

It can be seen that when  $S_1$  is executed first, the code is correct for both Assumption 1 and Assumption 2. However neither are correct when  $S_2$  is executed first and since it is impossible to predict which will be executed first  $CM(S_1, S_2)$  cannot be assumed to be correct for either Assumption 1 or Assumption 2.

### 7.4.3 Prerequisite - $PR(S_1, S_2)$

For the *prerequisite* relationship it is assumed that private memories are available.

It will be necessary to find which of the variables from one stanza may affect the other stanza and the consequents of  $PR(S_1, S_2)$ , by using Rule 7.3:

$$1. (X_1 \cup Y_1 \cup Z_1) \cap W_2$$

$$\text{i.e. } (a_2 \cup b_1 \cup a_1) \cap (c_1, d_1) = \emptyset$$

No variables are affected by the first condition.

$$2. (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (a_2 \cup b_1 \cup a_1) \cap ((b_2, e_1) \cup (a_2, b_1)) = (a_2, b_1)$$

By the second condition the variables 'a2' and 'b1' will be undefined upon completion of  $PR(S_1, S_2)$  and will be undefined in  $S_2$  until such time as they are set.

The symbolic execution network for  $PR(S_1, S_2)$  is given in Figure 7.11.

The consequents of its execution are:

$$a_1: \gamma + \epsilon, a_2: \omega, b_1: \omega, b_2: \zeta, c_1: \epsilon, d_1: \zeta, e_1: \zeta + 2 * \epsilon.$$

It can be seen that  $PR(S_1, S_2)$  is not correct for Assumption 1. However, the values of 'a1', 'b2', 'c1' and 'd1' correspond to those proposed in Assumption 2 and so  $PR(S_1, S_2)$  is correct for Assumption 2.

### 7.4.4 Conservative - $CV(S_1, S_2)$

For the *conservative* relationship it is assumed that private memories are available. It will be necessary to find which of the variables used in one stanza may affect the other stanza and the consequents of  $CV(S_1, S_2)$  by applying Rule 7.4.

$$1. (X_1 \cup Y_1 \cup Z_1) \cap W_2$$

$$\text{i.e. } (a_2 \cup b_1 \cup a_1) \cap (c_1, d_1) = \emptyset$$

No variables are affected by the first condition.



$$2. (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (a_2 \cup b_1 \cup a_1) \cap ((b_2, e_1) \cup (a_2, b_1)) = (a_2, b_1)$$

By the second condition the variables 'a2' and 'b1' will be undefined in  $S_2$  until such time they are set and upon completion of  $CV(S_1, S_2)$  will have the last value assigned to them in  $S_2$ .

The symbolic execution network for  $CV(S_1, S_2)$  is given in Figure 7.12.

The consequents of its execution are:

$$a_1: \gamma + \epsilon, a_2: \zeta + 2 * \epsilon, b_1: \zeta + \epsilon, b_2: \zeta, c_1: \epsilon, d_1: \zeta, e_1: \zeta + 2 * \epsilon$$

It can be seen that these values correspond to those proposed in both Assumption 1 and Assumption 2. Thus it can be said that  $CV(S_1, S_2)$  is correct for both Assumption 1 and Assumption 2.

#### 7.4.5 Consecutive - $CC(S_1, S_2)$

As mentioned in the previous section the *consecutive* relationships indicates that  $S_1$  and  $S_2$  are executed sequentially and so the availability of private memories will not effect this relationship. The symbolic execution of  $CC(S_1, S_2)$  is given in Figure 7.13. The consequents of its execution are:

$$a_1: \gamma + \epsilon, a_2: \zeta + 2 * \epsilon, b_1: \zeta + \epsilon, b_2: \zeta, c_1: \epsilon, d_1: \zeta, e_1: \zeta + 2 * \epsilon.$$

These are the same as the consequents for  $CV(S_1, S_2)$  and, hence,  $CC(S_1, S_2)$  is correct for both Assumption 1 and Assumption 2.

#### 7.4.6 Synchronous - $SN(S_1, S_2)$

For the *synchronous* relationship it is assumed that private memories are available. Firstly it will be necessary to find which variables are changed in both stanzas and will affect the consequents of  $SN(S_1, S_2)$ , by using Rule 7.5.

$$1. (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (a2 \cup b1 \cup a1) \cap ((b2, e1) \cup (a2, b1)) = (a2, b1)$$

The variables 'a2' and 'b1' will be undefined upon completion of  $SN(S_1, S_2)$ .

The symbolic execution network for  $SN(S_1, S_2)$  is given in Figure 7.14.

The consequents of its execution are:

$$a1:\gamma+\epsilon, a2:\omega, b1:\omega, b2:\zeta, c1:\epsilon, d1:\zeta, e1:\zeta+2*\epsilon.$$

It can be seen that  $SN(S_1, S_2)$  is not correct for Assumption 1. However, the values of 'a1', 'b1', 'c1' and 'd1' correspond to those proposed in Assumption 2 and so  $SN(S_1, S_2)$  is correct for Assumption 2.

#### 7.4.7 Inclusive - $IN(S_1, S_2)$

Again, for the *inclusive* relationship it is assumed that private memories are available. It will be necessary to apply the first condition of Rule 7.6.

$$1. (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$$

$$\text{i.e. } (a2 \cup b1 \cup a1) \cap ((b2, e1) \cup (a2, b1)) = (a2, b1)$$

Upon completion of  $IN(S_1, S_2)$  the variables 'a2' and 'b1' will have the last value set in  $S_1$ .

The symbolic execution network for  $IN(S_1, S_2)$  is given in Figure 7.15.

The consequents of its execution are:

$$a1:\gamma+\epsilon, a2:\gamma+\epsilon, b1:\delta, b2:\zeta, c1:\epsilon, d1:\zeta, e1:\zeta+2*\epsilon.$$

It can be seen that  $IN(S_1, S_2)$  is not correct for Assumption 1.

However, the values of 'a1', 'b2', 'c1' and 'd1' correspond to those proposed in Assumption 2 and so  $IN(S_1, S_2)$  is correct for Assumption 2.

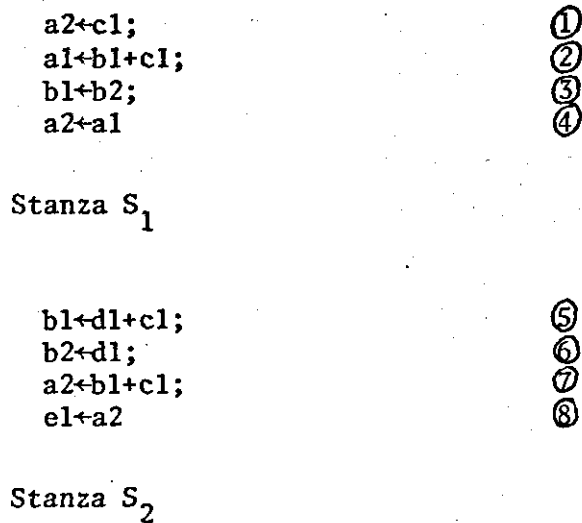


Figure 7.7

TWO STANZAS  $S_1$  AND  $S_2$

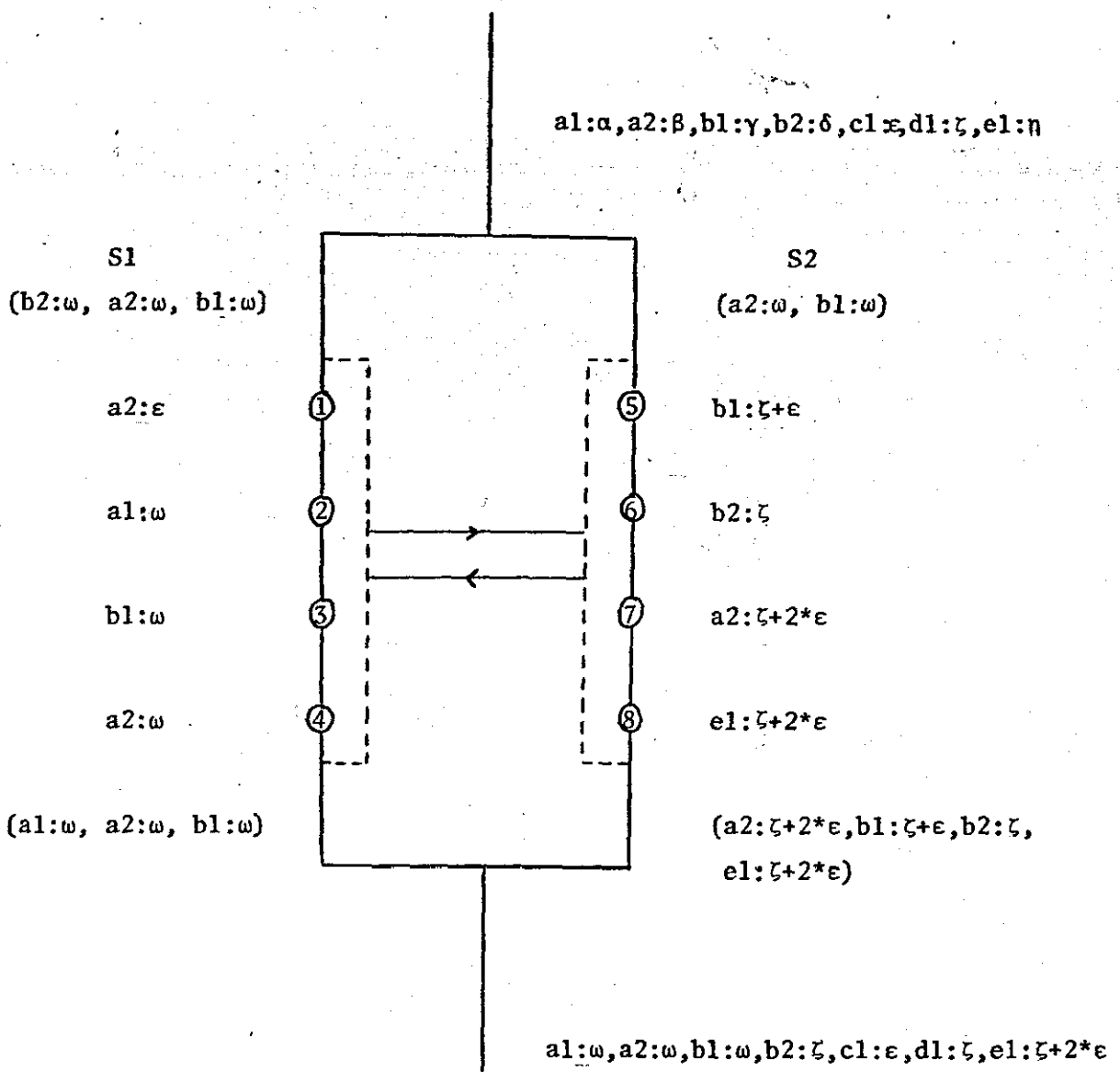


Figure 7.8

SYMBOLIC EXECUTION NETWORK OF  $CT(S_1, S_2)$  WITH PRIVATE MEMORIES

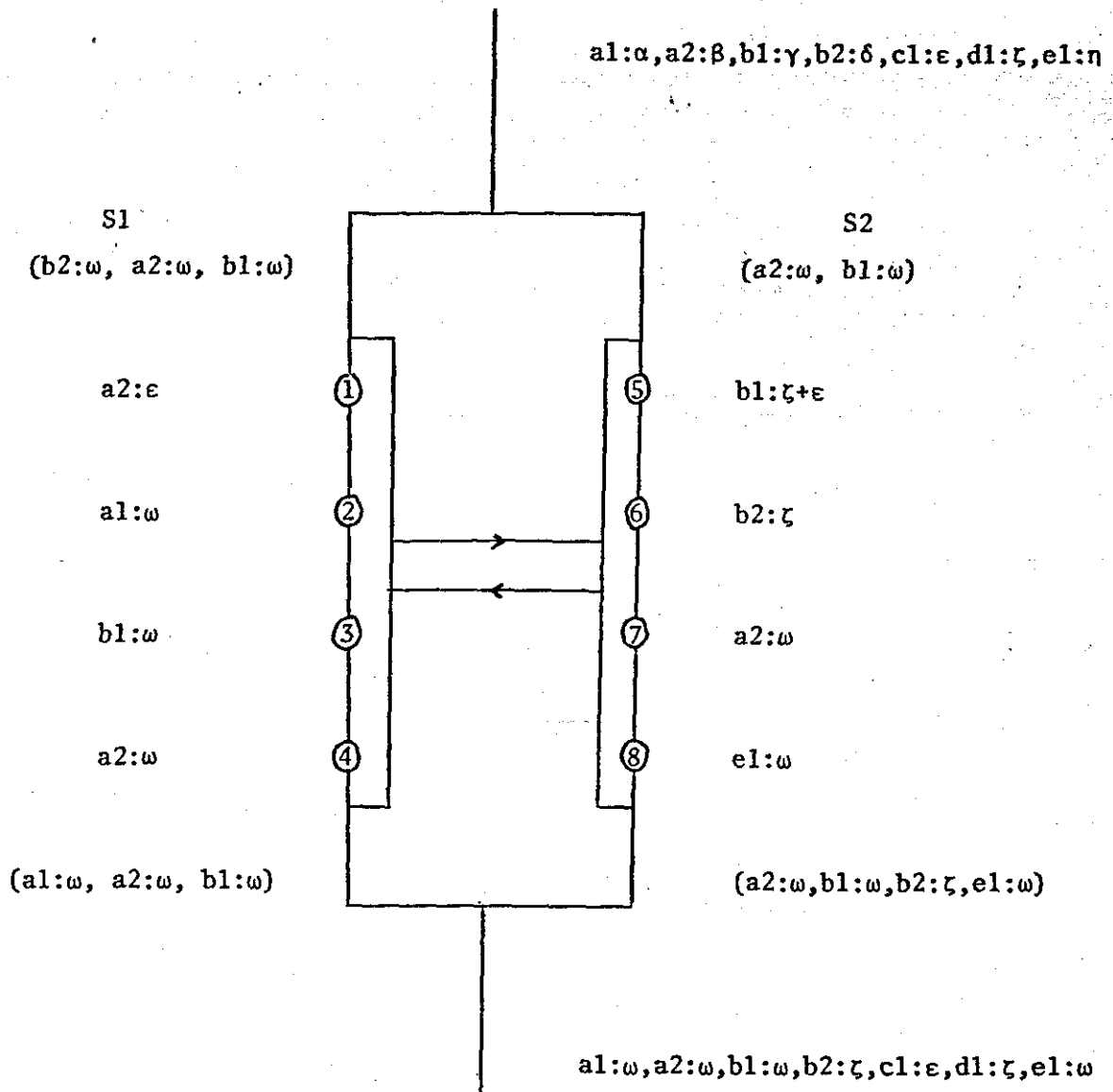


Figure 7.9

SYMBOLIC EXECUTION NETWORK OF  $CT(S_1, S_2)$  WITH ONLY SHARED MEMORY

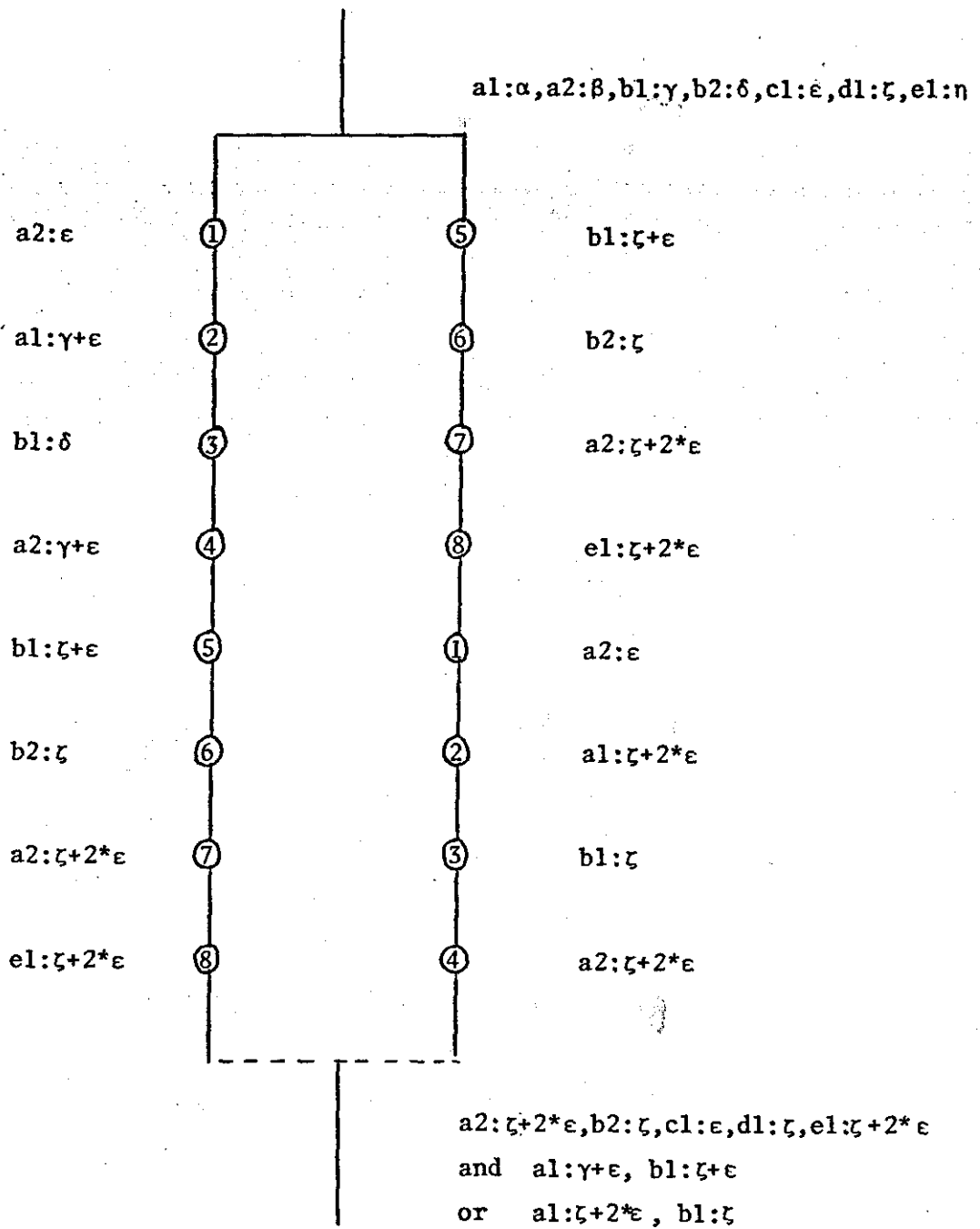


Figure 7.10

SYMBOLIC EXECUTION TREE OF  $CM(S_1, S_2)$

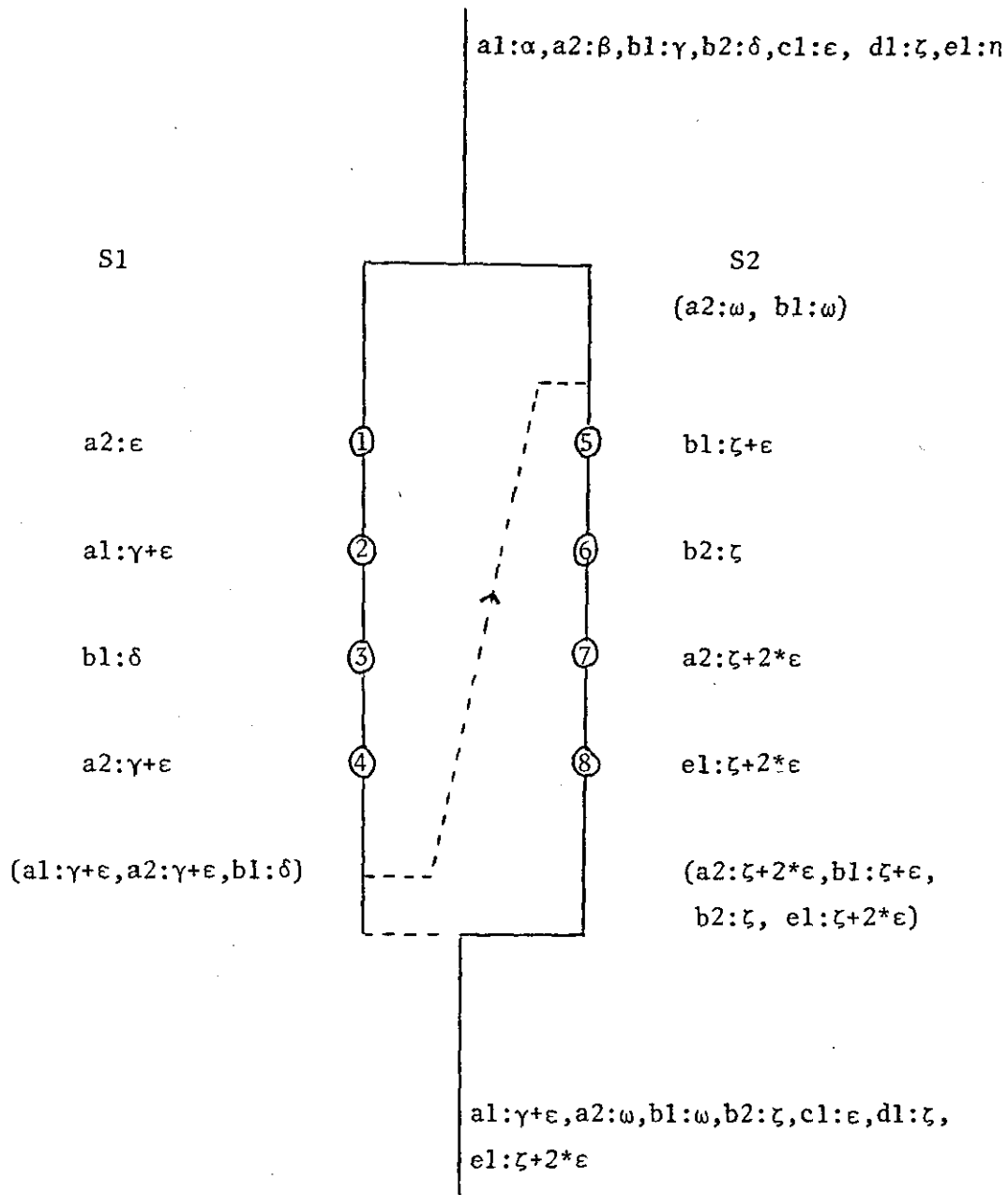


Figure 7.11

SYMBOLIC EXECUTION NETWORK OF  $PR(S_1, S_2)$

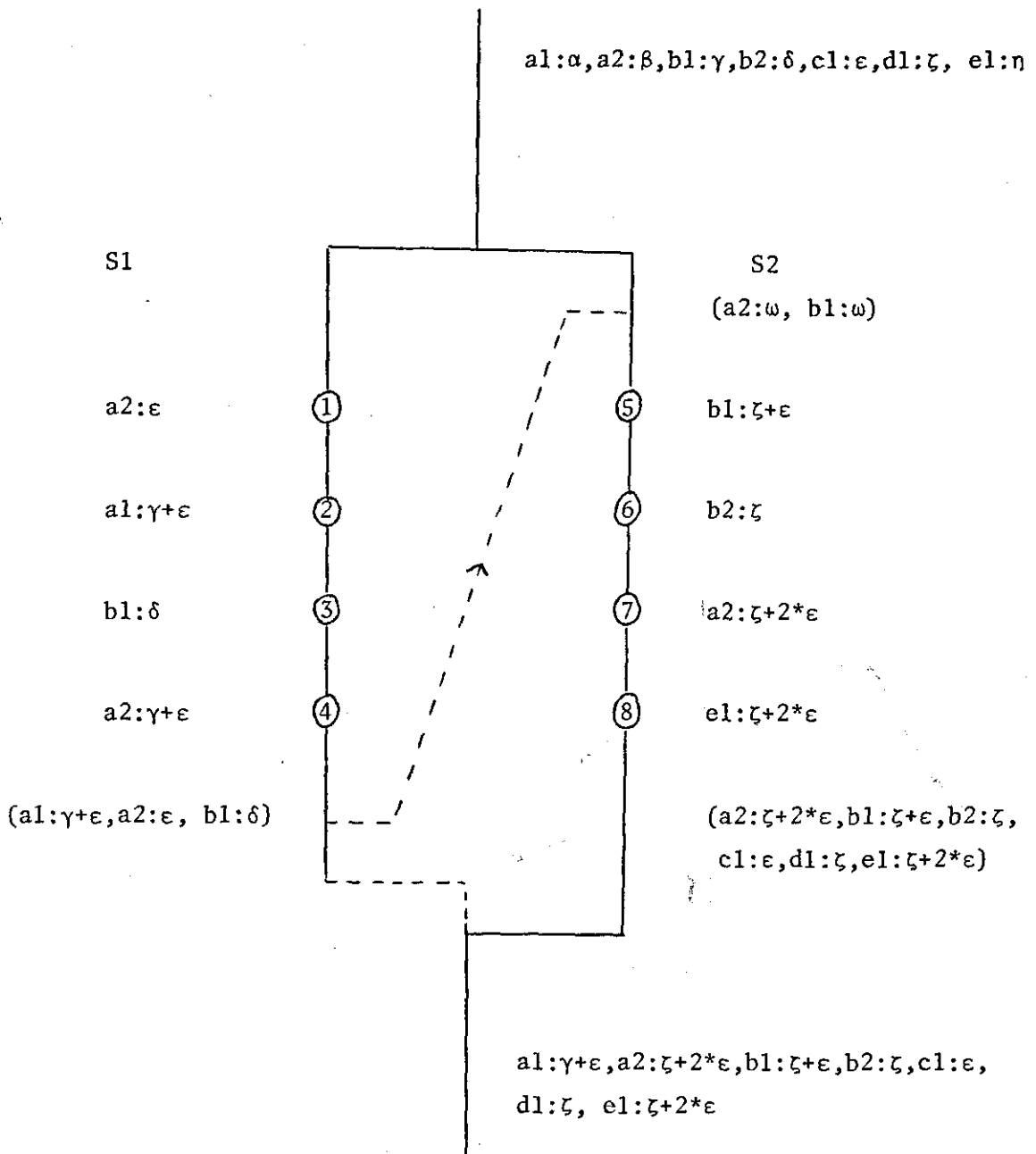


Figure 7.12

SYMBOLIC EXECUTION NETWORK OF  $CV(S_1, S_2)$



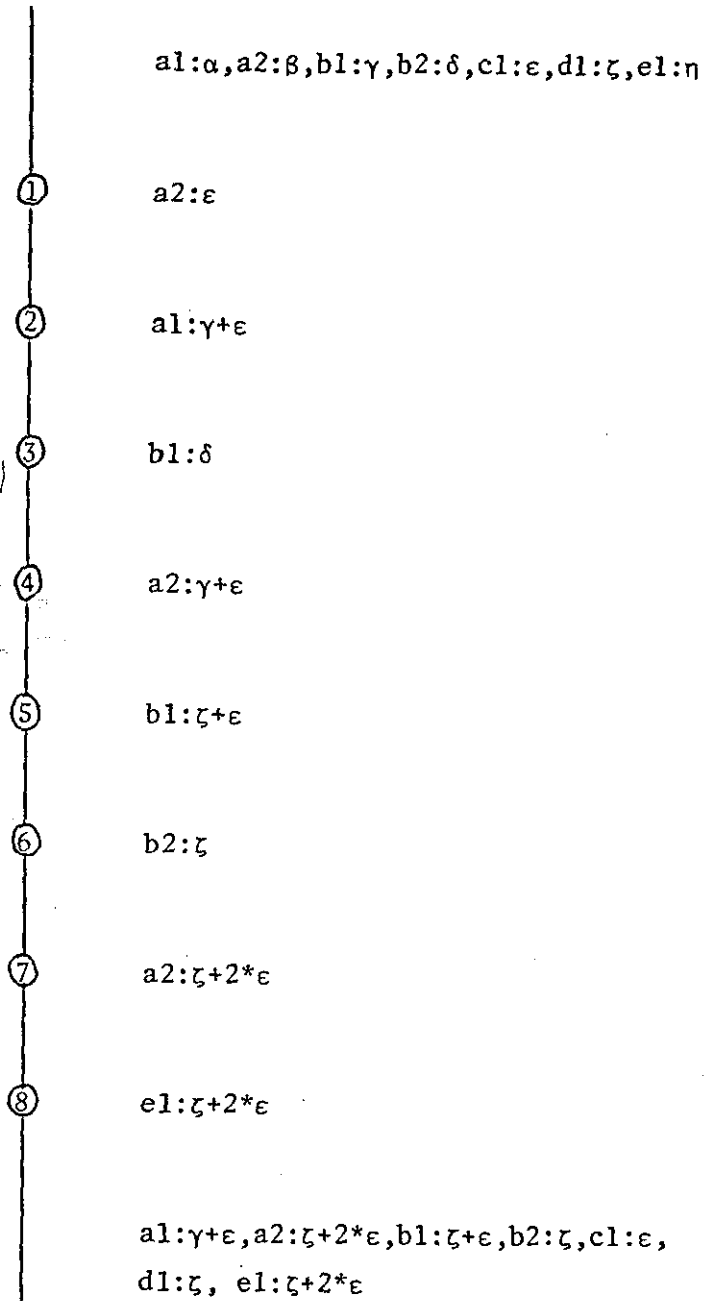


Figure 7.13

SYMBOLIC EXECUTION OF  $CC(S_1, S_2)$

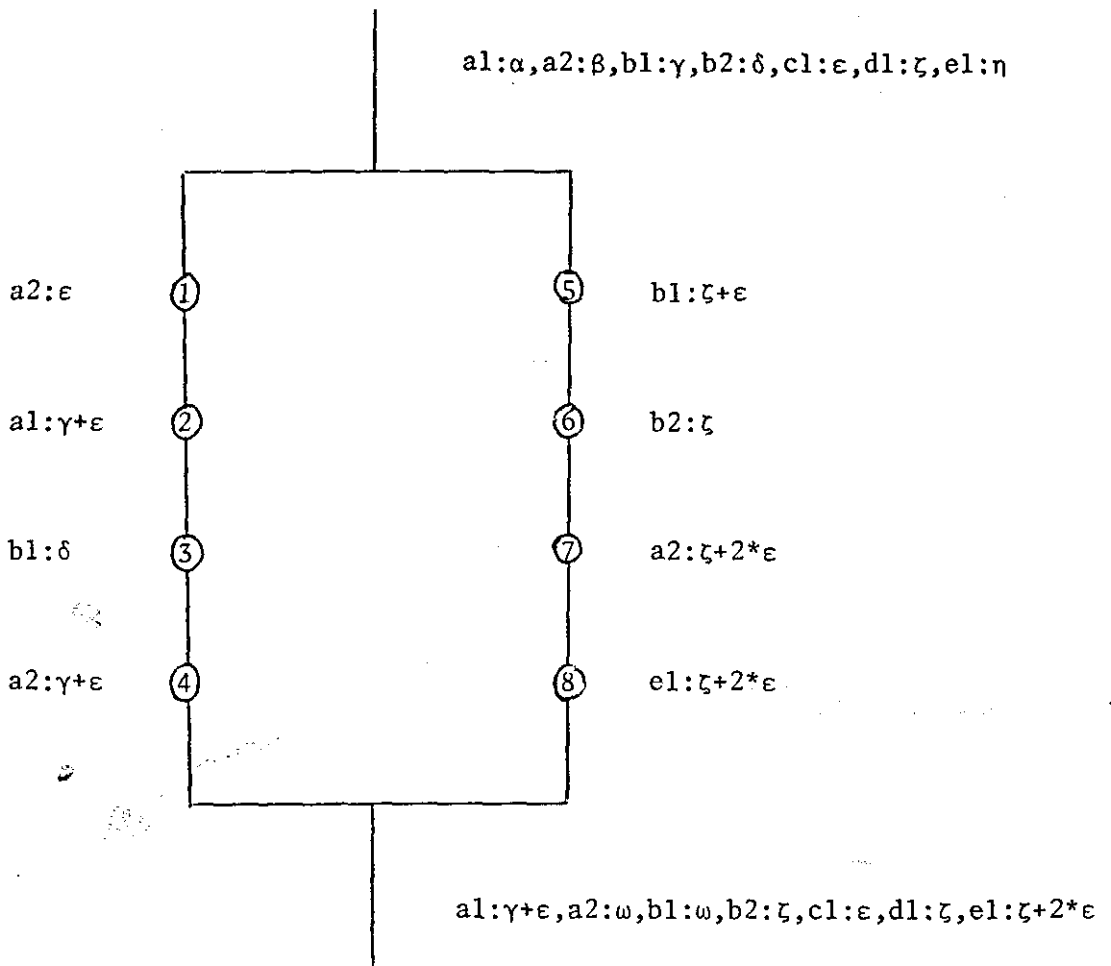


Figure 7.14

SYMBOLIC EXECUTION OF  $SN(S_1, S_2)$

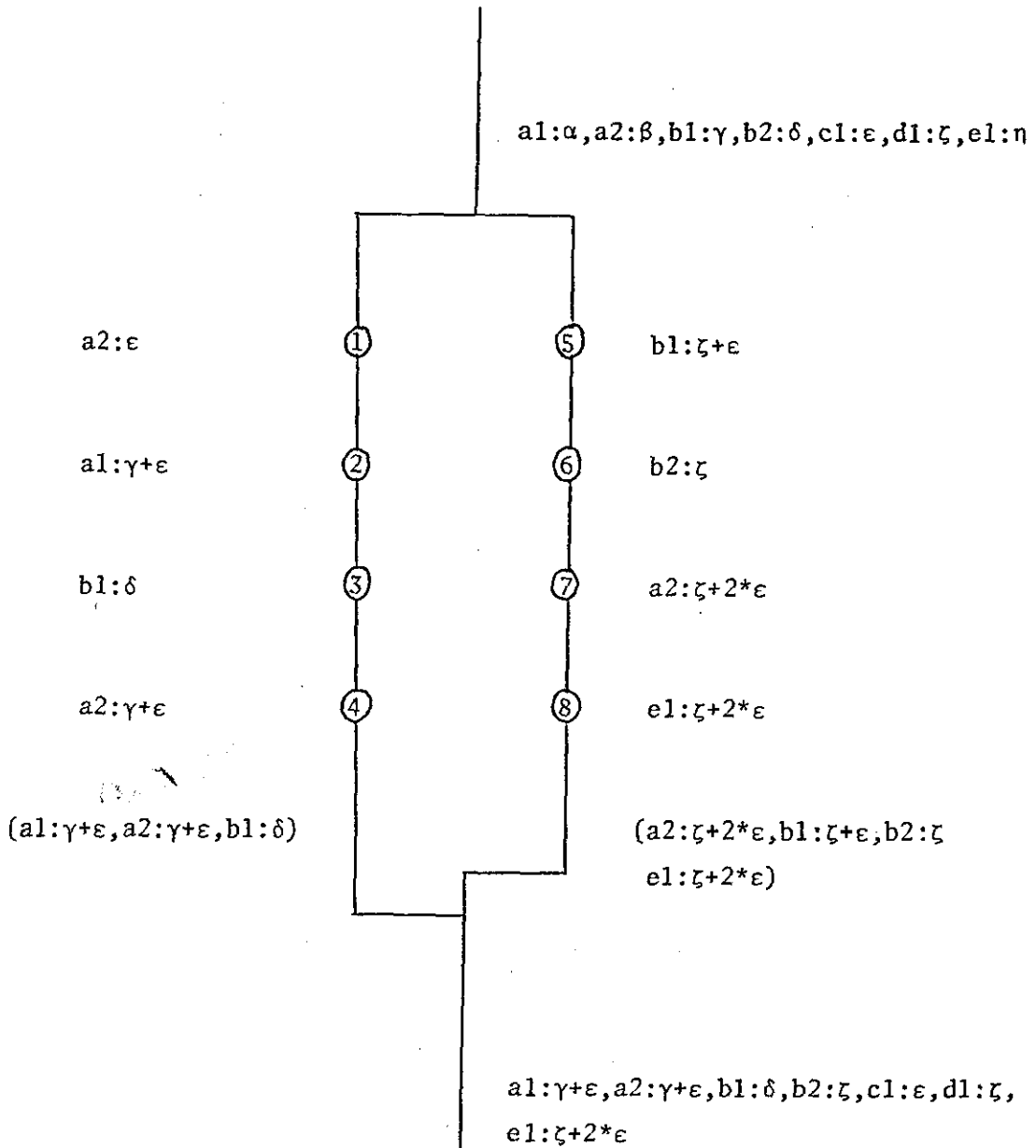


Figure 7.15

SYMBOLIC EXECUTION NETWORK OF  $IN(S_1, S_2)$

### 7.5 PROVING CORRECTNESS BETWEEN A NUMBER OF PARALLEL STANZAS

So far in this chapter techniques have been proposed that may be used to prove correctness between two stanzas that are to be executed in parallel. It is possible to adapt the Definitions 7.2 to 7.5 to describe indefiniteness and propagation of values for a number of stanzas that are to be executed in parallel. Thus the tests for correctness given in section 7.3 may be extended to find the consequents of a number of stanzas being executed in parallel.

CHAPTER 8

CONCLUSIONS

## 8.1 SUMMARY

When computer programs are to be run on parallel machines, there are basically two approaches to determining which parts of a program (i.e. stanzas) may be run on different processors. Such stanzas may be explicitly indicated by the programmer using special operators or the program may be implicitly divided into stanzas as part of the compilation procedure. In this thesis, both explicit and implicit parallelism have been considered.

As a consequence a number of relationships have been introduced that can be said to be the explicit relationships between stanzas. A subset of these relationships have been used to determine which parts of a serial program can be executed in parallel. The different effects on these relationships when using processors with their own private memories and those with only access to a shared memory were considered.

Methods by which both explicit and implicit parallel programs may or may not be optimised using standard serial techniques have been examined. It has been shown that some methods are readily amenable to parallel processing whereas others may detract from potential parallelism. Similarly, methods by which programs may be checked for correctness have been introduced, based upon the serial techniques of symbolic execution.

Within expressions it was assumed that the task of determining which operations could be executed in parallel would be too tedious to be done explicitly. However, the algorithm proposed for finding parallelism within expressions respects the ordering imposed on an expression by the programmer.

## 8.2 DETECTION OF IMPLICIT PARALLELISM AND CORRECTNESS OF PARALLEL PROGRAMS

It can be seen that the tests for detecting which relationships exist between the stanzas of a sequential program (see Chapter 5) and those for testing the correctness of a parallel program (see Chapter 7) have similarities. These are due to the fact that both techniques need to determine which variables may be undefined in particular circumstances.

When a sequential program is being transformed to a parallel program then at each stage in the testing of which relationship exists between stanzas it is necessary to determine if any variables used will be undefined. When one stanza is able to fetch a value of a variable that may or may not have been changed or, indeed, may be in the process of being changed by another stanza, then in the first stanza that variable is said to be undefined. Similarly, a variable is considered to be undefined if more than one stanza changes it, unless the order in which the stanzas store their results is specified. Such indefiniteness will be inconsequential if that variable is always reset before being subsequently fetched.

When a parallel program is being checked for correctness it is necessary to determine any indefiniteness and to propagate through all values, especially those that are undefined. When one stanza is able to fetch a value of a variable that may or may not have been changed or, indeed, may be in the process of being changed by another stanza then in the first stanza that variable is said to be undefined. Similarly, if more than one stanza changes a variable and the order in which the changes take place is not specified by the relationship then the variable must be considered undefined, after the parallel execution of these stanzas until it is redefined.

The similarities between the two techniques can be illustrated by reconsidering the two stanzas  $S_1$  and  $S_2$  given in Figure 7.7, when

they are to be executed on a machine with private memories. If  $S_1$  and  $S_2$  appear adjacently in a sequential program their tests to detect a parallel relationship between them would be as follows:

$$\begin{aligned} & (X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2) \\ & (a_2 \cup b_1 \cup a_1) \cap ((c_1, d_1) \cup \emptyset) = \emptyset \end{aligned} \quad (8.1)$$

The relationship is therefore at least *conservative*.

$$\begin{aligned} & (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap V_2 \\ & (a_2 \cup b_1 \cup a_1) \cap ((b_2, e_1) \cup \emptyset \cup (b_1, a_2)) = (a_2, b_1) \end{aligned} \quad (8.2)$$

taking  $V_2 = 1$ .

The relationship is therefore not *prerequisite*.

Thus, if all variables set in both  $S_1$  and  $S_2$  may be fetched without being reset then the relationship between  $S_1$  and  $S_2$  is *conservative*.

In subsection 7.4.4 it was shown that  $CV(S_1, S_2)$  was correct for Assumption 1 which required all outputs to be set. The indefiniteness of  $CV(S_1, S_2)$  was found by testing the same sets as those given above in equations (8.1) and (8.2).



### 8.3 OTHER APPLICATIONS OF PARALLEL PROCESSING

Within this thesis parallelism has been studied in Algol-type programs which are usually taken to be scientific programs. It would also be useful to exploit parallelism within other environments such as commercial programs and systems software.

Within a commercial environment many of the tasks that computers perform are inherently parallel. For instance a payroll program may be considered as many parallel processes, as one employee's pay is not dependent upon another's. Therefore, means of expressing explicit parallelism could be introduced into a commercial programming language such as COBOL. In addition the techniques described for finding implicit parallelism at the expression and stanza level should be adapted to handle commercial programs.

Baer and Ellis (1977) have suggested that the techniques of implicit detection of parallelism in programs cannot readily be applied to compilers. Obviously there will be some scope to apply techniques similar to those described here for determining implicit parallelism both at the expression and stanza levels. However, in most cases it would appear to be beneficial to rewrite parts of the compiler. This will mean that it is possible to have a compiler that operates in parallel and detects implicit parallelism in serial programs.

## 8.4 AREAS FOR FURTHER RESEARCH

Within this thesis a number of aspects of parallel processing have been considered. From these a number of areas where further research may be fruitful have become obvious. In the following subsections such areas are outlined and possible approaches to the problems suggested.

### 8.4.1 Automatic Stanza Formation

The stanzas considered in this thesis have been arbitrarily formed. In some cases the stanzas thus formed may not allow a large proportion of potential parallelism to be detected. If it were possible to have some inter-communication between the process that forms stanzas and the one that detects a particular relationship then it may be possible to 'optimise' the parallelism in a program. Such inter-communication may be possible if the compiler itself executes in parallel (as described in the previous section). However, the method by which the optimum size of a stanza may be determined will require further examination.

### 8.4.2 Detection of Parallelism Between Stanzas

In section 5.8 a number of program constructs were mentioned that still need to be examined and possible approaches suggested. It should be possible to develop methods by which all program constructs may be studied for parallelism. However, care would have to be exercised to ensure that the effort of finding parallelism in certain circumstances did not outweigh any parallelism that may be found.

It may be possible to develop new optimising techniques (as opposed to those based on serial programs) to optimise parallel programs. It should be realised that the techniques used for detecting implicit parallelism may be applied to explicit parallel programs to detect more parallelism and hence optimise them.

#### 8.4.3 Expression Parallelism

In subsection 3.4.2 a number of possible extensions to the algorithm for forming a balanced binary tree from an expression were suggested. Most of these extensions are readily implemented. However, for some expressions a tree of minimum height may not be found as explained in section 3.3. It may be possible to develop a method by which the final binary tree representation of an expression may be 'rebalanced' to minimise the height of the tree without affecting the ordering of the expression.

#### 8.4.4 Termination of a Correct Parallel Program

It should be possible to develop a technique that could be capable of indicating whether a correct parallel program terminates or not. The technique would, probably, be similar to that used in serial programs. However, allowances would have to be made for any indefiniteness introduced and the necessity, in many instances, for all parallel paths to terminate.

#### 8.4.5 Explicit Parallelism

Within this thesis a number of relationships have been introduced that may be used to express either implicit or explicit parallelism. In the previous chapter a method was suggested of how a particular relationship may be explicitly represented (see Figure 7.1). Further investigation may reveal alternative methods of representing explicit parallelism in line with existing constructs, some of which were described in section 1.4.

The methods of forming stanzas described in Chapter 4 and the tests outlined in the following chapter may be adapted to give guidelines

to be used when writing parallel programs. The explicit formation of stanzas will probably correspond to the formation of modules in a structured programming environment (Kernighan and Plauger, 1976).

The tests may require some simplification so that they can be readily applied by a programmer.

## REFERENCES

- AHO A.V. and ULLMAN J.D., 1977: '*Principles of Compiler Design*',  
Pub. Addison-Wesley.
- ALLEN F.E. and COCKE J., 1972: '*A Catalogue of Optimizing Transformations*',  
in '*Design and Optimization of Compilers*', Rustin R.(Ed.), Pub.  
Prentice-Hall Inc.
- ANDERSON A., 1965: '*Program Structures for Parallel Processing*',  
C.A.C.M., Vol.8, pp.786-788.
- BAER J.L. and BOVET D.P., 1968: '*Compilation of Arithmetic Expressions  
for Parallel Computation*',  
I.F.I.P. Congress Proc., Vol.1, pp.340-346.
- BAER J.L. and ELLIS C.S., 1977: '*Model, Design and Evaluation of a  
Compiler for a Parallel Processing Environment*',  
I.E.E.E. Trans. on Software Engineering, Vol.SE-3, pp.394-405.
- BAER J.L. and RUSSELL E.C., 1970: '*Preparation and Evaluation of  
Computer Programs for Parallel Processing Systems*',  
in '*Parallel Processor Systems, Technologies and Applications*',  
L.C. Hobbs (Ed.), Pub. Spartan Books, pp.375-415.
- BARNES J.G.P., 1976: '*RTL/2 Design and Philosophy*',  
Pub. Heyden.
- BARRON D.W., 1960: '*Recursive Techniques in Programming*',  
Pub. MacDonald/Elsevier.

- BARRON D.W., 1977: '*An Introduction to the Study of Programming Languages*',  
Pub. Cambridge University Press.
- BERNSTEIN A.J., 1966: '*Analysis of Programs for Parallel Processing*',  
I.E.E.E. Trans. on Electronic Computers, Vol. EC-15, pp.757-763.
- BINGHAM H.W., FISHER D.A. and REIGEL E.W., 1967: '*Automatic Detection of Parallelism in Computer Programs*',  
Burroughs Corporation, AD662 274.
- BINGHAM H.W. and REIGEL E.W., 1968: '*Parallelism in Computer Programs and in Machines*',  
Burroughs Corporation, AD667 907.
- BINGHAM H.W. and REIGEL E.W., 1969: '*Parallelism Exposure and Exploitation in Digital Computing Systems*',  
Burroughs Corporation, AD853 523.
- BRINCH HANSEN P., 1973: '*Concurrent Programming Concepts*',  
Computing Surveys, Vol.5, pp.223-245.
- CHEN T.C., 1975: '*Overlap and Pipeline Processing*',  
in '*Introduction to Computer Architecture*', Stone H.S. (Ed.)  
Pub. Science Research Associates, pp.375-431.
- ENSLOW P.H. Jr., 1977: '*Multiprocessing Organisations - A Survey*',  
Computing Surveys, Vol.9, pp.103-129.

- EVANS D.J. and SMITH S.A., 1977: '*On the Construction of Balanced Binary Trees for Parallel Processing*',  
Algorithm 99, Computer Journal, Vol.20, pp.378-379.
- FION L. and SUZUKI N., 1977: '*Non-determinism and the Correctness of Parallel Programs*',  
Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh,  
Pennsylvania, U.S.A.
- FIRESTONE R.M., 1971: '*Parallel Programming: Operational Model and Detection of Parallelism*',  
Ph.D. Thesis, New York University.
- FLYNN M.J., 1966: '*Very High Speed Computing Systems*',  
Proceedings of the I.E.E.E., Vol.54, pp.1901-1909.
- FREEMAN P., 1975: '*Software Systems Principles*',  
Pub. Science Research Associates.
- GONZALEZ M.J. and RAMAMOORTHY C.V., 1970: '*Recognition and Representation of Parallel Processable Streams in Computer Programs*',  
in '*Parallel Processor Systems Technologies and Applications*',  
L.C. Hobbs (Ed.), Pub. Spartan Books, pp.335-373.
- GONZALEZ M.J. and RAMAMOORTHY C.V., 1971: '*Program Suitably for Parallel Processing*',  
I.E.E.E. Trans. on Computers, Vol. C-20, pp.647-654.



- GOSDEN J.A., 1966: '*Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers*',  
Fall Joint Computer Conference 29, pp.651-666.
- GRIES D., 1977: '*An Exercise in Proving Parallel Programs Correct*',  
C.A.C.M., Vol.20, pp.921-930.
- HANTLER S.L. and KING J.C., 1976: '*An Introduction to Proving the Correctness of Programs*',  
A.C.M. Computing Surveys, Vol.8, pp.331-353.
- HELLERMAN H., 1966: '*Parallel Processing of Algebraic Expressions*',  
I.E.E.E. Trans. on Electronic Computers, Vol. EC-15, pp.82-91.
- HIGMAN B., 1977: '*A Comparative Study of Programming Languages*',  
Pub. MacDonald and Jane's.
- HOOGENDOORN C.H., 1975: '*Memory Access Problems in Multiprocessor Systems*',  
Ph.D. Thesis, University of Cambridge.
- HOPGOOD F.R.A., 1969: '*Compiling Techniques*',  
Pub. MacDonald/American Elsevier.
- JENSEN K. and WIRTH N., 1976: '*PASCAL - User Manual and Report*',  
Pub. Springer-Verlag.
- KENNEDY K., 1971: '*A Global Flow Analysis Algorithm*',  
International Journal of Computer Mathematics (Section A), Vol.3,  
pp.5-15.

- KERNIGHAN B.W. and PLAUGER P.J., 1976: *'Software Tools'*,  
Pub. Addison-Wesley.
- KNUTH D.E., 1968: *'The Art of Computer Programming'*,  
Vol. 1, Fundamental Algorithms, Pub. Addison Wesley.
- KNUTH D.E., 1971: *'An Empirical Study of FORTRAN Programs'*,  
Software Practice and Experience, Vol.1, pp.105-133.
- KNUTH D.E., 1973: *'The Art of Computer Programming'*,  
Vol.3, Sorting and Searching, Pub. Addison Wesley.
- KUCK D.J., 1975: *'Parallel Processing of Ordinary Programs'*,  
Dept. of Computer Science, University of Illinois at Urbana-  
Champaign, Report. No. UIUCD CS-R-75-767.
- KUCK D.J., 1977: *'A Survey of Parallel Machine Organization and  
Programming'*,  
Computing Surveys, Vol.9, pp.29-59.
- KUCK D.J., MURAOKA Y. and CHEN S.C., 1972: *'On the Number of Operations  
Simultaneously Executable in Fortran-like Programs and Their  
Resulting Speed Up'*,  
I.E.E.E. Trans. on Computers, Vol.C-21, pp.1293-1310.
- LEASURE B.R., 1976: *'Compiling Serial Languages for Parallel Machines'*,  
Dept. of Computer Science, University of Illinois at Urbana-Champaign,  
Report No. UIUCD CS-R-76-805.

- LEE J.A.N., 1974: *'The Anatomy of a Compiler'*,  
Second Edition. Pub. Van Norstrand Reinhold Company.
- MARTIN D.E. and ESTRIN G., 1967: *'Models of Computations and Systems'*,  
I.E.E.E. Trans. on Electronic Computers, Vol. EC-16, pp.59-69.
- MITCHELL G.H., 1972: *'Operational Research: Techniques and Examples'*,  
Pub. English University Press Ltd.
- NAUR P. (Ed.), 1962: *'Revised Report on the Algorithmic Language ALGOL 60'*,  
Computer Journal, Vol. 5, pp.349-369.
- OWICKI S., 1975: *'Axiomatic Proof Techniques for Parallel Programs'*,  
Ph.D. Thesis, Dept. of Computer Science, Cornell University,  
Ithaca, N.Y.
- RAMAMOORTHY C.V. and GONZALEZ M.J., 1969: *'A Survey of Techniques for  
Recognizing Parallel Processable Streams in Computer Programs'*,  
F.J.C.C., A.F.I.P.S. Conference Procs., Vol.35, pp.1-15.
- RAMAMOORTHY C.V., PARK J.H. and LI H.F., 1973: *'Compilation Techniques  
for Recognition of Parallel Processable Tasks in Arithmetic  
Expressions'*,  
I.E.E.E. Trans. on Computers, Vol. C-22, pp.986-998.
- REIGEL E.W., 1970: *'Parallelism Exposure and Exploitation'*,  
in *'Parallel Processors Systems, Technologies, and Applications'*,  
L.C. Hobb (Ed.), Pub. Spartan Books, pp.417-438.

- ROBERTS J.D., 1977: '*A Fast Discrete Fourier Transform Algorithm Suitable for a Pipeline Vector Processor*',  
R.C.S. 87, Department of Computer Science, University of Reading.
- ROBINSON S.K. and TORSUN I.S., 1976a: '*An Empirical Analysis of FORTRAN Programs*',  
Computer Journal, Vol.19, pp.56-62.
- ROBINSON S.K. and TORSUN I.S., 1976b: '*Simplicity: An Empirical Analysis of Algol and Cobol*',  
Private Communication.
- ROHL J.S., 1975: '*An Introduction to Compiler Writing*',  
Pub. MacDonald and Jane's/American Elsevier.
- RUSTIN R. (Ed.), 1972: '*Design and Optimization of Compilers*',  
Pub. Prentice-Hall Inc.
- SQUIRE J.S., 1963: '*A Translation Algorithm for a Multi-Processor Computer*',  
Draft Paper, Information System Laboratory, Department of Electrical Engineering, The University of Michigan.  
N.B. A finalised version of this appears in the unpublished Proc.  
18th A.C.M. National Conference, 1963.
- STONE H.S., 1967: '*One-Pass Compilation of Arithmetic Expressions for a Parallel Processor*',  
C.A.C.M., Vol.10, pp.220-223.

- STONE H.S., 1975: '*Parallel Computers*',  
in '*Introduction to Computer Architecture*', Stone H.S. (Ed.),  
Pub. Science Research Associates, pp.318-374.
- TOWLE R.A., 1976: '*Control and Data Dependence for Program Transformations*',  
Dept. of Computer Science, University of Illinois at Urbana-Champaign,  
Report No. UIUCD CS-R-76-788.
- WARD R.G., 1974: '*A Variable Delay Method for Improving Recognition of  
Parallel Processable Code in Computer Programs*',  
Computer Journal, Vol.17, pp.157-164.
- WICHMANN B.A., 1970: '*Some Statistics from ALGOL Programs*',  
N.P.L., CCU 11.
- WICHMANN B.A., 1973: '*ALGOL 60 Compilation and Assessment*',  
Pub. Academic Press.
- van WIJGAARDEN A. (Ed.), 1976: '*Revised Report on the Algorithmic  
Language ALGOL 68*',  
Pub. Springer-Verlag.
- WILKES M.V., 1965: '*Slave Memories and Dynamic Storage Allocation*',  
I.E.E.E. Trans. on Electronic Computers, Vol. EC-14, pp.270-271.
- WOODWARD P.M, and BOND S.G., 1972: '*Algol 68-R User's Guide*',  
Pub. H.M.S.O. London.

WOODWARD P.M., WETHERALL P.R. and GORMAN B., 1970; '*Official Definition of CORAL 66*',

Pub. H.M.S.O., London.

WULF W., JOHANSSON R.K., WEINSTOCK C.B., HOBBS S.O. and GESCHKE C.M., 1975:

'*The Design of an Optimizing Compiler*',

Pub. American Elsevier.

APPENDIX 1

ALGORITHM FOR CONSTRUCTING A BALANCED BINARY TREE

```

C*****
<make balanced tree> adds an item or a sub-tree to an existing tree,
  at the most suitable point for an entry of its size
_*****C
PROC make balanced tree=(REF INT next,randtop,optop,
                        INT last REF [] TREE this,
                        REF REF TREE orig,[] REF TREE randstack,
                        [] CHAR operators)
VOID:

```

```

C-----
next      indicates the next free position in the array of trees<this>,
          originally 0
randtop   indicates the top item of<randstack>
optop     indicates the number of entries in<operators>
last      indicates where in<this>references to the current set of sub-trees
          begins, initially 1
this      a stack of trees used to hold all subtrees formed
orig      contains the current sub-tree
randstack a stack containing sub-trees and operands
operators a stack of operators, which correspond to the operands
          this procedure will form a balanced tree of operands and operators,
          as long as the operator remains the same
-----C

```

```

      (INT temp;
      INT count,prev,pcount;
      CHAR oper<operators[optop];
      pcount<0;
      INT nooflevels<12;
      [1:2↑nooflevels-1] INT predefined;
      INT value<1;
      predefined[1]<1;
      FOR i FROM 2 TO nooflevels DO
        (value TIMES 2;predefined[value]<i;
        predefined[value+1:2*value-1]<predefined[1:value-1]);
      OP '>'=(TREE expra,exprb) INT:
        (INT lev<(level OF expra>level OF exprb!level OF expra!
        level OF exprb)+1;lev);

```

```

C*****
<pa>adjusts<point>so that instead of pointing to a node it points to
  to its father
_*****C
PROC pa=(REF INT point) VOID:
  (INT temp<point;
  FOR i1 FROM last TO next WHILE temp=point DO
    WHILE this[i1] IS father OF this[point] DO point<i1);

```

```

C-----
attach first element or sub-tree to null node
-----C
      next PLUS 1;
      level OF this[next]<level OF randstack[randtop];
      left OF this[next]<randstack[randtop];
      operator OF this[next]<"@";
      orig<left OF this[next];
      randtop MINUS 1;
      prev<next;
      count<2+(level OF this[next]-1);

```

```

C-----

```



the loop that builds up the tree

```
-----C
WHILE (optop>= LWB operators AND randtop>=LWB randstack!
oper=operators[optop] ! FALSE) DO
  (optop MINUS 1;
  next PLUS 1;
  temp<0;
  IF level OF randstack[randtop]>=level OF orig THEN
    count<2+(level OF orig-1)
  ELSE WHILE level OF randstack[randtop]>predefined[count] DO
    count PLUS 1
  FI;
  left OF this[next]←randstack[randtop];
  operator OF this[next]←oper;
  IF predefined[count]=1
    OR pcount=0
  THEN
```

C-----
 place one place above the previous entry

```
-----C
    father OF this[next]←this[prev];
    right OF this[next]←left OF this[prev];
    left OF this[prev]←this[next]
  ELSE
```

C-----
 place <temp>+ 1 places above the previous entry

```
-----C
    temp←predefined[count]-level OF this[prev];
    FOR i TO temp WHILE operator OF (father OF this[prev])#'"@'"
      DO pa(prev);
    father OF this[next]←father OF this[prev];
    IF operator OF (father OF this [prev])='@'" THEN
      operator OF (father OF this [next])←'"@'"
    ELSE this [prev] IS right OF (father OF this [prev]) THEN
      right OF (father OF this [prev])←this[next]
    ELSE left OF (father OF this [prev])←this[next]
    FI;
    father OF this [prev]←this[next];
    right OF this[next]←this[prev]
  FI;
```

C-----
 if a sub-tree of level greater than one has been added update count
 to allow for this

```
-----C
  IF level OF randstack[randtop]=1 THEN prev←next
  ELSE level OF randstack[randtop]>=level OF orig THEN
    prev←next;
    count<2+(level OF randstack[randtop])-1
  ELSE count PLUS 2+(level OF randstack[randtop]-1)-1;
  FOR il FROM last TO next-1 DO
    IF this[il] IS left OF this[next] THEN
      prev←il;
      father OF this[prev]←this[next];
      GOTO lz
    FI;
```

lz: SKIP

```
FI;
IF operator OF (father OF this[next])='@'" OR pcount=0 THEN
  orig←this[next]; pcount←1
```

```
FI;  
count PLUS 1:  
randtop MINUS 1:  
temp←next:
```

C-----  
update levels of all trees affected by this insertion  
-----C

```
IF operator OF this[temp]= "@" THEN  
level OF this[temp]←left OF this[temp]'>right OF this[temp]  
ELSE  
WHILE operator OF this[temp]#"@" DO  
(level OF this [temp]←left OF this[temp]'>'  
right OF this[temp];pa(temp))  
FI));
```

APPENDIX 2

ANALYSER

```

1  'BEGIN'
2  !CHARPUT!OUTPUT;OPENC(OUTPUT,LINE PRINTER,1);
3  !CHARPUT!MONITOR;OPENC(MONITOR,LINE PRINTER,2);
4  !INT!LENGTH OF LINE+81;          'C' CARDS          'C'
5  EVENT!OF!OUTPUT+(!INT!1)!INT!;
6  !BEGIN'
7      (I=+11!NEXTLINE(OUTPUT)!0!-1)
8  !END!;
9  OUT!OF!OUTPUT+(!INT!1)!INT!;
10 !BEGIN'
11     !IF!CHARNUMBER(OUTPUT)=LENGTH OF LINE 'AND' I#-33'THEN'
12         NEXTLINE(OUTPUT)
13     !FI!;
14     STANDARD OUT(1)
15 !END!;
16
17 EVENT!OF!MONITOR+(!INT!1)!INT!;
18 !BEGIN'
19     !IF!I=-11'THEN'NEXTLINE(MONITOR);0 'ELSE' -1 'FI'
20 !END!;
21 OUT!OF!MONITOR+(!INT!1)!INT!;
22 !BEGIN'
23     !IF!CHARNUMBER(MONITOR)=LENGTH OF LINE 'AND' I#-33'THEN'
24         NEXTLINE(MONITOR)
25     !FI!;
26     STANDARD OUT(1)
27 !END!;
28
29 !INT!L+12,M+12,N+11;          'C'HERE ONLY 12 STANZAS HANDLED  'C'
30 !INT!T,T1,SN+1;

```

```

31 'C' SN IS THE CURRENT STANZA NO, AND T + T1 ARE ALWAYS TEMPORARIES 'C'
32 !INT'LCOUNT+0,SCOUNT+0,MAXCOUNT+15;
33 'C' MAXCOUNT IS THE UPPER LIMIT OF VARIABLES TO BE USED IN ASTANZA 'C'
34 [1:20,1:M,1:N]'CHAR'WNAME,XNAME,YNAME,ZNAME;
35 !CLEAR'WNAME;
36 !CLEAR'XNAME;
37 !CLEAR'YNAME;
38 !CLEAR'ZNAME;
39 [1:M,1:N]'CHAR'WTEMP;
40 !CLEAR'WTEMP;
41 !STRING'LHS;
42 !STRING'TEMP;
43 !BOOL'YTYPE+'FALSE';
44 !STRING'SPCHAR="+-*/+)=('),";
45 !CHAR'C1;
46
47 !MODE!'SPROC'='STRUCT'('CHAR'TYPE,'STRING'NAME,'INT'SNO,'INT'TNO,'INT'POS,
48 'INT'LC);
49 [1:10]'SPROC'PROCS;
50 !INT'PROCPOINT+0;
51 !MODE!'SPARAM'='STRUCT'('CHAR'TYPE,'STRING'NAME);
52 [1:20]'SPARAM'PARAMS;
53 [1:20,1:6]'CHAR'PROCNAME;'CLEAR'PROCNAME;
54 'C' PROCEDURE NAMES LIMITED TO 6 CHARS AT THE MOMENT 'C'
55 !INT'NAMEPOINT+0;
56 !INT'PARAMPOINT+1;
57 'C' N.B. ALL ERRORS SHOULD BE RECOGNISED BY OTHER PARTS OF THE COMPILER 'C'
58 !PROC'ERROR=('INT'N)'VOID';
59 !BEGIN'
60 PRINT(("ERROR TYPE ",N));

```

```

61         FREE
62     'END':
63
64     'C' SHUFFLE REMOVES THE POS.TH. ITEM FROM ROLL + MOVES EVERYTHING ELSE
65         DOWN ONE PLACE
66     'PROC'SHUFFLE=( 'REF' T, J 'CHAR' ROLL, 'INT' POS) 'VOID':
67     'BEGIN'
68         'FOR' J 'FROM' POS 'TO' M-1 'DO' ROLL[J]+ROLL[J+1]#
69         'CLEAR' ROLL[M]
70     'END':
71
72     'C' ADDTO ADDS ITEM ON TO THE END OF ROLL
73     'PROC'ADDTO=( [ ] 'CHAR' ITEM, 'REF' I, 'CHAR' ROLL) 'VOID':
74     'BEGIN'
75         'INT' T+0;
76         'FOR' J 'TO' M 'WHILE' ROLL[J,1]# " " 'DO' T+J;
77         'IF' T=M 'THEN' ERROR(101) 'ELSE' ROLL[T+1,1]:='UPB'ITEM]+ITEM'FI'
78     'END':
79
80     'PROC'SKIPSPACES=( 'REF' 'CHAR' CH) 'VOID':
81     'BEGIN'
82         CH+" "; 'WHILE' CH=" " 'DO' (READ(CH); PUT(MONITOR,CH))
83     'END':
84
85     'PROC''VOID'SETAS:
86
87     'PROC''VOID'BLOCK:
88
89     'PROC''VOID'REORDER;
90

```

```

91      !PROC('REF' 'INT', 'REF' 'INT') 'VOID' 'TIDY';
92
93      'G' MATCH RETURNS TRUE IF C IS A MEMBER OF S          !C'
94      !PROC 'MATCH=( 'CHAR' C, [ ] 'CHAR' S) 'BOOL';
95      !BEGIN'
96          'BOOL' B+ 'TRUE';
97          'FOR' J 'TO' 'UPB' S 'WHILE' B 'DO' (C=S[J] | B+ 'FALSE?');
98          B
99      !END';
100
101      'G' MATCH KEY RETURNS A NUMBER = TO THE POSITION OF WORD IN WORDLIST , OR 0
102      IF THERE IS NO MATCH          !C'
103      !PROC 'MATCHKEY=( [ ] 'CHAR' WORD) 'INT';
104      !BEGIN'
105          [1:10,1:15] 'CHAR' WORDLIST;
106          'CLEAR' WORDLIST;
107          WORDLIST[1]+ "FOR "; WORDLIST[2]+ "IF ";
108          WORDLIST[3]+ "STEP "; WORDLIST[4]+ "UNTIL ";
109          WORDLIST[5]+ "DO "; WORDLIST[6]+ "THEN ";
110          WORDLIST[7]+ "ELSE "; WORDLIST[8]+ "BEGIN ";
111          WORDLIST[9]+ "END "; WORDLIST[10]+ "PROC ";
112      'G' EXPAND ON THIS AS NECESSARY TO HANDLE OTHER CONSTRUCTS    !C'
113          'INT' I1+0;
114          'FOR' J 'TO' 'UPB' WORDLIST 'WHILE' I1#0 'DO'
115              'IF' WORD=WORDLIST[J] 'THEN' I1+J 'FI';
116          I1
117      !END';
118
119      'G' MATCHNAME RETURNS THE POSITION OF NAME IN NAMELIST OR 0 IF NOT THERE    !C'
120      !PROC 'MATCHNAME=( [ , ] 'CHAR' NAMELIST# [ ] 'CHAR' NAME) 'INT';

```

```

121  BEGIN
122      INT J1+0;
123      FOR I1 TO UPB NAMELIST WHILE NAMELIST[I1,1]# " " AND J1=0 DO
124          IF NAMELIST[I1]=NAME[I1,2] UPB NAMELIST THEN
125              J1+I1
126          END IF
127      END FOR
128  END;

129  PROC GETNAME=( ) CHAR:
130  BEGIN
131      [1:N] CHAR NAME;
132      CLEAR NAME;
133      CHAR C+ " ";
134      INT T+1;
135      FOR J TO N WHILE MATCH(C, SPCHAR) DO
136          (SKIPSPACES(C); NAME[T]+C) PLUS 1;
137          (C="|" | FOR J1 FROM J+1 TO N WHILE C#"|" DO
138              (SKIPSPACES(C); NAME[TJ+C] PLUS 1));
139          NAME[T-1]+ " ";
140      END FOR
141      NAME
142  END;

143  PROC BACK='VOID';          'C' BACKSPACES          'C'
144  BEGIN READ(BACKSPACE) BACKSPACE(MONITOR) END;
145
146  PROC REA=(REF [ ] CHAR C) VOID;          'C' READS          'C'
147  BEGIN
148      READ(C); PUT(MONITOR,C)
149  END;
150

```



```

151      !PROC'READ=( 'REF''INT'K)'VOID';          !C' READS AN INTEGER      !C'
152      !BEGIN'
153          !STRING'K1+GETNAME;
154          K+0;
155          !FOR'KIN'TO'!UPB'K1'WHILE'K1[KIN]#" "!'DO'
156          !BEGIN'
157              !INT'K2+'ARS'K1[KIN];
158              !IF'K2>9'THEN'PRINT("NUMBER ? ");ERROR(200)'FI';
159              K+K+10+K2
160          !END';
161      BACK
162  !END';
163
164      !C' COPIES STANZA POS INTO STANZA PES      !C'
165      !PROC'COPY=( 'INT'POS,'INT'PES)'VOID';
166      !BEGIN'
167          WNAME[PES]+WNAME[POS];
168          XNAME[PES]+XNAME[POS];
169          YNAME[PES]+YNAME[POS];
170          ZNAME[PES]+ZNAME[POS]
171      !END';
172
173      !C' HANDLES CALLS TO PROCEDURES          !C'
174      !PROC'PROGCALL=( 'INT'POS,'REF''INT'LC)'VOID';
175      !BEGIN'
176          !:NO'CHAR'TEMP;
177          LHS+"";
178          COPY(POS,SN);
179          !IF!:(NO'OF'PROCS[POS])=0'THEN'
180

```

```

181      'IF'(SKIPSPACES(C1);C1#"(")'THEN'
182          PRINT("SHOULD BE NO PARAMETERS ")#ERROR(300)
183      'ELSE'ERROR(302)
184      'FI'
185  'ELSP'(BACK;SKIPSPACES(C1);C1#"(")'THEN'
186      PRINT("SHOULD BE SOME PARAMS ")#ERROR(301)
187  'ELSE'
188      PRINT((BACKSPACE," "));
189      'FOR'I1'WHILE'(BACK;REA(C1);C1#"")'DO?
190      'BEGIN'
191          'CHAR'T1+TYPE'OF'PARAMS[(POS'OF'PROCS[#OS])+I1-1];
192          TEMP+GETNAME?
193          'IF'T1="N"'THEN'ADDTO{TEMP,YNAME[SN]}
194          'ELSE'IF'T1="V"'THEN'ADDTO{TEMP,WNAME[SN]}
195          'ELSE'ERROR(305)
196          'FI'
197      'END'
198  'FI?;
199  LC'PLUS'(LC'OF'PROCS[POS]);
200  REORDER;
201  SKIPSPACES(C1);'IF'C1#"';"'THEN'ERROR(306)'FI'
202  'END';
203
204  'C' HANDLES SIMPLE ASSIGNMENT STATEMENTS
205  1PRDC'ASSIGNMENT='INT';
206  'BEGIN'
207      'INT'U1+1;
208      'INT'POS;
209      'INT'LC+1;
210      LHS+GETNAME;

```

'C'

```

211      'IF' (POS+MATCHNAME(PROCNAME, LHS); POS#0) 'THEN'
212      'IF' (BACK; SKIPSPACES(C1); C1#"?") 'THEN'
213          LC'MINUS'1;
214          TIDY(SN, LC); PRINT(" PROC CALL "); PROCCALL(POS, LC);
215      TIDY(SN, LC)
216      'ELSE' PRINT("PROC NOT ALLOWED ON LHS "); ERROR(200)
217      'FI'
218      'ELSF' (BACK; SKIPSPACES(C1); C1#"?") 'THEN'
219          PRINT((NEWLINE, "THIS SHOULD BE AN ASSIGNMENT ", NEWLINE))
220      'ELSE'
221          'CHAR' C2+ " ";
222          BACK;
223          'FOR' J 'TO' M 'WHILE' (SKIPSPACES(C2); C2#"#") 'DO'
224              'BEGIN'
225                  'INT' TEMP;
226                  WTEMP[J1]+GETNAME;
227                  'IF' (TEMP+MATCHNAME(PROCNAME, WTEMP[J1]); TEMP)>0 'THEN'
228                      'CLEAR' WTEMP[J1];
229                      J1'MINUS'1; PROCCALL(TEMP, LC)
230                  'FI';
231                  'IF' LHS=WTEMP[J1] 'THEN' YTYPE+ 'TRUE'
232                  'ELSE' J1+J1+1; LC'PLUS'1
233                  'FI';
234              BACK
235          'END'
236      'FI' &
237      LC
238      'END';
239      'C' THIS HANDLES ALL MULTIPLE USES AND ASSIGNS TO THE CORRECT NAME      'C'
240

```

```

241 REORDER+1VOID';
242 !BEGIN'
243 T+1;
244 'TO'M'WHILE'WTEMP[T,1]#* "'DO'
245 ('BOOL'B+'TRUE')
246 (('FOR'J1'TO'M'WHILE'{YNAME[SN,J1,1]#* "'AND'B)'DO'
247 (WTEMP[T]=YNAME[SN,J1]|SHUFFLE(WTEMP,T);SCOUNT'MINUS'1;
248 B+'FALSE');B)IT'PLUS'1));
249 T+1;
250 'TO'M'WHILE'WTEMP[T,1]#* "'DO'
251 ('BOOL'B+'TRUE')
252 (('FOR'J1'TO'M'WHILE'{ZNAME[SN,J1,1]#* "'AND'B)'DO'
253 (WTEMP[T]=ZNAME[SN,J1]|SHUFFLE(WTEMP,T);SCOUNT'MINUS'1;
254 B+'FALSE');B)IT'PLUS'1));
255 T+1;T1+1;
256 'TO'M'WHILE'WTEMP[T,1]#* "'DO'
257 ('BOOL'B+'TRUE');
258 T1+1;
259 'TO'M'WHILE'XNAME[SN,T1,1]#* "'AND'B)'DO'
260 ((WTEMP[T]=XNAME[SN,T1]|ADDTO(WTEMP[T],ZNAME[SN]);
261 SHUFFLE(WTEMP,T);SHUFFLE(XNAME[SN],T1);
262 SCOUNT'MINUS'1;B+'FALSE');
263 (BIT1'PLUS'1));
264 (BIT'PLUS'1));
265 'TO'M'WHILE'WTEMP[1,1]#* "'DO'
266 ('BOOL'B+'TRUE')
267 (('FOR'J1'TO'M'WHILE'{WNAME[SN,J1,1]#* "'AND'B)'DO'
268 (WTEMP[1]=WNAME[SN,J1]|SHUFFLE(WTEMP,1);SCOUNT'MINUS'1;
269 B+'FALSE');B)ADDTO(WTEMP[1],WNAME[SN]);
270 SHUFFLE(WTEMP,1));

```

```

271 'IF' ('BOOL'B+'FALSE'; 'FOR'J'TO'M'WHILE?WNAME[SN,J,1]#" "IAND'
272 'NOT'B'DO'
273 (LHS=WNAME[SN;J]IB+'TRUE'; SHUFFLE(WNAME[SN],J);
274 ADDTO(LHS?YNAME[SN]); SCOUNT'MINUS'1);B)
275 'THEN' 'SKIP'
276 'ELSEF'
277 ('BOOL'B+'FALSE'; 'FOR'J'TO'M'WHILE?XNAME[SN;J,1]#" "IAND'
278 'NOT'B'DO'
279 (LHS=XNAME[SN;J]IB+'TRUE'; SHUFFLE(XNAME[SN],J);
280 ADDTO(LHS?ZNAME[SN]); SCOUNT'MINUS'1);B)
281 'THEN' 'SKIP'
282 'ELSEF' ('BOOL'B+'FALSE'; 'FOR'J'TO'M'WHILE'YNAME[SN,#,1]#" "IAND'
283 'NOT'B'DO'
284 (LHS=YNAME[SN;J]IB+'TRUE'; SCOUNT'MINUS'1);B)
285 'THEN' 'SKIP'
286 'ELSEF' ('BOOL'B+'FALSE'; 'FOR'J'TO'M'WHILE'ZNAME[SN,#,1]#" "IAND'
287 'NOT'B'DO'
288 (LHS=ZNAME[SN;J]IB+'TRUE'; SCOUNT'MINUS'1);B)
289 'THEN' 'SKIP'
290 'ELSEF'YTYPE'THEN'ADDTO(LHS,YNAME[SN])
291 'ELSE'ADDTO(LHS,XNAME[SN])
292 'FI'
293 'END';
294
295 'G' TIDY IS USED TO COMPLETE A STANZA
296 TIDY<('REF''INT'NUM,'REF''INT'SC)'VOID':
297 'BEGIN'
298 'IF'5C>0'THEN'
299 'INT'PRESPOS=CHARNUMBER(MONITOR);
300 NEWLINE(MONITOR);

```

```

301 PUT(MONITOR, (" -+*- STANZA ", NUM=PROCPOINT));
302 NEWLINE(MONITOR);
303 NEWLINE(MONITOR);
304 'TO, PRESPOS=1 'DO' SPACE(MONITOR);
305 PRINT((NEWLINE, NEWLINE, "STANZA "; NUM, NEWLINE));
306 NEWLINE(OUTPUT);
307 PUT(OUTPUT, "W ");
308 PUT(OUTPUT, WNAME[ NUM, 1]);
309 'FOR' J 'FROM' 2 'WHILE' WNAME[ NUM, J, 1] # " " 'DO'
310 (PUT(OUTPUT, (";", WNAME[ NUM, J])); (J=(J+'9')*9) NEWLINE(OUTPUT));
311 PUT(OUTPUT, ";");
312 NEWLINE(OUTPUT);
313 PUT(OUTPUT, "X ");
314 PUT(OUTPUT, XNAME[ NUM, 1]);
315 'FOR' J 'FROM' 2 'WHILE' XNAME[ NUM, J, 1] # " " 'DO'
316 (PUT(OUTPUT, (";", XNAME[ NUM, J])); (J=(J+'9')*9) NEWLINE(OUTPUT));
317 PUT(OUTPUT, ";");
318 NEWLINE(OUTPUT);
319 PUT(OUTPUT, "Y ");
320 PUT(OUTPUT, YNAME[ NUM, 1]);
321 'FOR' J 'FROM' 2 'WHILE' YNAME[ NUM, J, 1] # " " 'DO'
322 (PUT(OUTPUT, (";", YNAME[ NUM, J])); (J=(J+'9')*9) NEWLINE(OUTPUT));
323 PUT(OUTPUT, ";");
324 NEWLINE(OUTPUT);
325 PUT(OUTPUT, "Z ");
326 PUT(OUTPUT, ZNAME[ NUM, 1]);
327 'FOR' J 'FROM' 2 'WHILE' ZNAME[ NUM, J, 1] # " " 'DO'
328 (PUT(OUTPUT, (".", ZNAME[ NUM, J])); (J=(J+'9')*9) NEWLINE(OUTPUT));
329 PUT(OUTPUT, ".");
330 NEWLINE(OUTPUT);

```

```

331         NUMPLUS'1
332         'F115
333         SC+0
334     !END';
335
336 'C' THIS WILL FIND THE TYPE OF KEY WORD AND ROUTE IT TO IT S HANDLING PROC 'C'
337     !PROC'KEYWORD='INT';
338     !BEGIN'
339         [1:50'CHAR'WORD;
340         'INT'K1;
341         'CHAR'A;
342         'CLEAR'WORD;
343         REA {A};
344         'FOR'J'TO'UPB'WORD+1'WHILE'(REA (A)IA#"")?DO'
345             WORD[J]+A;
346         (A#" "(READ(BACKSPACE));
347         K1←MATCHKEY(WORD);
348         K?
349     !END';
350
351 'C' CONDIT CONSTRUCTS THE ENVIROMENT FOR AN IF STATEMENT 'C'
352     !PROC'CONDIT='VOID';
353     !BEGIN'
354         'CHAR'C2+" ";
355         'INT'K1;
356         TIDY(SN,SCOUNT);
357         NEWLINE(OUTPUT);PUT(OUTPUT,"S ");
358         WNAMB[SN,1]←GETNAME;
359         'FOR'J'FROM'2'TO'M'WHILE'(BACK;REA(C2);C2#"T")'DO'
360     'BEGIN'

```

```

361          WNAME(SN,J1+GETNAME
362          'END';
363          BACK;
364          K1+KEYWORD;
365          (K1#6|ERROR(106));
366          SCOUNT+1;
367          TIDY(SN,SCOUNT);
368          REA(C1);BACK;
369          BLOCK;
370          TIDY(SN,SCOUNT);
371          SKIPSPACES(C1);
372          'IF'G1="'" THEN'
373          'BEGIN'
374              BACK;
375              K1+KEYWORD;
376              'IF'K1=7'THEN'REA (C1);
377              BACK;
378              BLOCK;
379              TIDY(SN,SCOUNT)
380          'FI'
381 'C' MAY BE NECESSARY TO READ BACKWARDS OVER KEYWORD IF NOT ELSE      'C'
382          'END'
383          'ELSE'READ(BACKSPACE)
384          'FI';
385          NEWLINE(OUTPUT);PUT(OUTPUT,"$ ");NEWLINE(OUTPUT)
386          'END';
387
388 'C' LOOP CONSTRUCTS A DO LOOP ENVIROMENT      'C'
389          'PROC'LOOP='VOID';
390          'BEGIN'

```



```

391      'STRING'CV;
392      'CHAR'C;
393      'INT'K1,K2;
394      TIDY(SN,SCOUNT);
395      NEWLINE(OUTPUT);PUT(OUTPUT,"#D  ");
396      CV+GETNAME;
397      (MATCHNAME(PROCNAME,CV)#0|PRINT("PROC NOT ALLOWED CV  ");ERROR(200));
398      BACK;REA(C);(C#"+"|ERROR(102));
399      REAI(K1);PUT(OUTPUT,(" ",K1));
400      K2+KEYWORD;
401      (K2#3|ERROR(103));
402      REAI(K1);PUT(OUTPUT,(" ",K1));
403      K2+KEYWORD;
404      (K2#4|ERROR(104));
405      REAI(K1);PUT(OUTPUT,(" ",K1));
406      K2+KEYWORD;
407      (K2#5|ERROR(105));
408      OUTP(OUTPUT,$10XTLS,CV);
409      REA (C1);
410      BACK;
411      BLOCK;
412      TIDY(SN,SCOUNT);
413      PUT(OUTPUT,"#O");NEWLINE(OUTPUT)
414  †END';
415
416  'C' ROUTINE HANDLES PROCEDURE DEFINITIONS
417  †PROC ROUTINE='VOID';
418  †BEGIN'
419      'CHAR'T+"R";
420      TIDY(SN,SCOUNT);

```

'C' FOR TIME BEING ALL PROCS REAL 'C'

```

421 'STRING'S+GETNAME;
422 'C' MAY CHECK TO SEE IF THIS PROC ALL READY EXISTS 'C'
423 PROC$(PROCPOINT'PLUS'1)+(T,S,SN,0,PARAMPOINT'LCOUNT);
424 'FOR'11'TO'UPB'S'WHILE'11<=2'URB'PROCNAME'DO'
425 PROCNAME[PROCPOINT;11]-S[11];
426 READ(BACKSPACE);READ(T);
427 'IF'T=""'THEN' 'SKIP' 'C' NO PARAMETERS 'C'
428 'ELSE'T=""('THEN'
429 'BEGIN'
430 'INT'COUNT+0;
431 'TO'UPB'PARAMS'WHILE'T#"')'DO'
432 'BEGIN'
433 (NO'OF'PROCS[PROCPOINT])'PLUS'1;
434 PARAMS[PARAMPOINT]+("N',(GETNAME));
435 'C' MAY CHECK IF PARAMETER IS ITSELF A PROCEDURE 'C'
436 READ(BACKSPACE);READ(T);
437 PARAMPOINT'PLUS'1;COUNT'PLUS'1
438 'END'
439 'END'
440 'ELSE'
441 PRINT(" PARAM ? ");ERROR(201)
442 'FIAS
443 SKIPSPACES(T);
444 SKIPSPACES(C1);BACK;
445 BLOCK;
446 'C' CHECK PARAMS + SET UP REST OF INFORMATION 'C'
447 (LC'OF'PROCS[PROCPOINT])'PLUS'1; 'C' ??? 'C'
448 'FOR'11'FROM'PARAMPOINT-NO'OF'PROCS[PROCPOINT]'TO'PARAMPOINT-1'DO'
449 'BEGIN'
450 'INT'T1;

```

```

451 'IF'(T1+MATCHNAME(WNAME[SN],NAME'OF'PARAMS[I1]))#0'THEN'
452 SHUFFLE(WNAME[SN],T1)
453 'ELSF'(T1+MATCHNAME(XNAME[SN],NAME'OF'PARAMS[I1]))#0'THEN'
454 SHUFFLE(XNAME[SN],T1)
455 'ELSF'(T1+MATCHNAME(YNAME[SN],NAME'OF'PARAMS[I1]))#0'THEN'
456 SHUFFLE(YNAME[SN],T1)
457 'ELSF'(T1+MATCHNAME(ZNAME[SN],NAME'OF'PARAMS[I1]))#0'THEN'
458 SHUFFLE(ZNAME[SN],T1)
459 'ELSE'PRINT("PARAMETER DECLARED AND NOT USED ");ERROR(209)
460 'FI'
461 'END';
462 SN'PLUS'1;SCOUNT+0
463 !END';
464
465 'G' SETAS GLOBALLY DECIDES WHAT A STANZA IS
466 SETAS+'VOID';
467 !BEGIN'
468 'INT'K1;
469 'IF'!C1#"'"'THEN'
470 YTYPE+'FALSE';
471 LCOUNT+ASSIGNMENT;
472 'IF'LCOUNT+SCOUNT>MAXCOUNT'THEN'TIDY(SN,SCOUNT)!FI';
473 SCOUNT'PLUS'LCOUNT;
474 REORDER;
475 'CLEAR'WTEMP;'CLEAR'LHS
476 'ELSE'K1+KEYWORD;
477 'IF'K1=1'THEN'LOOP
478 'ELSF'K1=2'THEN'CONDT
479 'ELSF'K1=8'THEN'BLOCK
480 'ELSF'K1=10'THEN'ROUTINE

```

LC'

```

481           'ELSE'ERROR(111)
482           'FI'
483           'FI'
484           'END';
485
486 'C' BLOCK IS CALLED RECURSIVELY-- EACH TIME A BLOCK IS ENTERED
487 BLOCK+'VOID';
488           'BEGIN'
489           'INT'K1;
490           SKIPSPACES(C1);
491           'IF'C1#"'"' THEN'BACK;SETAS
492           'ELSE'
493           BACK;
494           K1+KEYWORD;
495           'IF'K1=10'THEN'ROUTINE
496           'ELSE'
497           'IF'K1#8'THEN'ERROR(108)'FI';
498           SKIPSPACES(C1);BACK;
499           SETAS;
500           'WHILE'
501           (SKIPSPACES(C1);
502           'IF'C1#"'"' THEN'BACK;
503           'TRUE'
504           'ELSF'(BACK;K1+KEYWORD;K1#9)' THEN'
505           'FALSE'
506           'ELSE'BACK;
507           'TO'6'WHILE'(BACK;REA(C1);BACK;
508           (C1#"'"'|'FALSE'|'TRUE'))'DO''SKIP'?
509           'TRUE'
510           'FI')

```

'C'

```
511      'DO' SETAS;  
512      REA (C1)  
513      'FI'  
514      'FI'  
515      }END';  
516      }WHILE' (SKIPSPACES(C1);C1#"*")'DO' (BACK;PRINT(NEWLINE);BLOCK);  
517      BACK;  
518      TIOY(SN,SCOUNT);  
519      NEWLINE(OUTPUT);PUT(OUTPUT,"****"); NEWLINE(OUTPUT)  
520  
521      'END'  
522      'FINISH'  
523      ****
```

APPENDIX 3

DETECTOR

```

1  'BEGIN'
2  'CHARPUT'INPUT;OPENC(INPUT,CARDREADER,1);
3  'C' ALL RELATIONSHIPS ASSUME PRIVATE MEMORY IS AVAILABLE 'C'
4  'INT'L+12,M+12,N+12; 'C' ONLY 12 STANZAS + 1ST 12 CHARS SIGNIFICANT 'C'
5  'MODE''LISTS'='STRUCT'([1;M]'REF'[]'CHAR'NAME);
6  [1;L]'LISTS'W,X,Y,Z,WY,XYZ;
7  [1;L,1;M,1;N]'CHAR'WV,XV,YV,ZV;
8  [1;1]'CHAR'SP+" ";
9  'FOR'I'TO'M'DO''FOR'J'TO'N'DO'
10   'BEGIN'
11     (NAME'OF'W[I])[J]+[1;'UPB'SP]'CHAR'+SP;
12     (NAME'OF'X[I])[J]+[1;'UPB'SP]'CHAR'+SP;
13     (NAME'OF'Y[I])[J]+[1;'UPB'SP]'CHAR'+SP;
14     (NAME'OF'Z[I])[J]+[1;'UPB'SP]'CHAR'+SP
15   'END';
16   'CLEAR'WV;'CLEAR'XV;'CLEAR'YV;'CLEAR'ZV;
17   'MODE''CONTROLVARIABLE'='STRUCT'('INT'INIT,'INT'STEP,'INT'LIM,'STRING'CV);
18   [1;L]'CONTROLVARIABLE'LOOPSTACK;
19   'INT'LOOPPOINT+0;
20   'CHAR'CH1+" ";
21   'BOOL'COND+ 'FALSE';
22
23 'C' N,B. ALL ERRORS SHOULD BE RECOGNISED BY OTHER PARTS OF THE COMPILER 'C'
24 'PROC'ERROR=('INT'N)'VOID';
25 'BEGIN'
26   PRINT(("ERROR TYPE ",N));
27   FREE
28 'END';
29
30 'PROC'SKIPSPACES=('REF''CHAR'CH)'VOID';

```

```

31     'BEGIN'
32     CH←" ", 'WHILE' CH=" " 'DO' GET(INPUT, CH)
33     'END';
34
35 'C' OPERATOR E RETURNS TRUE IF LISTS A IS EMPTY                                'C'
36     'PRIORITY' E=9;
37     'OP' E=( 'LISTS' A) 'BOOL';
38     'BEGIN'
39         'BOOL' B←'FALSE';
40         'IF' (NAME' OF' A)[1]=SP 'THEN' B←'TRUE' 'FI';
41         B
42     'END';
43
44 'C' OPERATOR @ RETURNS TRUE IF ANY MEMBER OF LIST A IS ALSO A MEMBER OF B      'C'
45     'PRIORITY' @=3;
46     'OP' @=( 'LISTS' A, B) 'BOOL';
47     'BEGIN'
48         'BOOL' MATCH←'FALSE';
49         'FOR' I 'TO' M 'WHILE' 'NOT' MATCH 'DO' 'FOR' J 'TO' M 'WHILE' 'NOT' MATCH 'DO'
50             'IF' ((NAME' OF' A)[I]=(NAME' OF' B)[J]) 'AND' ((NAME' OF' A)[I]#SP) 'THEN'
51                 MATCH←'TRUE'
52             'FI';
53         MATCH
54     'END';
55
56 'C' OPERATOR X JOINS LISTS A AND B TO FORM A NEW LIST                          'C'
57     'PRIORITY' X=3;
58     'OP' X=( 'LISTS' A, B) 'LISTS';
59     'BEGIN'
60         'LISTS' AB;

```



```

61      'INT' TEMP←0;
62      'FOR' I 'TO' M 'WHILE' TEMP≠0 'DO'
63      'BEGIN'
64          (NAME 'OF' AB)[I]←(NAME 'OF' A)[I];
65          'IF' (NAME 'OF' A)[I]=SP 'THEN' TEMP←I 'FI'
66      'END';
67      (TEMP≠0|TEMP←M+1; PRINT("ARRAY FULL"));
68      'FOR' I 'FROM' TEMP 'TO' M 'DO'
69          (NAME 'OF' AB)[I]←(NAME 'OF' B)[I-TEMP+1];
70      AB
71      'END';
72
73      'C' PRINT OUT THE RELATIONSHIP BETWEEN TWO STANZAS      'C'
74      'PROC' PRINTREL=( 'INT' I, J, K) 'VOID';
75      'BEGIN'
76      OUTF( STANDOUT, $L "STANZAS" <2>, "AND" <2> "ARE ", C("CONSECUTIVE",
77          "CONSERVATIVE", "PREREQUISITE", "CONTEMPORARY"), (I, J, K)); PRINT(NEWLINE)
78      'END';
79
80      'C' RELATIONSHIP BETWEEN TWO AS-STANZAS      'C'
81      'PROC' ASSTANZA=( 'INT' I) 'VOID';
82      'BEGIN'
83          XYZ[I]←X[I]XY[I]XZ[I];
84          WY[I]←W[I]XY[I];
85          XYZ[I+1]←X[I+1]XY[I+1]XZ[I+1];
86          WY[I+1]←W[I+1]XY[I+1];
87          'IF' XYZ[I]@WY[I+1] 'THEN' PRINTREL(I, I+1, 1)
88          'ELSE' XYZ[I]@XYZ[I+1] 'THEN' PRINTREL(I, I+1, 2)
89          'ELSE' WY[I]@XYZ[I+1] 'THEN' PRINTREL(I, I+1, 3)
90          'ELSE' PRINTREL(I, I+1, 4)

```

```

91         'FI'
92     'END';
93
94 'C' RELATIONSHIP BETWEEN AN IF-STANZA AND ANOTHER STANZA 'C'
95     'PROC' PIFSTANZA=( 'INT' I1, 'BOOL' OT) 'VOID';
96     'BEGIN'
97         'INT' I+I1;
98         'INT' K+I+1;
99         PRINT((NEWLINE,"PIF STANZA",NEWLINE));
100        (OT||I'MINUS'1||I'MINUS'2);
101        WY[I+1]+W[I+1]XY[I+1];
102        XYZ[I+1]+X[I+1]XY[I+1]XZ[I+1];
103        WY[I+2]+W[I+2]XY[I+2];
104        XYZ[I+2]+X[I+2]XY[I+2]XZ[I+2];
105        'IF' 'NOT' OT 'THEN'
106            WY[I+3]+W[I+3]XY[I+3];
107            XYZ[I+3]+X[I+3]XY[I+3]XZ[I+3]
108        'FI';
109        PRINT((NEWLINE,"RELATIONSHIP IF THE CONDITION IS TRUE",NEWLINE));
110        'IF' XYZ[I+1]@WY[K] 'THEN' PRINTREL(I+1,K,1)
111        'ELSF' XYZ[I+1]@XYZ[K] 'THEN' PRINTREL(I+1,K,2)
112        'ELSF' (W[I]XWY[I+1])@XYZ[K] 'THEN' PRINTREL(I+1,K,3)
113        'ELSE' PRINTREL(I+1,K,4)
114        'FI';
115        'IF' 'NOT' OT 'THEN'
116            'IF' XYZ[I+2]@WY[K] 'THEN' PRINTREL(I+2,K,1)
117            'ELSF' XYZ[I+2]@XYZ[K] 'THEN' PRINTREL(I+2,K,2)
118            'ELSF' (W[I]XWY[I+2])@XYZ[K] 'THEN' PRINTREL(I+2,K,3)
119            'ELSE' PRINTREL(I+2,K,4)
120        'FI'

```

```

121         'FI'
122     'END';
123
124     'C' RELATIONSHIP BETWEEN A STANZA AND AN IF-STANZA           'C'
125     'PROC' AIFSTANZA=( 'INT' I1, 'BOOL' OT) 'VOID';
126     'BEGIN'
127         'INT' I+1;
128         PRINT((NEWLINE, "AIF STANZA", NEWLINE));
129         (OT I I'MINUS'2 I I'MINUS'3);
130         WY[I]+W[I]XY[I];
131         XYZ[I]+X[I]XY[I]XZ[I];
132         WY[I+2]+W[I+2]Y[I+2];
133         XYZ[I+2]+X[I+2]Y[I+2]XZ[I+2];
134         'IF' 'NOT' OT 'THEN'
135         XYZ[I+3]+X[I+3]XY[I+3]XZ[I+3];
136         WY[I+3]+W[I+3]Y[I+3]
137         'FI';
138         'IF' XYZ[I]@W[I+1] 'THEN'
139         PRINT((NEWLINE, "STANZA ", I, " MUST BE COMPLETED BEFORE THE A IF ",
140             "STANZA IS STARTED", NEWLINE));
141         'ELSE'
142     'C' ESTABLISH THE RELATIONSHIP FOR THE TRUE AND FALSE PARTS           'C'
143         PRINT((NEWLINE, "RELATIONSHIP IF THE CONDITION IS TRUE", NEWLINE));
144         'IF' XYZ[I]@WY[I+2] 'THEN' PRINTREL(I, I+2, 1)
145         'ELSE' XYZ[I]@XYZ[I+2] 'THEN' PRINTREL(I, I+2, 2)
146         'ELSE' WY[I]@XYZ[I+2] 'THEN' PRINTREL(I, I+2, 3)
147         'ELSE' PRINTREL(I, I+2, 4)
148         'FI';
149         'IF' 'NOT' OT 'THEN'
150         PRINT((NEWLINE, "RELATIONSHIPS FOR THE PATH TAKEN IF THE ",

```

```

151         "CONDITION IS FALSE",NEWLINE));
152         'IF'XYZ[I]@WY[I+3]'THEN'PRINTREL(I,I+3,1)
153         'ELSF'XYZ[I]@XYZ[I+3]'THEN'PRINTREL(I,I+3,2)
154         'ELSF'WY[I]@XYZ[I+3]'THEN'PRINTREL(I,I+3,3)
155         'ELSE'PRINTREL(I,I+3,4)
156         'FI'
157     'FI'
158 'END'
159
160 'C' RELATIONSHIP BETWEEN TWO ADJACENT IF-STANZAS 'C'
161 'PROC'ADJIFSTANZA=( 'INT'I1, 'BOOL'OT, 'BOOL'LOT) 'VOID';
162 'BEGIN'
163     'INT'I+1, K1, K2;
164     PRINT((NEWLINE, "TWO ADJACENT IF STANZAS", NEWLINE));
165     (LOT'I' MINUS'1 | I' MINUS'2);
166     (OT'I' MINUS'2 | I' MINUS'3);
167     WY[I+1]+W[I+1]XY[I+1];
168     XYZ[I+1]+X[I+1]XY[I+1]XZ[I+1];
169     'IF'LOT'THEN'K1+I+2
170     'ELSE'
171         K1+I+3;
172         WY[I+2]+W[I+2]XY[I+2];
173         XYZ[I+2]+X[I+2]XY[I+2]XZ[I+2]
174     'FI'
175     WY[K1+1]+W[K1+1]XY[K1+1];
176     XYZ[K1+1]+X[K1+1]XY[K1+1]XY[K1+1]XZ[K1+1];
177     'IF'OT'THEN'K2+K1+1
178     'ELSE'
179         K2+K1+2;
180

```

```

181      WY[K1+2]+W[K1+2]*Y[K1+2];
182      XYZ[K1+2]+X[K1+2]*Y[K1+2]*Z[K1+2]
183  'FI';
184  'IF'XYZ[I+1]@W[K1]'THEN'
185      PRINT((NEWLINE,"SECOND IF STANZA IS DEPENDENT ON THE TRUE PART ",
186            "OF THE FIRST IF STANZA",NEWLINE))
187  'ELSE'
188      PRINT((NEWLINE,"RELATIONSHIP FOR THE TRUE PART OF THE FIRST IF ",
189            "STANZA & THE TRUE PART OF THE SECOND IF STANZA",NEWLINE));
190      'IF'XYZ[I+1]@WY[K1+1]'THEN'PRINTREL(I+1,K1+1,1)
191      'ELSE'XYZ[I+1]@XYZ[K1+1]'THEN'PRINTREL(I+1,K1+1,2)
192      'ELSE'(W[I]*WY[I+1])@XYZ[K1+1]'THEN'PRINTREL(I+1,K1+1,3)
193      'ELSE'PRINTREL(I+1,K1+1,4)
194      'FI';
195      'IF'NOT'OT'THEN'
196          PRINT((NEWLINE,"RELATIONSHIP FOR THE TRUE PART ",
197                "OF THE FIRST IF STANZA & THE FALSE PART OF THE",
198                " SECOND IF STANZA",NEWLINE));
199          'IF'XYZ[I+1]@WY[K1+2]'THEN'PRINTREL(I+1,K1+2,1)
200          'ELSE'XYZ[I+1]@XYZ[K1+2]'THEN'PRINTREL(I+1,K1+2,2)
201          'ELSE'(W[I]*WY[I+1])@XYZ[K1+2]'THEN'PRINTREL(I+1,K1+2,3)
202          'ELSE'PRINTREL(I+1,K1+2,4)
203          'FI'
204      'FI'
205  'FI';
206  'IF'NOT'LOT'THEN'
207      'IF'XYZ[I+2]@W[K1]'THEN'
208          PRINT((NEWLINE,"SECOND IF STANZA IS DEPENDENT ON THE FALSE",
209                " PART OF THE FIRST IF STANZA",NEWLINE))
210      'ELSE'

```

```

211 PRINT((NEWLINE,"RELATIONSHIP FOR THE FALSE PART OF THE ",
212 "FIRST IF STANZA & THE TRUE PART OF THE SECOND IF ",
213 "STANZA",NEWLINE));
214 'IF'XYZ[I+2]@WY[K1+1]'THEN'PRINTREL(I+2,K1+1,1)
215 'ELSF'XYZ[I+2]@XYZ[K1+1]'THEN'PRINTREL(I+2,K1+1,2)
216 'ELSF'(W[I]XWY[I+2])@XYZ[K1+1]'THEN'PRINTREL(I+2,K1+1,3)
217 'ELSE'PRINTREL(I+2,K1+1,4)
218 'FI';
219 'IF'NOT'OT'THEN'
220 PRINT((NEWLINE,"RELATIONSHIP FOR THE FALSE PART OF ",
221 "THE FIRST IF STANZA & THE FALSE PART OF THE ",
222 "SECOND IF STANZA",NEWLINE));
223 'IF'XYZ[I+2]@WY[K1+2]'THEN'PRINTREL(I+2,K1+2,1)
224 'ELSF'XYZ[I+2]@XYZ[K1+2]'THEN'PRINTREL(I+2,K1+2,2)
225 'ELSF'(W[I]XWY[I+2])@XYZ[K1+2]'THEN'PRINTREL(I+2,K1+2,3
226 )
227 'ELSE'PRINTREL(I+2,K1+2,4)
228 'FI'
229 'FI'
230 'FI'
231 'FI'
232 'END';
233
234 'C' READS IN THE OUTPUT FROM THE ANALYSER
235 'PROC'LREAD=('REF',LISTS'A,'REF'[?]'CHAR'AV)'VOID';
236 'BEGIN'
237 'CHAR'CH+" ";SKIPSPACES(CH);
238 'IF'CH#" ";THEN'BACKSPACE(INPUT);
239 'FOR'J'TO'M'WHILE'CH#" ";DO'
240 'BEGIN'

```

```

241      'INT' TEMP+1, TEMP1+0;
242      SKIPSPACES(CH); BACKSPACE(INPUT);
243      (NAME'OF'A)[J]+('FOR'K'TO'N'WHILE'(GET(INPUT,AV[J,K]); CH+AV[J,K];
244      (CH#"["TEMP1+K));
245      CH#"."AND'CH#""); 'DO' TEMP+K; [1,TEMP]'CHAR'
246  'C' IF THIS IS AN ARRAY ELEMENT PUT THE INDEX AT THE FRONT OF NAME      'C'
247      +('IF' TEMP1>1'THEN'
248      [1,TEMP1-1]'CHAR' TEM+AV[J,1,TEMP1-1];
249      AV[J,1,TEMP-TEMP1+1]+AV[J,TEMP1,TEMP];
250      AV[J,TEMP-TEMP1+2,TEMP]+TEM
251      'FI';
252      AV[J,1,TEMP]); 'FOR'J1'FROM' TEMP+1'TO'N'DO'AV[J,J1]+# "
253      'END'
254      'FI'
255  'END';
256
257  'C' PRINTS STANZAS      'C'
258      'PROC' LPRINT=( 'LISTS'A)'VOID';
259      'BEGIN'
260          PRINT(NEWLINE);
261          'FOR'J'TO'M'DO'PRINT(((NAME'OF'A)[J],# "));
262          PRINT(NEWLINE)
263      'END';
264
265  'C' FORMS W,X,Y+Z SETS      'C'
266      'PROC' BREAD=( 'INT'I)'VOID';
267      'BEGIN'
268          (CH1="W" | LREAD(W[I], WV[I]); SKIPSPACES(CH1));
269          (CH1="X" | LREAD(X[I], XV[I]); SKIPSPACES(CH1));
270          (CH1="Y" | LREAD(Y[I], YV[I]); SKIPSPACES(CH1));

```

```

271      (CH1="Z" | LREAD(Z[I], ZV[I]); SKIPSPACES(CH1));
272      OUTF(STANDOUT, $L"STANZA"<2>L"W"$S, I); LPRINT(W[I]);
273      PRINT("X"); LPRINT(X[I]); PRINT("Y"); LPRINT(Y[I]);
274      PRINT("Z"); LPRINT(Z[I])
275      'END';
276
277      'C' THIS RETURNS NUMBER = TO THE POSITION OF CV AS AN INDEX TO AV      'C'
278      'PROC' POSITION=([]'CHAR'AV, []'CHAR'CV)'INT';
279      'BEGIN'
280          'INT' NO+1;
281          'INT' T+2;
282          'INT' UPBD+'UPB'CV;
283          'BOOL' B+'TRUE';
284          'TO' 'N' 'WHILE' 'B' 'AND' 'T<'N' 'DO'
285              'IF' 'AV[T, T+UPBD-1]'#CV' THEN'
286                  'TO' 'N' 'WHILE' '(AV[T]'#", "'AND'AV[T]'#")' 'DO' T+T+1;
287                  'IF' 'AV[T]'=#", "' THEN 'NO' PLUS'1; T+T+1
288                  'ELSE' B+'FALSE'
289                  'FI'
290          'ELSE' B+'FALSE'
291          'FI';
292      NO
293      'END';
294
295      'C' THIS GIVES THE CONSTANT ASSOCIATED WITH THE NO.TH SUB+SCRIPT OF AV      'C'
296      'PROC' GIVECON=([]'CHAR'AV, 'INT'NO)'INT';
297      'BEGIN'
298          'INT' N1+2;
299          'INT' CONST+0, T, T1;
300          'TO' 'NO-1' 'DO' 'WHILE' 'AV[N1]'#", "' 'DO' N1+N1+1;

```



```

301      (AV[N1]="", "IN1+N1+1);
302      'FOR' J1 'FROM' N1 'TO' N1 'WHILE' ('CHAR' C=AV[J1]; C#+" 'AND' C#="-" 'AND' C#", "
303          'AND' C#"") 'DO' T+J1;
304      'IF' T<N 'THEF' AV[T+1]#"", " 'AND' AV[T+1]#"") 'THEN'
305          'FOR' J1 'FROM' N1+1 'TO' N1 'WHILE' AV[J1]#"") 'AND' AV[J1]#"", " 'DO' T1+J1;
306          'FOR' J1 'FROM' T+2 'TO' T1 'DO'
307              ('CHAR' C+AV[J1];
308                  'INT' T2+'ABS' C;
309                  'IF' T2<10 'THEN' CONST+CONST*10+T2
310                  'ELSE' ERROR(11)
311                  'FI')
312      'FI')
313      (T<N) (AV[T+1]="-" | -CONST|CONST) 10)
314  'END';
315
316  'C' COMPARE NM WITH THE NAME OF AV 'C'
317  'PROC' COMPARE=(['CHAR' NM, ['CHAR' AV, 'REF' | 'INT' DF, 'INT' CON1, 'INT' POS1) 'C'
318                                          'VOID';
319  'BEGIN'
320      'INT' CON2, T, T1;
321      'IF' ('FOR' J1 'TO' N1 'WHILE' AV[J1]#"") 'DO' T+J1;
322          'FOR' J1 'TO' N1 'WHILE' AV[J1]#" " 'DO' T1+J1;
323              T+T1;
324              (T<N) NM=(['1; T1-T] 'CHAR' +AV[T+1; T1]) | 'FALSE')
325          'THEN' CON2+GIVECON(AV, POS1); DF+'ABS' (CON1-CON2)
326          'FI'
327  'END';
328
329  'C' NOINDEX RETURNS TRUE IF CV DOES NOT APPEAR IN AV 'C'
330  'PROC' NOINDEX=(['CHAR' AV, ['CHAR' CV, 'REF' | 'BOOL' B1) 'BOOL';

```

```

331     'BEGIN'
332         'BOOL'B+'TRUE';
333         'INT'UPBD+'UPB'CV;
334         'INT'T;
335         B1+'TRUE';
336         'FOR'J'FROM' 'LWB'AV'TO' 'UPB'AV'WHILE'B'DO'
337             'IF'AV[J,1]#"["'THEN'
338                 T+2;
339                 'WHILE'T+UPBD<N'DO'
340                     'IF'AV[J,T;T+UPBD-1]=CV'THEN'B+'FALSE';B1+'FALSE';T+N
341                     'ELSE'T+T+1
342                     'FI'
343             'ELSE'AV[J,1;UPBD]=CV'AND'AV[J,UPBD+1]#" "'THEN'B+'FALSE'
344             'FI';
345     B
346     'END';

```

'C' INDEX RETURNS TRUE IF ALL MEMBERS OF A ARE INDEXED BY THE CONTROL VAR  
OR A CONSTANT DIFFERENCE OF IT

```

349     'PROC'INDEX=( 'LISTS'A,[,]'CHAR'AV;[,]'CHAR'CV)'BOOL';
350     'BEGIN'
351         'BOOL'B+'FALSE';
352         'BOOL'B1+'TRUE';
353         'FOR'J'FROM' 'LWB'AV'TO' 'UPB'AV'WHILE'AV[J,1]#" "'AND'B1'DO'
354             'IF'AV[J,1]#"["'THEN'
355                 'INT'UPBD+'UPB'CV;
356                 'INT'T+2;
357                 'WHILE'T+UPBD<N'DO'
358                     'IF'AV[J,T;T+UPBD-1]=CV'THEN'B+'TRUE';T+N
359                     'ELSE'T+T+1
360

```

```

361                                     'FI');
362                                     B1+B
363                                     'ELSE'
364                                     B1+'FALSE'
365                                     'FI');
366                                     B1
367 'END';
368
369 'C' RETURNS TRUE IF W AND X DO NOT BOTH USE THE SAME ARRAY 'C'
370 'PROC'WANDX=('LISTS'X1,[,]'CHAR'XV1,[,]'CHAR'WV1,[,]'CHAR'C)'BOOL';
371 'BEGIN'
372 'BOOL'B+INDEX(X1,XV1,C);
373 'IF'B'THEN'
374 'FOR'J1'TO'M'WHILE'WV1[J1,1]#" "AND'B'DO'
375 'FOR'J0'TO'M'WHILE'XV1[J0,1]#" "AND'B'DO'
376 'BEGIN'
377 'STRING'S1+
378 ('INT'T,T1;
379 'FOR'J2'TO'N'WHILE'WV1[J1,J2]#"#"DO'T+J2;
380 'FOR'J2'TO'N'WHILE'WV1[J1,J2]#" "DO'T1+J2;
381 'IF'T>=T1'THEN'[1;1]'CHAR'+#" "
382 'ELSE'[1;T1-T+1]'CHAR'+WV1[J1/T+2;T1]
383 'FI');
384 'IF'S1#" "
385 'THEF'
386 ('STRING'S2+
387 ('INT'T,T1;
388 'FOR'J2'TO'N'WHILE'XV1[J0,J2]#"#"DO'T+J2;
389 'FOR'J2'TO'N'WHILE'XV1[J0,J2]#" "DO'T1+J2;
390 [1;T1-T-1]'CHAR'+XV1[J0,T+2;T1]);

```

```

391             S1=S2)
392             'THEN'B←'FALSE'
393             'FI'
394         'END'
395     'FI';
396     B
397 'END';
398
399 'C' FOR EVERY NAME ESTABLISH DIFFERENCE IN USAGE TIME 'C'
400 'PROC' CONTRAST=( 'INT' I, [ , ] 'INT' POS, [ , ] 'CHAR' WU, [ , ? ] 'CHAR' XU, [ , , ] 'CHAR' YU,
401 [ , , ] 'CHAR' ZU) 'VOID';
402 'BEGIN'
403     'INT' UP←'UPB' POS;
404     [ 1, UP ] 'INT' ST, LIM, INIT, SD, CONST, DF;
405     'FOR' J TO UP 'DO'
406     'BEGIN'
407         ST[J]←STEP'OF' LOOPSTACK[LOOPPOINT+1-J];
408         LIM[J]←LIM'OF' LOOPSTACK[LOOPPOINT+1-J];
409         INIT[J]←INIT'OF' LOOPSTACK[LOOPPOINT+1-J];
410         SD[J]←-1; DF[J]←-1
411     'END';
412     'STRING' COMP;
413     'BOOL' NOTCON←'TRUE';
414     'INT' DIF←-1, DIFF←-1;
415
416 'C' THESE 2 PROCS ARE USED LOCALLY WITH GLOBAL REFS 'C'
417 'PROC' SET=( 'INT' LLL, [ , , ] 'CHAR' AV) 'VOID';
418 'BEGIN'
419     'FOR' J1 'FROM' LLL 'TO' M 'WHILE' NOTCON 'AND' AV[I, J1, 1]#"" 'DO'
420 'BEGIN'

```

```

421         'FOR' J2 'TO' UP 'DO'
422         'BEGIN'
423             COMPARE (COMP, AV[I, J1], DIF, CONST[J2], POS[J2]);
424             'IF' DIF = +1 'OR' DIF = 0 'THEN' 'SKIP'
425             'ELSE' (DIF / 'ST[J2]) * ST[J2] = DIF
426                 'AND' DIF < LIM[J2] - INIT[J2] 'THEN'
427                 SD[J2] + DIF / 'ST[J2]
428             'FI'
429             'DIF' ← -1
430         'END'
431     'END'
432 'END'
433
434 'PROC' GIVECOMP = ([, , ] 'CHAR' AV, [INT' J] [ ] 'CHAR' ;
435     'BEGIN'
436         'INT' T, T1;
437         'FOR' J0 'TO' UP 'DO' CONST[J0] + GIVECON (AV[I, J], POS[J0]);
438         'FOR' J1 'TO' N 'WHILE' AV[I, J, J1] # " ] " 'DO' T + J1;
439         'FOR' J1 'TO' N 'WHILE' AV[I, J, J1] # " " 'DO' T1 + J1;
440         [1; T1 - T - 1] 'CHAR' + AV[I, J, T + 2; T1]
441     'END'
442
443 'FOR' J 'TO' M 'WHILE' XU[I, J, 1] # " " 'AND' NOT CON 'DO'
444 'BEGIN'
445     COMP + GIVECOMP (XU, J);
446     SET (J + 1, XU);
447     SET (1, YU);
448     SET (1, ZU);
449     SET (1, WU)
450 'END'

```

```

451      'FOR' J 'TO' M 'WHILE' YU[I, J, 1] # " " 'AND' 'NOTCON' 'DO'
452      'BEGIN'
453          COMP+GIVECOMP(YU, J);
454          SET(J+1, YU);
455          SET(1, ZU);
456          SET(1, WU)
457      'END';
458      'FOR' J 'TO' M 'WHILE' ZU[I, J, 1] # " " 'AND' 'NOTCON' 'DO'
459      'BEGIN'
460          COMP+GIVECOMP(ZU, J);
461          SET(J+1, ZU);
462          SET(1, WU)
463      'END'; PRINT(NEWLINE);
464      'IF' 'NOT' 'NOTCON' 'THEN' PRINT(("EACH ITERATION MUST BE DONE ",
465          "SEQUENTIALLY"))
466      'ELSE'
467          PRINT(("ADJACENT ";NEWLINE));
468          'FOR' J 'TO' UP 'DO' (SD[J]#-1) PRINT("ALL      ") | PRINT(SD[J])
469      'FI'; PRINT(NEWLINE)
470  'END';
471
472  'C' RANK RETURNS TRUE IF THE POSITION OF CV IS THE SAME FOR ALL OF X , Y & Z ,
473  AND SETS POS TO THAT NUMBER
474  'PROC' RANK=( [, ] 'CHAR' XV1, [, ] 'CHAR' YV1, [, ] 'CHAR' ZV1, 'REF' 'INT' POS,
475      [, ] 'CHAR' CV) 'BOOL';
476  'BEGIN'
477      'BOOL' B+'FALSE';
478      'IF' XV1[1, 1] # " " 'THEN' POS+POSITION(XV1[1], CV)
479      'ELSE' YV1[1, 1] # " " 'THEN' POS+POSITION(YV1[1], CV)
480      'ELSE' POS+POSITION(ZV1[1], CV)

```

```

481         'FI'
482         'FOR' J1 'FROM' 2 'TO' M 'WHILE' Xv1[J1,1]# " " 'AND' 'NOT' B 'DO'
483             (POS#POSITION(Xv1[J1],CV)IB+'TRUE');
484         'FOR' J1 'TO' M 'WHILE' Yv1[J1,1]# " " 'AND' 'NOT' B 'DO'
485             (POS#POSITION(Yv1[J1],CV)IB+'TRUE');
486         'FOR' J1 'TO' M 'WHILE' Zv1[J1,1]# " " 'AND' 'NOT' B 'DO'
487             (POS#POSITION(Zv1[J1],CV)IB+'TRUE');
488     B
489     'END' )
490
491     'C' REMOVE FROM W ANYTHING NOT INDEXED BY CV
492     'PROC' FROMW=( 'LISTS' WA, 'REF' [,] 'CHAR' WB, [,] 'CHAR' CV) 'VOID' ;
493     'BEGIN'
494         'FOR' J 'TO' M 'WHILE' WB[J,1]# " " 'DO'
495         'IF' INDEX(WA,WB[J],CV) 'THEN'
496             'FOR' J1 'FROM' J-1 'BY' -1 'TO' 1 'WHILE' WB[J1,1]# " " 'DO'
497                 'BEGIN'
498                     WB[J1]+WB[J1+1];
499                     WB[J1+1,1]# " "
500                 'END'
501             'ELSE'
502                 WB[J,1]# " "
503             'FI'
504         'END' )
505
506     'C' DETECTS A CONDITIONAL
507     'PROC' CONDITIONAL=( 'REF' 'INT' I, 'INT' J, 'REF' 'BOOL' ONLY THEN,
508                         'REF' 'BOOL' LAST OT) 'VOID' ;
509     'BEGIN'
510         SKIPSPACES(CH1); BREAD(I);

```

'C'

'C'

```

511      (EX[I]'AND'EY[I]'AND'EZ[I]]'SKIP'ERROR(4));
512      'TO'2'WHILE'CH1#"$" 'DO'
513          (BREAD(I'PLUS'1);ONLYTHEN+'NOT'ONLYTHEN);
514      (CH1#"$" 'ERROR(3)'SKIPSPACES(CH1));
515      'IF' 'ODD'(J-1)' THEN'
516          (COND='TRUE' 'ADJIFSTANZA(I,ONLYTHEN,LASTOT)
517              'COND+'FALSE'
518              'AIFSTANZA(I,ONLYTHEN))
519      'ELSE'
520          I'PLUS'1;COND+'TRUE';LASTOT+ONLYTHEN
521      'FI'
522  'END';
523
524  'C' MOVES MAY BE USED TO CHECK ON MOVEMENT OF A CV AS AN INDEX      'C'
525      'PROC' MOVES=( 'INT'NO)'VOID';
526      'BEGIN'
527          'INT'DUM;
528          ERROR(NO)
529      'END';
530
531
532  'C' SINGLE LOOPS                                                    'C'
533      'PROC' ONCE=( 'REF' 'INT' I)'VOID';
534      'BEGIN'
535          [1;L,1;M,1;N]'CHAR'WX;
536          'INT' POS+1;      'C' THIS WILL EVENTUALLY BE SET IN AN ELSEF TEST      'C'
537  'C' TEST FOR TOTAL INDEPENDENCE 'C'
538      'IF' EY[I]'AND'EZ[I]
539          'AND' WANDX(X[I],XV[I],WV[I],CV'OF'LOOPSTACK[LOOPPOINT])
540          'THEN' PRINT((NEWLINE,"TOTALLY INDEPENDENT",NEWLINE))

```



```

541 'C' TEST FOR TOTAL DEPENDENCE
542     'ELSF' 'NOT'
543         'BEGIN'
544             'BOOL' B+ 'TRUE';
545             'FOR' I1 'TO' 3 'WHILE' B= 'TRUE' 'DO'
546                 B+ 'CASE' I1 'IN'
547                     INDEX(X[I],XV[I],CV'OF' LOOPSTACK[LOOPPOINT]),
548                     INDEX(Y[I],YV[I],CV'OF' LOOPSTACK[LOOPPOINT]),
549                     INDEX(Z[I],ZV[I],CV'OF' LOOPSTACK[LOOPPOINT])
550             'ESAC'
551         B
552     'END'
553     'THEN'
554     PRINT((NEWLINE,"TOTALLY DEPENDENT",NEWLINE))
555     'ELSF' RANK(XV[I],YV[I],ZV[I];POS,CV'OF' LOOPSTACK[LOOPPOINT])
556     'THEN' MOVES(20)
557     'ELSE'
558         WX[I]+WV[I];
559         FROMW(W[I],WX[I],CV'OF' LOOPSTACK[LOOPPOINT]);
560         CONTRAST(I,POS,WX,XV,YV,ZV)
561     'FI'
562 'END';
563
564 'C' NESTED LOOPS.....ONLY 2 DEEP HERE
565 'PROC' TWICE=( 'REF' 'INT' I) 'VOID';
566 'BEGIN'
567     'INT' POSI+1, POSO+1;
568     'BOOL' IIL+ 'FALSE';
569     'BOOL' ILID+ 'FALSE';
570     'IF' EY[I] 'AND' EZ[I] 'AND'

```

'C'

'C'

```

571          WANDX(X[I],XV[I],WV[I],CV'OF'LOOPSTACK[LOOPPOINT])'AND'
572          WANDX(X[I],XV[I],WV[I],CV'OF'LOOPSTACK[LOOPPOINT-1])
573 'THEN'PRINT((NEWLINE,"BOTH LOOPS ARE TOTALLY INDEPENDENT",NEWLINE))
574 'ELSE'
575 'IF'NOT'
576 'BEGIN'
577     'BOOL'B+'TRUE'
578     'FOR'I1'TO'3'WHILE'B'DO'
579         B+'CASE'I1'IN'
580             INDEX(X[I],XV[I],CV'OF'LOOPSTACK[LOOPPOINT]),
581             INDEX(Y[I],YV[I],CV'OF'LOOPSTACK[LOOPPOINT]),
582             INDEX(Z[I],ZV[I],CV'OF'LOOPSTACK[LOOPPOINT])
583         'ESAC'
584     B
585 'END'
586 'THEN'PRINT((NEWLINE,"THE INNER LOOP IS DEPENDENT",NEWLINE));
587     ILID+'TRUE'
588 'ELSE'IRANK(XV[I],YV[I],ZV[I],POSI,CV'OF'LOOPSTACK[LOOPPOINT])
589 'THEN'MOVES(21)
590 'ELSE'IL+'TRUE'
591 'FI'
592 'IF'ILID'THEN'LOOPPOINT'MINUS'1;ONCE(I);LOOPPOINT'PLUS'1
593 'ELSE'
594 'IF'NOT'
595 'BEGIN'
596     'BOOL'B+'TRUE'
597     'FOR'I1'TO'3'WHILE'B'DO'
598         B+'CASE'I1'IN'
599             INDEX(X[I],XV[I],CV'OF'LOOPSTACK[LOOPPOINT-1]),
600             INDEX(Y[I],YV[I],CV'OF'LOOPSTACK[LOOPPOINT-1]),

```

```

601             INDEX(Z[I],ZV[I],CV'OF'LOOPSTACK[LOOPPOINT-1])
602             'ESAC';
603             B
604             'END'
605             'THEN'(IILIONCE(I));
606             PRINT((NEWLINE,"THE OUTER LOOP IS DEPENDENT",NEWLINE))
607             'ELSE' RANK(XV[I],YV[I],ZV[I]; POSO,CV'OF'LOOPSTACK[LOOPPOINT-1])
608             'THEN' MOVES(22)
609             'ELSE'
610             FROMW(W[I],WV[I],CV'OF'LOOPSTACK[LOOPPOINT-1]);
611             CONTRAST(I,(POSI,POSO),WV,XV,YV,ZV)
612             'FI'
613             'FI'
614             'FI'
615             'END';
616
617             'C' SPOTS KEY SYMBOLS FROM ANALYSER                                'C'
618             'C' # = DO-STANZA      S = IF-STANZA                                'C'
619             'PROC' KEY=( 'REF' 'INT' I) 'VOID';
620             'BEGIN'
621             SKIPSPACES(CH1);
622             'IF' CH1="0" 'THEN'                                'C'END OF LOOP 'C'
623             SKIPSPACES(CH1);
624             (LOOPPOINT>0|LOOPPOINT'MINUS'1;I+I-1|ERROR(1))
625             'ELSE' CH1#"D" 'THEN' ERROR(2)
626             'ELSE' GET(INPUT,((INIT'OF'LOOPSTACK[LOOPPOINT'PLUS'1]),
627             (STEP'OF'LOOPSTACK[LOOPPOINT]),LIM'OF'LOOPSTACK[LOOPPOINT]));
628             CV'OF'LOOPSTACK[LOOPPOINT]+
629             'BEGIN'
630             [1;N]'CHAR'AR;'INT'TEMP+0;'CHAR'CH;SKIPSPACES(CH);

```

```

631         'FOR'K'TO'N'WHILE'CH#" 'DO'
632         (TEMP'PLUS'1;AR[TEMP]+CH;GET(INPUT,CH));
633         [1;TEMP]'CHAR'+AR[1;TEMP]
634     'END';
635 PRINT((NEWLINE,"CV      ",CV'OF'LOOPSTACK[LOOPPOINT],INIT'OF'LOOPSTACK[LOOPPOINT],
636     STEP'OF'LOOPSTACK[LOOPPOINT],LIM'OF'LOOPSTACK[LOOPPOINT],NEWLINE));
637     SKIPSPACES(CH);
638     'IF'CH1="#"'THEN'KEY(I)
639     'ELSE'CH1="$"'THEN'
640         'BEGIN'
641         'BOOL'ONLYTHEN+'FALSE',LASTOT+'FALSE';
642         'BOOL'B+'FALSE';
643         'BOOL'B1;
644         'INT'I1;
645         COND+'FALSE';
646         CONDITIONAL(I,1,ONLYTHEN,LASTOT);
647         'IF'ONLYTHEN'THEN'I1+I-2'ELSE'I1+I-3'FI';
648         B+NOINDEX(WV[I1],CV'OF'LOOPSTACK[LOOPPOINT],B1);
649         'IF'B'THEN'PRINT(("FOR ANY GIVEN LOOP THE SAME ",
650             "PATH IS ALWAYS TAKEN "));
651         'IF'LOOPPOINT=1'THEN'
652             ONCE(I1'PLUS'1);('NOT'ONLYTHEN|ONCE(I1'PLUS'1))
653         'ELSE'LOOPPOINT=2'THEN'
654             TWICE(I1'PLUS'1);('NOT'ONLYTHEN|TWICE(I1'PLUS'1))
655         'ELSE'PRINT(("ONLY DOUBLE LOOPS",NEWLINE))
656         'FI'
657         'ELSE'B1'THEN'PRINT(" PATH DECIDABLE ");
658 'C' SPECIAL ROUTINES NEEDED TO SPOT SWITCH-OVER 'C'
659     'IF'LOOPPOINT=1'THEN'
660         ONCE(I1'PLUS'1);('NOT'ONLYTHEN|ONCE(I1'PLUS'1))

```

```

661         'ELSE' LOOPPOINT=2'THEN'
662             TWICE(I+'PLUS'1);('NOT'ONLYTHENITWICE(I+'PLUS'1))
663         'ELSE' PRINT(("ONLY DOUBLE LOOPS",NEWLINE))
664         'FI'
665         'ELSE' PRINT("PATH NOT KNOWN")
666         'FI';PRINT(NEWLINE)
667     'END'
668     'ELSE' BREAD(I);
669         'IF' LOOPPOINT=1'THEN'ONCE(I)
670         'ELSE' LOOPPOINT=2'THEN'TWICE(I)
671         'ELSE'
672             PRINT((NEWLINE,"ONLY DOUBLE LOOPS"?NEWLINE))
673         'FI'
674     'FI'
675 'END';
676
677 SKIPSPACES(CH1);
678 'BEGIN'
679     'INT' I+0;
680     'TO' L'WHILE'CH1#"*" 'DO'
681     'BEGIN'
682     I'PLUS'1;
683     'IF' CH1="#" 'THEN' KEY(I)
684     'ELSE'
685     'BOOL' ONLYTHEN+ 'FALSE', LASTOT+ 'FALSE';
686     'FOR' J'TO'2'WHILE'CH1#"*" 'AND' CH1#"#" 'AND' CH1#"@" 'DO'
687     'BEGIN'
688     'IF' CH1="#" 'THEN'
689     CONDITIONAL(I,J,ONLYTHEN, LASTOT)
690

```

```
691         'ELSE'
692         'BEGIN'
693             BREAD(I);
694             'IF' 'ODD' (J-1) 'THEN'
695                 'IF' 'COND=' 'FALSE' 'THEN' 'ASSTANZA(I-1)
696                 'ELSE' 'PIFSTANZA(I-1, ONLYTHEN);
697                 COND+' 'FALSE'
698             'FI'
699         'ELSE'
700             I'PLUS'1
701         'FI'
702     'END'
703 'FI'
704 'END'
705 'FI'
706     'END'
707 'END' 'BACKSPACE(INPUT)
708 'END'
709 'FINISH'
710 ****
```

APPENDIX 4

SAMPLE PROGRAM

```

1      'BEGIN'
2      'PROC'(C11,C12):
3      'BEGIN'
4          C11←TWO;
5          TWO←C11+C12;
6          C12←ONE+C12;
7      'END';
8
9      'FOR'K←2'STEP'2'UNTIL'14'DO'
Stanza 1 [ 10      'BEGIN'
11          A[K]←ONE+TWO;
12          A[Y+1]←THREE;
13      'END';
14
15      B1←A1+C1;
Stanza 2 [ 16      A1←C1-D1;
17      D1←E1*C1;
18      J1←J1;
19      F1←G1;
20
21      'IF'AC=BC'THEN'F1*JK;'ELSE'EF+KJ;
Stanzas 3,4 and 5 [ 22
23      C((A1,B1));
Stanza 6 [ 24
25      'IF'ONE=A0'THEN'TWO←A[2];'ELSE'A[2]←TWO;
Stanzas 7,8 and 9 [ 26
27      A1←B1+EF;
Stanza 10 [ 28
29      'FOR'I←1'STEP'2'UNTIL'11'DO'B[I]←B[I]+4;
Stanza 11 [ 30
31      'END';
32      ****

```



		#D	+2	+14	
		2			
Stanza 1	[	3	W ONE	.TWO	.THREE ;
		4	X A[K]	.A[K+1]	;
		5	Y	;	
		6	Z	;	
		7	#0		
		8			
Stanza 2	[	9	W C1	.E1	.JK .KJ ;
		10	X B1	.J1	.K1 ;
		11	Y A1	.D1	;
		12	Z	;	
		13			
Stanza 3	[	14	\$		
		15	W AC	.BC	;
		16	X	;	
		17	Y	;	
		18	Z	;	
		19			
Stanza 4	[	20	W JK	;	
		21	X EF	;	
		22	Y	;	
		23	Z	;	
		24			
Stanza 5	[	25	W KJ	;	
		26	X EF	;	
		27	Y	;	
		28	Z	;	
		29			
		30	\$		
		31			
Stanza 6	[	32	W ONE	;	
		33	X	;	
		34	Y TWO	.A1	.B1 ;
		35	Z	;	
		36			
		37	\$		
Stanza 7	[	38	W ONE	.AC	;
		39	X	;	
		40	Y	;	
		41	Z	;	
		42			
Stanza 8	[	43	W A[2]	;	
		44	X TWO	;	
		45	Y	;	
		46	Z	;	
		47			
Stanza 9	[	48	W TWO	;	
		49	X A[2]	;	
		50	Y	;	
		51	Z	;	
		52			
		53	\$		
		54			
Stanza 10	[	55	W B1	.EF	;
		56	X A1	;	
		57	Y	;	
		58	Z	;	
		59			
		60	#D	+1	+2 +11 I
		61			
Stanza 11	[	62	W B[1+4]	;	
		63	X B[1]	;	
		64	Y	;	
		65	Z	;	
		66	#0		
		67			
		68	****		

- Stanza 1 - Do-stanza  
Each iteration is totally independent (i.e. they are all *contemporary*)
- Stanzas 2,3 and 4 - If-stanza
- Stanza 5 - As-stanza  
If stanza 2 is true or false then the If-stanza and the As-stanza are *contemporary*)
- Stanza 6 - Procedure Call
- Stanzas 7,8 and 9 - If-stanza  
If stanza 7 is true then the Procedure Call and If-stanza are *conservative*, otherwise the Procedure Call and If-stanza are consecutive.
- Stanza 10 - As-stanza  
There is nothing available to compare with the stanza,
- Stanza 11 - Do-stanza  
Every two iterations are adjacents (i.e. pairs of iterations are *contemporary* but each pair must be executed in a *consecutive manner*)

