Specification and analysis of the sequencing of computing operations in high level languages: with particular reference to parallel processing

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/14456](https://dspace.lboro.ac.uk/2134/14456)

Publisher: © Fatemeh Abdollahzadeh

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

![Creative Commons License](https://creativecommons.org/licenses/by-nc-nd/2.5/)

**Attribution-NonCommercial-NoDerivs 2.5**

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Noncommercial.** You may not use this work for commercial purposes.

**No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

For the full text of this licence, please go to: [http://creativecommons.org/licenses/by-nc-nd/2.5/](http://creativecommons.org/licenses/by-nc-nd/2.5/)
SPECIFICATION AND ANALYSIS OF THE SEQUENCING
OF COMPUTING OPERATIONS IN HIGH LEVEL LANGUAGES,
WITH PARTICULAR REFERENCE TO PARALLEL PROCESSING

BY

FATEMEH ABDOLLAHZADEH, M.Sc., M.A.

A Doctoral Thesis
Submitted in Partial Fulfilment of the Requirements
for the Award of Doctor of Philosophy
of Loughborough University of Technology
June, 1981.

Supervisors: PROFESSOR D.J. EVANS, Ph.D., D.Sc.,
DR. D.J. COOKE, Ph.D.,
Department of Computer Studies

© Fatemeh Abdollahzadeh, 1981.
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.

F. ABDOLLAHZADEH
ACKNOWLEDGEMENTS

I would like to express my appreciation and thanks to my advisors, Professor D.J. Evans and Dr. D.J. Cooke, for providing motivation and assistance but especially to Dr. D.J. Cooke for his enlightening criticism and patience throughout the work.

Also, I would like to express my thanks to Miss J.M. Briers for typing this thesis.
## CONTENTS

### Chapter 1: INTRODUCTION

1.1 General Description .................................................. 1
1.2 Classification of Computer Structures and System Organization .......................................... 2
1.3 Level and Types of Parallelism .................................... 4
1.4 Existing Techniques for Recognizing and Detecting Parallelism ........................................... 9
1.5 The New Method to Detect Parallelism ......................... 13

### Chapter 2: A NEW STRATEGY FOR THE CONSTRUCTION OF BINARY TREES FOR PARALLEL PROCESSING

2.1 Introduction .......................................................... 15
2.2 Definitions and Previous Algorithms ......................... 16
  2.2.1 Previous Algorithms ........................................... 17
    2.2.1.1 Williams's Algorithm .................................. 20
    2.2.1.2 Ashoke's Algorithm .................................... 25
    2.2.1.3 Other Methods for Recognizing Parallelism Within Arithmetic Expressions .................. 38
2.3 Tree Height Reduction by a New Strategy ..................... 41
  2.3.1 The Strategy of the Method .................................. 41
  2.3.2 Definitions and Concepts .................................... 42
2.4 Formation of a Balanced Binary Tree .......................... 51
  2.4.1 Algorithm A .................................................. 52
  2.4.2 Algorithm B .................................................. 55
  2.4.3 The Basic Algorithm ....................................... 57
2.5 Comparison of the Algorithms ................................... 58
2.6 On the Evaluation of a Polynomial ............................. 63
2.7 Conclusions .......................................................... 66

### Chapter 3: A SURVEY OF PREVIOUS METHODS OF DETERMINING PARALLELISM IN A PROGRAM

3.1 Interdependencies Between Parts of a Program .......... 67
3.2 Existing Techniques to Exploit and Detect Parallelism ......................................................... 68
  3.2.1 Methods Based on Graph Theory ......................... 69
  3.2.2 Methods Based on the Structure of the Program ......................................................... 72
### Chapter 4: A NEW METHOD FOR DETECTING POTENTIAL PARALLELISM IN A PROGRAM

- 4.1 Introduction ............................................ 75
- 4.2 Some Basic Definitions of Language Theory ........ 76
- 4.3 Grammars .................................................. 77
- 4.4 Restricted Grammars ................................. 79
- 4.5 Ambiguity .................................................. 85
- 4.6 Operator Precedence Relations from Associativity and Precedence ............... 86

### Chapter 5: A NEW GRAMMAR FOR RECOGNIZING PARALLELISM IN ARITHMETIC EXPRESSIONS

- 5.1 Motivation for the Grammar ........................... 95
- 5.2 The New Grammar ...................................... 100
- 5.3 An Extension of the Grammar ....................... 101
- 5.4 Further Extension of the Grammar .................. 115

### Chapter 6: PARSING ALGORITHMS FOR CONTEXT SENSITIVE GRAMMARS

- 6.1 General Parsing Methods ............................ 117
- 6.2 A Survey of Some Methods for the Parsing of Content Sensitive Languages .......... 118
  - 6.2.1 Woods's Algorithm ............................... 118
  - 6.2.2 The Kuno Algorithm ............................ 125
  - 6.2.3 Tanaka and Fu's Algorithm ...................... 128
- 6.3 A Left to Right Parsing Algorithm (L.R.) .......... 130
- 6.4 A Left-most (Canonical) Parsing Algorithm ........ 134
- 6.5 A Combined Algorithm for Parsing .................. 137
  - 6.5.1 Introduction ..................................... 137
  - 6.5.2 The Algorithm ................................... 138
- 6.6 Non-Context-Free Grammars Generating Context-Free Languages ....................... 143

### Chapter 7: SOME INCONSISTENCIES IN DIFFERENT FORTRAN COMPILERS

- 7.1 Introduction ............................................ 147
- 7.2 Some FORTRAN Compilers ........................... 149
- 7.3 Inconsistencies ......................................... 151
  - 7.3.1 The Exact Sequence Occurring During the Evaluation of a DO Loop .......... 151
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.1.1</td>
<td>Comparison of DO Loop Elaboration of Three Compilers</td>
<td>163</td>
</tr>
<tr>
<td>7.3.2</td>
<td>The Value of the Control Variable After the Execution of a DO Loop</td>
<td>165</td>
</tr>
<tr>
<td>7.3.3</td>
<td>The Evaluation of the DO-Loop Compiler in a Certain Language</td>
<td>165</td>
</tr>
<tr>
<td>7.3.4</td>
<td>An Unclear Point About Assignment Statements</td>
<td>170</td>
</tr>
<tr>
<td>7.3.5</td>
<td>&quot;Call by Name&quot; (Substitution) in FORTRAN</td>
<td>172</td>
</tr>
<tr>
<td>7.4</td>
<td>The Effect of Inconsistencies in Parallel Execution</td>
<td>176</td>
</tr>
</tbody>
</table>

Chapter 8: SUMMARY AND CONCLUSION

8.1 Summary and Conclusion ............................................. 179

REFERENCES ........................................................................... 184

APPENDIX I: TWO ALGORITHMS FOR CONSTRUCTING A BALANCED BINARY TREE ... 192

APPENDIX II: SOME FORTRAN PROGRAMS AND THEIR RESULTS ................... 196
CHAPTER 1

INTRODUCTION
1.1 GENERAL DESCRIPTION

During the past few years, the advancement of hardware technologies has initiated new lines of thought amongst computer scientists.

With the ability to perform faster computation and the availability of high speed hardware devices, the motivation for parallel activities in both the "system organization" and "system software" areas has followed.

The term "parallel processing" is used in the literature to describe many different aspects: sometimes it indicates that arithmetic operations act on all the bits within a word rather than on one bit at a time, sometimes it refers to multiple processors, to array machines and often to multi-programmed machines. In this thesis the term "parallel processing" is used to indicate the execution of several 'tasks' at the same time on different processors or processing units of a machine. A task is a part of a program and can vary from a micro-instruction to a whole program.

For sequential computers, a great deal of work is done at compile time to decrease the execution time of a program. Code optimization is the term generally applied to attempts to make object programs more 'efficient', e.g. faster running. There are many possibilities for code optimization for sequential programming languages, and various techniques can be employed at different phases of the compilation process (see Aho and Ullman, 1972). However, if we have a machine capable of performing parallel operation, then we might attempt to arrange the order of the execution to create as many simultaneous parallel computations as possible. For example, suppose that we have a four register machine in which four operators can be simultaneously executed, then the expressions:

\[ a_1 + a_2 + a_3 + a_4 + \ldots + a_8 \]

can be done as shown in Figure 1.1. In the first step we would load \( a_1 \) into the first register, \( a_3 \) into the second, \( a_5 \) into the third and \( a_7 \) into
the fourth. In the second step, we would add $a_2$ to register 1, $a_4$ to register 2, $a_6$ to register 3, and $a_8$ to register 4. After this step, registers 1, 2, 3 and 4 would contain the quantities $a_1+a_2$, $a_3+a_4$, $a_5+a_6$ and $a_7+a_8$ respectively. At the third step, we would add register 2 to register one and register 4 to register 3. At the fourth step we would add register 2 to register 1. Thus, the elapsed time taken to execute the above expression in a machine with four processors is 4 (time units) whereas on a serial machine it is 7.

FIGURE 1.1
1.2 **CLASSIFICATION OF COMPUTER STRUCTURES AND SYSTEM ORGANIZATION**

Several different architectures for parallel computers have been designed and built. Stone (1975) describes in detail the classification of computer systems (Figure 1.2) due to Flynn (1966). These definitions are:

1. **SISD computer**
2. **SIMD computer**
3. **MISD computer**
4. **MIMD computer.**

The SISD computer (Single Instruction Stream-Single Data Stream) is the sequential computer, where, at any time, at most one instruction is in execution and affects only one item of data. Most programming languages are designed to run on this type of computer.

The SIMD computer (Single Instruction Stream-Multiple Data Stream) is a machine in which an instruction commands more than one processing element, i.e. the instruction can operate on a data vector which is supplied by a multiple data stream. One well known class of SIMD machine is the array processor (e.g. ILLIAC IV which contains an array of 64 fast, floating point processors).

Other SIMD machines include associative processors in which the data is content addressable. The architecture of associative processors is discussed by Yau and Fung (1977).

The MISD computer (Multiple Instruction Stream-Single Data Stream) is a machine in which each item of data is used simultaneously by several instructions. At the present time no examples of this type of machine are in practical use although its nearest equivalent is the line printer.

The MIMD computer (Multiple Instruction Stream-Multiple Data Stream) is a machine of several inter-connected computers each of which may carry
**FIGURE 1.2: Flynn's classification with some example.**
out different instructions on different items of data at any one time.

Multiprocessors are a subclass of MIMD multiple computer systems in which processors have common access to primary memory and the input/output channels. There is a single operating system which controls the entire complex. The concepts of sharing and interaction form the basis of multiprocessing techniques. Enslow (1977) has explained how multiprocessors are organized. He also (1980) has considered loosely-coupled systems which communicate one with another at the input-output level. The important characteristic of this type of system is that all data transfer operations between the two systems are performed as input-output operations. Another point is that one processor has no capability of directly controlling another. Also there is no direct access to primary memory. This implies that the sharing of information among components on various processors is greatly curtailed, and the control is forced to work with information that is often out-of-date or inaccurate.

These systems introduce problems of interconnection, synchronization and transmission of data and control information. Compilation on this type of system is also a problem that has not been studied to any significant extent.

"Pipelining" is a technique that has emerged as an important attribute and an economical way of speeding up computer systems. The idea is based on the fact that the execution of many machine instructions consumes several clock periods, usually using the same hardware iteratively, if such hardware is replicated serially, then a number of operations may be followed through the processors at once, instead of waiting for the whole operation to finish and then the next one to start, in other words pipelining divides a computational process into several subprocessors. Of course, the speed-up is limited by the pipeline length (and by the number of clock cycles required by individual operations).
So pipelining is a technique of decomposing a serial process into many overlapped subprocesses. As an example, consider the process of executing an instruction. Usually it involves fetching the operand and incrementing the program counter (IF), decoding (ID), executing (IE) and storing the operand (IS). The process can be divided into four subprocesses, (IF, ID, IE and IS), and executed in parallel (Figure 1.3). Thus, for a set of distinct tasks (instruction), i.e. \( p_i \)'s, where \( 2 \leq i \leq n \), when

- IF fetches (and increments the program counter) \( p_{q+3} \),
- ID decodes \( p_{q+2} \),
- IE executes \( p_{q+1} \) and
- IS stores \( p_q \), where \( 1 \leq q \leq n-3 \).

Figure 1.4 shows how \( n \) instructions (distinct tasks) pass through a fourfold pipelined processor.

The pipelining techniques can be applied to any type of Flynn's classification without affecting the definitions of any of these systems (Chen, 1975). Some "super machines" have special pipeline processing capabilities, such as CDC 7600, CRAY-1 (for further details see Kuck, 1980).

Ramamoorthy and Li (1977) have presented theoretical consideration and reviewed many problems concerning pipeline machines that operate in either sequential or vector pipeline mode.
FIGURE 1.3: Pipelining an instruction

FIGURE 1.4: Tasks executing by a fourfold pipelined processors
1.3 LEVELS AND TYPES OF PARALLELISM

In a parallel processing environment, all possible tasks can be considered as comprising of four levels: machine, instruction, block and program.

In the machine level, the parallelism can be detected within micro-instructions or between micro-instructions. This is very machine oriented (Freeman, 1975) and will not be considered here.

At the program level, parallelism can be detected either in individual programs or groups of programs. This is known as inter-program parallelism or multiprocessing and has been discussed by Enslow (1977).

This thesis will consider some methods of detecting parallelism within expressions (instruction level) and their possible extension to the whole program. At each level parallelism can be detected explicitly or implicitly.

If a programmer indicates which parts of his program can be executed by different processors in parallel, then the parallelism is explicit otherwise it is implicit.

Some statements for parallel processing have been used by Anderson (1965) for ALGOL 60. For example FORK initiates parallel tasks and JOIN waits for them to finish.

Cosden (1966) shows that there is a great variety of parallel activity in loops. He introduced a parallel loop construct (i.e. PARALLELFOR) where each iteration of a loop may be executed in parallel.

Brinch Hansen (1972), has presented structured programming concepts which have simple axiomatic properties and permit extensive compile time checking and generation of efficient machine code. His idea is sufficiently safe to use not only within operating systems but also within user programs to control the use of local resources.
Owicki (1975) has presented a method for verification of parallel programs based on Hoare's (1969) axiomatic approach for proving partial correctness. She provided axioms and inference rules for two parallel languages. The deductive system can be used to establish the correctness of a program which uses semaphores, events, or any of the common synchronizing tools. These proofs are quite complex because the verification of the interference-free property requires that each assertion be tested for invariance with respect to assignment statements executed on other processors.

In the explicit expression of parallelism the process is entirely the responsibility of the programmer. This introduces risks for program reliability.

Using the "implicit" approach, a program (written to run on a serial machine) is divided into segments of code, which can be executed on a parallel computer. This analysis is performed during the compilation stage. In such a situation the programmer may be unaware of the detection of parallelism in his program, and even of the machine in use. Also there is no need for a sequential program to be re-written to run efficiently on a parallel computer.

In this thesis new methods of detecting implicit parallelism within computer programs are presented.
1.4 EXISTING TECHNIQUES FOR RECOGNIZING AND DETECTING PARALLELISM

Algorithms and data structures which have been designed for conventional uniprocessor systems (SISD) cannot be transposed to a multiprocessor machine (SIMD and MIMD) without great losses in efficiency.

A compiler is a program which accepts as data an input program written in a programming language (such as FORTRAN or ALGOL) called the source language and produces as output an equivalent program in another language (such as machine code or assembly language) called the object language.

During the different stages of the compilation process, the possible parallel paths can be detected.

Various methods have previously been proposed for recognizing and detecting parallelism. These methods ascertain which parts of the micro-instructions, expression and group of statements are most suitable for execution in parallel.

Several algorithms have been devised to detect parallelism at the expression level, and expressions are the key building blocks in many programming languages.

In Chapter 2 a review of some of these methods will be considered. The balancing of arithmetic trees is investigated and an improved algorithm presented. This algorithm not only gives trees of minimal height but makes use of vacancies in the tree in such a way so as to allow another smaller tree to be inserted into it; the height reduction and the overlapping of trees both being features which can be exploited by parallel processing systems.

The methods already in existence for recognizing parallelism at the statement level, depend on the structure of a program. These program analysis techniques can be used in the compiling of programs for parallel computers.

Some of these techniques relate to the control flow through a program
and its analysis, others deal directly with the usage of and access to
data areas.

Another approach is data driven sequencing in which a computation is
sequenced by the availability of data.

In Chapter 3, these techniques are discussed in more detail.
1.5 THE NEW METHOD TO DETECT PARALLELISM

Existing techniques which are used for recognizing parallelism in a program can be used during the semantic analysis and code generating stages and at the run time.

To date no one has attempted to detect relationships between parts of a program during syntax analysis, i.e. changing the grammar used for a serial program to one more suitable for a parallel program.

In Chapter 4, we show how an operator precedence grammar can be changed to an operator grammar, by making use of the property of associativity. This operator grammar yields several different parse trees, one of which is more suitable for parallel execution.

In Chapter 5, we give a context sensitive grammar, based on ambiguous context-free grammar in Chapter 4, which described the structure of arithmetic expressions for a parallel programming language. The grammar gives rise to generalised operator precedence relations capable of indicating the associativity of like operators. This Chomsky type 1 grammar is capable not only of deriving semantically equivalent expressions but of actually performing the tree balancing. It also detects possible vacancies in a subexpression into which other smaller subexpressions can be inserted. The applicability of these grammars is discussed, as is the extension of tree-balancing to entire programs.

In Chapter 6, we consider sentences in the language generated by the grammar introduced in Chapter 5. A few existing parsing algorithms are described, and a new algorithm is presented. This is not necessarily as powerful as a recognizer for the general context-sensitive grammars, but is more suitable for the grammar presented in Section 5.3. This seems to yield a situation in which a non context-free grammar generates a context-free language. In this case, the recognizer has the same power as a
recognizer for context-free languages. In Chapter 7, we show that certain inter-relationships between operators in a language gives rise to situations which inhibit parallelism; the information derived from the grammatical analysis locate situations that dis-allow any parallel execution. However, further semantic checks are required before a parallel execution, free of side-effects, can be guaranteed.

The inter-relationships between some FORTRAN operators (which can only be partially specified by precedence relations) have also been investigated. This has not only given rise to extra information which could be used in parallel execution but has disclosed a lack of proper definition in certain statements in FORTRAN 66 which cause serious discrepancies in their implementation in various widely used compilers.

The thesis closes with a discussion of how grammatical structures may be used to avoid variation in language implementation and to convey semantic information which may be used in multiprocessor systems.

Chapter 8, summarizes the work and suggests extensions and areas for further work.
CHAPTER 2

A NEW STRATEGY FOR THE CONSTRUCTION OF BINARY

TRÉES FOR PARALLEL PROCESSING
2.1 INTRODUCTION

This chapter is concerned with the detection of parallelism in an arithmetic expression at compilation time. The problem of parallelism of arithmetic expressions has been investigated by several authors in the past, and various methods have been proposed for recognising parallelism at the expression level. These methods determine which parts of the expressions are most suitable for execution in parallel.

Methods of forming tree representations of the arithmetic expressions are well known and are given in (Knuth, 1967).

Since any operation that appears at the same level, in a tree representation of an expression, may be executed in parallel, in most of these methods, tree formation of the expression is used to attempt to make from it a balanced binary tree of minimum height.

In these algorithms it is assumed that:
1. There is an arbitrarily large number of processors.
2. Memory units are arbitrarily large.
3. There is no more than one reference to an atom, i.e., atoms are distinct.
4. All the atoms are simple variables.

Descriptions of two algorithms are given in the following section. Some simple arithmetic expressions will be used where necessary to illustrate the working of these algorithms.

In Section 3, the concept of 'vacancies' is defined and a new method which produces a binary tree representation of an arithmetic expression which is of minimum height wherever possible will be presented. In Section 5, all the algorithms are compared and it is shown that the new algorithm is better. In the following section the application of the new algorithm in polynomial evaluation is investigated, and it is noted that Mumro-Patterson (1973) give the binary tree of minimum height for a polynomial. In this report an arithmetic expression \( E \) is defined to be a well formed string of operators,
operand and parentheses. The operations will be assumed to be n-ary \((n \geq 1)\) with an imposed precedence relation on them. An arithmetic expression is simple iff it is parenthesis-free.

2.2 DEFINITIONS AND PREVIOUS ALGORITHMS

Some definitions are included which will be required later for the parsing of expressions to be executed on a parallel computer with any number of arithmetic units or processors.

Definition 2.1:

The symbols ' ::= ' and ' '|' belong to the Backus-Naur form (BNF) formalism. In general ' ::= ' means 'is' and ' '|' means 'or'. Also sequences of characters enclosed in the brackets <> represent metalinguistic variables whose values are sequences of symbols.

The syntax of operands and operators of an arithmetic expression can therefore be defined as follows:

\[
\begin{align*}
\text{<adding operator> ::= } &+ - \\
\text{<multiplying operator> ::= } &\times / \\
\text{<operand> ::= } &\text{<constant> | <variable> \\
\text{<factor> ::= } &\text{<operand> | <factor> }\times <\text{<factor>} \\
\text{<term> ::= } &\text{<factor> | <term> }\times\text{<multiplying operator> <factor> \\
\text{<simple arithmetic expression> ::= } &\text{<term> | <adding operator> <term> | <simple arithmetic expression > <adding operator> <term> }.
\end{align*}
\]

Definition 2.2:

An arithmetic expression is defined to be a well formed string of the elements described in Definition 2.1, i.e.,

\[
\text{<arithmetic expression> ::= <simple expression>}
\]
in terms of the definition of simple expression in Definition 2.1.
Definition 2.3:
An arithmetic expression $E$ may be specified by a list notation as:
$E=(op, a_1, a_2, a_3, \ldots, a_n)$, whose first element belongs to the operators set and the others are an ordered list of operands where each one is an atom, simple expression or another subexpression in the expression.

Definition 2.4:
A tree used in the following algorithms is assumed to be a structure (record) with five fields as follows:

1. **Left subtree:** Is a reference to the tree that is the left hand son of the present tree. This can be null if the present tree is an atom or simple variable.

2. **Right subtree:** The conditions for this subtree are the same as the left subtree but the right subtree refers to the right hand son of the present tree.

3. **Father of tree:** This is a reference to the tree that is the father of the present tree. This tree is null if the present tree has no father.

4. **Operator:** This is the operator which joins the left and right sub-trees. If the tree is an atom or simple variable this is the name of the variable.

5. **Level:** This is the level of the operator in the tree. The level (height) of a leaf is assumed to be one and the level of the other nodes depends on the level of left and right sub-trees.

The level of the tree, $T$, will be shown by $h(T)$. Figure 2.1 shows how a binary tree may be constructed for an arithmetic expression.

Definition 2.5:
The algorithm $A$ is **optimal** if and only if for any other algorithm $A'$, $h(T) \leq h(T')$. 
FIGURE 2.11: A tree structure for \((5.5+3) \times 7\)
where $T$ and $T'$ are the execution-trees constructed by algorithms $A$ and $A'$, respectively, for an arbitrary arithmetic expression.

In this thesis logarithms are with respect to base 2 and $\lceil\log_b n\rceil = b$, where $b$ is the smallest integer such that $b-1 < n \leq b$.

2.2.1 Previous Algorithms

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

These methods consider which parts of the expressions are most suitable for execution in parallel. Descriptions of them are given in the following sub-sections.

2.2.1.1 Williams's Algorithm

Williams has considered how a balanced binary tree of minimum height (level) is systematically constructed. In her algorithm three cases are considered.

Case I: If the operators of an expression are associative, the tree of lowest height can be constructed in the following manner: at the beginning it is assumed that the tree is null, for the tree of level one, one element can be inserted, at level two, two elements, at level three, up to four elements, at level four, up to eight elements, ... and at level $n$, up to $2^{n-1}$ elements.

Hence for $n$ operands the level of the tree is $h(T) = \lceil \log_2 n \rceil$. This is shown in the following example.

Example I: Let $E = a + b + c + d + e$. Then, the stage of generation of the execution tree $T$ for $E$ is shown in Figure 2.2a. Since the number of elements is $4 < 5 < 8$, the height of the tree is 4, and we can add three more variables to this tree without increasing the height of the tree, i.e., if $E = a + b + c + d + e + f + g + h$, the level of the tree is still 4, as shown in Figure 2.2b.

Case II: If the operators are not associative, then the tree of highest level
FIGURE 2.2a: Representation tree for $E = a + b + c + d + e$

FIGURE 2.2b: Representation tree for $E = a + b + c + d + e + f + g + h$
is constructed, i.e., for each operator the height of the tree will be increased by one. Hence, the number of levels is the number of operands minus one, so for \( n \) operands the level of the tree is \( h(T) = n - 1 \). This is shown in the following example.

**Example 2:** Let \( E = a + b + c + d + e \), then the level of the tree is four as shown in Figure 2.3.

**Case III:** If the elements are themselves other expressions, and the operators are associative, then the height of the constructed tree depends on the height of the left and right subtrees.

Assume the level of left and right subtrees are \( L_1 \) and \( L_2 \) respectively, then the level of the constructed tree \( T \) is:

\[
h(T) = \max[L_1, L_2] + 1
\]

The algorithm claims that inserting a subtree at the top of the tree is preferable to extending the tree below the lowest existing level, so as to provide further possible extensions to the tree. Figure 2.4 gives an example of how a subtree may be joined to a tree. For further details, see (Williams 1975). The following example shows the application of Williams's algorithm.

**Example 3:** Let \( E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + b_8 + b_7 * c_1 + b_9 + (b_5 + b_6) * b_4 + a_9 + b_1 + b_2 + (c_4 * c_5 + c_3) + c_2 + b_3 \).

To make a balanced tree for \( E \) the following steps are required.

1. A balanced tree is made for each subexpression. Hence the trees for
   \( E_1 = b_8 + b_7 * c_1 + b_9 \), \( E_2 = (b_5 + b_6) * b_4 \) and \( E_3 = (c_1 * c_5 + c_3) + c_2 \)
   will be made.

2. By Definition 2.3 we now have,
   \[ A = (+, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, E_1, E_2, a_9, b_1, b_2, E_3, b_3) \]

3. With the list \( A \) completed in step 2, the algorithm joins the operands together by using Case I and Case II whenever the operands are atoms, otherwise by using Case III.

The execution tree \( T \) for expression \( E \) is shown in Figure 2.5. The height of the tree is \( h(T) = 8 \).
FIGURE 2.3: Representation tree for $E = a^+ b^+ c^+ d^+ e$

FIGURE 2.4: Addition of a subtree to a tree
FIGURE 2.5: Execution tree for $E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + b_8 + b_9 + (c_1 + b_2)$

$(c_4 \cdot c_5 + c_3) \cdot c_3 + b_3$ by Williams's algorithm
As the writer has stated, after inserting several subtrees 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. Hence, any suitable positions available earlier in the tree are no longer accessible.

2.2.1.2 Ashoke's Algorithm

Similar cases to those discussed by Williams are also given by Ashoke (1976). Ashoke's algorithm allows various operators to have different execution times. Multiplication may reasonably be assumed to be much longer than addition. If the execution time (unit time) for '+' (addition) be $e_+$, for '*' (multiplication) be $e_*$ and so on, then, by these possibilities the cases discussed in subsection 2.2.1.1 might be reconsidered as follows:

Case I: The tree construction is the same as before except that $h(T) = \lceil \log n \rceil \ast e$ where $e$ is the execution time for the operator, thus, if the operator is '+' then $h(T) = \lceil \log n \rceil \ast e_+$ and if the operator is '*' then $h(T) = \lceil \log n \rceil \ast e_*$.

Case II: If the execution time for the operator is $e$, then $h(T) = (n-1) \ast e$.

Case III: $h(T) = \max[h(t_1), h(t_2)] + e$ where $t_1$ and $t_2$ are subexpressions and $e$ is the execution time for the operator.

In this algorithm, $h(T)$ denotes the time to execute an arithmetic expression.

Another aspect is that the algorithm applies the distribution law to reduce the height of the tree.

Most programmers use parentheses so as to keep the length of the expression short and to make it more comprehensible and readable. Two different types of parentheses are needed in a programming language, such that one type is used to dictate the order of evaluation and the other type is used in places where it does not matter if the parentheses are removed by the distribution law.

Since distribution is a complex phenomenon, application of the distribution law to reduce the height of an execution tree of an arithmetic
expression needs a very careful strategy to be chosen.

Ashoke demonstrates in his thesis that distribution may or may not reduce the tree height. He also proves that partial distribution is more efficient than total distribution, and that the use of the distribution law is more efficient if the operators have different execution times. The distribution law is quoted in the following definition.

**Definition 2.6:**

Let \( E_1, E_2 \) and \( E_3 \) be arithmetic expressions, then

\[
E_1(\cdot(E_2+\cdot E_3)) = E_1E_2 + E_1E_3.
\]

The expression on the right hand side is said to be the distributed form of that on the left hand side. Distribution can be applied repeatedly and in any order, e.g.

\[
(a+b)(c+g\cdot h) = (a+b)c + (a+b)g\cdot h
\]

\[
= a\cdot (c+g\cdot h) + b\cdot (c+g\cdot h)
\]

\[
= a\cdot c + b\cdot c + (a+b)\cdot g\cdot h
\]

\[
= a\cdot (c+g\cdot h) + b\cdot c + b\cdot g\cdot h,
\]

etc. which are different forms of an expression.

In the following examples, some of the effects of the application of the distribution law on the execution tree are shown. Hereafter the execution time for ' + ' and ' * ' will be denoted by \( \theta_+ \) and \( \theta_- \) respectively. Example 4 shows that distribution may increase the tree height.

**Example 4:** Let \( E = a\cdot b\cdot (c+d) \), \( E' = a\cdot c + a\cdot d + b\cdot c + b\cdot d \) be the distributed form of \( E \) and \( \theta_+ = \theta_- = 1 \). Then the height of the execution tree \( E' \) is 3, whereas the execution tree for \( E \) has height 2. The execution trees for \( E \) and \( E' \) are shown in Figure 2.6. Example 5 shows that distribution may decrease the tree height.

**Example 5:** Let \( E = a\cdot (b\cdot c\cdot d\cdot f\cdot g + k+h) \),

\[
E' = a\cdot b\cdot c\cdot d\cdot f\cdot g + a\cdot k + a\cdot h
\]

and \( \theta_+ = \theta_- = 1 \).

Then, the height of the execution tree for \( E \) is 5, whereas the execution tree
FIGURE 2.6: Distribution increases the height of the execution tree
for the distributed form $E'$ has height 4. The execution trees for $E$ and $E'$ are shown in Figure 2.7.

The following examples demonstrate that varying the execution time of the operators affects the usefulness of the application of distribution.

**Example 6:** Let $E = a*b*c*(d+f)+g+h$, $E' = a*b*c*d+a*b*c*f+g+h$ be the distributed form of $E$. Two cases can be considered:

**Case I:** If the execution times for different operators are equal, then, the height of the execution trees $E$ and $E'$ are equal. The execution trees for $E$ and $E'$ are shown in Figure 2.8.

**Case II:** The execution times for different operators are different. If $\theta_+ = 1$ and $\theta_* = 2$, then, the height of the execution tree of $E$ is 7, whereas the height of that for $E'$ is 6. Hence, the application of the distribution law may decrease the tree height when execution times for the various operators are different.

To what extent does distribution help to reduce the tree height? The following example answers this question.

**Example 7:** Let $E = (a+b)*(c*d*e*f*g+k*\%+m)$, and by application of distribution any one of the following examples can be obtained:

- $E_1 = a*c*d*e*f*g + a*k*\% + b*c*d*e*f*g + b*k*\% + a*m + b*m$
- $E_2 = a*c*d*e*f*g + a*k*\% + b*c*d*e*f*g + b*k*\% + (a+b)*m$
- $E_3 = (a+b)*(c*d)*(e*f*a) + (a+b)*(k*\%*m)$
- $E_4 = (a+b)*(c*d*e*f*g) + (a+b)*(k*\%) + (a+b)*m$
- $E_5 = (a+b)*(c*d*e*f*g + k*\%) + (a+b)*m$

Assuming $\theta_* = 3$ and $\theta_+ = 1$, then the height of the execution trees for $E, E_1, E_2, E_3, E_4$ and $E_5$ are 13, 12, 12, 10, 13 and 14 respectively. The corresponding execution trees of expressions $E, E_1, E_2, E_3, E_4$ and $E_5$ are shown in Figures 2.10, 2.11, 2.12, 2.13, 2.14 and 2.15.

If $\theta_+ = \theta_* = 1$ then the height of the execution trees for $E, E_1, E_2, E_3, E_4$ and $E_5$ are 5, 6, 6, 4, 5 and 6 respectively.
FIGURE 2.7: Distribution decreases the height of the execution tree
FIGURE 2.8: Distribution does not change the tree height when $\theta_+ = \theta_*$. 
\[ h(E) = 7 \]

\[ h(E') = 6 \]

**FIGURE 2.9**: Distribution changes the tree height when \( \theta = 2 \) and \( \theta' = 1 \).
$h(E) = 13$

FIGURE 2.10: Execution tree for E
FIGURE 2.11: Execution tree for $E_1$
FIGURE 2.12: Execution tree for E₂
FIGURE 2.13: Execution tree for $E_3$
$h(E_4) = 13$

**FIGURE 2.14:** Execution tree for $E_4$
FIGURE 2.15: Execution tree for $E_5$

$h(E_5) = 14$
In this example, the execution tree for $E_3$ is the minimum height execution tree and is the best form of distribution of $E$. The other types take much more time and reach a higher level.

Ashoke, by proving some theorems shows how to use distribution in order to effectively reduce the height of the execution tree of an expression without enumerating all possibilities.

The following example shows the application of Ashoke's algorithm, when there is no need for application of the distribution law.

**Example 8**: Let $E$ be the same expression (taken from Ashoke and Mukhapadhyay's paper) as in Example 3, where $\theta_+=1, \theta_*=2.5$ and $\theta=3.2$.

Figure 2.16 shows that the parts of the expression $E$ have been re-ordered in terms of the execution time of each subexpression. The height of the tree is $h(T)=7.7$ time units.

If all operations require the same execution time, then the height of the execution tree is $h(T)=8$, i.e., the same as the height of the execution tree for this expression as constructed by Williams's algorithm.

In Ashoke's algorithm, the execution times for the various operations on distinct machines are different, and the tree height depends on the execution time for each operation. Hence this method is machine dependent, i.e., from one machine to the other one the tree height will change. The other point to notice is that the application of the distribution law usually increases the number of processors required, for instance in Example 7 expression $E$ needs 4 processors but $E_1, E_2, E_3, E_4$ and $E_5$ need 10, 9, 5, 6 and 5 processors respectively. It also changes the order of the expression.

2.2.1.3 Other Methods for Recognizing Parallelism within Arithmetic Expressions

In previous sub-sections two methods have been described for the detection of parallelism within expressions.

Other possible methods have been studied by various authors. Squire (1963)
FIGURE 2.16: Execution tree for $E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + b_1 + b_2 + \ldots + (b_4 + b_5 + c_1 + b_6 + c_2 + b_7 + c_3 + b_8 + c_4 + c_5)$ by Ashoke's algorithm when $\theta_1 = 1$, $\theta_2 = 2.5$ and $\theta_3 = 3.2$
has proposed an algorithm based on information relating to operands, the related operator and the height of a tree representation at which an operation may be performed. The algorithm involves using both right to left and left to right scans, and is consequently very involved.

Stone (1967) proposed an algorithm to generate, in one pass of an arithmetic expression, a type of a reverse polish expression. Basically the algorithm attempts to join two subtrees $t_1$ and $t_2$ of the same level, i.e., $h(t_1) = h(t_2)$, to form a new subtree of level $h(t_1 + t_2) = h(t_1) + 1$.

Subtraction and division are handled by using inversion. This method can produce a full balanced binary tree of height $n$ when there are $2^n$ variables linked by a single type of associative operator.

Baer and Bovet (1968) have proposed an algorithm to satisfy the following aims:

a. To obtain the syntactic tree of minimum number of levels.
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.

Kuck (1977) applies distribution over expressions so that a tree representation is of minimum height where distributivity can not be used to reduce the height, associativity and commutativity only are used.

Ward (1974) proposed a method to create a tree representation of several assignment statements that appear adjacently. If none of the statements use the same variable then all of their corresponding trees may be executed in parallel.

None of the above methods seems to give a better tree representation for an arithmetic expression than those of Williams or Ashoke, the advantages of which were described earlier.
2.3 TREE HEIGHT REDUCTION BY A NEW STRATEGY

2.3.1 The Strategy of the Method

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

Ashoke examines the application of the distribution law to expressions such that a tree representation is of minimum height. This may involve performing extra operations such as shown in Example 7. (See Figures 2.10 and 2.13). The distributed form $E_3$ requires eleven operations whereas in the original form $E$ only nine operations are required. However $E_3$ is completed in only four levels whilst $E$ takes five.

When Example 3 is compared with Example 8, one finds that in Example 8 (see Figure 2.16), the subexpressions have been re-ordered, whilst in Example 3 (see Figure 2.5) they have not, i.e., although the levels of the two execution trees are the same, William's algorithm is much better than Ashoke's algorithm.

In this section by reconsidering Case II and III of the sub-sections 2.2.1.1 and 2.2.1.2, we will show a method to construct a binary tree of minimum number of levels for an arithmetic expression without reordering any part of it.

According to Case II of the algorithms described in sub-sections 2.2.1.1 and 2.2.1.2, if the operators of an expression are not associative, then the tree of the greatest height is constructed (see Figure 2.3). Hence for an expression with $n$ operands the tree height is equal to $n-1$.

Since $a^{mn} = a^{m+n}$, where $m$ and $n$ are positive integers, the execution tree for the expression $E$ in Example 2 can be reconstructed as it is shown in Figure 2.17.

According to Case III of the previous described algorithms, whenever
the operands of the expression are themselves other expressions, say, $t_1$ and $t_2$, then the height of the constructed tree is $h(t_1 + t_2) = \max[h(t_1), h(t_2)] + 1$.

To construct a tree $T = t_1 + t_2$, it is possible to insert one of these execution trees $t_1$ or $t_2$ into the other one. This idea is illustrated by the following example.

Example 9: Let $E_1 = a + b + c + d + e$, $E_2 = f * h$ and $E = E_1 + E_2$. The execution trees $t_1$, $t_2$ and $T$ for $E_1$, $E_2$ and $E$ are shown in Figures 2.18a, 2.18b and 2.19 respectively.

It can be seen that there are 'vacancies' in execution tree $t_1$, which can be exploited in reducing the height of the tree. Intuitively an obvious way of doing this is by proper 'accommodation' of right sized tree into these 'vacancies'.

Since the level of tree $t_2$ is suitable to be inserted into the right hand side of tree $t_1$, the two trees can be added together without increasing the level of $T = t_1 + t_2$.

Formal definition for the terms 'vacancies' and 'accommodation' will be given in the next sub-section.

2.3.2 Definitions and Concepts

We use the following definitions.

**Definition 2.7:**

An arithmetic expression is called a sum term, if it is an atom (simple variable or constant) or the final operation is '+' (addition). Similarly, an arithmetic expression is called a product term, if it is an atom or the final operation is '*' (multiplication).

**Definition 2.8:**

If $T$ is an execution tree such that the root node is '+' ('*') and $T_1$ and $T_2$ are left- and right-subtrees, respectively, then $T = T_1 + T_2$ ($T = T_1 * T_2$) is the algebraic notation for $T$. 
FIGURE 2.17: Execution tree for $E = a \uparrow b \uparrow c \uparrow d \uparrow e = a (b \uparrow c \uparrow d \uparrow e)$

$E$ = a \uparrow b \uparrow c \uparrow d \uparrow e = a (b \uparrow c \uparrow d \uparrow e)$

FIGURE 2.18: Execution trees for $t_1$ and $t_2$

$T = t_1 + t_2$

FIGURE 2.19: Accommodation of tree $t_2$ into the 'vacancy' of tree $t_1$
Definition 2.9:

\( \text{Add}(t_1, t_2, \ldots, t_n) \) is an execution-tree obtained by 'adding' the subtrees \( t_1, t_2, \ldots, t_n \).

Definition 2.10:

\( \text{Mult}(t_1, t_2, \ldots, t_n) \) is an execution-tree obtained by 'multiplying' the subtrees \( t_1, t_2, \ldots, t_n \).

Definition 2.11:

Vacancies of an execution tree \( T \), denoted as \( V_T \) is a set obtained as follows:
1. If \( T \) is a leaf (simple variable or constant) then \( V_T = \emptyset \) (the empty set).
2. If \( T = T_1 + T_2 \), let \( h(T), h(T_1) \) and \( h(T_2) \) be the integers \( \ell, \ell_1 \) and \( \ell_2 \) respectively. There are two cases to be considered here:
   a. \( |\ell_1 - \ell_2| = \ell' > 1 \), then \( V_T = \{\ell - 1, \ell - 2, \ldots, \ell - n > \min(\ell_1, \ell_2)\} \) and all of the vacancies belong to one of the subtrees only.
      This is because at least one of \( \ell_1 \) or \( \ell_2 \) must be equal to \( (\ell - 1) \).
   b. \( |\ell_1 - \ell_2| = 0 \). This means that the subtrees have the same height and \( \ell_1 = \ell_2 = \ell - 1 \).
3. If \( T = T_1 \times T_2 \), \( V_T \) is the same as \( V_T \) for \( T = T_1 + T_2 \). The vacancy is just part of a tree in which time is wasted waiting for operands to be computed.

Definition 2.12:

If \( V_T \) denotes the vacancies of \( T \), then an element of \( V_T \) is called a vacancy.

Definition 2.13:

Let \( T \) and \( t \) be execution trees, then \( T \) accommodates \( t \) if and only if \( h(T + t) = h(T) \).
Theorem 1:
Let T and t be two trees such that \( h(T) > h(t) \geq 1 \). Then T accommodates t if and only if T has a vacancy \( \forall \geq h(t) \).

1. If \( h(T) > h(t) \), T accommodates t when T has a vacancy \( \forall \geq h(t) \).

   Proof: The proof of this Theorem is obtained by contradiction. T accommodates t \( \Rightarrow h(T+t) = h(T) \). (By Definition 2.13) (a)
Suppose T does not have a vacancy \( \forall \geq h(t) \). Then \( h(T+t) = h(T) + 1 \). (b)
That is a contradiction (by (a) and (b)), hence T has a vacancy \( \forall \geq h(t) \).

2. If T has a vacancy \( \forall \geq h(t) \), then \( h(T) > h(t) \).

   Proof: Let T have a vacancy \( \forall \geq h(t) \), hence T accommodates t (by Definition 2.13).

Lemma 1:
Let A and B be two execution trees. If B accommodates A, then \( h(B) > h(A) + 1 \).

Proof: By Theorem 1, B accommodates A if and only if B has a vacancy \( \forall \geq h(A) \).
Hence, there exists a subtree \( B_1 \) of B which has a vacancy \( \forall_1 \geq \forall \geq h(A) \).
These definitions are clarified by the following examples:
If T=\( \text{add}(t_1, t_2) \), to find \( h(T) \), there are three cases to be considered:
1. \( h(t_1) > h(t_2) \)
2. \( h(t_1) < h(t_2) \)
3. \( h(t_1) = h(t_2) \).

1. If \( h(t_1) > h(t_2) \), there may be a vacancy in \( t_1 \) which can accommodate \( t_2 \).
2. If \( h(t_2) > h(t_1) \), there may be a vacancy in \( t_2 \) which can accommodate \( t_1 \).
3. There is not a suitable vacancy in any of them to accommodate the other one.

Example 10: \( h(t_1) > h(t_2) \)
Let \( t_1 = a*b+c+d+f+g \) and \( t_2 = h+k \).
The execution trees for \( t_1 \) and \( t_2 \) are shown in Figure 2.20a.

To insert \( t_2 \) in \( t_1 \) without re-ordering the expressions (i.e. first \( t_1 \) and then \( t_2 \)), \( h \) (right subtree of \( t_1 \)) must be less than \( h \) (left subtree of \( t_1 \)) and \( h \) (left subtree of \( t_1 \)) must be less than or equal to \( h(t_2) \), otherwise the height of \( T \) is \( h(T) = \max(h(t_1), h(t_2)) + 1 \). It can be seen that there is a vacancy in \( t_1 \) to accommodate \( t_2 \). The execution tree for \( T = t_1 + t_2 \) is shown in Figure 2.20b.

**Example 11:** \( h(t_1) < h(t_2) \)

Let \( t_1 = a*b \) and \( t_2 = a_1 + b_1 * c_1 * d_1 \) (See Figure 2.21a).

In this example, the height of the left subtree of \( t_2 \) is less than that of its right subtree and equal to \( h(t_1) \), then it is possible to insert \( t_1 \) somewhere on the left side of \( t_2 \), otherwise they will be joined together and \( h(t_1 + t_2) = h(t_2) + 1 \). The execution tree for \( T = t_1 + t_2 \) is shown in Figure 2.21b.

**Example 12:** \( h(t_1) = h(t_2) \)

Let \( t_1 = a*b + c \) and \( t_2 = a_1 + b_1 * c_1 \) (see Figure 2.22b).

In this example, there is no suitable vacancy in trees \( t_1 \) or \( t_2 \), for inserting one of them into the other. Hence, the height of the tree \( T = t_1 + t_2 \) is \( h(t_1 + t_2) = h(t_1 \text{ or } t_2) + 1 \). The execution tree for \( T \) is shown in Figure 2.22b.

Of course, the insertion is feasible if the operator of each tree is associative. The following example shows that in spite of existing vacancies, since the operator of the tree is not associative, it is not possible to insert another tree even though it may be of the right size, unless this vacancy (wasted processors) may be usable by a subsequent computation.

**Example 13:** Let \( t_1 = (a*b + c + d) + (f + g) \) and \( t_2 = h + k \) (See Figure 2.23a).

Although, there exists a suitable vacancy in the right subtree of \( t_1 \), \( t_2 \) cannot be inserted, because the operator on the fourth level of \( t_1 \) is not associative.
FIGURE 2.20: Execution trees for $t_1$, $t_2$ and $T = t_1 + t_2$ by accommodating $t_2$ into $t_1$. 
FIGURE 2.21: Execution trees for t1, t2 and T by accommodating t1 into t2
FIGURE 2.22: Execution trees for $t_1$, $t_2$ and $T = t_1 + t_2$
FIGURE 2.23: Execution trees for $t_1$, $t_2$ and $T = t_1 + t_2$
The height of tree $T$ is then $h(T) = \max(h(t_1), h(t_2)) + 1 = 4 + 1 = 5$.

In Example 12, the left subtree of $t_2$ has been inserted into the right subtree of $t_1$. Thus this new tree and the right subtree of $t_2$ are joined together. In this case, the height of the tree is the same as before, but vacancies have been created in the resulting tree for further insertion (see Figure 2.22b).

It should be noted that sometimes, if the expression is not executed several times, it may not be worth while to change it for parallel processing, as in the above case (Figure 2.22b).

The theorem, definitions and examples bring out a general strategy to reduce the total height of the execution tree of an arithmetic expression without moving any part of the expression around (i.e., applying the commutativity of operators).

### 2.4 FORMATION OF A BALANCED BINARY TREE

In the previous section we introduced a new technique to reduce the height of the tree for an arithmetic expression.

In this section the method for forming a balanced binary tree will be described. To construct a tree representing an arithmetic expression three cases must be considered as follows:

a. The expression is of the form $(op, t_1, t_2, \ldots, t_n)$ where $t_1, t_2, \ldots, t_n$ can be simple operands or other subexpression and $op$ is a nonassociative operator.

b. The expression is of the form $(op, t_1, t_2, \ldots, t_n)$ where $t_1, t_2, \ldots, t_n$ are simple variables (trees of $h(t_i) = 1$, $1 \leq i \leq n$) and $op$ is an associative operator.

c. The expression is like case b, except that each $t_i$, $1 \leq i \leq n$ can be either a simple operand or another subexpression, (a tree of $h(t_i) > 1$).
An expression can have any combination of the above cases. In case a, even if there exists a vacancy of the correct size in one of the trees it is not possible to insert any other subexpression into it. Therefore a tree, say \( T = t_1 \text{opt}_2 \text{opt}_3 \text{op}, \ldots, \text{opt}_n \) with \( h(T) = h(\max[h(t_1), h(t_2), \ldots, h(t_n)]) + n - 1 \) is constructed. The representation tree for \( T \) is shown in Figure 2.24. We should explain here that these available vacancies (or in other words these waiting processors) are usable for other computations at the same time that this tree is being computed.

We will commence with a new algorithm A which makes a balanced binary tree of minimum height for case b. Then, we introduce a second new algorithm, B, to construct a tree for case c.

To construct a tree for case b, \( t_1 \) and \( t_2 \) can be joined by forming a new tree say \( AD_1 \), whose left and right subtrees are \( t_1 \) and \( t_2 \), similarly \( t_3 \) can be joined to \( AD_1 \) by forming another tree, \( AD_2 \), whose right subtree is \( AD_1 \) and left subtree is \( t_3 \). Then \( t_4 \) can be inserted into the right subtree of \( AD_2 \) (without increasing the height). Thus, the right subtree of \( AD_2 \) is \( t_3 \text{opt}_4 \).

Similarly \( t_5 \) can be added by forming another tree, \( AD_3 \) with \( AD_2 \) and \( t_5 \) as its left and right subtrees. This process is continued either by inserting each \( t_i \), where \( i \leq n \), into the constructed tree or by forming another tree whose left subtree is the constructed one of \( t_1, t_2, \ldots, t_{i-1} \) and right subtree is \( t_i \). Then the constructed tree is a balanced binary tree of height \( \lceil \log_2 n \rceil \).

### 2.4.1 Algorithm A

Two stacks will be used, for storing symbols already scanned, operators are stored in \( \text{OPSTACK} \) and operands are stored in \( \text{STACK} \). A stack of trees, \( AD_i \), where \( i = 1, 2, 3, \ldots \), is required to keep any new node constructing during the process. The pointer \( \text{TP} \) points to the top of this stack. A tree is assumed to be a record as described in Definition 2.4 of Section 2. \( \text{TEMP} \) and \( \text{EXPER} \)
FIGURE 2.24: The representation tree for

\[ T = t_1 \text{ op } t_2 \text{ op } t_3 \text{ op } \ldots \text{ op } t_n \]

where \( \text{op} \) is a nonassociative operator.
are references to trees.

The expression $E$ is of the form $(o_P, t_1, t_2, \ldots, t_n)$ where $o_P$ is an associative operator and $t_i, 1 \leq i \leq n$ is a tree of the following form:

```
<table>
<thead>
<tr>
<th>empty</th>
<th>empty</th>
<th>empty</th>
<th>simple variable</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>left subtree</td>
<td>right subtree</td>
<td>father</td>
<td>the value of the variable</td>
<td>level of the tree</td>
</tr>
</tbody>
</table>
```

$Empty$ is an empty tree and $origin$ is the resulting tree.

The algorithm works as follows:

```plaintext
Begin  
(making a new node AD[1] whose left subtree is $t_1$ and right subtree is $t_2$)  
step 1 $\xi$=h($t_1$)+1;  
| TP=TP+1; (use an element of stack AD)  
| AD[TP]=$(t_1, t_2, empty, \xi, op);$  
| origin=AD[TP];  
step 2 while there are more operators and simple operands ($t_i, i \leq n$) do  
| begin  
| if $h(right of AD[TP])=h(left of AD[TP])$  
| then  
| (Joining AD[TP] and $t_i$)  
| $\xi$=h(AD[TP])+1;  
| TP=TP+1;  
| AD[TP]+AD[TP-1], $t_i$, empty,$\xi$, op);  
| origin=AD[TP]  
| else  
| (inserting $t_i$ into a suitable vacancy on the right-hand subtree of AD[TP])  
| temp=right of AD[TP];  
| exper=father of temp;  
| while $h(temp)\geq 2$ and ($h(left of temp)$>$h(right of temp)$) do (temp=right of temp;  
| exper=father of temp  
| ); (this action shows which node we are looking at)  
```

if \( h(\text{temp}) = 1 \)
\[ \text{then } \ell + 2 \]
else \( \ell + h(\text{temp}) \)
TP + TP + 1
AD[TP] + (\text{temp, } t_i, \text{exper, } \ell, \text{op})
if \( i = n \) (the last simple operand)
(adjust level of AD[TP])
then
if \( \ell = h(\text{temp}) \)
\[ \text{then } \ell + \ell + 1; \]
while (father of exper#empty) and
\( h(\text{right of exper}) = h(\text{left of exper}) \)
do
(adjust level of exper)
\[ \ell(\text{exper}) = \ell(\text{left of exper}) + 1; \]
origin + AD[TP - 1]
end

2.4.2 Algorithm B

This algorithm constructs a tree of minimum height for any arithmetic expression of the form \((\text{op, } t_1, t_2, \ldots, t_n)\), where \text{op} is an associative operator and each \( t_i, 1 < i \leq n \) is a simple operand (a tree of height one) or another subexpression.

To make a tree of minimum height for the expression, the following cases will be considered.

a. If \( t_1, t_2, \ldots, t_i, i < n \) are simple operands then the algorithm A will be applied to make a tree of height \( \lceil \log_2 i \rceil \). The constructed tree is being used instead of \( t_1 \) to \( t_i \).

b. If the two successive operands, say \( t_i \) and \( t_{i+1} \), are other subexpressions of different height, then if \( h(t_i) > h(t_{i+1}) \), the operator of the root node of \( t_i \) is the same as \text{op} and \( h(\text{left subtree of } t_i) > h(\text{right subtree of } t_i) \), then if there exists a suitable vacancy on the right subtree of
t_i, t_{i+1} can be inserted into it. In this case t_i is the resulting tree.
c. If h(t_i)<h(t_{i+1}) and all the above conditions for the right subtree of
t_i hold for the left subtree of t_{i+1}, then t_i can be inserted into the
left subtree of t_{i+1}. In this case t_{i+1} is the resulting tree.
d. If h(t_i)=h(t_{i+1}) or the conditions described in a and b, do not exist.
Then, the resulting tree of h(t_i)+1 is a tree whose left and right sub-
trees are t_i and t_{i+1} respectively.
This process will be continued until the stack becomes empty i.e., all
the operands have been used.
The algorithm works as follows:

while there are more subexpressions or simple variables do
begin
  step1: if all t_i are simple variables or constants then apply
        the algorithm A
          if for some j<i, t_j is simple variable or constant
          then apply the algorithm A and use the resulting tree
          instead of t_i.
  step2: t_i and t_{i+1} are subtrees of the height l_1 and l_2, and their
         root nodes are associative operators.
          if l_1-l_2 (t_i has vacancies), then
            temp=right subtree of t_i; l+=h(temp);
            while l>=l_2 and operator of temp=op
            (the operator of the right subtree of t_i should be op)
            do (temp=right of temp; l+=h(temp));
            l_3+=max(l_1,l_2)+1;
            TP+=TP+1; (use an element of stack AD)
            AD[TP]={(temp,t_{i+1},father of(father of temp),l_3,op)}
            (t_i has a vacancy to accommodate t_{i+1})
            h(t_i+t_{i+1})=h(t_i)
            t_{i+1}+=t_i; (the resulting tree)
          else if l_1-l_2 (t_{i+1} has vacancies) and the operator of their
            root nodes are associative
            then temp=left subtree of t_{i+1}; l+=h(temp)
            while l>=l_2 and operator of temp=op do
            (temp=left of temp; l+=h(temp));
\[ i_3 \max(i_1, i_2) + 1; \]
\[ TP + TP + 1; \]
\[ AD[TP] + (i_1, \text{temp}, \text{father of (father of temp), } i_3, \text{op}) \]

(now father of temp is AD[TP], therefore father of (father of temp) should refer to AD[TP] as its left subtree)

\[ (t_i \text{ is inserted in } t_{i+1} \text{ and } h(t_i, \text{op}, t_{i+1}) = h(t_i) \]

if \[ i_1 = i_2 \text{ or } |i_1 - i_2| = 1. \]
then
\[ i_3 = \max(i_1, i_2) + 1; \]
\[ T = (t_i, t_{i+1}, \text{empty, } i_3, \text{op}) ; t_{i+1} + T \]

(step 2 is feasible if the operators of the root nodes of \( t_i \) and \( t_{i+1} \) are associative)

**step3:** if the operators of the root node are not associative

then \( T = t_i + t_{i+1} \) and \( h(T) = \max(i_1, i_2) + 1 \)

**step4:** \( t_{i+1} + T \)

end.

### 2.4.3 The Basic Algorithm

In addition to all declarations in the algorithms A and B, a fixed string of operators, `opers = -"?$/\()\+-*/+"` and an array of integers, `priorities = (-1, 0, 0, 1, 2, 6, 6, 7, 7, 8)` are needed to show the operators being constructed and their priorities.

The expression is scanned one character at a time, from left to right. Various actions are taken to find all simple variables, subexpressions and operators which are needed to construct a balanced binary tree.

These actions are as follows:

1. Put a dummy operator '?' of priority -1 on **OPSTACK**.
2. Read a new symbol 'S' of the expression.
3. If \( S = "\)" , then repeat action 2.
4. If \( S \) is an operand, store it on top of **STACK**.
5. If \( S \) is an operator, compare its priority (p) with the one on top of **OPSTACK** (q).
6. If p>q add it to OPSTACK.

7. If p\neq q, then the two operands from the top of STACK are joined into a subtree by the operator from the top of OPSTACK applying either the new algorithm B if this operator is associative or the method for the nonassociative operators. Then the two operands and the operator are removed from the top of respective stacks and the resulting subtree is stored on top of the STACK.

8. If S is an opening parenthesis, this indicates the start of a subexpression. Hence the symbol "(" on the input string should initiate some special action. This process continues until a matching ")" closes the subexpression. This subexpression is stored on top of STACK.

9. If S="\$", this indicates the end of an expression has been reached. Hence, the final tree must be constructed. This is done by applying the algorithm B with the remaining items on OPSTACK and STACK.

The result of this algorithm is shown in the following example.

**Example 14:**

Let \( E=a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8+b_7\ast b_9+(b_5\ast b_6)\ast b_4+a_9+b_1+b_2+(c_4\ast c_5+c_3)\)

\[ +c_2\ast b_3 \]

be the same expression as in Example 3 and 8.

The execution tree has been shown in Figure 2.25. The height of the tree is \( h(T)=7 \).

2.5 **COMPARISON OF THE ALGORITHMS**

The previously described algorithms handle addition, multiplication and exponentiation, also subtraction and division can be handled by negating and reciprocating.

Ashoke's algorithm makes use of the commutativity of subexpressions.
FIGURE 2.25: Execution tree for 

\[ E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + b_8 + b_7 + b_6 + b_4 + a_9 + b_1 + b_2 + (c_4 + c_5 + c_6) + c_2 + b_3 \]

by the new algorithm.
Williams's algorithm and the proposed new one do not reorder any sub-expression or single variable of the expression. Figures 2.5, 2.16 and 2.25 show the different execution trees for the arithmetic expression:

\[ E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + b_8 + b_7 + c_1 + b_9 + (b_5 + b_6) + b_4 + a_9 + b_1 + b_2 + (c_4 * c_5 + c_3) + c_2 + b_3 \]

by Williams, Ashoke and the new algorithm respectively, the corresponding trees being of heights 8, 8 and 7.

Ashoke's algorithm uses the distribution law to reduce the height of the tree (assuming different execution times for various operators). In Example 7 we have shown that distribution may or may not decrease the tree height and may increase the number of processors even in case \( E_3 \) which reduces the height of the tree.

Hence, in speeding up numerical computation, the applying of the distribution law is still debatable. If a certain expression has to be evaluated for many different values, it might be worthwhile to apply distribution to reduce the height of the execution tree. Since each machine takes a different execution time for each operation, Ashoke's algorithm is machine dependent.

The new algorithm attempts to reduce the height of the execution tree by using 'vacancies' in subexpressions. When Examples 3, 8 and 14 are compared (see Figures 2.5, 2.16 and 2.25), the new algorithm is better in terms of Definition 2.5 in Section 2. Also, if there were more subtrees with height less than 6, they could be inserted into the right subtree of \( T \) in Figure 2.25 without increasing the height of the tree, but in Figure 2.5 the height of the tree would be increased. Williams (1978) claims that "this situation is in line with the tree being formed systematically from components". This is not the case because in the algorithm insertions are always at the next available position in the tree, i.e., the algorithm takes the subexpressions successively from its right hand side, so any
suitable positions available earlier in the tree are not accessible (see Figure 2.5). The new algorithm takes subexpression always from its left hand side for insertion in the tree and whenever any vacancy is available in the tree, the new subexpression is accommodated in it.

In Williams's algorithm, the height of the tree is enumerated by using a numerical code (an array of size $2^{12}$). This has three drawbacks:

1. To make the trees for any subexpression and expression itself, the value of the array elements must be calculated.
2. It takes time to compute the level of each constructed tree after attaching a variable to it, because its level should be compared with the value of the array elements.
3. The level of the tree can not be more than twelve, but the proposed new algorithm makes the tree without any restriction on its level.

The new algorithm saves time in two respects:

1. The construction of the tree for $n$ simple variables with $n-1$ numbers of any single associative operator is quicker.
2. Execution of the expression is quicker.

Programs for Williams's algorithm and the proposed new one were written in Algol 68-R and run on the ICL 1904S machine at Loughborough University within a loop (because the time to calculate an expression is short).

The time to make the trees for nine simple arithmetic expressions by the new algorithm is four time units less than the time by Williams's. Also the time to execute the trees for fifty arithmetic expressions (using the algorithm B to reduce the height of the tree) is three time units less than the time by Williams's. Therefore the new algorithm is more efficient in terms of time as well as level.

Since most operations and functions are not commutative and the application of commutativity allows the reordering of the operands within
an expression, we shall ignore its application in arithmetic expressions, even though it might reduce the level of the tree. For instance, in Example 14., if $b_2$ and $b_3$ added together and then combined with $a_9 + b_1$, then the resulting tree would be one level less than the tree in Figure 2.25.

Only one pass is necessary for the three algorithms to make an execution tree for an arithmetic expression.

The conditions for the operations and operands are the same in each of them, except that Ashoke uses the commutativity of the operators.

(For further details of this approach see also Williams (1978) and Ashoke (1976)).
2.6 ON THE EVALUATION OF A POLYNOMIAL

In previous sections general arithmetic expressions, where operands are distinct, have been investigated. There are some kinds of expressions which need special attention because all the atoms are not distinct. The evaluation of polynomials is one area which needs careful and further study.

A polynomial of degree n, denoted as \( p_n(x) \), is defined as \( p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \), where \( a_0, a_1, a_2, \ldots, a_n \) are called 'coefficients' and \( x \) is called the 'variable' or 'atom'.

If we consider the general situation in which the coefficients and variables are not related, then it is possible that in a specific case more simplification could be carried out.

Polynomial evaluation of this case has been investigated by Ashoke (1976), assuming different execution times for different operators. The algorithm he described is a modification of the Munro-Patterson algorithm (1973), (which the execution time for different operators is taken to be the same).

The Munro-Patterson algorithm uses the factorization to reduce the height of the tree for a polynomial. This is shown in the following example.

Example 15:

Let \( p_{11}(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{11} x^{11} \).

The Munro-Patterson algorithm changes \( p_{11}(x) \) to the form \( p'_{11}(x) \) as follows:

\[
p'_{11}(x) = (a_0 + a_9 x) + x^2 ((a_4 + a_3 x) + x^2 (a_6 + a_7 x)) + x^8 (a_0 + a_{11} x) + x^{24}
\]

By using a recursive procedure, every time a polynomial of degree \( q = \lceil \log(n+1) \rceil - 1 \) is constructed which is multiplied by \( x^0, x^\lceil \log(n+1) \rceil, x^2 \lceil \log(n+1) \rceil, \ldots \) and \( x^m \lceil \log(n+1) \rceil \) where the degree \( x^m \lceil \log(n+1) \rceil \) of \( p_q(x) \) \( \leq n \). The execution trees for \( p_{11}(x) \) and \( p'_{11}(x) \) are shown in Figures 2.26 and 2.27 respectively.

In a polynomial we can rarely find a suitable vacancy in a subexpression. Hence, neither the method described in sub-section 2.4, nor the distribution law can be used to reduce the height of the execution tree of a polynomial, when it is evaluated by the Munro-Patterson algorithm which is the quickest one.
FIGURE 2.26: Execution tree for $p_{11}(x)$
FIGURE 2.27: Execution tree for $p_{11}(x)$ by Munro-Patterson algorithm
2.7 CONCLUSIONS

In this chapter a new method is proposed for constructing an execution tree for an arithmetic expression, without reordering its variables or subexpressions. This method can be used at the compilation time and produces a suitable machine-oriented code for efficient parallel execution. In a real machine only a finite number of processors are available. Hence one of the areas which may be considered is forming a balanced binary tree for an expression which uses a limited number of processors. Especially in a polynomial of degree $n$, it may be possible to evaluate and store $x^2, x^4, x^8, \ldots, x^{2p}$ and $x^n$ and use them wherever needed, provided the time for fetching and storing them is less than for re-calculating them. For example, in Figure 2.27 the number of processors is much less than that in Figure 2.26 although they both evaluate the same polynomial.

For efficient parallel computation data has to be organized in a way which permits simultaneous fetching and storing, and the resources have to be shared among processes executable in parallel. Hence, an optimal algorithm which reduces the tree height of an expression must be efficient in terms of storing and fetching the data and organising the resources.

When the variables are real, applying the distribution law to reduce the level of the tree, might not give the correct answer. This is particularly important as one of the main areas in which parallel processing will be useful is the solving of large sets of numeric equations.

In many expressions (or subexpressions) there might be some vacancies which can not accommodate another subexpression from the same expression. In this case these vacancies can be used for other subsequent computation.
CHAPTER 3

A SURVEY OF PREVIOUS METHODS OF DETERMINING PARALLELISM IN A PROGRAM
3.1 INTERDEPENDENCIES BETWEEN PARTS OF A PROGRAM

A program designed to run on a serial computer is assumed to pass from one statement to the one immediately following, unless a control statement dictates otherwise. However, each statement, or group of statements, is not necessarily dependent on their predecessors. Hence, when various parts of a program are independent it may be possible to execute them in parallel. Thus, any approach to determine parallelism, at this level, will have to study dependencies between the different parts of a program.
3.2 EXISTING TECHNIQUES TO EXPLOIT AND DETECT PARALLELISM

Several methods have already been studied to determine parallelism at different levels of a program (particular statements, expressions, groups of assignment statements, iteration of a loop and so on).

These methods are considered in two categories. One of these techniques uses graph theory for recognizing parallelism and other methods are generally based on aspects of the structure of the program.

3.2.1 Methods Based on Graph Theory

Kuck (1975, 1977, 1980) and his co-workers at U.C.L.A. are the first who have used graph theory to exploit parallelism in a program. Rammamoorthy and Gonzalez (1969, 1970, 1971) have studied a more formal approach based on a connectivity matrix of a graph representation of a program. More recently Dennis (1973) outlined a method based on a data flow graph.

The greater part of Kuck's work is based on FORTRAN-like programming languages. The first distinction he makes is between "program segments" containing loops and those that are loop free. In both cases, each statement is considered and its dependency on other statements is determined. He defines five kinds of program dependencies (see Kuck, 1977, 1980), which arise in languages and describes how to build a dependence graph. Then he sketches various ways of removing dependencies from a graph that are derived in an exact way from a given program, which can lead to faster computation. These dependencies are illustrated in Figure 3.1 (Kuck 1980).

He has also considered a program segment containing a loop which may be of three types: acyclic, linear cyclic or non-linear cyclic, and the methods to detect parallelism in these loops.
Example 3.1:

\[ S_1 : A+B+C \]
\[ S_2 : B+A^3 \]
\[ S_3 : A+2^C \]
\[ S_4 : p+B_{>6} \]

\[
\begin{cases}
\text{if } p \text{ then } S_5 : D=1 \\
\text{else } S_6 : D=0
\end{cases}
\]

The relation between the statements are as follows:

- \( S_2 \) is \textbf{flow-dependent} on \( S_1 \) (\( S_1 \rightarrow S_2 \)) because of A,
- \( S_2 \) is \textbf{anti-dependent} on \( S_1 \) (\( S_1 \leftarrow S_2 \)) because of B,
- \( S_3 \) is \textbf{output-dependent} on \( S_1 \) (\( S_1 \rightarrow S_3 \)) because of A,
- and \( S_3 \) is \textbf{input-dependent} on \( S_1 \) (\( S_1 \leftarrow S_3 \)) because of C.

\( S_5 \) and \( S_6 \) are \textbf{control dependent} on \( S_4 \) (\( S_4 \rightarrow S_5 \) and \( S_4 \rightarrow S_6 \)).

The work of Rammamoorthy is also based on the fact that computational processes can be modelled by oriented graphs. Each node in the graph represents a single task and each oriented edge represents the permissible transition to the next task. The graph can be represented by a \((N \times N)\) connectivity matrix. The digit 1 is used in position \( I,J \) in the matrix to represent a directed edge between nodes \( I \) and \( J \), where \( 1 \leq I,J \leq N \). A zero in the position \( I,J \) in the matrix shows that there is no connection between the \( I^{th} \) and \( J^{th} \) nodes. All nodes that make a strongly connected subgraph can be considered as a single task, and if there is no relation between the subgraphs, they can be executed in parallel.

The work of Dennis is based on a data flow graph. A data flow graph is a directed graph with two types of nodes: links and actors. Arcs in the graph carry values from one actor to others by means of links (input links and output links). For further details see Syre (1980) and Dennis (1973). Many people have tried to extend the capabilities of data flow graphs. In particular David Misunas (1975) used new actors to introduce data structures
FIGURE 3.1: The dependence types between the various statements of Example 3.1
and to merge two structures or extract substructures.

3.2.2 Methods Based on the Structure of the Program

In 1966, Burroughs Corporation commenced researching into the automatic recognition of the parallelism within computer programs (Bingham et al. and Reigel 1967, 1968 and 1970). Also Bernstein (1966) presented another technique based on set theory.

The Burroughs Corporation developed an algorithm to detect implicit parallelism in various parts of the program structures like loops and conditionals.

Bernstein has considered how a memory location may be used by a group of instructions \( p \). He divided the memory location into four types as follows:

1. The location \((W)\) which is only fetched during the execution of \( p \).
2. The location \((X)\) which is only stored during the execution of \( p \).
3. The first operation on the location \((Y)\) is a fetch and then stored.
4. The first operation on the location \((Z)\) is a store and then fetch.

Then, two program segments \( p_1 \) and \( p_2 \) can be executed in parallel if the following conditions hold:

1. The inputs of \( p_1 \) must not be used as the outputs of \( p_2 \), i.e.
   \[
   (W_1 U Y_1 U Z_1) \cap (X_2 U Y_2 U Z_2) = \emptyset
   \]

2. The outputs of \( p_2 \) must not be used as the inputs of \( p_1 \), i.e.
   \[
   (X_1 U Y_1 U Z_1) \cap (W_2 U Y_2 U Z_2) = \emptyset
   \]

3. The locations changed in both \( p_1 \) and \( p_2 \) must be reset before being used.

With these conditions Bernstein has suggested that two program segments may be commutative.

Towel (1976) has considered the inter-relationships between data dependencies and control dependencies.
Williams (1978) defines the relationships that may exist between sub-programs as follows:

1. $S_1$ and $S_2$ are **contemporary** if they can be executed at the same time and the locations used in any order.

2. $S_1$ and $S_2$ are **commutative** if each one may be executed before or after the other one but not at the same time.

3. $S_1$ and $S_2$ are **prerequisite** if $S_1$ fetches its inputs before $S_2$ stores its output.

4. $S_1$ and $S_2$ are **conservative** if $S_1$ must store its outputs before $S_2$ does.

5. $S_1$ and $S_2$ are **consecutive** if $S_1$ must store its outputs before $S_2$ fetches its inputs.

6. $S_1$ and $S_2$ are **synchronous** if both have the same inputs.

7. $S_1$ and $S_2$ are **inclusive** if $S_2$ must store its outputs after $S_1$ fetches its inputs but before $S_1$ stores its outputs.

Williams has used these definitions to detect parallelism in a sub-program and the relationships between different sub-programs. She considered these definitions for loop, conditional statements and procedures on machines with private and shared memories or just shared memory.
CHAPTER 4

A NEW METHOD FOR DETECTING POTENTIAL PARALLELISM IN A PROGRAM
4.1 INTRODUCTION

In the preceding chapters we described existing techniques which can be used for recognizing parallelism between different parts of a program at compilation time.

Each of these methods looks at parallelism in a computer program during semantic analysis, code generating and run time, from a special point of view. All of these methods are feasible because a compiler can assist in producing reliable programs (by means of semantic analysis). For example, a compiler could check that the types of operands are compatible, and could often warn the user if there is a possibility that a variable is used before being defined or if an array is out of bounds. Thus, a compiler can also find the properties of each operator or operand, that is why recognizing parallel relationships is possible during semantic analysis. But to date no one has attempted to detect relationships between different parts of a program during syntax analysis, i.e., changing the grammar used for serial programs to one more suitable for a parallel program. To date, we are not quite sure whether it is possible to find a complete grammar for parallel programming. In the following sections and chapters we will consider this possibility.
4.2 SOME BASIC DEFINITIONS OF LANGUAGE THEORY

Definition 4.1:
A letter (or character) is a single indivisible token or symbol and an alphabet is a set of letters.

Definition 4.2:
A set of finite-length strings over a set of finite alphabet (T) is called a language.

The Algol and Fortran programming languages are included in this definition.

Let $T^*$ denote the set containing all strings over T including $\Lambda$. For example if $T=\{a,b\}$ then,

$$T^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, bbb, \ldots\}$$

$T^+$ is the set of all strings over T excluding $\Lambda$. Every language over an alphabet T is a subset of $T^*$. 
4.3 GRAMMARS

A grammar is a mathematical system for defining a language and a device for giving a useful structure to the sentences in the language.

Two finite disjoint sets of symbols are used in a grammar. These are, the set of non-terminal symbols, which is denoted by "N", the set of terminal symbols "T" which is the alphabet over which the language is defined, and $N \cup T = V$ where $V$ is called the vocabulary. The vital part of a grammar is a finite set $P$ of productions which describe how the sentences of the language are to be generated.

A production is a pair of strings, or, precisely an element of $V^*N^* \times V^*$
i.e., the first component is any string containing at least one non-terminal symbol and the second one is any string. The pair of strings is shown as a "rewriting rule" of a form $a \rightarrow B$ rather than $(a, B)$.

**Definition 4.3: (Phrase-Structure grammar) P.S.G.**

A grammar is a four-tuple $G = (N, T, P, S)$, where:
1. $N$ is a finite set of non-terminal symbols.
2. $T$ is a finite set of terminal symbols, disjoint from $N$, i.e. $N \cap T = \emptyset$.
   
   $V$, the vocabulary, is defined by $V = N \cup T$.
3. $P$ is a finite set of "rewriting rules" of the form $a \rightarrow B$ called a production, where $a \in V^*N^*$ and $B \in V^*$. (i.e., $a$ consists of three strings juxtaposed, the first and third from $V^*$, the second is an element of $N$).
4. $S$ is a special symbol in $N$ called the sentence (or start) symbol.

**Example 1:**

An example of a grammar is $G_1 = (\{C, S\}, \{a, b\}, P, S)$, where $P$ consists of

- $S \rightarrow aCb$
- $aC \rightarrow aaCb$
- $C \rightarrow A$
The non-terminal symbols are $C$ and $S$ and terminal symbols are $a$ and $b$.

**Definition 4.4:**

Let $G=(N,T,P,S)$ be a grammar, we can define a relation $\Rightarrow$ (directly derives) on $(N \cup T)^*$ as follows:

If $xAy$ is a string in $(N \cup T)^*$ and $A \Rightarrow B$ is a production in $P$, then $xAy \Rightarrow xBy$.

Relation $\Rightarrow$ (derives in a non-trivial way) is used to denote the transitive closure of $\Rightarrow$.

Relation $\Rightarrow^*$ is used to denote the reflexive and transitive closure of $\Rightarrow$.

The language generated by a grammar $G$, denoted $L(G)$, is the set of strings generated by the grammar $G$, i.e.,

$$L(G) = \{\alpha | \alpha \in T^* \text{ and } S \Rightarrow^* \alpha\}.$$ 

A string containing no non-terminal symbols, generated by a grammar $G$, called a sentence.
4.4 **RESTRICTED GRAMMARS**

Let $G=(N,T,P,S)$ be a P.S.G. as described above, such a grammar is called Chomsky Type 0.

If $W\rightarrow Y \in P$

and $W = Z_1 U Z_2$ and $Y = Z_1 V Z_2$

where $Z_1, Z_2 \in V^*$, $U \in N$ and $V \in V^+$,

then $G$ is said to be **context sensitive**, or Chomsky Type 1,

(in this definition the string $Z_1$ and $Z_2$ may be regarded as the context in which $U$ may be replaced by $V$).

An alternative restriction is that $W$ and $Y$ should be such that

$$1 \leq |W| \leq |Y|$$

If the replacements may be carried out regardless of context then we may replace 'contexts' $Z_1$ and $Z_2$ by the empty string, $\lambda$, and obtain the weaker restriction that if $W\rightarrow Y \in P$, then $W \in N$ and $Y \in V^+$. This restriction is satisfied by **context-free** or Chomsky type 2 grammars.

Finally, if $P$ consists only of productions of the form $W\rightarrow Y$ where

$W \in N$, $Y \in T \cup TN$,

then $G$ is said to be regular or Chomsky Type 3 grammar, some texts also refer to such a grammar as **right linear**.

The languages generated by any of these kinds of grammar are similarly named. So, a P.S.G. generates a phrase-structure language (P.S.L.), a C.S.G. generates a C.S.L., C.F.G. generates a C.F.L. and a regular grammar generates a regular language (or a regular set).

Type 0 languages can be recognized by Turing Machines and are much too general for computer languages; Type 3 languages can be recognized by Finite-State Automata, and most of the work done at the lexical scan is defined as a finite state language. Type 1 and 2 languages can be interpreted as systems of phrase structure description and are very
suitable for programming languages. In this thesis only Type 1 and 2 grammars and languages will be considered.

It is clear that for both the grammars and languages,

\[
\text{Type 0} \supset \text{type 1} \supset \text{type 2} \supset \text{type 3}.
\]

The context free grammars (type 2 in the Chomsky classification) are the most important because of its applications to programming languages and compiling processes. The use of a context-free grammar is enough to specify most of the syntactic structure of a programming language (for further details, see Aho and Ullman (1972)).

Now we will describe how a grammar defines a language and later specify what features or programming languages cannot be specified by a C.F.G.

How does a context free grammar define a language?

To answer this question we will consider the following C.F.G. for an arithmetic expression:

\[
E + E + E | E * E | (E) | I,
\]

where the non-terminal \(E\) is an abbreviation for an expression. The production \(E \rightarrow (E)\) shows the one instance where an \(E\) can be replaced by \((E)\) in any string of grammar symbols, i.e.,

\[
E * E \rightarrow (E) * E,
\]

If we have:

\[
E \Rightarrow (E) \Rightarrow (I).
\]

Such a sequence of replacement is called a derivation on \((I)\) from \(E\). This derivation shows that one particular instance of an expression is the string \((I)\).

A grammar defines a language in a recursive manner, We define a special kind of string called a sentential form of a grammar \(G=(N,T,P,S)\) recursively as follows:

1. \(S\) is a sentential form.

2. If \(\alpha \beta \gamma\) is a sentential form and \(\beta \rightarrow \delta\) is in \(P\), then \(\alpha \delta \gamma\) is also a sentential form. Thus every sentential form \(\alpha \in T^*\) is a sentence.
The notation \( \alpha \Rightarrow^k \beta \) is used to denote the \( k \)-fold product of the relation \( \Rightarrow \). That is to say, \( \alpha \Rightarrow^k \beta \) if there is a sequence \( \alpha_0, \alpha_1, \ldots, \alpha_k \) of \( k+1 \) strings (not necessarily distinct), such that \( \alpha = \alpha_0, \alpha_{i-1} \Rightarrow \alpha_i \) for \( 1 \leq i \leq k \) and \( \alpha_k = \beta \). This sequence of strings is called a derivation of length \( k \) of \( \beta \) from \( \alpha \) in \( G \).

**Example 4.2:**

The string \((1+1)\) is a sentence of grammar (4.1) because

\[
E \Rightarrow (E) \Rightarrow (E+E) \Rightarrow (I+E) \Rightarrow (1+1)
\]

(4.2a)

The above strings appearing in this derivation are all sentential forms of this grammar.

We can write \( E \Rightarrow^* (I+I) \). It is easy to show that the language of the C.F.G. (4.1) is the set of all arithmetic expressions involving only the binary operators \( '+', '*' \) and \( '(', ')' \) and the operand \( 'I' \).

The derivation of the above example could continue from \((E+E)\) as follows:

\[
(E+E) \Rightarrow (E+I) \Rightarrow (I+I)
\]

(4.2b)

where we have replaced each non-terminal in (4.2b) by the same right side as in Example 4.2, but the order of replacement is different.

**Definition 4.5:**

A derivation is the "left most derivation", if only the left most non-terminal in a sentential form is replaced at each step. "Right most" derivations are those in which the right most non-terminal is replaced at each step. The left most derivation is sometimes called the canonical derivation.

For example, the derivation like the one in (4.2a) is a left most derivation and in (4.2b) is a right most one.

**Derivation Tree:**

In a grammar it is possible to have several derivations that are equivalent, in the sense that all derivations use the same productions at
the same places, but in a different order. The determination of when two derivations are equivalent is a complex matter for unrestricted grammars, but for context-free grammars we can define a convenient graphical representative of an equivalence class of derivations called a derivation tree. The ability to construct such trees assists greatly in the understanding of C.F.G.'s.

The parse tree for (1+1) implied by the derivation of Example 4.2 as shown in Figure 4.1, does not show the order in which symbols are replaced, i.e., if the derivation of Example 4.2 were continued as in line (4.2b) the same parse tree as Figure 4.1 would result. The following example illustrates this matter.

**Example 4.3:**

The sentence 1+1+1 has two distinct left most derivations:

\[

e \Rightarrow e + e \\
\Rightarrow 1 + e \\
\Rightarrow 1 + e + e \\
\Rightarrow 1 + 1 + e \\
\Rightarrow 1 + 1 + 1
\]

and

\[

e \Rightarrow e + e \\
\Rightarrow e + e + e \\
\Rightarrow 1 + e + e \\
\Rightarrow 1 + 1 + e \\
\Rightarrow 1 + 1 + 1
\]

with the two corresponding parse trees as shown in Figure 4.2.

**Example 4.4:**

The sentence 1+1*1 has two distinct left most derivations:

\[

e \Rightarrow e + e \\
\Rightarrow e + e + e \\
\Rightarrow 1 + e + e \\
\Rightarrow 1 + 1 + e \\
\Rightarrow 1 + 1 + 1
\]

and

\[

e \Rightarrow e * e \\
\Rightarrow e + e * e \\
\Rightarrow 1 + e * e \\
\Rightarrow 1 + 1 * e \\
\Rightarrow 1 + 1 * 1
\]

with the two corresponding parse trees shown in Figure 4.3.

Because of the commonly assumed precedence of + and * we say that the parse tree of Figure 4.3(a) is correct, but in Example 4.3 both operators
FIGURE 4.1: Parse tree for (I+I)

(a) E + E
    E
    I

(b) E + E
    I

FIGURE 4.2: The parse tree for I+I+I

(a) E + E
    E
    I
    I

(b) E + E
    E
    I
    I

FIGURE 4.3: The parse tree for I+I*I

(a) E * E
    E
    I
    I

(b) E + E
    E
    I
    I
are + hence, both derivations are mathematically correct. It is desirable to modify the grammar so that a certain structure is imposed on the language generated. Otherwise a programmer and a compiler may have differing opinions as to the meaning of some sentences.

The grammar (4.1) does not give a unique structure for its language. In the next section, we will consider ambiguity in detail for context free grammars and will show two unambiguous grammars equivalent to the grammar (4.1) by imposing left or right associativity and precedence rules on the grammar.
4.5 AMBIGUITY

A C.F.G.: is ambiguous if it produces more than one parse tree for some sentence, in other words if it produces more than one left most derivation for some sentence. (For further details see Aho and Ullman (1972)).

We can dis-ambiguate the grammar (4.1) by specifying the left associativity and precedence of the arithmetic operators. However, it is possible to incorporate these rules into the grammar and hence make it unambiguous.

By introducing one non-terminal for each precedence level in the grammar (4.3) we have:

\[
\text{expression} \rightarrow \text{expression} + \text{term} | \text{term} \\
\text{term} \rightarrow \text{term} * \text{factor} | \text{factor} \\
\text{factor} \rightarrow (\text{expression}) | \text{identifier} ,
\]

which is an unambiguous grammar. Hereafter we will show it by:

\[
E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
F \rightarrow (E) | I .
\]

(4.3)

Thus, according to the grammar (4.3) the order of the evaluations of the operators in Figures 4.2(b) and 4.3(a) is correct. However the order given by the Figure 4.2(a) is not acceptable to the grammar, even though it is computationally 'valid'. To illustrate this matter, we need more specifications about the properties of an associative operator. In the next section we will give some definitions and relations which will be useful.
4.6 OPERATOR PRECEDENCE RELATIONS FROM ASSOCIATIVITY AND PRECEDENCE

Definition 4.6:
A C.F.G. is said to be an operator grammar if no production of it takes the form:

\[ U \rightarrow xXYy, \]

where \( X \) and \( Y \) are non-terminal symbols (\( N \)) and \( x, y \in V^* \) are strings of symbols either or both of which may be null. If \( G = (N, T, P, S) \) is an operator grammar, then it is clear that no sentential form generated from \( S \) can contain adjacent non-terminal characters.

In an operator grammar there are three relations, some or all of which may hold between two terminal symbols \( a \) and \( b \):

1. \( a \sim b \), if there is a production \( U \rightarrow xAby \) where \( \delta \) is either \( A \) or a single non-terminal. That is, \( a \sim b \) if \( a \) appears immediately to the left of \( b \), or if one non-terminal separates them. For example, in the grammar (4.3), the production \( F \rightarrow (E) \) implies that \( (\sim) \).
2. \( a \rightharpoonup b \), if there is a production \( U \rightarrow xAb \) and a derivation \( A \Rightarrow ZaU \) where \( U \) is either \( A \) or a single non-terminal.
3. \( a \rightharpoonup b \), if there is a production \( U \rightarrow xAby \) and a derivation \( A \Rightarrow ZbU \), where \( Z \) is either \( A \) or a single non-terminal.

The trees for \( a \rightharpoonup b \) and \( a \sim b \) are shown in Figure 4.4.

Definition 4.7:
A precedence grammar is an operator grammar for which no more than one of the three relations \( \sim, \rightharpoonup \) and \( \rightharpoonup \) holds between any pair of terminal symbols. These relationships are called precedence relations of the terminal symbols.

We suppose that \( $ \) marks each end of the string, and we define \( $ \rightharpoonup b \) and \( a \rightharpoonup $ \), if there are the derivations, \( $ \rightarrow \gamma b \delta \) or \( $ \rightarrow \delta a \gamma \) and \( \gamma \in N \cup \{ A \} \).

In the parse tree of a sentence of a precedence grammar, a part of the sentence which has higher precedence is at the lower level of the tree, i.e., it is evaluated sooner than the parts which have lower precedence, (see Figure 4.4).
FIGURE 4.4: The parse trees indicating $a > b$ and $a < b$
Example 4.5:

Grammar (4.3) is an operator precedence grammar and the precedence relations for this grammar are shown in Figure 4.5.

Example 4.6:

Another example of an operator precedence grammar is:

\[
\begin{align*}
E & \rightarrow T + E | T \\
T & \rightarrow F \ast T | F \\
F & \rightarrow (E) | I
\end{align*}
\]  

(4.4)

and its precedence relations are shown in Figure 4.6.

In Example 4.5, + and * are left associative. The precedence relations between them are + ≫+ and * ≫*.

In Example 4.6, + and * are right associative. The precedence relations between them are + ≪+ and * ≪*. We are free to create operator precedence relations in any way we see as suitable. As shown in grammar (4.3), left associativity is incorporated into the grammar, but in grammar (4.4) right associativity is used. Both grammars give the 'correct' results to an arithmetic expression (including only + and *), and are unambiguous.

What will happen if grammars (4.3) and (4.4) are used together to evaluate an arithmetic expression? In other words what will happen if we are free at any moment to choose right or left associativity to evaluate an expression? We answer this question in the following example.

Example 4.7:

Let us evaluate \( E=I+I+I+I+I \). \( E \) can be evaluated by using both the grammars (4.3) and (4.4). The parse trees for \( E \) are shown in Figures 4.7(a) and 4.7(b) respectively.

We are free also to make other trees as well by using the above precedence relations for + from both grammars (4.3) and (4.4). Some of these trees are shown in Figure 4.8.
<table>
<thead>
<tr>
<th>+  *  (  )  I  $</th>
</tr>
</thead>
<tbody>
<tr>
<td>+  &gt;  &lt;  &lt;  &gt;  &lt;  &gt;</td>
</tr>
<tr>
<td>*  &gt;  &gt;  &lt;  &gt;  &lt;  &gt;</td>
</tr>
<tr>
<td>(  &lt;  &lt;  &lt;  =  &lt;</td>
</tr>
<tr>
<td>)  &gt;  &gt;  &gt;  &gt;  &lt;  &gt;</td>
</tr>
<tr>
<td>I  &gt;  &gt;  &gt;  &gt;  &gt;</td>
</tr>
<tr>
<td>$  &lt;  &lt;  &lt;  &lt;  &lt;</td>
</tr>
</tbody>
</table>

**FIGURE 4.5:** Operator precedence relations for grammar (4.3)

<table>
<thead>
<tr>
<th>+  *  (  )  I  $</th>
</tr>
</thead>
<tbody>
<tr>
<td>+  &lt;  &lt;  &lt;  &gt;  &lt;  &gt;</td>
</tr>
<tr>
<td>*  &gt;  &lt;  &lt;  &gt;  &lt;  &gt;</td>
</tr>
<tr>
<td>(  &lt;  &lt;  &lt;  =  &lt;</td>
</tr>
<tr>
<td>)  &gt;  &gt;  &gt;  &gt;  &gt;</td>
</tr>
<tr>
<td>I  &gt;  &gt;  &gt;  &gt;  &gt;</td>
</tr>
<tr>
<td>$  &lt;  &lt;  &lt;  &lt;  &lt;</td>
</tr>
</tbody>
</table>

**FIGURE 4.6:** Operator precedence relations for grammar (4.4)
FIGURE 4.7: Two parse trees for $E = I + I + I + I + I$
FIGURE 4.81 Three different parse trees for $E = I + I + I + I + I$
The trees in Figure 4.8 show that we are free to choose the relation + →+ or +←+ at any step we like.

If we consider the following grammar $G$:

$$
E \rightarrow E+T \mid T+E \mid T \\
T \rightarrow T*F \mid F*T \mid F \\
F \rightarrow (E) \mid I 
$$

The relation between the two '+'s could be ← or →. We indicate these two relations by a single relation '⊥'. The relations for grammar (4.5) are shown in Figure 4.9.

The relation ⊥ shows that when there are two or more successive + or * operators, they can be evaluated in any order (because of associativity).

This grammar is ambiguous because

$$
E \rightarrow E+T \Rightarrow E+T+T \Rightarrow T+T+T \rightarrow I+I+I \\
E \rightarrow T+E \rightarrow I+E \Rightarrow I+T+E \rightarrow I+I+I
$$

**Definition 4.8:**

We say $a\vDash b$ (or $a\Rightarrow b$ and $a\Leftarrow b$), if there are two productions

$$
W \rightarrow aAbB \text{ and } W \rightarrow aBbB
$$

and two corresponding derivations

$$
A \Rightarrow UaZ \text{ and } B \Rightarrow ZbU,
$$

where $Z$ is either $A$ or a single non-terminal.
<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>I</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-Z</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>*</td>
<td>&gt;</td>
<td>Z</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>(</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>)</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>I</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>

**FIGURE 4.9:** The relations for grammar (4.5)
CHAPTER 5

A NEW GRAMMAR FOR RECOGNIZING PARALLELISM

IN ARITHMETIC EXPRESSIONS
5.1 MOTIVATION FOR THE GRAMMAR

In the previous chapters we have said that the aim was to produce a tree like figure 4.8(a), i.e., we would like to balance a tree from left to right. Since the grammar (4.5) is ambiguous, its application may produce several different types of trees for an arithmetic expression, one of which is the one we require. Therefore, we need a method which can only produce a tree like figure 4.8(a) for any expression. At the present time we only use simple operands (variables) and the '+' operation in an expression. It is also assumed that there are no side effects in the evaluation of the operands.

In the process of making a balanced binary tree (B.B.T.) for an expression, we show an expression by $E_n$ where it has $n \geq 1$ operands and $n-1 \geq 0$ operators; i.e.

$$
E_1 = I_1 \\
E_2 = (I_1 + I_2) \\
E_3 = (I_1 + I_2) + I_3 \\
E_4 = ((I_1 + I_2) + (I_3 + I_4)) \\
\vdots \\
E_n = ((I_1 + I_2) + (I_3 + I_4)) + \ldots + I_n.
$$

(5.1)

The trees for $E_n$ where $1 \leq n \leq 6$ are shown in Figure 5.1. To aid the discussion about B.B.T. we define the term 'complete B.B.T'.

Definition 5.1:

We say a balanced binary tree is complete when its left and right subtrees are balanced and equal, i.e. both balanced subtrees have the same height and the same number of operands.

For example the trees in Figures 5.1(b) and (d) are C.B.B.T's. Hereafter whenever an expression makes a B.B.T. we put it in the brackets.

By using the above description, the tree for $E_n$ can be made in the following manner:
FIGURE 5.1: The trees for $H_n$ where $1 \leq n \leq 6$
\[ E_1 = I \]
\[ E_2 = I + I = (E_1 + E_1) = 2E_1 \]
\[ E_3 = (I + I) + I = (E_2) + (E_1) \]
\[ E_4 = E_3 + I = (E_2) + (E_1) + (E_1) = (E_2) + (E_2) = 2E_2 \]
\[ E_5 = E_4 + I = (E_4) + (E_1) \]
\[ E_6 = E_5 + I = (E_4) + (E_1) + (E_1) = (E_4) + (E_2) \]
\[ E_7 = E_6 + I = (E_4) + (E_2) + (E_1) = (E_4) + (E_2) + (E_1) \]
\[ E_8 = E_7 + I = (E_4) + (E_2) + (E_1) + E_1 = (E_4) + (E_2) + (E_1) + E_1 = (E_4) + (E_2) + (E_1) + E_1 = (E_4) + (E_2) + (E_1) + E_1 \]
\[ \vdots \]
\[ E_n = \begin{cases} 2E_{p-1} & \text{where } n = 2^P \\ E_{2^p-1} + E_q & \text{where } 2^{P-1} < n < 2^P \text{ and } q < 2^P - 1 \end{cases} \]

The balanced binary trees for \( E_n \), where \( 2 \leq n \leq 8 \) are shown in Figure 5.2.

From Figure 5.2 we can see that the process can be extended by recognizing the symmetry of binary trees, i.e., when a balanced binary tree is completed, the height of the tree being increased by one and a new balanced subtree will be formed to insert in the right hand side of the previous tree to complete it, otherwise the right subtree is smaller than the left one.

Let us again consider \( E_n \) to see what mathematical order exists for making a B.B.T. for it.

The height of the execution tree for,
\[ \begin{align*}
E_1 & \text{ to } E_2 \text{ is } 1 \\
E_3 & \text{ to } E_4 \text{ is } 2 \\
E_5 & \text{ to } E_8 \text{ is } 3 \\
E_9 = 2^3 + 1 & \text{ to } E_{16} = 2^4 \text{ is } 4 \\
\vdots & \vdots \\
\end{align*} \]

and for \( E_{2^n-1} \) to \( E_{2^n} \) is \( n \).

The insertion of the \((2^n+1)\)th to \(2^{n+1}\)th operands will be in the same manner as the insertion of the first \(2^n\) operands. Thus, when the number
FIGURE 5.2: The B.E.T.'s for E , $2 \leq n \leq 8$ with the right place for each operator.
of operators is $2^n-1$, the tree is a complete B.B.T. of height $n$. But if the number of operations is $m$, $2^{n-1}-1 < m < 2^n-1$, then the tree has a complete B.B.T. as its left subtree and a tree like $E_p + E_q$, where $1 \leq q < p < 2^{n-1}$ as its right subtree.
5.2 THE NEW GRAMMAR

So far, we know of no published grammar which recognizes an arithmetic expression in this manner.

Here we will show a grammar which produces only the trees shown in Figures 5.1 and 5.2.

In Figure 5.2, it has been shown that for \( E_n \) where \( 2^{p-1} \leq n < 2^p \), the tree is \( E_{\frac{2 \cdot n}{2}} + E_{\frac{n}{2}} \) where \( q \leq 2^{p-1} \).

Thus, the \( \left( \frac{n}{2} \right) \) operator is at the top of the tree, and the \( \left( \frac{2^{p-1}}{2} \right) = 2^{p-2} \) operator is one level lower, the \( (2^{p-3}) \) operator is 2 levels lower and so on (see the relation (5.3)).

To show how the grammar works, whenever any B.B.T. is completed, we put it between '(' and ')' (see relations (5.1) and (5.2)). To increase the level of the tree (when there are more operands to add), there should be another B.B.T. as right subtree, not necessarily of the same height as the left one (see Figures (5.1) and (5.2), which is again between '(' and ')'). Therefore, we have two B.B.T.'s as:

\[
\text{(right C.B.B.T.)} + \text{(left B.B.T.)}
\]

To add these two subtrees, ')' + '(' should be reduced to '+'. Thus a rule (production) like \( ++ \) is required.

We claim that the following set of productions, say \( Q \), is able to produce a B.B.T. for any arithmetic expression (including any number of a single associative operator and simple operands);

\[
Q = \left\{ 
\begin{array}{l}
(1) \quad E+a \\
(2) \quad p++ \\
(3) \quad OEC\rightarrow EPE \\
(4) \quad p\rightarrow CPO \\
(5) \quad S\rightarrow C$ \\
(6) \quad PE$\rightarrow CPE$ \\
(7) \quad S$\rightarrow OE$ \\
(8) \quad S$\rightarrow OS$
\end{array}
\right\} \quad (5.4)
\]
The corresponding grammar is: $GPE=\{E,P,O,C,\},\{+,a\},Q,S\}$, where $S$ is the starting symbol, $P$, $O$ and $C$ are used instead of '+','(',')' in order to have only non-terminal symbols at the left hand side of the production. $\$ and $a$ are used as end marker and simple operand respectively.

To change only one symbol in each production, rule (3) should be re-written as the rules (3.a):

- $EEE\rightarrow EPE$
- $EEC\rightarrow EZE$
- $OEC\rightarrow EEC$

**Example 5.1:**

Let $E_7 = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 \$ be an expression.

$E$ should be parenthesised thus:

$$E_7 = ((a_1 + a_2) + (a_3 + a_4) + ((a_5 + a_6) + a_7))$$

in which the right subtree is a C.B.B.T. of height 2 and the left one is a B.B.T. of height 2 and the tree for $E$ itself is a B.B.T. of height 3.

The tree can be constructed as follows:

1. $E_7 = a + a + a + a + a + a + a + \$
2. $\rightarrow EPE + a + a + a + a + a + \$
3. $\rightarrow OEC + a + a + a + a + a + \$
4. $\rightarrow OECPOEC + a + a + a + \$
5. $\rightarrow OEPEC + a + a + a + \$
6. $\rightarrow OECCPEPE + a + \$
7. $\rightarrow OECCPOEC + a + \$
8. $\rightarrow OECPEC + a + \$
9. $\rightarrow OECPECPE + \$
10. $\rightarrow OECPEPE + \$
11. $\rightarrow OECPOEC + \$
12. $\rightarrow OOEPEC + \$
It is clear from line (8) that the two successive E's do not combine because they do not have the same height, i.e. the second E is too low. The term $a_7$ makes another subexpression which combines with the second E in line (10). However, there is a redundant 'C' between the last two E's at line (9). Thus, the rule (6) deletes this 'C' and then the two subexpressions can be combined in line (11). Line (13) has been reduced to line (14) by two applications of rule (5) and line (14) to (15) by two applications of rule (7). In line (14), the number of O's (open brackets) shows the height of the B.B.T. for the expression. Some steps of the derivation tree are shown in Figure 5.3, and the complete derivation tree is shown in Figure 5.4, (where the dotted lines show the context of the rules).

To assist in the parsing of expressions involving two or more operands a delimiter at the beginning of the string is used in addition to that marking the (right-hand) end. Hence in what follows some grammars generate $E$ while others generate $E$. 
* =>
\[
\begin{align*}
O & \quad E & \quad C \\
E & \quad P & \quad E \\
an + a + a + a + a + a + a + a \\
\end{align*}
\]

* =>
\[
\begin{align*}
O & \quad E & \quad C \\
E & \quad P & \quad E \\
E & \quad P & \quad E \\
an + a + a + a + a + a + a + a \\
\end{align*}
\]

* =>
\[
\begin{align*}
O & \quad E & \quad C \\
E & \quad P & \quad E \\
E & \quad P & \quad E \\
O & \quad E & \quad C \\
E & \quad P & \quad E \\
E & \quad P & \quad E \\
an + a + a + a + a + a + a + a \\
\end{align*}
\]

continued ........
FIGURE 5.3: Some step showing the construction of the derivation tree for \( E_7 \).
FIGURE 5.4.1
The derivation tree for E7
5.3 AN EXTENSION OF THE GRAMMAR

The grammar (5.4) can be used for any associative operator. It can also be extended for any expression including more than one type of associative operator. In this section, the grammar will be constructed for an expression including addition (+) and multiplication (*).

Since the precedence of '*' is greater than '+' (see the grammar (4.5) and figure 4.9), whenever these two operators are adjacent in an expression, some productions have to check this property and operate "*" before '+'. Of course, for any subexpression including only one type of operator, the method is reduced to that used before.

We now give some illustrative examples showing how we cope with expressions including both + and *.

In the following examples $T_n$ shows a subexpression which has $(n-1)'*$ operators where $n>1$.

Example 5.2:

Let $E=a_1+a_2+a_3*a_4*a_5$ be an expression. $E$ should be parenthesised thus:

$E = (a_1+a_2)++(a_3*a_4)*a_5$.

Now $a_1+a_2$ makes a B.B.T. of height one. There is a '*' operator after $a_3$, and therefore $a_3$ should operate with $a_4$ and $a_5$, i.e., the left and right hand side operators of each operand are now checked. To add two operands $a_1$ and $a_2$, the following operator must be a '+'. The productions like

$T+a$

$(E)++T+T+$

can check this situation. The second and third operators are '+' and '*' respectively. Therefore '*' operates first and the production

$(T)+T*T$

(5.6)

is used. It is necessary here to say that when an operator with higher precedence is used (here it is '*'), no checking is needed for
the type of the other operators involved unless the operator is left or right associative. Now we have

$$\text{($)$(E_2) + (T_2) \times a_5\text{)}$$

In this state we have + and *, thus $T_2$ and $a_5$ must be multiplied first, but '()' is between $T_2$ and * and so the production

$$\ast T\hat{\rightarrow}\ast T$$

(5.7)

can be used to delete it.

Now the productions

(a) $++\rightarrow(+,$
(b) $\rightarrow\$ ,
(c) $E\rightarrow T$ ,
(d) $(E)\rightarrow E+E$
(e) $E\rightarrow(E$ ,

and (f) $S\rightarrow E$ ,

are used to force the derivation:

$$E \rightarrow $(E)+(T)$$
$$\rightarrow $(E+T)$
$$\rightarrow $(E+T$'
$$\rightarrow $(E+E$
$$\rightarrow $(E)$
$$\rightarrow $(E$'
$$\rightarrow $(E$
$$\rightarrow $(E$
$$\rightarrow $(E$
$$\rightarrow $(E$
$$\rightarrow $(E$
$$\rightarrow $(E$
$$\rightarrow $(E$
$$\rightarrow $(E$

The B.B.T. of minimum height for E is shown in Figure 5.5.
Example 5.3:

Let $X = a_1a_2 + a_3 + a_4 + a_5$. E should be paranthesised thus:

$$X = ((a_1a_2) + (a_3 + a_4)) + a_5.$$

$X$ can be reduced to $S$ in the following manner:

$$X \Rightarrow T*T+T+T+T \Rightarrow (T+T)+T \Rightarrow (E)+T \Rightarrow E + T \Rightarrow E \Rightarrow S$$

The B.B.T. for this expression is shown in Figure (5.6).

This expression is very similar to the one shown in Example 5.2, thus its tree should be similar to the one in that example, but as the Figure 5.6 shows, it is not. This is because $E_2 = a_3 + a_4$ makes a B.B.T. of height one and with $T_2 = a_1a_2$ makes the B.B.T. $T_2 + E_2$ and then $a_5$ is being inserted one level higher. Now we consider the rules which make this tree.

Rules (5.5) and (5.6) reduce the expression $X$ to

$$\Rightarrow (T)+a_3+a_4+a_5 \Rightarrow (T)+(E)+T$$

The rules (5.8) and $(E) \Rightarrow (E)+E$ reduce the above form to:

$$T+(E)+T \Rightarrow (T)+E+T \Rightarrow ((E)+T \Rightarrow ((E)+T \Rightarrow ((E)+E \Rightarrow ((E)+E \Rightarrow ((E)+E \Rightarrow ((E)+$
FIGURE 5.5: The B.B.T. for $E = a_1 + a_2 + a_3 * a_4 * a_5$.

FIGURE 5.6: The B.B.T. for $E = a_1 * a_2 + a_3 + a_4 + a_5$. 
We claim that the following set of productions is able to produce a B.B.T. for any arithmetic expression including any number of two associative operators with different precedence.

In this set of production, like the grammar (5.4), O, C, P and M are used instead of ',(', ')', '+' and '*' respectively in order to have only non-terminal symbols at the left hand side of the production and hence can form the context sensitive classifications.

\[
\begin{align*}
1. & \quad T \rightarrow a \\
2. & \quad P \rightarrow P \ \\
3. & \quad M \rightarrow M \ \\
4. & \quad OTC \rightarrow TMT \\
5. & \quad OEC \rightarrow EPTP \\
6. & \quad OEC \rightarrow EPEP \\
7. & \quad OEC \rightarrow TPEP \\
8. & \quad OEC \rightarrow TPPTP \\
9. & \quad OEC \rightarrow OEPEC \\
10. & \quad OEC \rightarrow OTEPC \\
11. & \quad OEC \rightarrow OTPTC \\
12. & \quad $TP \rightarrow TP$ \\
13. & \quad M \rightarrow CMD \\
14. & \quad P \rightarrow CPO \\
15. & \quad MTCP \rightarrow CMTP \\
16. & \quad PEP \rightarrow CPEPO \\
17. & \quad PEP \rightarrow CPITO \\
18. & \quad OEP \rightarrow OEPO \\
19. & \quad OT$ \rightarrow TME$ \\
20. & \quad MT$ \rightarrow CMT$ \\
21. & \quad $ \rightarrow C$ \\
22. & \quad E$ \rightarrow T$
\end{align*}
\]
In the above examples, there are only two sub-expressions, and the B.B.T.'s are not very complicated.

In the following examples, some expressions with more than two sub-expressions are considered.

Example 5.4:

Let $X = a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+...
Some steps of constructing the parse tree for \( E \) are shown in Figure 5.7.

The combination of the sub-expressions \( E_5 \) and \( T_2 \) show that, since the height of \( T_2 \) is less than \( E_5 = E_4 + E_1 \), and there is a vacancy at the right hand side of \( E_5 \) (see Chapter 2), \( E_1 \) and \( T_2 \) are joined and make a B.B.T. to combine with \( E_4 \). In this case, two sub-expressions \( E_5 \) and \( T_2 \) add together without increasing the height of the tree. The same situation arises in \( E_6 = E_4 + E_2 \) and \( T_2 \), i.e. \( E_2 \) and \( T_2 \) are combined first and then \( E_4 \) is joined to their resulting tree. This means that, the grammar (5.10) not only produces a B.B.T. for any sub-expression, it also systematically looks for some vacancies in a sub-expression into which another smaller sub-expression can be inserted.

Example 5.5:

Let \( E = \$T_S + E_5 + T_4 + E_2\$ \), be an expression with four sub-expressions \( T_S, E_5, T_4 \) and \( E_2 \). By using the grammar (5.10), \( T_S \) and \( E_5 \) make a B.B.T. of height four and \( T_4 \) and \( E_2 \) make one of height three. These two trees are joined to produce a tree of height 5 (see Figure 5.8).

This example shows that the grammar can check successive sub-expressions combining the smaller sub-trees first and then a combined tree of these two with the third. Therefore it joins \( T_4 \) and \( E_2 \) first which both have a lower height than \( T_S + E_5 \) and then adds the produced tree \( T_4 + T_2 \) to the tree \( T_S + E_5 \).

The grammar (5.10) may be improved to produce a tree of lower height (whenever possible). It may also be possible to reduce the number of productions in the grammar.
\[ E = \$a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a \$ \]

\[ \Rightarrow \]

\[ \$ a + a + a + a \$ \]

\[ \Rightarrow \]

\[ \$ a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a \$ \]

\[ \Rightarrow \]

\[ \$ a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a \$ \]

\[ \Rightarrow \]

\[ \$ a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a \$ \]

**FIGURE 5.7**: Some steps of building a B.B.T. for \( E \) by using the grammar (5.10).
FIGURE 5.8: Representation of the tree for $E = T_8 + E_5 + T_4 + E_2$ by using the grammar (5.10)
5.4 **FURTHER EXTENSION OF THE GRAMMAR**

An important problem in constructing a compiler for a non-trivial programming language is the syntactic analysis which is specified by formulae in the language (e.g. BNF).

As mentioned earlier there is a significant and useful class of formal languages called precedence languages. Some existing programming languages are in fact precedence languages, and many of them are almost precedence languages; apart from a small number of terminal character pairs where more than one precedence relation exist.

Floyd (1963) produced a precedence grammar which defines a language closely comparable to ALGOL 60, and contains none of the syntactic ambiguities of ALGOL 60. However, it is much more concise (using 43 non-terminal characters rather than 109), and assigns more natural constructions to several grammatical forms.

Another classification of formal grammars is the set of so-called operator grammars. In these grammars some or all of the three relations '>', '<' and '=' may hold between two terminal characters (see Example 4.7 in Chapter 4). It is therefore probable that such a grammar could be used, in a similar way to our modified operator precedence grammars, to detect parallelism within a program.

The grammar (5.10) is an operator grammar which includes two associative operators '+' and '*'. It is feasible to extend it for the logical and set theoretical associative operators such as 'or' and 'AND', and union and intersection.

If we can extend the grammar (5.10) to generate a programming language like ALGOL 60 or FORTRAN, then some of the parts of the language which can be executed in parallel will be detected during the syntax analysis stage of the compiling process.
CHAPTER 6

PARSING ALGORITHMS FOR CONTEXT SENSITIVE GRAMMARS
6.1 GENERAL PARSING METHODS

In previous chapters we described how a context sensitive grammar can be used to define the syntax of an arithmetic expression for a parallel programming language.

Here we now show how to check whether an input string is a sentence of the given language and how to construct the parse tree for the string.

Definition 6.1:

A parser for a grammar G, is a program that takes as input a string W and produces as output either a parse tree, if W is a sentence of L(G), or an error message indicating that W is not a sentence of L(G).

In other words, we say a sentence of a language has been parsed when we know one of its derivation trees.

There are many parsing algorithms for context-free grammars. Two basic types of parsers are bottom-up and top-down and all the parsing algorithms are based on these two methods.

6.1.1 Top-down Parsing

Top-down parsers attempt to produce a parse tree for the input string from the top and the procedure works down to the leaves. It begins with a tree containing one node, which is the starting symbol of a grammar and comes down until it reaches the leaves which make the input string.

6.1.2 Bottom-up Parsing

This is another general approach, that is in a sense opposite to that of top-down parsing. Bottom-up parsing starts with the leaves (the input symbols) and attempts to build the tree upwards, towards the starting symbol.

Both these methods (with many different algorithms) have been discussed for C.F.G.'s (Aho, Ullman and Gries).
6.2 A SURVEY OF SOME METHODS FOR THE PARSING OF CONTEXT-SENSITIVE LANGUAGES

A few methods have previously been proposed for the parsing of context-sensitive languages.

Descriptions of some of these algorithms are given in the following subsection.

6.2.1 Woods's Algorithm

The algorithm proposed by Woods (1970) is based on a C.S.G. in which the general format for rules is

\[ \alpha X \beta \rightarrow \alpha \gamma \beta \]

where \( X \in N, \alpha, \beta \in V^* \) (\( V = N \cup T \) is called the vocabulary) and \( \gamma \in V^+ \). \( \alpha \) and \( \beta \) specify the context of the rule and the string \( \gamma \) is called the constituent of the rule.

In context-free analysis, a given reduction can not be performed until all its constituents have been formed. In context-sensitive analysis, a given reduction must wait not only for all its constituent and context symbols to be formed but also for any reduction to be performed which may require some of the constituents of this rule for their context, because when a reduction has been performed, its constituents are no longer available.

The level of any reduction is called the start time for that symbol, and the analysis can be represented by any ordering of the reduction in which the level of the reduction is monotonically increasing. Furthermore, ordering the reductions within a level in order of their left to right positions in the string gives a canonical order for listing the reductions used in an analysis.

If we know the start times of all reductions which use a symbol for its context, then we can compute the time at which the symbol becomes available to be used as a constituent. This number is one greater than the
maximum of its own start time and the start times of all reductions (if any) which use this symbol in the context. This number is called the "maturation" level for the symbol, and indicates the level at which it becomes available for use as a constituent.

The "birthday" of a symbol is the time at which it becomes available for use as a context, i.e., as soon as it is created, or one unit later than the start time of the reduction which forms it. By convention, the values of birthday and maturation for terminal symbols are 0. When this information is available then the level for a given reduction is the maximum of the birthdays of its context symbols and the maturation levels of its constituents, i.e., since the context will not be changed, they can be used as soon as they are created, but the constituent symbols cannot be reduced to the non-terminal symbol on the left hand side of the corresponding rule until any other reductions which use them as context have been computed.

Figure 6.1 shows a context sensitive grammar and a parse tree for one of its strings, abc, (Woods, 1970). The birthday and maturation level of the symbols are shown as a pair of superscripts. The maturation level is the second of the pair and is always greater than or equal to the birthday. In this figure, the solid arcs connect each non-terminal symbol on the left hand side of the rule to its constituents, while dotted arcs connect it to the symbols (if any) to specify the contents.

Woods considers a non-deterministic parser. Such a parser accepts a string if any combination of its possible choices perform the reduction of that string to S. The parser operates as follows: the input string is loaded into a working storage with a birthday and maturation level of 0 assigned to each symbol, and a variable called the parser level is set to 0.

For each parser level the algorithm scans the working string from left
(a) A context sensitive grammar

(b) A context sensitive structure diagram with birthdays and maturation levels indicated.
to right, and at each position it runs through all the rules to see if any of them can apply at that position. If it is so, the maximum value of the birthdays of its context symbols and the maturation levels of its constituents is computed. If this number is not equal to the current parser level, then the reduction is not performed and scanning is continued; if this value is equal to the parser level, then the parser has three choices as follows:

a. It can perform the reduction, increase the parser level by 1, and begin scanning from the beginning of the working string.
b. It can perform the reduction and continue scanning at the same level looking for further reductions at that level.

or
c. It can continue scanning as if the reduction did not apply.

When a reduction is performed at the same level \( \ell \), the new symbol is given a birthday and maturation level of \( \ell + 1 \), and the maturation level of all the context symbols are increased to \( \ell + 1 \). The algorithm halts when the parser attempts to scan beyond the end of the working string at some level.

At the moment that a given reduction is performed, the reductions which may use some of its constituents as contexts are those which have already been performed (since their constituents will be replaced by this reduction and will not be available for later reduction).

The proposed context-sensitive recognition algorithm follows each of the possible paths which the non-deterministic parser could take and will find each analysis of the input string once and once only.

This algorithm uses a list of the grammar rules, a working string, a pushdown stack, a pointer \( i \) which marks the scanned position in the working string, a pointer \( r \) which marks the position in the rule list and a register \( \ell \) which contains the parser level.

When a reduction is performed a copy of the working string with the
value of \( i \), the number of the rule which has been applied and the level \( \ell \) 
is loaded into the top of the pushdown stack, in order that we may be able 
to pursue the other two alternatives later.

If the string has been reduced to the single symbol \( S \), then an analysis 
has been successful and the pushdown stack at this point contains a complete 
record of the reductions performed. If at any stage, the level of the last 
reduction is less than \( \lambda \) (the parser level ) then there are no reductions 
on the stack with level \( \ell \). Hence the parser level can be decreased to \( \ell-1 \) 
to continue with the second alternative of the non-deterministic parser and 
the working string is replaced by the copy which was saved on the stack, Thus 
the string is restored to its original state before the reduction was 
attempted. The top entry on the pushdown stack is then deleted. If, on 
the other hand the level of the last reduction is equal to \( \lambda \), then the 
second alternative of the parser for that reduction must already have been 
considered. Hence, the reduction is undone and the third alternative of the 
parser is pursued. Since all three choices for the reduction have been 
considered, it can now be deleted from the stack. Thus, the algorithm 
pursues each possible path of the non-deterministic parser and finds each 
analysis of the input string once.

The operation of this algorithm on the string abc for the grammar of 
Figure 6.1(a) is represented in Figure 6.2. Solid lines point to the push-
down stack represent the adding of an entry to the stack prior to 
performing a reduction.

We changed this algorithm (slightly) in order to parse the sentences 
of a grammar which might reduce the constituents to more than one non-
terminal at the left hand side of a production. In this case, the 
constituents of the reduction are replaced by more than one non-terminal 
with birthday and maturation level of \( \ell+1 \). The parse tree for the string and
Working string

i,r,& pushdown stack i,r,&

<table>
<thead>
<tr>
<th>Working string</th>
<th>begin at</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>1,1,0</th>
<th>1,7,0 + a</th>
<th>00 00 00</th>
<th>1,7,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A b c 00</td>
<td>begin at</td>
<td>1,1,1</td>
<td>1,6,1 + A b c 00</td>
<td>1,6,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B 22 c 00</td>
<td>begin at</td>
<td>1,1,2</td>
<td>1,6,1 + A b c 00</td>
<td>1,6,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A b c 00</td>
<td>continue</td>
<td>1,7,0 + A b c 00</td>
<td>1,7,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A b c 11</td>
<td>begin at</td>
<td>1,1,1</td>
<td>1,6,1 + A b c 11</td>
<td>1,6,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B 22 c 11</td>
<td>begin at</td>
<td>1,1,2</td>
<td>A b 22 c 11</td>
<td>2,5,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B 23 c 33</td>
<td>begin at</td>
<td>1,1,3</td>
<td>A b 23 c 33</td>
<td>2,4,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A C 44 c 34</td>
<td>begin at</td>
<td>1,1,4</td>
<td>A b c 44 34</td>
<td>2,3,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A C B 55</td>
<td>begin at</td>
<td>1,1,5</td>
<td>A b c 45 55</td>
<td>1,2,5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B 55</td>
<td>continue</td>
<td>1,2,5</td>
<td>A b 45 55</td>
<td>1,2,5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A C 55</td>
<td>continue</td>
<td>1,2,5</td>
<td>A b c 45 55</td>
<td>1,2,5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A C 34</td>
<td>continue</td>
<td>2,3,4</td>
<td>A b c 44 34</td>
<td>2,4,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B 33</td>
<td>continue</td>
<td>2,4,3</td>
<td>A b 23 33</td>
<td>2,5,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B 33</td>
<td>continue</td>
<td>2,5,2</td>
<td>A b 23 33</td>
<td>2,5,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(sucessful analysis)

print out contents of pushdown stack

end

continued............
FIGURE 6.2: Analysis of the string abc
the content of the pushdown stack of the new algorithm for the string $a+a+a+a$ for the grammar (5.4), after halting is represented in Figures 6.3 and 6.4 respectively, (in Figure 6.3, the symbol $\dagger$ shows that some other reductions are made and then deleted.)

6.2.2 The Kuno Algorithm

The Kuno algorithm (1967) takes a context sensitive grammar in the usual notation and finds every analysis of a string once. This algorithm has a number of features notably the avoidance of repetitive analysis of sub-strings for different analyses of the entire string. This makes it extremely efficient on some grammars. The algorithm is not specified in detail and contains steps such as "find a sequence of registers containing register $p$" and other similar operations whose enumeration procedure is not clear. Nevertheless it would seem that reasonably efficient enumeration and search procedures could be devised.

The Kuno and Woods algorithms reflect a basic difference in parsing strategy.

The Kuno algorithm works in parallel on all analyses at once, avoiding the recomputing of substring analyses by saving the analyses of each sub-string. The Woods algorithm, pursuing one alternative analysis at a time, remembers those points along the way where a choice was made and returns to pursue them later.

The major advantage of the Kuno parallel approach is the time saving. The disadvantage is the increased amount of storage space required to save all the sub-string analyses. In Woods's algorithm space is saved by forgetting things and recomputing them. The matching rule applies to a complete intermediate string and the context checking is a straightforward match.

This algorithm is basically an efficient approach except for the effort
FIGURE 6.3: The pushdown stack after the algorithm halts
FIGURE 6.4: The parse tree for $E = a + a + a + a^*$ with the birthday-and maturation levels indicated.
required for the context checking. It seems that the Kuno and Woods algorithms are somewhat complementary to each other, in the sense that one tends to do well on grammars for which the other performs poorly and vice-versa.

We can say that the Kuno algorithm is better for grammars which do not involve a great deal of context dependence.

6.2.3 Tanaka and Fu's Algorithm

Tanaka and Fu (1977) have extended the Cocke-Kasami-Younger's algorithm which parses a context-free grammar (the essence of which is the construction of a triangular parse table T, whose elements are $t_{ij}$, for $1 \leq i \leq n$ and $1 \leq j \leq n-i+1$, where $n$ is the length of the string. The input string belongs to the related language if and only if the start symbol is in $t_{1n}$). This algorithm can parse the strings of the grammars where productions have the following forms:

1. $A \rightarrow a$
2. $A \rightarrow BC$
3. $AB \rightarrow CB, AB \rightarrow AC$
4. $AB \rightarrow C$

(6.1)

where $a \in \Sigma$ and $A, B, C \in \mathbb{N}$.

Therefore, this algorithm can not parse either of the grammars (5.4) or (5.10) which have the following sets of productions:

\{
    (1) $E \rightarrow a$
    (2) $P \rightarrow$
    (3) $OE \rightarrow EPE$
    (4) $P \rightarrow CPO$
    (5) $\rightarrow C$
    (6) $PE \rightarrow CPE$
    (7) $S \rightarrow OE$
    (8) $S \rightarrow OS$
\}

(5.4)
and

\{ (1) \ T \mapsto a , \\
(2) \ P \mapsto , \\
(3) \ M \mapsto * , \\
(4) \ OTC \mapsto \text{TMT} , \\
(5) \ OECP \mapsto \text{EPTP} , \\
(6) \ OECP \mapsto \text{EPEP} , \\
(7) \ OECP \mapsto \text{TPEP} , \\
(8) \ OECP \mapsto \text{TPTP} , \\
(9) \ OECC \mapsto \text{OEPEC} , \\
(10) \ OECC \mapsto \text{OTPEC} , \\
(11) \ OECC \mapsto \text{OTPTC} , \\
(12) \ $TP \mapsto $TP \mapsto , \\
(13) \ M \mapsto \text{CMO} , \\
(14) \ P \mapsto \text{CPO} , \\
(15) \ MTCP \mapsto \text{CMTP} , \\
(16) \ PEP \mapsto \text{CPEPO} , \\
(17) \ PEP \mapsto \text{CPTPO} , \\
(18) \ OEP \mapsto \text{OEPO} , \\
(19) \ OT$ \mapsto \text{TM$} , \\
(20) \ MT$ \mapsto \text{CM$} , \\
(21) \ + \mapsto C$ , \\
(22) \ E$ \mapsto T$ , \\
(23) \ OEC$ \mapsto \text{EPE$} , \\
(24) \ OEC$ \mapsto \text{TPE$} , \\
(25) \ PT$ \mapsto \text{CPE$} , \\
(26) \ E$ \mapsto OE$ , \\
(27) \ S$ \mapsto $S$ \\
\} . (5.10)
6.3 A LEFT TO RIGHT PARSING ALGORITHM (L.R.)

As stated earlier, in a context-sensitive analysis a given reduction must wait for two things:

1. For all its constituents and context symbols to be formed.
2. For any reduction to be done which may require some of the constituents of this rule for its context.

Since the second case does not happen in the grammar (5.4), there is no need for the most of the evaluation and memory locations used in the Woods' algorithm. The algorithm makes use of a list of the grammar rules, a working string, a pointer POS (which marks the scanned position in the working string), a pointer I (which marks the position in the rule list), a pointer TOS (which marks the end of the working string after each reduction), the integer variable TOP (which shows the number of rules) and a boolean variable FOUND (which becomes TRUE whenever a part of working string matches the right hand side of a rule, otherwise it is FALSE).

Assuming that the input string has been loaded into the working string, the operation of the reduction of the working string to the start symbol S is as follows:

begin
    TOP=number of rules
    POS=1
    TOS=N (the number of symbols of the input string)
    while working string ≠ S$ do
        POS=1
        while POS<TOS do
            begin FOUND=False
                for I to TOP while not FOUND do
                    begin
                        compare the right hand side of the rule [I] with the working string beginning at position POS
                        if it matches then FOUND=True;
replace the matched part of the working string
by the symbol(s) at the left hand side of the
rule [I];
adjust POS (the first position after the reduction
in the working string)
end;
if not FOUND and POS<TOS
then POS+POS+1
end.

The explanation of the algorithm is as follows:
It starts scanning from the beginning of the string, checks the current rule
with the current position and, if it applies, the reduction is performed. It
continues scanning from the position after the reduction. This process is
carried on until the end of the string. Then it starts again from the first
position of the string using the above process.

If the string has been reduced to the symbol S, followed by $, then an
analysis has been found.

This algorithm can parse these C.S.G.'s in which the constituents of each
production are not used as the context of any other rules, in other words, a
given reduction is not bound to wait for any reduction to be formed which may
require some of the constituents of this rule as the contexts. (Like the
grammar (5.4)).

The operation of the deterministic algorithm and the parse tree for the
string a+a+a+a$ of the grammar (5.4) are given in Figures 6.5 and 6.6.
\[
a+a+a+a^$ \Rightarrow E+a+a+a^$
\Rightarrow EPa+a+a^$
\Rightarrow EPE+a+a^$
\Rightarrow EPEPa+a^$
\Rightarrow EPEPE+a^$
\Rightarrow EPEPEPa^$
\Rightarrow EPEPEPE^$
\Rightarrow OECEPEPE^$
\Rightarrow OECEOEC^$
\Rightarrow OEEP^$
\Rightarrow OEPE^$
\Rightarrow OECE^$
\Rightarrow OE^$
\Rightarrow OS^$
\Rightarrow S^$
\]

**FIGURE 6.5**: The parsing of \(a+a+a+a^\$\) using the left to right parsing algorithm
FIGURE 6.6: The parse tree for a+a+a+a$ by the left to right parsing algorithm.
6.4 A LEFT-MOST (CANONICAL) PARSING ALGORITHM

This algorithm scans the string from left to right and at each position runs through all the rules to see if any of them can apply to the string at that position. For each reduction which can apply, it performs the reduction and commences the scanning from the beginning of the working string.

If it runs through all the rules and none of them apply to the string, it means that the string does not belong to the language, otherwise the operation continues until the string reduces to the start symbol S.

The operation of this parser is very similar to the Left to Right Parsing Algorithm (see Section 6.3), except that after each reduction it commences scanning from the beginning of the string.

The operation of the algorithm (which is again deterministic) and the parse tree for the string $a+a+a+a$ of the grammar (5.4) are represented in Figures 6.7 and 6.8.
a+a+a+a$ $\Rightarrow$ E+a+a+a$

$\Rightarrow$ EP+a+a$
$\Rightarrow$ EPE+a+a$
$\Rightarrow$ OEC+a+a$
$\Rightarrow$ OECP+a+a$
$\Rightarrow$ OECPE+a$
$\Rightarrow$ OECPEPa$
$\Rightarrow$ OECPEPE$
$\Rightarrow$ OECPOEC$
$\Rightarrow$ OEPEC$
$\Rightarrow$ OEEC$
$\Rightarrow$ OEECC$
$\Rightarrow$ OOECC$
$\Rightarrow$ OOECC$
$\Rightarrow$ OS$
$\Rightarrow$ S$

FIGURE 6.7: The parsing of a+a+a+a$ using the left-most parsing algorithm
FIGURE 6.8: The parsing of $a+a+a+a$ using the leftmost parsing algorithm.
6.5 A COMBINED ALGORITHM FOR PARSING

6.5.1 Introduction

In Chapter 5, we showed that the grammar (5.4) is only able to produce a balanced binary tree for arithmetic expressions using any number of a single associative operator and simple operands. An extension of the grammar (5.4), is the grammar (5.10). This is able to produce a balanced binary tree for any expression with any number of two different associative operators (of different precedence), and simple operands. Hence, the grammar (5.10) is more general than the grammar (5.4).

It might be possible to find an efficient algorithm to parse the grammar (5.10) and its extension to include any expression incorporating all the various arithmetic operations.

The Woods's algorithm is very complicated and uses a pushdown stack to keep all the reduced forms of the working string so that any reduction is unnecessary. Then the parser can look at the pushdown stack, remove all the redundant reductions and start again from the right position. If the string is very long or (and) the number of the rules in a grammar is very large, then the algorithm requires too many memory locations and extra time.

The Kuno algorithm is more efficient than Woods's for a grammar with only a few context-sensitive productions. However, when more contexts occur in the rules, the algorithm requires more context checking; and this is where Woods's method begins to benefit from the use of the contexts to direct the course of the parsing. In the extreme case where the contexts of the productions are sufficient to determine the sequence of reductions in an analysis, Woods's algorithm proceeds directly to each analysis, while Kuno's has to do a great deal of context checking and (or) a large number of spurious analyses.

The left to right algorithm can parse strings generated by the grammar
which have one type of associative operator. The left-most parsing algorithm can recognize strings generated by both the grammars (5.4) and (5.10). These two algorithms are very simple and they do not require too much storage (the pushdown stack in the Woods's algorithm), and the time for them to reduce a string to the starting symbol (S) is less than the other algorithms. This is because of the properties of the grammars (5.4) and (5.10). Although they are context-sensitive, a recognizer which can parse their string is not necessarily as powerful as the recognizer for the general C.S.G's.

6.5.2 The Algorithm

The set of strings generated by the grammar (5.4) is a subset of that generated by (5.10) and the left to right parsing algorithm can parse the strings generated by the grammar (5.4) but not the ones generated by the (5.10). Hence, it is possible to make a recognizer by combining these two algorithms to operate in the following manner:

Step 1: Scan the string and whenever it sees two different adjacent operators, mark their position and their precedence.

Step 2: Then, for each substring (including only one type of associative operator) use the left to right parsing algorithm to reduce it as much as possible (but not necessarily to one sub-expression). Now we have some sub-expressions enclosed in the number of parentheses (according to their level).

Step 3: The parser starts scanning from the beginning of the string by using again the left to right algorithm reducing all )op( combinations to op, (see the productions (13) and (14) of the grammar (5.10)), and carry on this process until reaching the end of the string.
Continue steps 2 and 3 until the string becomes \$S\$. This indicates that the string belongs to the language generated by the grammar.

The following example makes the operation of this algorithm clear.

**Example 6.1:**

Let the string \(W\) be

\[a_1 + a_2 + a_3 + a_4 + a_5 + a_6 \cdot a_7 + a_8 + a_9 + a_{10} + a_{11} + a_{12} + a_{13} + a_{14} + a_{15}\]

The recognizer parses the string in the following manner:

Start scanning from the beginning of the string and mark the `+` and `*` operators at the left and right hand sides of \(a_6, a_7, a_{14}\).

Now use the left to right parsing algorithm to reduce the substring,

\[a_1 + a_2 + a_3 + a_4 + a_5\]
\[a_6 \cdot a_7\]
\[a_8 + a_9 + a_{10} + a_{11} + a_{12} + a_{13}\]
and
\[a_{14} \cdot a_{15}\]
to \(\text{ECCPTP, OTC, POOECCPOECP and OT}\) respectively.

Now we have the string,

\(\text{ECCPTPOTCPOOECCPOECPOT}\).

Again, using the left to right algorithm the following reductions are performed at each step:

\(\text{ECCPTPOTCPOOECCPOECPOT}\)
\[\Rightarrow \text{ECPPTPOECPPEPT}\]
\[\Rightarrow \text{ECPPOECPPOECPPEPT}\]
\[\Rightarrow \text{EPEPEPECPOECP}\]
\[\Rightarrow \text{OECEPEPE}\]
\[\Rightarrow \text{EPEP}\]
\[\Rightarrow \text{OEC}\]
\[\Rightarrow \text{E}\]
\[\Rightarrow \$S\]
All the reduction operations are given in the parse tree in Figure 6.9. We have also parsed this string by the left most parsing algorithm, and the parse tree is given in Figure 6.10.
FIGURE 6.91 The parse tree by the combined algorithm.
FIGURE 6.10: The parse tree by the left most parsing algorithm.
6.6 NON-CONTEXT-FREE GRAMMARS GENERATING CONTEXT-FREE LANGUAGES

It is well known that the family of languages generated by arbitrary phrase structure grammars is precisely the family of recursively enumerable sets, and those grammars with only context-free rules generate context-free languages, which form a proper subset of recursive sets. However, the way in which the rules containing restrictions interact to generate non-context-free languages is not well understood.

The grammar (5.10) is a C.S.G., but the strings generated by this grammar can also be generated by a C.F.G. (see Chapter 4). We have also shown that the parser for this string is not as powerful as the one for general C.S.L's. It would seem that this is either because this C.S.G. generates a C.F.L. or because the rules of this grammar have at most one right and (or) left context.

Consider now the first case. Some investigators have derived sufficient conditions relating to the form of the rules of a non-C.F.G. G; to guarantee that L(G) is context-free (Baker (1974), Aggarwal (1979)), i.e., the addition of certain context-sensitive rules does not increase the generative power of a grammar. Two of these results are presented below.

Baker (1974) has proved that grammars obeying a particular weak restriction on the form of rules generate only context-free languages. The restriction is that in each non-context-free rule, the right side must contain a string of terminals longer than any terminal string appearing between non-terminals on the left side of the rule. He shows that previous results by other investigators all deal with aspects of the same mechanism and therefore, can be obtained as corollaries of his theorem.

Aggarwal (1979) introduced a new type of grammar (SVMT-bounded grammars) which can generate only context-free languages and he has demonstrated that certain grammars satisfying this property do not satisfy
previously obtained conditions which guarantee the generation of a C.F.L. Therefore, this property enlarges the class of grammars generating only C.F.L.'s. Here we define a strict partial ordering on a set and the conditions for the SVMT-bounded grammars.

Definition 6.2:
A strict partial ordering on the set \( V = N \cup T \) is a relation \(<\) on \( V \) such that:
1. \(<\) is transitive, that is, \( x < y \) and \( y < z \) imply \( x < z \).
2. \( x \not< x \) for all \( x \in V \) (i.e. \(<\) is irreflexive).

Definition 6.3:
If \( G = (N, T, P, S) \) is a grammar, the string \( \alpha \) is a "SVMT-string" (single variable or multiple terminal string) if \( \alpha \in V \cup T^* \).

Definition 6.4:
Suppose that \( G = (N, T, P, S) \) is a grammar and \(<\) is a strict partial ordering on \( V \cup T^* \) then, \(<\) is an "SVMT-ordering of \( G \)" if:
1. \( A < \alpha \) for \( \alpha \in N \cup T^+ \),
2. \( A < x \) for all \( A \in N \) and \( x \in T^+ \),
3. \( x < y \) for all \( x, y \in T^* \) such that \( |x| < |y| \),
and 4. whenever \( A < B \) and \( B \in N \), then \( A \in N \cup \{ \lambda \} \).

Definition 6.5:
If \( G = (N, T, P, S) \) is a grammar and \( \alpha \in V^+ \), then the "SVMT-normal form" of \( \alpha = \alpha_1 \alpha_2 \ldots \alpha_n \) where each \( \alpha_i \), \( 1 \leq i \leq n \) is a non-empty SVMT string, and if \( n \geq 2 \) for all \( i, 1 \leq i \leq n-1 \), at least one of the two strings \( \alpha_i \) and \( \alpha_{i+1} \) is a single non-terminal.

Definition 6.6:
If \( G = (N, T, P, S) \) is a grammar and \(<\) is an SVMT-ordering on \( G \), then the
rule $\alpha_1 \alpha_2 \ldots \alpha_n \beta_2 \ldots \beta_m$ (written in SVMT-normal form) is "SVMT-bounded with respect to $<$", if there exists a $k$, $1 \leq k \leq m$ such that $\alpha_i \beta_k$ for all $i$, $1 \leq i \leq n$.

Definition 6.7:

If each rule in $P$ of the grammar is SVMT-bounded with respect to $<$, then $G$ is a SVMT-bounded grammar.

In the grammar (5.10) we have:

$.O,E,C<P<T<M$ and $S<$E<0

but $P<P$ and $T<T$ according to rules 6 and 12 of the grammar. Hence, if this grammar generates a C.F.L., then the partial ordering can be reflexive (i.e. $A<A$).

Although the grammar (5.10) violates the SVMT-bounded condition, it nevertheless seems to generate a C.F.L. This is worthy of further investigation.
CHAPTER 7

SOME INCONSISTENCIES IN DIFFERENT FORTRAN COMPILERS
7.1 INTRODUCTION

In a parallel processing environment it is important to be aware of the possibility of a variable being accessed by more than one process at any one time. By considering the usage of variables it is possible to determine where processes may be executed in parallel. Also an awareness of the use of a variable will enable programmers to express the parallelism in their program, as well as allowing the correctness of the parallel constructs to be tested.

The aim of this chapter is to investigate the possibility of using extra information in parallel execution of other parts of a program. For this purpose we choose FORTRAN which is one of the oldest, and one of the most widespread programming languages.

There are many different FORTRAN compilers in everyday use. Several programs have been run on some compilers and since the output of each program on each compiler is different there is no way to show which one is right or wrong. Even if these problems are never resolved, it is important to list them, so that constructs that may give rise to errors can be avoided, not only in parallel execution but also in sequential execution.

To aid discussion of these points we define some terms:

Definition 7.1:

Two implementations of a language are inconsistent if their outputs for any program are not equal.

Inconsistencies are assumed to be "legal" or "illegal but acceptable". We say that the output of a program is legal if it satisfies the USA FORTRAN Standard, and is illegal but acceptable if it does not.
Definition 7.2:

We say there is a "conflict" between two answers from two equivalent program segments, on one compiler, when we expect the same answer from both, but the answers are different and we do not know which one is right.

The different outputs of one program from different compilers is the effect of the semantic analysis of each compiler. We say that there is discrepancy between the semantic analysis of compilers and that this might be considered as an ambiguity (in FORTRAN) which causes the production of different parse trees for the same sentences on different compilers.

Here some of the most important points will be considered and a brief description of each FORTRAN compiler used will be given.

Finally, another aim of considering these inconsistencies is to show that they disclose a lack of proper definition of certain statements in FORTRAN 66 (STANDARD FORTRAN) and serious discrepancies in their implementation in various widely used compilers.
7.2 SOME FORTRAN COMPILERS

To investigate some of the most important inconsistencies the following FORTRAN compilers have been used:

A. ICL 1900 FORTRAN (unoptimised)

This compiler is the extended FORTRAN based on U.S.A.S.I. FORTRAN (formerly A.S.A. FORTRAN) as defined in USA Standard X3.9 - 1966 and used on 1900 machines.

B. SOFOR FORTRAN Compiler

SOFOR is a fast FORTRAN compiler written by Southampton University (hence Southampton FORtran). It provides for the fast processing of small FORTRAN programs and is mostly used for teaching.

SOFOR compiler is much faster than the ICL compilers, as the source program is compiled directly into a binary program without access to subroutine libraries.

The FORTRAN language implemented by SOFOR is more restricted than that implemented by the ICL compilers.

C. ICL 1900 Optimizing FORTRAN Compiler

This optimizing compiler attempts to produce the fastest program from the source statement given to it. The cost of having a faster object program is the longer compilation time.

The optimizing techniques listed below are some of those used at the present time.

1. DO loop optimization
2. Statement block optimization
3. Function reference optimization

Knuth (1971) and Robinson and Torson (1976) have carried out empirical studies on FORTRAN programs and suggested several types of optimization.
The side effects of the optimization may be responsible for some of these inconsistencies, but this matter is not clear yet, because the optimization should not affect the output of program.

In the following section the ICL 1900, SOFOR and ICL 1900 optimized FORTRAN compilers are referred to as A, B and C respectively.
7.3 **INCONSISTENCIES**

Some of the most important inconsistencies which exist in FORTRAN compilers occur in the areas listed below:

1. The exact sequence occurring during the evaluation of a "DO loop".
2. The value of the control variable after execution of a DO loop.
3. The evaluation of the DO-loop parameters in a certain compiler.
4. An unclear point about assignment statements.
5. "call by name (substitution)" in the FORTRAN language.

In the following subsections each of them will be considered.

7.3.1 **The Exact Sequence Occurring during the Evaluation of a DO Loop**

The exact sequence occurring during the evaluation of a DO loop may be different for each compiler. Before considering these discrepancies, an exact definition of DO Statement (DO loop) in the USA. FORTRAN Standard will be given.

**Definition 7.3:**

"A DO statement is one of the form

\[
\text{DO } n \quad I = m_1, m_2, m_3 \\
\text{or} \\
\text{DO } n \quad I = m_1, m_2
\]

where:

1. \( n \) is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GOTO of any form, arithmetic IF, RETURN, STOP, PAUSE or DO statement nor a logical IF containing any of these forms.
2. \( I \) is an integer variable name; this variable is called the control variable.
3. \( m_1 \), called the initial parameter, \( m_2 \), called the terminal; and \( m_3 \),
called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m3 is not explicitly stated, a value of one is implied for the incrementation parameter. At the time of execution of the DO statement, m1, m2 and m3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special...

A DO statement is used to define DO loop. The action succeeding execution of a DO statement is described by the following six steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.
2. The range of the DO is executed.
3. If control variable reaches the terminal statement, then after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.
4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, then the action described starting at step 2 is repeated, with the understanding that the range in question is that of the DO, whose control variable has been most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, then the DO is said to have been satisfied and the control variable becomes undefined.
5. ...
6. Upon exiting from the range of a DO by the execution of a GOTO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the preceding paragraphs."

"The control variable, initial parameter, terminal parameter and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO".
The execution of a DO loop in USA FORTRAN Standard language (1966) is carried out in the following flowchart (Figure 7.1).

FIGURE 7.1: Execution of a DO loop in Standard FORTRAN

In this subsection we try to show the inconsistencies in the DO loop as implemented in each of the compilers A, B and C. It has been mentioned in the definition of the DO loop in Standard FORTRAN that the initial, terminal and incrementation parameters are each either an integer constant or an integer variable reference, i.e. they are not allowed to be a function
call or an expression. It is necessary to know that in the compilers A, B and C the DO loop parameters are allowed to be function calls. A function call for the evaluation of parameters is investigated to show some discrepancies in the evaluation of the DO loop on compilers A, B and C.

Some of the inconsistencies have been shown by the outputs of the following examples on these compilers.

**Example 7.1:**

This program (see Figure 7.2) has four separate DO loops, each one has been used to show a specific aspect. The function K is the subprogram of the program and is used in each of the DO loops for a particular reason. The function K is:

```fortran
FUNCTION K(KK)
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME , M,I
M = M - NPRIME(I)
I = I+1
WRITE(2,51) M, I, KK
51 FORMAT(1H 'M, I,KK = ',319)
KK=I
K =I
RETURN
```

**FIGURE 7.2a**: The FUNCTION K

In this function, an array of prime numbers is used to show the exact value of I, M and KK in each call. The outputs of this example and others show that in some calls there is not a reasonable relation between I and NPRIME(I).

In the main program, before each DO loop N, I and M are reset to one. Now we investigate each DO loop separately. The first DO loop is:

```fortran
DO 4 II= K(NPRIME(3)) , K(NPRIME(2)) , K(NPRIME(1))
WRITE(2,50) II
WRITE(2,50) M
4 CONTINUE
```

**FIGURE 7.2b**: The first DO loop of Example 7.1
The output from compilers A, B, and C are:

<table>
<thead>
<tr>
<th></th>
<th>Compiler A</th>
<th>Compiler B</th>
<th>Compiler C</th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK</td>
<td>M, I, KK</td>
<td>M, I, KK</td>
<td>M, I, KK</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>M, II=</td>
<td>M, II=</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>M, II=</td>
<td>M, II=</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

respectively.

These outputs suggest that:

a. The order of the evaluation of the DO loop parameters in compilers A and B has to be:

1. Initial parameter.
2. Terminal parameter.
3. Incrementation parameter,

and in C it has to be:

1. Terminal parameter.
2. Incrementation parameter.
3. Initial parameter.

b. The first value of the control variable of the DO loop is not equal for each compiler and that is the result of a.

c. The values of M are not equal for each output, but as is shown in the function K the value of M depends on the value of I, so there should not be any inequality in the value of M.

The second DO loop is:
\begin{verbatim}
N=1
DO 5 II= K(N), K(N), K(N)
WRITE(2,53) II,M
5 CONTINUE

FIGURE 7.2c: The second DO loop of Example 7.1

The outputs from compilers A,B and C are:

\begin{tabular}{ccc}
M ,I, KK=& 4 & 2 1 \\
M ,I, KK=& 12 & 3 2 \\
M ,I, KK=& 24 & 4 3 \\
II,M IN DO 5=& 2 & 24 \\
\end{tabular}

and

\begin{tabular}{ccc}
M ,I, KK=& 4 & 2 1 \\
II,M IN DO 5=& 2 & 4 \\
\end{tabular}

respectively.

These outputs show that:

a. In A all the three parameters of the DO statement have been evaluated, whereas in B only the initial parameter and in C two of the three parameters have been evaluated.

b. Before going into the DO loop the value of M, I and N is one, hence the value of M in the first call of K should have been changed to M= $1^{\ast}\text{NPRIME}(1) = 2$, but in A and B the value of M is equal to 4 and in C it is equal to 3, which are not the expected answers.

The third DO loop is:

\begin{verbatim}
DO 6 II= NPRIME(K(N))+1, NPRIME(K(N))+5, NPRIME(K(N))
WRITE(2,52) II,M
6 CONTINUE

FIGURE 7.2d: The third DO loop of Example 7.1

The outputs from compilers A,B and C are:
\end{verbatim}
The outputs show that:

a. In A and C all three parameters have been evaluated, but the orders of the evaluation of them as shown in the first DO loop are not the same. In B only the initial parameter has been evaluated.

b. The values of M on each compiler are different and also are different from the values of M in the first DO loop. The conditions of M, I and N before each call of K are equal, so the values of M for the three successive calls should be 2, 2*3 and 2*3*5 respectively, but in A are 4, 4*3 and 4*3*2, and in C are 3, 3*2, and 3*2*4 respectively.

The fourth DO loop is:

```
DO 7 II= 1,5
    J =K(N)
    WRITE(2,54) II,M
    WRITE(2,50) J
7 CONTINUE
```

In this DO loop the parameters are integer constants. The values of M from function K are incorrect. The values of M should be 2, 2*3, 2*3*5, 2*3*5*7 and 2*3*5*7*11 respectively. The outputs of A, B and C show the following values for M:
4, 4*3, 4*3*2, 4*3*2*7 and 4*3*2*7*11
and 3, 3*2, 3*2*4, 3*2*4*7 and 3*2*4*7*11

respectively.

The output of Example 7.1 from each compiler is given in full in Appendix II at the end of the thesis.

Example 7.2:

In this program (see Figure 7.3), two more DO loops are added.

Before execution of each DO loop all the variables are reset to one. Now, we investigate these additional DO loops. The first loop is:

```
DO 3 II=K(1), K(2), K(3)
WRITE(2,55) M, II
3 CONTINUE
```

The output of this DO loop from the compiler B shows that to evaluate the initial parameter, K(1) has to be called, i.e. the formal parameter KK of function K should be a value but in the body of K, KK is treated as a variable. Here the actual parameter is 1 and cannot be changed, therefore at run time the execution of the program stops at this point. For the compilers A and C, KK is changed to I and the outputs are:

```
M ,I, KK=  2   2   1
M ,I, KK=  6   3   2
M ,I, KK=  30  4   3
M, II IN DO 3=  30  2
```

and

```
M ,I, KK=  2   2   2
M ,I, KK=  6   3   3
M ,I, KK=  30  4   1
M, II IN DO 3=  30  4
```

As is shown the values of M in this DO loop are equal in each successive call from the compilers A and C. It is important to explain that in this DO loop in each call of K the actual parameter KK is an integer constant, but in the other DO loops it is a reference requiring the evaluation of an array element or the evaluation of an integer
variable. One of the reasons that the values of $M$ are different from one call of $K$ to another, or from one compiler to another, should be as in the above explanation, but it is not the complete reason, because as we will show in Tables 7.1, 7.2 and 7.3, the outputs of equal loops in various programs from one compiler are different.

To show this discrepancy, the output of the second DO loop (which is the same as the first loop of Example 7.1) is given here. It is not equal to the output from Example 7.1.

The output from the compiler $A$ is:

\begin{verbatim}
M, I, KK= 2 2 7
M, I, KK= 6 3 5
M, I, KK= 18 4 3
M, II= 2
M, II= 18
\end{verbatim}

This result is even worse than the one of Example 7.1, because the actual parameters for function $K$ have to be:

$$\text{NPRIME(3)} = 5 , \text{NPRIME(2)} = 3 \text{ and } \text{NPRIME(1)} = 2$$

but they are:

$$\text{NPRIME(4)} = 7 , \text{NPRIME(3)} = 5 \text{ and } \text{NPRIME(2)} = 3$$

respectively.

The outputs of the other DO loops are given in Tables 7.1, 7.2 and 7.3, and the complete output for each compiler is given in Appendix II at the end of the thesis.

Example 7.3:

The main program in this example is the same as that in Example 7.2, (Figure 7.4), but the formal parameter $KK$ is not changed in the body of function $K$ (Figure 7.2a), i.e., line 8 of this function has been deleted. Although the value of $M$ is not dependent on $KK$, here the values of $M$ from $A, B$ and $C$ are equal, (see Tables 7.1, 7.2 and 7.3).

The last DO loop in this example is:
DO 8 II = 5, 4
WRITE(2,56) II
8 CONTINUE

In this DO loop the initial parameter is greater than the terminal parameter. Compilers A and C execute the range of the DO loop once and continue the execution of the next part of the program but B does not execute the DO loop, it stops.

<table>
<thead>
<tr>
<th>The first loop</th>
<th>Example 7.1</th>
<th>Example 7.2</th>
<th>Example 7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>30</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>M, II IN DO 6=</td>
<td>30</td>
<td>2</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The second loop</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>12</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>M, II=</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>M, II=</td>
<td>12</td>
<td>18</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The third loop</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>II,M IN DO 5=</td>
<td>2</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The fourth loop</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>II,M IN DO 6=</td>
<td>2</td>
<td>24</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The fifth loop</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>II,M IN DO 7=</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>II,M IN DO 7=</td>
<td>2</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>II,M IN DO 7=</td>
<td>3</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>168</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>II,M IN DO 7=</td>
<td>4</td>
<td>168</td>
<td>4</td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>1848</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>II,M IN DO 7=</td>
<td>5</td>
<td>1848</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The sixth loop</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M,II IN DO 8=</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 7.1:** The evaluation of identical DO loops in Examples 7.1, 7.2 and 7.3 on compiler A
### TABLE 7.2: The evaluation of identical DO loops in Examples 7.1, 7.2, and 7.3 on compiler B

<table>
<thead>
<tr>
<th>Loop Type</th>
<th>Example 7.1</th>
<th>Example 7.2</th>
<th>Example 7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>The first loop</td>
<td>M, I, KK=</td>
<td>2 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td></td>
<td>6 3 2</td>
</tr>
<tr>
<td>Error at run time stops</td>
<td></td>
<td></td>
<td>30 4 3</td>
</tr>
<tr>
<td>The second loop</td>
<td>M, I, KK=</td>
<td>2</td>
<td>2 5</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>6 3 3</td>
<td>6 3 3</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>12 4 2</td>
<td>30 4 2</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>The third loop</td>
<td>M, I, KK=</td>
<td>4 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 5=</td>
<td>4 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 6=</td>
<td>2 4</td>
<td>2 4</td>
</tr>
<tr>
<td>The fourth loop</td>
<td>M, I, KK=</td>
<td>4 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 6=</td>
<td>2 4</td>
<td>2 4</td>
</tr>
<tr>
<td>The fifth loop</td>
<td>M, I, KK=</td>
<td>12 3 2</td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>12 3 2</td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>30 4 1</td>
<td>6 3 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>30 4 1</td>
<td>6 3 1</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>210 5 1</td>
<td>210 6 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>210 5 1</td>
<td>210 6 1</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>2310 6 1</td>
<td>2310 6 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>2310 6 1</td>
<td>2310 6 1</td>
</tr>
<tr>
<td>The sixth loop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example 7.1</td>
<td>Example 7.2</td>
<td>Example 7.3</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>2 2 2</td>
<td>2 2 2</td>
</tr>
<tr>
<td>The first loop</td>
<td>M, I, KK=</td>
<td>6 3 3</td>
<td>6 3 3</td>
</tr>
<tr>
<td></td>
<td>M, II IN DO 3=</td>
<td>30 4 1</td>
<td>30 4 1</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>2 2 3</td>
<td>2 2 3</td>
</tr>
<tr>
<td>The second loop</td>
<td>M, I, KK=</td>
<td>4 3 2</td>
<td>6 3 2</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>20 4 5</td>
<td>20 4 5</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>The third loop</td>
<td>M, I, KK=</td>
<td>3 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>6 3 2</td>
<td>6 3 1</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 5=</td>
<td>3 6 6</td>
<td>3 6</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>3 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td>The fourth loop</td>
<td>M, I, KK=</td>
<td>6 3 2</td>
<td>6 3 1</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>24 4 3</td>
<td>30 4 1</td>
</tr>
<tr>
<td></td>
<td>M, II=</td>
<td>8 24</td>
<td>8 24</td>
</tr>
<tr>
<td></td>
<td>M, II, M IN DO 6=</td>
<td>8 24</td>
<td>8 30</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>3 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td>The fifth loop</td>
<td>II, M IN DO 7=</td>
<td>6 3</td>
<td>1 3</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>2 6</td>
<td>2 6</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>24 4 4</td>
<td>24 4 3</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>3 24</td>
<td>3 24</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>168 5 4</td>
<td>168 5 4</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>4 168</td>
<td>4 168</td>
</tr>
<tr>
<td></td>
<td>II, M IN DO 7=</td>
<td>1848 6 5</td>
<td>1848 6 5</td>
</tr>
<tr>
<td></td>
<td>M, I, KK=</td>
<td>5 1848</td>
<td>5 1848</td>
</tr>
<tr>
<td>The sixth loop</td>
<td>II, M IN DO 8=</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**TABLE 7.3:** The evaluation of identical loops in Examples 7.1, 7.2 and 7.3 on compiler C
7.3.1.1 Comparison of DO loop Elaboration on the Compilers A, B and C

The discrepancies in the elaboration of DO loops on the compilers A, B and C are summarised as follows:

1. The order of the evaluation of the DO loop parameters is different.

   In the compilers A and B it is:
   1. Initial parameter.
   2. Terminal parameter.
   3. Incrementation parameter.

   In the compiler C the order of their evaluation is:
   1. Terminal parameter.
   2. Incrementation parameter.
   3. Initial parameter.

2. If the value of the initial parameter is greater than the value of the terminal parameter, compilers A and B evaluate the range of the DO loop once and continue the evaluation of the next part, but compiler B does not evaluate the DO loop and stops.

3. If the actual parameter of a function is a value, then compiler B does not change it in the body of the function (or procedure) and it stops, but compilers A and C do change it and continue. Hence if the DO loop parameters are function calls, it affects the evaluation of the DO loop, i.e., B does not continue, but A and C do.

4. If a function is called in the DO loop parameters, then the evaluation of these parameters on each compiler is different (see the second DO loop in Example 7.1, Figure 7.2c).

5. In the Examples 7.1, 7.2 and 7.3, when KK, the actual parameter, is changed in the body of the function K then the value of M is different on each compiler (see the value of M in the first and the second DO loop in Examples 7.1 and 7.3 in Figures 7.2b and 7.2c).
This case needs further discussion. We do not know the relationship between KK (actual parameter), I and M in the function K. When KK does not change, the values of I and M seem to be right, but if KK changes in the body of K, the value of M not only is different from one compiler to another, it is also different from one program segment to another on the same compiler. In Example 7.1, in the first DO loop (Figure 7.2b), the value of M in the third call of K on compiler A, should be:

\[ M = 6 \times \text{NPRIME}(3) = 6 \times 5 \]

but it is:

\[ M = 6 \times \text{NPRIME}(1) = 6 \times 2 \]

This means that although I is increased in each call there is a conflict between I and \text{NPRIME}(I).

Hence, as far as the outputs of the Examples 7.1, 7.2 and 7.3 suggest, there is some interrelationship between the variables of a function, and also between them and a formal parameter which changes in the body of the function. This needs further work.

6. The values of the control variable after execution of the DO loop are different on different compilers. In compilers A and B this value is equal to the value of the terminal parameter plus the value of the incrementation parameter. In the compiler C the value of the control variable is equal to one, but as mentioned earlier, this value is defined to be "undefined" in the USA Standard FORTRAN, i.e. no value is assigned to it.

7. If the DO loop parameters are functions, then the evaluation of these functions may affect the value of the other parameters, and cause a different number of executions of the DO loop range. The output of the execution of the third loop in Example 7.1 (Figure 7.2d), shows that the range of the DO loop is executed once on A and C, and twice on B. This is difficult to show in a flowchart.
We ought to mention that, the function K in Examples 7.1, 7.2 and 7.3 has also been written without the WRITE statement and the outputs of these examples on the different compilers were the same as above.

The following flowcharts show the execution of a DO loop on the compilers A, B and C, but the number of the executions of the DO loop range when its parameters are expressions or function references is not clear (see the second loop in Example 7.1, Figure 7.2c).

7.3.2 The Value of the Control Variable after Execution of a DO Loop

This area has been discussed earlier in this section and also will be considered again in subsection 7.3.4.

7.3.3 The Evaluation of the DO-Loop Parameters in a Certain Compiler

As stated earlier the exact sequence of the evaluation of the DO loop parameters is different on various compilers. Here it is interesting to show that even this sequence is different on the same compiler for two programs which should have the same output. The following example contains two similar DO loops; they should be computationally equivalent but the presence of a different number of "WRITE" statements within the loops affects their elaboration.
FIGURE 7.51: The execution of a DO loop in compiler A.
Evaluation of m1

Evaluation of m2

Evaluation of m3

\[ I = m_1 \]

\[ I \leq m_2 \]

\[ I = I + m_3 \]

\[ I \leq m_2 \]

\[ I = I + m_3 \]

**FIGURE 7.61** The execution of a DO loop in compiler B.
FIGURE 7.7: The execution of a DO loop in compiler C.
Example 7.4:

In this example the function K is the same function as in Example 1 (see Figure 7.8). The two DO loops that have been used in the main program are:

```
M=1
I=1
DO 4 II =K(1),K(2),K(3)
  4 CONTINUE
WRITE(2,50) M
M =1
I =1
DO 5 II =K(1),K(2),K(3)
WRITE(2,50) M
WRITE(2,50) II
  5 CONTINUE
```

The output from the first DO loop is:

```
M= 2 1 2 ---- +  KK=2
M= 6 2 3 ---- +  KK=3
M= 30 3 3 ---- +  KK=3
M= 210 4 1 ---- +  KK=1
M=2310 5 2 ---- +  KK=2
```

and from the second DO loop it is:

```
M= 2 1 2
M= 6 2 3
M= 30 3 1
```

The output from the first DO loop shows two evaluations of K(2) (terminal parameter), two evaluations of K(3) (incrementation parameter) and one evaluation of K(1) (initial parameter), but in the second DO loop each is only evaluated once.

Several programs have been run with some changes in the function K and the value of M is different in each program (see 7.3.1.1), but the number of calls of K in each program is 5.

This program has also been run on the compilers A and B, and in both cases the results are equal to those from the second DO loop shown above.

It is not reasonable that the evaluation of the DO loop parameters should have any side effects, but as this output shows, in this compiler there is a strange occurrence which needs further investigation.
7.3.4 An Unclear Point About Assignment Statements

The output from the following example shows something unexpected in the execution of an assignment statement.

Example 7.5:

In this program (Figure 7.9), the function SUM has been declared as an integer function.

```fortran
INTEGER FUNCTION SUM (L,N,II,JJ,KK)
J=0
DO 3 II= JJ, KK
  J= J+N
3 CONTINUE
WRITE(2,56) J, N
56 FORMAT ( 1H, 'SUM=',219)
WRITE (2,58) II
58 FORMAT ( 1H, 'II=',16)
L=J
SUM=J
RETURN
END
```

**FIGURE 7.9a:** The function SUM

This function is assigned once to an integer variable AA and once to a real variable JJ. The relevant part of the main program is:

```fortran
LL =10
JJ =1
I =1
AA =1.1
AA = SUM (LL, NN(I), I, 1, 10)
WRITE (2,55) AA
WRITE (2,57) I
JJ = SUM (LL, NN(I), I, 1, 10)
WRITE (2,54) JJ, LL
WRITE (2,99) (NN(I),I=1, 11)
WRITE (2,57) I
```

**FIGURE 7.9b:** A part of the main program in Example 7.5

The function SUM has not been declared as an integer in the main program and the outputs using the compilers A and C for this program are:
SUM = 10 1
II = 11
AA = 0.11000000 E 01
I = 11
SUM = 110 11
II = 11
JL, LL = 1 110
NN = 1 2 3 4 5 6 7 8 9 10 11
I = 12
I = 11

and
SUM = 10 1
II = 1
AA = 0.11000000 E 01
I = 1
SUM = 10 1
II = 1
JL, LL = 1 10
NN = 1 2 3 4 5 6 7 8 9 10 11
I = 1
I = 1

respectively. This program stopped in B at the compilation stage. The results from A and C show that the function SUM has been executed but its result has been assigned to neither the real variable AA nor the integer variable Jl.

The program was then altered so that the function SUM was declared as an INTEGER. This time the outputs of the compilers A, B and C are:

SUM = 10 1
II = 11
AA = 0.11000000 E 02
I = 11
SUM = 110 11
II = 11
JL, LL = 110 110
NN = 1 2 3 4 5 6 7 8 9 10 11
I = 12
I = 11

and
SUM = 10
II = 1
AA = 0.10000000 E 02
I = 1
SUM = 10 1
II = 1
J1,II = 10 10
NN = 1 2 3 4 5 6 7 8 9 10 11
I = 1
I = 1

respectively.

These outputs show that the value of the function SUM has been assigned to both the real variable AA and the integer variable J1. The outputs suggest that, before calling each function in the main program, it is necessary to declare the function, otherwise, its result will not be assigned to any variable.

The other point is that, the value of the control variable of an implied DO loop (like DO loop) after execution is different from one compiler to another. This example shows that in the compilers A and B the value of the control variable is equal to the value of the terminal parameter plus the value of the incrementation parameter, but in the compiler C, it is equal to 1 (see Figures 7.5, 7.6 and 7.7).

As shown in these outputs, the values of SUM and J1 are not equal, because the value of II before calling SUM in the compilers A and B is equal to 11 and in C is equal to 1, so they refer to different addresses (different array elements) and work with different array variables.

7.3.5 "Call by Name" (Substitution) in FORTRAN

"Call by name" does not exist in the FORTRAN language.

In Example 7.5, the index I in NN(I) does not change when I is changed during the execution of the function SUM. Since the value of I is equal to 1 at the call of SUM, NN(I) is equal to NN(1) and it does not change to NN(2), NN(3), ..., and NN(10) (see the value of SUM at the first line of each output).
To make the point clear, it is necessary to define "call by value", "call by reference" and "call by name (substitution or embedded function)".

1. **Call by Value**

   A parameter is called by "value" in a procedure call, if there is absolutely no way for it to change the value of the actual parameter outside the procedure body, i.e., the actual parameter has a location allocated for a value of the type of the formal parameter in the data area of the procedure.

2. **Call by Reference**

   A parameter is called by "reference" in a procedure call if its actual parameters can be used to transmit a value back from the procedure, i.e., the address of the actual parameter is located in the data area of the procedure and if the value of the actual parameter is changed during the call of the procedure, then the value of it will be changed outside the procedure.

3. **Call by Name (substitution)**

   Here, not only the value of the actual parameter but also the parameter itself can be changed.

   If the following function:
   
   ```
   FUNCTION SUM (L, N, II, JJ, KK)
   J = 0
   DO 3 II = JJ, KK
   J = J+N
   3 CONTINUE
   L = J
   SUM = J
   RETURN
   END
   ```
   
   is called by J1 = SUM (L, NN(I), I, 1, 10), the result should be \( \sum_{I=1}^{10} NN(I) \), i.e., whenever the control variable II is changed the actual parameter NN(I) should be changed to another element of NN (assumed to be an array). That is, within the procedure, each access to the formal parameter N must
cause the actual parameter specification \(NN(I)\) to be evaluated. It is difficult to see how this can be implemented efficiently by such a textual substitution. Since the actual parameter corresponding to \(N\) changes whenever the value of \(II\) changes, we should recompute \(N\) every time we refer to the formal parameter in the procedure. One way of implementing "call by name" parameters is to have a special routine (what Ingerman called a "thunk" (1961)) in "call by name" the thunk evaluates and returns the address into the body of the procedure.

There is no such concept as thunk in the FORTRAN language. Calling a procedure with a subscripted actual parameter means that during the execution of the procedure, even if the index variable changes, the variable is the one that has been called as the actual parameter (see Figures 7.9a, 7.9b and their outputs).

What has to be done in order to have a call by name instead of by reference? Since there is no facility like a thunk in FORTRAN, programmers need to emulate such a feature for themselves. To do this an EXTERNAL function can be declared in the block which calls the procedure to evaluate and return the address of the actual parameter into the subprogram whenever the address changes.

In the Example 7.5 the function \(SUM\) has been assigned to both AA and J1, and during the execution of \(SUM\), \(NN(II)\) does not change. The actual parameter \(NN(I)\) can be changed by declaring an EXTERNAL function, say \(N\), in the main program, use it as the actual parameter for \(SUM\) and change the assignment statement:

\[
J = J + N
\]

\[
to \quad J = J + N(II)
\]

in the body of \(SUM\).

The function \(N\) (thunk) is:
FUNCTION N(Il)
DIMENSION NN(Il)
N = NN(Il)
RETURN
END

In the main program we have

AA = SUM (LL, N, I, 1,10)

and output for this assignment statement is:

SUM = 55
II = 11
A = 0.55000000 E 01

which shows

SUM = 1+2+..+10 = \sum_{I=1}^{10} NN(I).

The reader should bear in mind that according to the USA Standard FORTRAN if the actual parameter is an expression, it is said to be "called by name". So as not to add to the confusion caused by the use of the term "call by name" in USA Standard FORTRAN, it is probably better to refer to the general usage of "call by name" as "call by embedded function".
7.4 THE EFFECT OF INCONSISTENCIES IN PARALLEL EXECUTION

The investigation of these inconsistencies discloses a lack of proper definition of certain statements within the definition and hence the variation in several widely used compilers. It is generally accepted that proving the correctness of non-trivial programs is difficult. Hence, all these points must be considered very carefully not only in a parallel execution but first in a sequential one, so as not to have unexpected results. To make this matter clear, we will consider the effect of one discrepancy on different compilers.

Suppose we have:

\[
\begin{align*}
&\text{DO } N1 \ I1 = 1, 10 \\
&\text{DO } N2 \ I2 = 1, 10 \\
&N2 \ A( I1, I2) = B( I1, I2) \\
&\text{DO } N3 \ I3 = 1, 10 \\
&N3 \ B( I3, I2) = A( I1, I2) \\
&N1 \ CONTINUE
\end{align*}
\]

The statement labelled by N2 is totally independent, i.e., there is no dependency between the assignment statements of each iteration, and they can be evaluated separately at the same time on ten different processors. The loop "N3" is also totally independent, but for the execution of "N3" the value of \( A( I1, I2) \) is required. \( I2 \) is the control variable of N2 and, as seen earlier, its value is different from one compiler to another and in the USA Standard FORTRAN it is undefined (see 7.3.1.1). Hence, the values of the elements of the array B may not be equal, in loop "N3", in either parallel or sequential execution.

Therefore, before any execution on any system (sequential or parallel), we must be sure that there are no inconsistencies, conflicts or ambiguities between different implementations or within a single implementation. This problem is not particular to FORTRAN implementations, similar situations exist in other languages. Badii (1981) has found several inconsistencies between two different PASCAL implementations.
We should have a proper and logical definition for each sentence or statement in order to have an unambiguous and unique interpretation for both serial and parallel execution.
CHAPTER 8

SUMMARY AND CONCLUSION
In a parallel processing environment there are basically two approaches to determine which parts of a program may be run on different processors at the same time. Parallelism can be explicitly indicated by the programmer or it must be detected and distinct tasks generated as part of the compilation process.

We have presented a technique to detect parallelism in an arithmetic expression by constructing a balanced binary tree of minimal height and which makes use of the vacancies in the tree in such a way so as to allow smaller trees to be inserted within it. This method can be used at compilation time to produce machine-oriented code suitable for efficient parallel execution.

The applicability of this work to speed up the evaluation of arithmetic expressions is demonstrated. The basic algorithm is then reformulated so as to act, not directly on arithmetic operators, but on tables of operator properties and on grammars which incorporate the restructuring used in the tree manipulation algorithm. The grammars involved are ambiguous context-free grammars giving rise to generalised operator precedence relations capable of indicating associativity of like operators.

We have also constructed a context-sensitive grammar to define the syntax of an arithmetic expression for a parallel programming language by using the associativity of operators. This grammar is capable of performing the tree balancing and the detection of vacancies in an expression into which other smaller sub-expressions can be inserted. In other words, it performs the same task as the technique described in Chapter 2, during the syntax analysis.

An algorithm is also presented to parse strings generated by the grammar. The algorithm has been compared with some parsers for general context-sensitive-grammars and it is shown that, although the grammar is
context-sensitive, the parser is not as powerful as the parser for the
general C.S.G. This seems to yield a situation in which this non-context
free grammar might generate a context-free language.

Finally, we have investigated the possibility of using extra
information in the parallel execution of other parts of a program. For
this purpose we chose FORTRAN which is one of the most widely used
programming languages, and found some important inconsistencies which
exist in different FORTRAN compilers. These inconsistencies show a lack
of proper definition of certain statements in the standard FORTRAN and
serious discrepancies in their implementation in several compilers in
common usage. The investigation of how such statements might be executed
in parallel was thus thought impractical. The results in this thesis
suggest several directions for future work as follows:

1. In the algorithm of Chapter 2, if the height of two subexpressions
is not the same, then, one of them can be inserted as a subtree of
the other (without affecting the re-ordering of the expression).
This algorithm can be extended in order to insert the right subtree
of a subexpression into the left one of a following expression, or
vice versa. In this case, the height of the resulting tree might be
decreased, or if not, larger vacancies could be created in the
resultant tree into which further insertions could then be made (see
Figure 2.22b). This method forms the basis of the manipulations
carried out by the new grammar in Chapter 5 (see Figures 5.6 and 5.7).
In many expressions there are vacancies which cannot accommodate
another sub-expression from the current expression, but which may be
used by other subsequent computations.
In a real machine only a finite number of processors are available.
Hence, one of the areas which may be considered is forming a balanced
binary tree for an expression which makes best use of a limited number of available processors, especially in a polynomial of degree \( n \). Another point is that, for efficient parallel computation, data must be organized in order to permit simultaneous fetching and storing, and resources have to be shared amongst the processors. Hence, an optimal algorithm which reduces the tree height of an expression must be efficient in terms of storing and fetching the data and also organizing the resources. Such algorithms would almost certainly be machine/system dependent.

2. As mentioned earlier, Floyd (1963), produced a precedence grammar which defines a language closely resembling ALGOL 60, but which has none of the syntactic ambiguities of ALGOL 60. It is possible that an operator grammar could be used in a similar way to our modified operator precedence grammar to detect parallelism within a complete program. If the grammar (5.10) can be extended to generate a programming language like ALGOL 60, then some of the parts of the language which can be executed in parallel will be detected during the syntax analysis phase of compiling.

3. In Chapter 6, we remarked that some investigators (e.g. Baker (1974) and Aggarwal (1979)) have derived sufficient conditions which when satisfied by the rules of a non C.F.G. guarantee that the corresponding language is context-free i.e., the addition of context restriction does not increase the generative power of a grammar. Aggarwal (1979) proved that SVMT-bounded grammars can generate only context-free languages. His conditions require a strict partial ordering (irreflexive) which is not satisfied by two non-terminal symbols (P and T) in the grammar (5.10) (i.e. \( P<P, T<T \) and \( P>P, T>T \)).
However, the language generated by the context-sensitive grammar (5.10) can also be generated by an ambiguous context-free grammar (see Chapter 4) and the parser for recognizing the strings generated by this grammar is not as powerful as the recognizer for those of a general C.S.L. It may therefore be that this grammar generates a context-free language. If this is so, then the class of grammars generating only C.F.L.s will be enlarged.

4. Finally, since it is generally accepted that proving the correctness of non-trivial programs is difficult, all inconsistencies which are found in different FORTRAN compilers must be considered very carefully; not only in a parallel execution but initially in sequential ones as well, so as not to have unexpected results (particularly when these 'incorrect' results look reasonable). Some of the inconsistencies may result from the semantic analysis carried out by each compiler. The discrepancies between the semantic analyses of these compilers should be considered as ambiguities which cause the production of different parse trees for the same sentence on different compilers. We have given explanations for some of these inconsistencies, but others need further investigation. Some outstanding areas of difficulty are:

a. If the DO loop parameters are function calls, then the evaluation of these parameters on different compilers is different, and also cause a different number of executions of the DO loop range.

b. When the actual parameter of a function is changed within its body, the value of the variables are different on each compiler, i.e., there are some interrelationships between the variables of a function, and also between them and the formal parameters which change in the body of the function.

c. There is a conflict in the evaluation of the parameters of DO loops in one compiler, when the number of 'WRITE' statements in the same loop is changed.
We have given an improved tree balancing algorithm and demonstrated how such transformation can be carried out by means of a suitable grammar. The efficient parsing of sentences generated by the grammar must now be investigated further. Extensions of the grammar to describe a complete algorithmic language are also sought.
REFERENCES


APPENDIX I

TWO ALGORITHMS FOR CONSTRUCTING A BALANCED BINARY TREE
Program for algorithm A of chapter 2

PROC BALTREE = ('REF' 'INT' NEXT, RANDTOP, OPTOP,
               'REF'[] TREE THIS, 'REF' 'REF' 'TREE' ORIG,
               ['REF' 'TREE' RANDSTACK, [] 'CHAR' OPERATORS) 'VOID':

'BEGIN'
  'INT' NEWRAND ← RANDTOP; 'CHAR' OPER ← OPERATORS [OPTOP];
  'INT' HEIGHT ← NEXT 'PLUS' 1; RANDTOP ← 1;
  THIS [NEXT] ← RANDSTACK [RANDTOP];
  RANDTOP 'PLUS' 1;
  'INT' PREV ← NEXT;
  'IF' RANDTOP <= 'UPB' RANDSTACK
       'THEN' OPTOP ← 'MINUS' 1; NEXT 'PLUS' 1;
       THIS [NEXT] ← (THIS [PREV], RANDSTACK [RANDTOP], EMPTY, 'ABS' (OPER), 2);
       FATHER OF RANDSTACK [RANDTOP] ← THIS [NEXT];
       RANDTOP 'PLUS' 1;
       FATHER OF RANDSTACK [RANDTOP] ← THIS [NEXT]
       FATHER OF THIS [PREV] ← THIS [NEXT];
       FI;
  WHILE (OPTOP >= 'LWB' OPERATORS 'AND' RANDTOP <= 'UPB' RANDSTACK)
       OPER ← OPERATORS [OPTOP] 'FALSE') 'DO'
'BEGIN'
  'IF' LEVEL OF (RIGHT OF AID1) = LEVEL OF (LEFT OF AID1)
  'THEN' HEIGHT ← LEVEL OF AID1 + 1;
       NEXT ← 'PLUS' 1;
       IF RANDTOP <= 'UPB' RANDSTACK
       'THEN'
       IF LEVEL OF THIS [NEXT] = LEVEL OF TEMP
       'THEN' LEVEL OF THIS [NEXT] 'PLUS' 1
       FI
       FI;
       WHILE ((FATHER OF EXPER)'ISNT' EMPTY)'AND'
       (LEVEL OF (RIGHT OF EXPER) = LEVEL OF (LEFT OF EXPER))
       'DO'
       EXPER ← EXPER + LEVEL OF (LEFT OF EXPER) + 1;
       EXPER ← FATHER OF EXPER
       FI;
       RANDTOP 'PLUS' 1;
       OPTOP ← 'MINUS' 1
'END'; ORIG ← AID1
'END';
'PROC' BALTREEB = 'VOID';
'BEGIN'
'OP' 'INT':
>('INT' LEV ← (LEVEL 'OF' EXPRA > LEVEL 'OF' EXPRB) +1; LEV);
'PROC' MAKENEXT='VOID':
(T1←STACK [KHar] 'OP' STACK [T2];
TP←TP+1;
AD[TP]←(STACK[KHar],STACK[T2],EMPTY,27,T1);
FATHER 'OF' STACK [KHar]←AD[TP];
FATHER 'OF' STACK [T2] ← AD[TP];
STACK [T2] ← AD[TP];
);
'IF' 'ABS' (LEVEL 'OF' STACK[KHar]−LEVEL 'OF' STACK[T2]) >1
'THEN'
'IF' LEVEL 'OF' STACK[KHar]>LEVEL 'OF' STACK[T2]
'C'
'-------------------------------------------------------------
INSERT T2 INTO RIGHT SUBTREE OF KHar
'-------------------------------------------------------------'C'

'THEN' TEMP 'OF' STACK[KHar];
FATHER 'OF' TEMP 'OF' STACK [KHar];
'IF' LEVEL 'OF' TEMP = LEVEL 'OF' (LEFT 'OF' STACK[KHar])
'THEN'
'WHILE' (LEVEL 'OF' STACK[T2]<LEVEL 'OF' TEMP)
 'AND' ('ROUND' (OPERATOR 'OF' TEMP) =27)
 'DO'
 (FATHER 'OF' (RIGHT 'OF' TEMP)+ TEMP;

TEMP 'OF' TEMP
);
EXPER 'OF' TEMP;
T1←TEMP 'OP' STACK [T2];
TP ← TP+1;
AD[TP]←(TEMP,STACK[T2],EXPER,27,T1);
FATHER 'OF' TEMP 'OF' AD[TP];
FATHER 'OF' STACK [T2] ← AD[TP];
RIGHT 'OF' EXPER 'OF' AD[TP];
'IF' LEVEL 'OF' EXPER = T1
'THEN' LEVEL 'OF' EXPER 'PLUS' 1
'FI';
'IF' LEVEL 'OF' EXPER = LEVEL 'OF' STACK[KHar]
'THEN' STACK[KHar] ← EXPER
'FI';
STACK[T2] ← STACK[KHar]
'ELSE' MAKENEXT
'FI'
'ELSE'

continued....
INSERT KHAR INTO LEFT SUBTREE OF T2

```
C'
'-------------------------------------------------------------'

TEMPL=LEFT 'OF STACK[T2];
FATHER 'OF TEMPl=STACK[T2];
'IF' LEVEL 'OF TEMPl>LEVEL 'OF 'STACK[KHAR])
'THEN'
'WHILE' (LEVEL 'OF TEMPl>LEVEL 'OF 'STACK[KHAR])
'AND' ('ROUND' (OPERATOR 'OF 'TEMP)=27)
'DO'
(FATHER 'OF 'LEFT 'OF 'TEMP) TEMPl;
TEMPl=LEFT 'OF 'TEMP
);
EXPER=FATHER 'OF 'TEMP;
Tl=TEMPl>STACK[KHAR];
TP=TP+1;
AD[TP]={(STACK[KHAR], TEMP, EXPER, 27, Tl);
FATHER 'OF 'TEMP=AD[TP];
FATHER 'OF 'STACK[KHAR]=AD[TP];
LEFT 'OF 'EXPER=AD[TP];
'IF' LEVEL 'OF 'EXPER<=Tl
'THEN' LEVEL 'OF 'EXPER='PLUS'1
'FI';
'IF' LEVEL 'OF 'EXPER =LEVEL 'OF 'STACK[T2]
'THEN' STACK[T2]=EXPER
'FI';
'ELSE' MAKENEXT
'FI';
'FI'

'C'
'-------------------------------------------------------------'

MAKE A TREE WITH KHAR AND T2 AS ITS RIGHT AND LEFT SUBTREES

Program for algorithm B of chapter 2
This Appendix consists of source listings and corresponding output from various programs discussed in Chapter 7, and consequently they are numbered as Figures of that chapter.
BLOCK DATA
DIMENSION NPRIME(10)
COMMON /B11/NPRIME,M,I
DATA NPRIME /2,3,5,7,11,13,17,19,23,29/,M/1/,I/1/
END

MASTER EXAMP
DIMENSION NPRIME(10)
COMMON /B11/NPRIME,M,I
WRITE(2,49) (NPRIME(I), I=1,10)
M=1
I=1
DO 4 II=K(NPRIME(3)) ,K(NPRIME(2)) ,K(NPRIME(1))
WRITE(2,50) II
WRITE(2,57) M
4 CONTINUE
I=1
N=1
M=1
DO 5 II=K(N),K(N),K(N)
WRITE(2,53) II,M
5 CONTINUE
I=1
N=1
M=1
DO 6 II=NPRIME(K(N))+1,NPRIME(K(N))+5,NPRIME(K(N))
WRITE(2,52) II,M
6 CONTINUE
I=1
J=1
N=1
M=1
DO 7 II=1,5
J=K(N)
WRITE(2,54) II,M
7 CONTINUE
WRITE(2,50) II
STOP
49 FORMAT(1H ,'/' '/' '/' '/' '/' '/' '/' '/' '/' ,10I
50 FORMAT(1H ,'
II= ',I5)
52 FORMAT(1H ,'
II,M IN DO 6= ',1I7)
53 FORMAT(1H ,'
II,M IN DO 5= ',1I7)
54 FORMAT(1H ,'
II,M IN DO 7= ',1I7)
57 FORMAT(1H ,'
M = ',I5)
END

continued.....
FUNCTION K(KK)
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
M=M*NPRIME(I)
I=I+1
WRITE(2,51) M,I,KK
51 FORMAT (1H , ', M, I, KK=' ', 3I9)
KK=I
K=I
RETURN
END
FINISH

FIGURE 7.2: Program for Example 7.1

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>2 3 5 7 11 13 17 19 23 29</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 2 5</td>
<td></td>
</tr>
<tr>
<td>6 3 3</td>
<td></td>
</tr>
<tr>
<td>12 4 2</td>
<td></td>
</tr>
<tr>
<td>II= 2</td>
<td></td>
</tr>
<tr>
<td>M = 12</td>
<td></td>
</tr>
<tr>
<td>4 2 1</td>
<td></td>
</tr>
<tr>
<td>12 3 2</td>
<td></td>
</tr>
<tr>
<td>24 4 3</td>
<td></td>
</tr>
</tbody>
</table>

II, M IN DO 5=

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>4 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 3 2</td>
<td></td>
</tr>
<tr>
<td>24 4 3</td>
<td></td>
</tr>
</tbody>
</table>

II, M IN DO 6=

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>4 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 3 2</td>
<td></td>
</tr>
<tr>
<td>24 4 3</td>
<td></td>
</tr>
</tbody>
</table>

II, M IN DO 7=

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>4 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 3 2</td>
<td></td>
</tr>
<tr>
<td>24 4 3</td>
<td></td>
</tr>
</tbody>
</table>

II, M IN DO 7=

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>12 3 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 4 3</td>
<td></td>
</tr>
</tbody>
</table>

II, M IN DO 7=

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>168 5 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>168 6 5</td>
<td></td>
</tr>
</tbody>
</table>

II, M IN DO 7=

<table>
<thead>
<tr>
<th>M, I, KK=</th>
<th>1848 6 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1848 5 4</td>
<td></td>
</tr>
</tbody>
</table>

II= 6

Output from compiler A
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
<th>17</th>
<th>19</th>
<th>23</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>12</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II=</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M = 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 5=</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 6=</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 6=</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>2</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>24</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>3</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>168</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>4</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>1848</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>5</td>
<td>1848</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II=</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output from compiler B

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
<th>17</th>
<th>19</th>
<th>23</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>M, I, KK=</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>20</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II=</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M = 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 5=</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>24</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 6=</td>
<td>8</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>24</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>3</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>168</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>4</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK=</td>
<td>1848</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN DO 7=</td>
<td>5</td>
<td>1848</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II=</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output from compiler C
BLOCK DATA
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
DATA NPRIME /2,3,5,7,11,13,17,19,23,29/,M/1/,I/1/
END

MASTER EXAMP
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
DO 3 II=K(1),K(2),K(3)
WRITE (2,55) II,M
3 CONTINUE
M=1
I=1
DO 4 II=K(NPRIME(3)),K(NPRIME(2)),K(NPRIME(1))
WRITE (2,50) II
WRITE (2,57) M
4 CONTINUE
I=1
N=1
M=1
DO 5 II=K(N),K(N),K(N)
WRITE (2,53) II,M
5 CONTINUE
N=1
I=1
M=1
DO 6 II=NPRIME(K(N))+1,NPRIME(K(N))+5,NPRIME(K(N))
WRITE (2,52) II,M
6 CONTINUE
I=1
J=1
N=1
M=1
DO 7 II=1,5
J=K(N)
WRITE (2,54) II,M
7 CONTINUE
WRITE (2,50) II
DO 8 II=5,4
WRITE (2,56) II,M
8 CONTINUE
STOP
50 FORMAT (1H ,')
57 FORMAT (1H ,')
52 FORMAT (1H ,')
53 FORMAT (1H ,')
54 FORMAT (1H ,')
55 FORMAT (1H ,')
56 FORMAT (1H ,')
END

Continued....
FUNCTION K(KK)
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
M=M*NPRIME(I)
I=I+1
WRITE(2,51) M, I, KK
51 FORMAT (1H8,'M, I, KK=', 319)
KK=I
K=I
RETURN
END
FINISH

FIGURE 7.3: Program for Example 7.2

Output from compiler A
<table>
<thead>
<tr>
<th>Statement</th>
<th>M, I, KK</th>
<th>M, I, KK</th>
<th>M, I, KK</th>
</tr>
</thead>
<tbody>
<tr>
<td>II, M IN</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DO 3=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>M, I, KK</td>
<td>30</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>II= 4</td>
<td>4</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>M = 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II, M IN</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>DO 5=</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>M, I, KK</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK</td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>II, M IN</td>
<td>8</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>DO 6=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M, I, KK</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>II, M IN</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>DO 7=</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M, I, KK</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>II, M IN</td>
<td>24</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>DO 7=</td>
<td>3</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>M, I, KK</td>
<td>168</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>II, M IN</td>
<td>4</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>DO 7=</td>
<td>5</td>
<td>1848</td>
<td></td>
</tr>
<tr>
<td>M, I, KK</td>
<td>1848</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>II= 5</td>
<td>6</td>
<td>1848</td>
<td></td>
</tr>
<tr>
<td>II, M IN</td>
<td>5</td>
<td>1848</td>
<td></td>
</tr>
<tr>
<td>DO 8=</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output from compiler C
BLOCK DATA
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
DATA NPRIME /2,3,5,7,11,13,17,19,23,29/,M/1/,I/1/
END

MASTER EXAMP
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
DO 3 II=K(1),K(2),K(3)
WRITE(2,55) II,M
3 CONTINUE
M=1
I=1
DO 4 II=K(NPRIME(3)),K(NPRIME(2)),K(NPRIME(1))
WRITE(2,50) II
WRITE(2,57) M
4 CONTINUE
I=1
N=1
M=1
DO 5 II=K(N),K(N),K(N)
WRITE(2,53) II,M
5 CONTINUE
N=1
I=1
M=1
DO 6 II=NPRIME(K(N))+1,NPRIME(K(N))+5,NPRIME(K(N))
WRITE(2,52) II,M
6 CONTINUE
I=1
J=1
N=1
M=1
DO 7 II=1,5
J=K(N)
WRITE(2,54) II,M
7 CONTINUE
WRITE(2,50) II
DO 8 II=5,4
WRITE(2,56) II,M
8 CONTINUE
STOP
50 FORMAT(1H, 57 FORMAT(1H, 'II= ', 'M = ', 'II,M IN DO 6=', 'II,M IN DO 5=', 'II,M IN DO 7=', 'II,M IN DO 3=', 'II,M IN DO 8=', 'Continued ....
FUNCTION K(KK)
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
M=M*NPRIME(I)
I=I+1
WRITE(2,51) M,I,KK
51 FORMAT (IH 'M, I, KK=' , 3I9)
K=I
RETURN
END

FIGURE 7.4: Program for Example 7.3

Output from compiler A
\begin{verbatim}
M, I, KK =  2  2  1
M, I, KK =  6  3  2
M, I, KK =  30  4  3
II, M IN DO 3 =  2  30
M, I, KK =  2  2  5
M, I, KK =  6  3  3
M, I, KK =  30  4  2
II =  2
M =  30
M, I, KK =  2  2  1
II, M IN DO 5 =  2  2
M, I, KK =  2  2  1
II, M IN DO 6 =  4  2
II, M IN DO 6 =  7  2
M, I, KK =  2  2  1
II, M IN DO 7 =  1  2
M, I, KK =  6  3  1
II, M IN DO 7 =  2  6
M, I, KK =  30  4  1
II, M IN DO 7 =  3  30
M, I, KK =  210  5  1
II, M IN DO 7 =  4  210
M, I, KK =  2310  6  1
II, M IN DO 7 =  5  2310
II =  6

\textsc{DO-LOOP CONTROL - TERMINAL VALUE < INITIAL VALUE}
\end{verbatim}

**Output from compiler B**

\begin{verbatim}
M, I, KK =  2  2  2
M, I, KK =  6  3  3
M, I, KK =  30  4  1
II, M IN DO 3 =  4  30
M, I, KK =  2  2  3
M, I, KK =  6  3  2
M, I, KK =  30  4  5
II =  4
M =  30
M, I, KK =  2  2  1
M, I, KK =  6  3  1
II, M IN DO 5 =  3  6
M, I, KK =  2  2  1
M, I, KK =  6  3  1
M, I, KK =  30  4  1
II, M IN DO 6 =  8  30
M, I, KK =  2  2  1
II, M IN DO 7 =  1  2
M, I, KK =  6  3  1
II, M IN DO 7 =  2  6
M, I, KK =  30  4  1
II, M IN DO 7 =  3  30
M, I, KK =  210  5  1
II, M IN DO 7 =  4  210
M, I, KK =  2310  6  1
II, M IN DO 7 =  5  2310
II =  6
II, M IN DO 8 =  5  2310
\end{verbatim}

**Output from compiler C**
MASTER EXAMPLE
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
M=1
I=1
DO 4 II=K(1),K(2),K(3)
4 CONTINUE
WRITE (2,50) M
M=1
I=1
DO 5 II=K(1),K(2),K(3)
WRITE (2,50) M
WRITE (2,50) II
5 CONTINUE
WRITE (2,50) M
STOP
50 FORMAT (1H1, M=' ',I5)
END

FUNCTION K(KK)
DIMENSION NPRIME(10)
COMMON /BL1/NPRIME,M,I
M=M*NPRIME(I)
WRITE (2,55) M,I,KK
55 FORMAT (1H1, M=' ',3I5)
I=I+1
KK=I
K=I
RETURN
END
FINISH

FIGURE 7.8: Program for Example 7.4

| M   | 2   | 1   | 2   |
| M   | 6   | 2   | 3   |
| M   | 30  | 3   | 3   |
| M   | 210 | 4   | 1   |
| M   | 2310| 5   | 2   |
| M   | 2310|     |     |
| M   | 2   | 1   | 6   |
| M   | 6   | 2   | 4   |
| M   | 30  | 3   | 5   |
| M   | 30  |     |     |
| M   | 4   |     |     |
| M   | 30  |     |     |

Output from compiler C
MASTER VECTOR
DIMENSION NN(ll)
DATA NN/1,2,3,4,5,6,7,8,9,10,11/
   LL=10
   I=1
   J1=1
   AA=1.1
WRITE(2,59) AA
   AA=SUM(LL,NN(I),I,1,10)
WRITE(2,55) AA
WRITE(2,57) I
   J1=SUM(LL,NN(I),I,1,10)
WRITE(2,54) J1,LL
WRITE(2,99) (NN(I),I=1,11)
WRITE(2,57) I
   K=1
   DO 5 I=1,10
   K=K+1
5 CONTINUE
WRITE(2,57) I
STOP
59 FORMAT(1H 'I',16.8)
55 FORMAT(1H 'AA=' ,E16.8)
57 FORMAT(1H 'I=', I6)
54 FORMAT(1H 'J1, LL=',2I6)
99 FORMAT(1H 'NN=',11I3)
END

INTEGER FUNCTION SUM (L,N,II,JJ,KK)
   J=0
   DO 3 II=JJ,KK
      J=J+N
   3 CONTINUE
56 FORMAT(1H 'SUM=',2I6)
      WRITE(2,56) J,N
      WRITE(2,58) II
58 FORMAT(1H 'II=',I6)
      L=J
      SUM=J
      RETURN
      END
FINISH

FIGURE 7.9: Program for Example 7.5
Output from compiler A

```
0.11000000E 01
SUM =  10  1
II =  11
AA =  0.11000000E 01
I =  11
SUM =  110  11
II =  11
J1, LL =  1  110
NN =  1  2  3  4  5  6  7  8  9  10  11
I =  12
I =  11
```

Output from compiler C

```
0.11000000E 01
SUM =  10  1
II =  1
AA =  0.11000000E 01
I =  1
SUM =  10  1
II =  1
J1, LL =  1  10
NN =  1  2  3  4  5  6  7  8  9  10  11
I =  1
I =  1
```
Program for Example 7.4 with 'INTEGER SUM'
Output from compiler A

Output from compiler B

Output from compiler C