A NEW COMPUTER PROGRAMMING LANGUAGE

A NEW COMPUTER PROGRAMMING LANGUAGE FOR THE PARALLEL PROCESSING ENVIRONMENT

By

JOHN N. WOLKOWSKI, B. Eng.

A Thesis

Submitted to the School Of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Engineering

McMaster University

(May) 1972

MASTER OF ENGINEERING (1972)
(Electrical Engineering)

McMASTER UNIVERSITY
Hamilton, Ontario.

TITLE: A New Computer Programming Language for the Parallel Processing Environment

AUTHOR: John. N. Wolkowski, B. Eng. (McMaster University)

SUPERVISOR: Dr. E. Della Torre

NUMBER OF PAGES: v 149

ABSTRACT

A new high-level, high-order computer programming language designed to complement multi-processor, parallel computing systems is presented. These systems permit a high-order of operation by performing many instructions simultaneously, thus producing significant increases in computing speed.

The proposed language is so constructed as to give the user a free and natural format, to express problems which exhibit natural or inherent parallelism. In order to demonstrate some of the main features, a small subset of the language has been written, and implemented as a sequential simulation.

In order to relate the language to hardware schemes, a parallel processing array computer is briefly examined.

A core language to communicate with parallel computing systems may be constructed from the concepts developed.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to the Department of Electrical Engineering for providing its facilities and assistance and to Dr. E. Della Torre for his guidance and encouragement.

Thanks are given to Karl Siemens, Bob Tiernay and the other graduate students in the department for many and useful discussions.

The author also wishes to thank Mrs. J. Wolkowski for her assistance and long hours in typing this manuscript.

TABEL OF CONTENTS

CHAPTER ONE

INTRODUCTION

This thesis describes the basis on which the propositions for a new programming language, to communicate with parallel computing systems, are founded. For purposes of discussion, the acronym PALTRAN (PAralleL TRANslation) will be adopted as a name for the language.

Parallel Processing and the basis for Paltran will be discussed in the next chapter. Chapter three describes the proposed statement forms and syntax of the language. The version of the language written for a Digital Equipment Corporation PDP-8 minicomputer is discussed in Chapter four. A general model of an array processor, on which Paltran commands are based, is presented in the next section.

The remainder of this chapter is devoted to the motives and needs for parallel processing and generally related language descriptions.

Motivation for Parallel Processing

Several reasons can be cited for justifying interest in parallel processing organizations, namely, functional and hardware or economic. Even if application or functional requirements cannot justify a parallel organization, the economics of new hardware technologies may still do so.

Functional reasons for parallel processing may be divided into four areas. These include, very large computational problems, problems with inherent parallelism, multi-programming and simultaneous use by many users, and reliability and graceful degradation. The latter two areas are somewhat divorced from high-level language considerations, while the former are directly dependent on efficient communications with the hardware.

History has shown that, at any given point, very large processing and computation problems place requirements on a system that exceed those which can be implemented with conventional computer organization using state-of-the-art circuit and memory speeds available at that point. In spite of the increasing speeds of logic circuits, memories and input/output equipment, requirements for processing and computation capabilities have increased at a somewhat comparable rate. As computer power increases, there is a continuing pressure to solve problems for which solutions have been impossible in the past. Examples of such problems include the global weather problem, nuclear physics problems, economic planning, traffic management, and others in which an array of data points are processed.

Problems, or parts of problems, may contain structures that are inherently parallel. Such parallelism may result from the presence of several data streams which can be processed simultaneously under the control of a single instruction stream. Row or column operations on matricies are typical examples. Another type of inherent parallelism results from the requirement to perform several relatively independent processing operations and different sets of data between two points in the program.

Hardware and economic reasons for parallel processing may sometimes outweigh functional ones. Some of the new developments and technologies, such as large-scale integrated circuits, LSI semiconductor memories, nondestructive-read out plated wire memories, optical techniques, etc., may be used more effectively in some form of parallel organization. A highly parallel organization or functional organization provides an approach to the repetitive use of large arrays. In a parallel system implemented with LSI arrays, a few chips may be unique, but a large percentage could be identical, thus providing design and manufacturing economies.

Recent advances in LSI memory techniques also tend to make parallel organizations attractive. It is no longer economically necessary to design around a large central memory. The total internal memory capacity can be distributed over a number of smaller memory modules which can be accessed in parallel.

Programming Languages in the Parallel Environment

Computer programming languages have evolved considerably; however, they share the problems encountered with natural languages. While there are many rules and regulations to be observed, there exists no formal definition or means of constructing them. The exception to this is ALGOL[1], which was the first language in which the syntax was defined with a formal notation. It is interesting however, that a large class of users found the notation difficult to read. The very value of the formal definition contributed to a lesser usage of the language simply because it discouraged people. The view has been taken that the only complete definition of a language is a compiler for that language. This is justified in some respects since the rule and regulations are well-documented and a program can be written and executed rather than be limited to abstract discussion.

Since parallel processing is in its formative stages, with many eventualities still unexplored, a formal definition for Paltran will not be attempted at this time. Rather, the language will be described in terms of its technical characteristics with the view that future compilers will be based on them. Since parallel processing systems will tend to be somewhat unique and have their own individual language requirements, a detailed description of Paltran will not be given. The various forms of input/output statements, compiler directives, etc. will be determined by the needs of individual installations.

Before discussing parallel processing further, certain general objectives as to the proposed form of the language may be stated. The first languages to gain wide acceptance were those that enabled problems in the scientific and engineering fields to be solved. FORTRAN[2] is the best example of this. It is also interesting to note that the first electronic digital computers were "number crunchers" rather than list, string or character processors. Existing computers that are organized in a parallel fashion are all primarily designed for mathematical computation. In this regard, Paltran is to be a procedure-oriented, problem-oriented, and problem solving language, designed primarily for numeric, scientific calculation. As experience with parallel processing in general is gained, list and string, as well as business data processing capabilities, may be added. This implies that Paltran is to be a general purpose computer programming language and is not to be restricted to specific machines or applications.

Experience with other languages has shown that certain types of operations or commands tend to be common to these languages. In general, some means of expressing mathematical problems (equations, formulae, etc.) are provided. The results of calculations performed during these operations can usually be tested or examined, and decisions to alter data or program flow can be made. Some method of repeating portions of programs many times in the form of loop control commands are provided as well as means for effecting data transmission (I/O), data naming, memory allocation, etc. Some languages also have provisions for error checking and debugging.

In addition to these standard features, certain mathematical operations with inherent parallelism should be incorporated in Paltran. Such operations include, matrix operations, numerical integration and differentiation, sum and continued product operations, simultaneous evaluation of functions over given ranges, etc. Since data involved in these operations tend to be grouped in large structures, suitable means for shaping or manipulating the physical form of the data would be desirable.

With these preliminaries in mind, we see that Paltran is to be a new language with capabilities of describing mathematical operations that have natural parallel structures.

The next chapter further discusses parallel processing; in light of this, a preliminary outline of Paltran is described.

CHAPTER TWO

PARALLEL PROCESSING AND PALTRAN

Introduction

Computing speed can increase by several orders of magnitude by permitting many operations to be done at the same time. It will be generally agreed that increases in computation speed is a good thing; in some instances, the objective of having faster computers may be seriously questioned. The long-standing problem has been that of idling processors while INPUT/OUTPUT equipment and other peripherals took their time. The earliest and most common form of parallel processing was (and still is) in fact simultaneous I/O computation. Numerous satellite computers, connected to data channels would process all input and output (i.e. formatting, conversion, peripheral control, etc.) perhaps through as many as three or more buffering stages. These satellite computers were relatively slow, while a fast central processor was reserved exclusively for calculation. However, as new and larger problems are tackled (where I/O is relatively unimportant), the age of the processor-bound computer is fast approaching.

Virtually every computer made to this day incorporates only one central processor and runs in a sequential mode when executing instructions. There are configurations of two central processor machines that run in a foreground-background mode, as well as configurations with many peripheral processors in addition to the one central processor. These are attempts to reduce the time needed to execute a sequence of instructions, but still only one major operation at a time is actually done. In order to decrease the time taken to do a given problem, two solutions are apparent. One way (the most obvious up to now) is to increase the operating speed of the hardware. However, there are limitations to this solution which are finite in nature. The switching times of electronic components are now comparable to the time taken by pulses to propagate (i.e., at approximately the time taken by light to travel a similar distance) along wiring between these components. Since there is a limit on how small components (and systems) can be made, a maximum possible operating speed will be reached. In fact, modern 4th-generation computers are faced by these problems. The other way of increasing through-put is to do more things at once.

Forms of Parallel Processing

Certain problems lend themselves more to parallel execution than others. If arrays or vectors of many elements are involved, the savings are very large. For example: If one processor is available and it is required to multiply an n x n matrix by a scalar, then n^2 sequential multiplications must occur. If n^2 or more processors are available, the multiplications can be carried out all at the same time and a speed factor of n^2 is gained. For some problems, it may be possible to segment the program into parts which can be run independently and, at the same time, combining results at the end.

On a smaller scale, parallel processing can take on slightly different forms. Loops like Fortran DO and Algol FOR can sometimes be eliminated; arithmetic strings can be decomposed into independent substrings and calculated in parallel.

The analysis of existing programs in order to discover parallelism in the algorithm has received considerable attention in the literature[3]. When two successive operations reference distinct variables, they may be performed simultaneously or in either order. Subexpression analysis has been performed to discover parallelism within arithmetic expressions. As an example, consider the two statements:

$X = A+B + C*D$

$Y = E/G + F*H$

The input sets for the expressions are disjoint and each statement may be calculated independently and at the same time. On a smaller scale, the subexpressions A+B, C*D, E/G, and F*H can also be calculated in parallel. Numerous software and hardware[4] algorithms have been developed to implement this aspect of parallel processing. In general, the programmer has no control over how a given program will be broken down for execution.

Scope of Parallel Processing

System organization as found in existing or proposed hardware can be examined in the light of the previously discussed forms of parallel processing. Parallel processing is interpreted broadly to include any type of system organization in which multiple operations are accomplished simultaneously, or multiple hardware control or processing units are working simultaneously. This includes multicomputer systems, multiprocessors, associative processors, array or network processors, and functionally partitioned systems. Parallel processors have been broadly classified into three categories[5] which consist of general purpose network computers, special purpose network computers characterized by global parallelism, and nonglobal computers where each module is only semiindependent or locally parallel.

General purpose network computers can be further divided into parallel networks with a common central control and parallel networks with many identical processing elements, each capable of independent execution. Special purpose network computers can be divided into pattern processors and associative processors.

Another approach to classification is to consider classes of units capable of parallel operation, namely: control functions, functional processing units and data streams.

Parallel control units can simultaneously provide independent instruction streams, either operating on parts of the same problem or on different problems. Parallel functional processing units, which may be identical or which may differ, can simultaneously operate on either a single or multiple data stream under control of a single or multiple control unit. Parallel data streams may be operated upon under the control of an identical instruction sequence performing the same operation on each data stream simultaneously, or by independent instruction sequences operating independently on each data stream. Although this approach to classification is relatively clear cut from the conceptual standpoint, many of the parallel processing systems that have been developed or proposed do not fall neatly into one or the other of these categories. In fact, some of them involve parallel control units, parallel functional processing units, and parallel data streams in the same system.

If existing parallel systems are grouped together, the following four categories can be established.

1. Multicomputers and Multiprocessors

These systems consist of several complete computers interconnected in a manner which facilitates the transfer of data and the assignment of processing tasks between them. This permits individual units to work on different parts of the same problem or on different programs in an overall problem. Other systems include several processing units sharing common memory and common I/O equipment. A high level control unit may be used to control the transfer of data and to assign tasks and sequences of operations between the different processors. The IBM 9020[6] is an example of such a system.

2. Associative Processors

An associative processor utilizes associative memory techniques but with the inclusion of additional processing logic at each cell or word location to permit actual processing operations on each word of data. As associative memory is a device capable of retrieving stored data by means of testing part or all of the contents of each word simultaneously (by hardware means) in order to find one or many desired words. An associative processor performs one operation on N operands simultaneously. An example of such a system is the Goodyear Associative Processor[7].

3. Network or Array Processors

Network or array processors involve a large number of processing units interconnected in some form of network, frequently a matrix. Some array processors have completely distributed control functions, although recent designs include a central control in addition to the local control within each processing unit. The ILLIAC IV[8] system and Litton's Block Oriented Computer[9] (BOC) are examples.

4. Functional Organizations

A functional organization is one in which a number of functional modules are provided to permit performing different types of operations concurrently on a different data item within a single program or on different programs. This type of organization is an extension of the multiprocessor organization in which several different types of processors are used. The processors are smaller and are functionally organized. The further distinction is that each of the functional organizations in this type of machine may not have an internal programmable control unit. In other words, each of the functional units is hard-wired to perform a specific type of operation. The CDC 6600[10] is typical of these systems.

The choice of programming languages to complement the above systems is not an easy one. Multicomputer and functionally organized systems support virtually all of the common high-level languages as well as several levels of assembly and macro languages. Associative processors and array processors are restricted almost entirely to machine languages at this point in time. A general purpose language to satisfy the needs of all these systems, in an efficient manner, is not likely to be found. The practicality of designing one may be questioned since various types of hardware cannot accommodate divergent classes of problems.

If Paltran is to be a general purpose language, a problem written in another, similar language (e.g. FORTRAN), must also be capable of solution in Paltran. Conversely, any high-order mathematical operations found in Paltran should be expressible as subroutines or subprograms in other languages. A parallel computing system suitable for Paltran operation should have a default "sequential mode." This mode can be entered whenever a high-order operation cannot run efficiently on the principle hardware, however, the effect should be completely transparent to the user. As with any computing system, the user should be aware of the machine's capabilities and make suitable alowances.

Paltran Objectives

The primary objective of Paltran is to give the user a free and natural format for expressing problems that exhibit natural parallelism. These problems are to be of a mathematical nature and will be array-oriented, hence the basic data structure will be the array (vectors, matricies, etc.)*. Paltran will therefore tend to be associated with network or array processing systems, although this is not a hard-and-fast rule. Simulations of Paltran compilers on sequential machines may in fact become quite popular until more experience with parallel

* An interesting historical note: As early as 1957, there existed the Matrix Compiler[11] which ran on the UNIVAC machines. It provided users with language and facilities for performing a number of operations on matricies including addition, multiplication, inversion and transposition.

processing is made.

It is not necessarily a function of Paltran compilers to recognize parallel structures in sequentially written programs, nor to optimize existing code so as to take advantage of the given hardware. In general, the programmer is not free to initiate his own parallel structures by use of such operations as FORK and JOIN[12].

The popularity of time-sharing and interactive terminals indicates that Paltran should be formulated as an on-line language. A batch version presents little problem since this would be a proper subset of on-line Paltran and somewhat less restrictive in format.

Mathematical Objectives

The mathematical objectives sought in the Paltran language are two-fold; standard arithmetic operations must be available as exemplified in other programming languages, and a special set of operations dealing with parallel structures must be included. The problem then becomes one of finding these operations and determining their feasibility for inclusion in a general purpose programming language. The requirement that these operations be of a general nature is important; otherwise, the ensuing language will become extremely complex, replete with specific and particular operations, and have a cumbersome and perhaps unreadable notation.

The availability of hardware in the form of array processors strongly suggests that operations on arrays be included in the language. Linear arrays (vectors) and rectangular arrays (matricies) are the most common and parallel operations such as vector addition and multiplication of a matrix by a scalar are readily visualized and straightforward to implement. Matrix multiplication and inversion consists of row and column operations which can be done in parallel. Similarly, the determinant, eigenvalues, and eigenvectors of a square matrix can be found in a similar manner.

Operations on higher order arrays (i.e. 3-dimensional and greater) can be performed in terms of matrix and vector operations.

Experience has shown that two other classes of mathematical operations should be included in Paltran. These include numerical integration and differentiation, and sum and continued product operations.

Integration

In some cases, functions can be integrated by direct analytical means; in others, it may be profitable to use numerical techniques due to the complexity involved. Some functions have no analytic solution and must be integrated numerically; the same applies to experimental data. Numerical integration consists of dividing a function into small increments, over a given range, and summing the areas bounded by the function in these intervals. The time-consuming step in numerical integration involves evaluating the function repeatedly, especially if the function is complex (i.e. consisting of many terms). The availability of many processors enables these evaluations to take place simultaneously, greatly reducing the time needed to calculate the integral. If the range of integration is large (or the function changes rapidly in the range), it may be subdivided and the process carried out in several passes. Infinite integrals can be tackled in this manner -- the range of integration can be split, for example.

$$\int_a^{\infty} f(x)dx = \int_a^{2a} f(x)dx + \int_{2a}^{3a} f(x)dx + \ldots + \int_{na}^{(n+1)a} f(x)dx + \ldots$$

When the value of the integral for an interval (an, a(n+1)) changes by less than some given amount from the previous interval (a(n-1), a(n-2)), the process may be terminated.

The Trapezoidal Rule or Simpson's Rule can be used effectively in the parallel environment, however, some reservation is needed. While it is possible to divide the function (or data) into increasingly small intervals while maintaining good speed, a point will be reached where further subdivision will actually impair accuracy due to accumulation of round-off error and loss of precision. While integration will be included as a Paltran operation, the accuracy expected and actual methods used will be determined by the individual compiler and programming system.

The availability of a successful integral operator makes possible the convenient formulation and solution of a very large class of problems; however, the inclusion of explicit statements for solutions to Exponential, Fresnel, Elliptic, etc. integrals would overly complicate the language. The following do occur frequently enough to merit inclusion in the language as functions:

The Gamma Function

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} \, dt \qquad z > 0$$

noting $\Gamma(z+1) = z!$

The Error Function

$$\mathrm{erf}(z) = (2/\pi^{\frac{1}{2}}) \int_0^z \exp(-t^2) \, dt$$

Bessel Functions of the First Kind

$$J_v(z) = \frac{2(z/2)^v}{\Gamma(v+\frac{1}{2})\pi^{\frac{1}{2}}} \int_0^{\pi/2} \cos(z\sin\phi)(\cos\phi)^{2v}\, d\phi$$

Differentiation

The derivative of some function $f(x)$ at x_0 is defined:

$$f'(x_0) = \lim_{x\to x_0} \frac{f(x)-f(x_0)}{x-x_0}$$

An approximation to the derivative may be defined:

$$f(x,\dot{x_0}) = \frac{f(x) - f(x_0)}{x-x_0} \qquad x \neq x_0$$

where $f(x,x_0)$ is termed the first finite divided difference relative to the arguments (x,x_0). The first finite divided difference is related to the first derivative provided that the continuity and differentiability restrictions of the mean value theorem are met. The concept of second, third, etc., differences can be extended to permit approximations to higher derivatives. As an example, the table below lists the divided differences for the function $f(x) = x^3 - 2x^2 + 7x - 5$.

i	x_i	$f(x_i)$	DD_1	DD_2	DD_3	DD_4	DD_5
0	0	-5					
1	1	1	6				
2	3	25	12	2			
3	4	55	30	6	1		
4	6	181	63	11	1	0	
5	7	289	108	15	1	0	0

Divided Differences for

$x^3 - 2x^2 + 7x - 5$

A vector of n elements will yield n-1 first differences, n-2 second differences, etc. A lower triangular matrix (or upper triangular matrix) of dimensions (n-1) containing the (n-1) divided differences can be generated from the vectors (of length n) of the functional values and their respective spacing. It should be noted that if f(x) is a polynomial of degree n and m data points are taken where $m > n$, there will appear zero entries for differences of order greater than n. This is also illustrated in the above table. As with integration, functional evaluations can take place in parallel, and thus shorten the time needed to evaluate the divided difference matrix.

When base-point values are equally spaced so that $x_1 - x_0 = x_2 - x_1 = \ldots\ldots = x_n - x_{n-1} = h$, some simplification of the divided-difference table (and corresponding matrix) is possible. This enables the finite forward, central, and backward differences of the function to be calculated; these are identical except for subscripts of the base points, and the forward differences can be related to the divided differences by $f(x,x_0) = \Delta f(x_0)/h$. The forward difference operator is represented by Δ , noting: $\Delta f(x) = f(x+h) - f(x)$ etc.

Finite differences appear in many areas such as interpolation, solution of differential equations, boundary value problems, etc.

Extreme care must be taken where any operations involving differentiation is performed as errors tend to be magnified rather than smoothed as with integration. While proper analysis is necessary before any computer solution is attempted, this need is even greater with Paltran. High-order operations used indiscriminately can create more problems rather than solve them.

Integration and Differentiation of Polynomials

Consider as an example the polynomial, $p(x) = 3x^3 + 2x + 5$. Integration and differentiation of this polynomial yield: $\int p(x)dx = 4 \cdot 3x^4 + 2 \cdot 2x^2 + 1 \cdot 5x$ + a constant and $D_x p(x) = 1/3 \cdot 3x^2 + \frac{1}{2}x$ noting that $\int x^n dx = \frac{1}{(n+1)} x^{(n+1)}$ and $D_x x^n = n \cdot x^{n-1}$.

If vectors containing the coefficients of polynomials are available, integration or differentiation can take place directly. Thus, if all polynomials are represented by the series $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots\ldots + a_n x^n$ the integral polynomial is, to an arbitrary constant

$p'(x) = a_0 x + \frac{1}{2} a_1 x^2 + \ldots\ldots + \frac{1}{n+1} a_n x^{n+1}$

and the differential polynomial is

$p''(x) = 0 + a_1 + 2a_2 x + \ldots\ldots + n a_n x^{n-1}$.

Thus the vectors

$$\begin{vmatrix} a_0 \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ a_n \end{vmatrix} \quad \begin{vmatrix} a_0 \\ a_1/2 \\ a_2/3 \\ \cdot \\ \cdot \\ a_n/(n+1) \end{vmatrix} \quad \begin{vmatrix} 0 \\ a_1 \\ 2a_2 \\ \cdot \\ \cdot \\ n \cdot a_n \end{vmatrix}$$

represent the polynomial and its integral and derivative respectively, taking account of the shifts in the powers of x.

Since polynomials are used extensively in scientific work, these forms of integration and differentiation should also be included in Paltran.

Sum and Continued Product Operations

As an introduction to sum and continued product operations let us consider the factorial function. The factorial of some number n is defined as: $n! = n(n-1)(n-2)\ldots3\cdot2\cdot1$ and $0! = 1$. Normally this operation requires (n-1) subtractions and (n-1) multiplications to be performed, however if n/2 processors are available the factorial can be computed as follows:

- An indexed LOAD instruction generates the vector whose elements are 1,2,3,....n and each pair of elements (1,2), (3,4), ... (n-1, n) or (n,1) if n is odd, is assigned storage to a single processor.
- The product of each pair is determined simultaneously and the results are again paired and multiplied until only a single number remains.

Expressions of the form

$$\sum_{i=0}^{n} e_i = e_0 + e_1 + \ldots\ldots + e_n \qquad \text{and}$$

$$\prod_{j=0}^{m} E_j = E_0 \times E_1 \times \ldots\ldots \times E_m$$

can be evaluated in this manner, noting that the factorial function is simply a specific case of the continued product operation. The savings are increased with the complexity of the expression, since all e_i or E_j are evaluated simultaneously.

The availability of the sum and product operators make possible the generation and solution of many functions; the following examples occur frequently and will be included in Paltran:

The product operation permits the factorial function to be calculated; as a direct consequence, the number of combinations and permutations of n (dissimilar) things taken r at a time $nCr = \binom{n}{r} = n!/(n-r)!r!$ and

$nPr = n!/(n-r)!$ can be found.

Combining the preceeding operation and the sum operation the Bernoulli Numbers can be readily found by using the recursive definition:

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k \quad \text{for } n \geqslant 1 \quad \text{and } B_0 = 1,\ B_1 = -\tfrac{1}{2}$$

$$B_{2n+1} = 0 \text{ for all integral } n \geqslant 0.$$

The same techniques can be used for determining the coefficients of, or solving orthogonal polynomials. Any set of polynomials ($f_n(x)$) with the property

$$\int_a^b w(x)f_n(x)f_m(x)dx = 0 \quad \text{for } m \neq n$$
$$= h_n \quad \text{for } m=n$$

is called a set of orthoganol polynomials on the interval (a,b) with respect to the weighting function w(x). The four most common are the Chebyshev, Hermite, Laguerre and Legendre polynomials; as an example the Laguerre Polynomial has explicit expression:

$$L_n(x) = \sum_{m=0}^{n} (-1)^m \binom{n}{m} X^m/m!$$

which is a combination of the sum operation, factorial operation and binomial coefficient $\binom{n}{m}$.

It should be noted that these polynomials can also be determined from a general recursive definition and this may be the better method if a family of polynomials is required.

Finally, the sum and product operations can be combined to determine interpolating polynomials, that is the polynomial of lowest degree which passes through n points (x_i, y_i), $i = 1,2,3,....n$ is given by:

$$p(x) = \sum_{i=1}^{n} \left(y_i \prod_{k=1}^{n} (x-x_k)/(x_i-x_k) \right) \quad k \neq i$$

where $x_i \neq x_k$ for $k \neq i$.

Functional Analysis

The process to be described may be considered too specific to be included in a general purpose programming language, however it is a good example of how parallel processing can make possible methods which have been impracticle in the past.

The availability of many processing units makes possible a somewhat unusual technique of analyzing functions for the occurence of zeros, maxima or minima. Let $f(x) = y$ be some function defined over the range (xmax, xmin) in which at least one xero, maximum or minimum is assumed to exist. The objective is then to find the value x^* for which $f(x^*) = Q$, where Q is the required condition. In operation, the function is evaluated over (xmax, xmin) in n steps (assuming n processors) and the resulting array searched for the required condition, noting the value of x for which this occurs. If n is sufficiently large, the answer may be found on the first pass, if not, new values of (xmax,xmin) are established around x^* for which $f(x^*) \cong Q$ and the process repeated until sufficient accuracy has been attained. In general there may be any number of zeros or extrema in the given range.

The speed at which the answer is found is quite considerable since only one effective function evaluation is performed per iteration. If a large number of processors are available the value of x^* approaches the final value by several orders of magnitude for each pass, thus necessitating only a few iterations. It should be noted however, that new ranges (xmax, xmin) cannot be set arbitrarily close to the approximation x^* , since round-off and truncation error can accumulate and create problems of overshoot. Programs simulating one hundred processors configured linearly, have shown that usually less than ten iterations are needed to solve any arbitrary equation ($f(x) = 0$), to accuracies of order 10^{-5} or better, even when the initial range chosen is very wide. An example of such a simulation appears in the Paltran - 8 manual. (Appendix IV)

The power of this method becomes even more apparent if a rectangular configuration of processors is used to investigate functions of two variables. The method of solution is similar, however additional considerations as to locating extrema must be made. Consider the following table which lists values for $y^2+x = z$ over the ranges $-3 \geqq x,y \geqq +3$:

X	-3	-2	-1	0	1	2	3
Y							
-3	6	7	8	9	10	11	12
-2	1	2	3	4	5	6	7
-1	-2	-1	0	1	2	3	4
0	-3	-2	-1	0	1	2	3
1	-2	-1	0	1	2	3	4
2	1	2	3	4	5	6	7
3	6	7	8	9	10	11	12

VALUES OF $y^2 + x$

The function describes a parabolic surface, and for any value of x there always occurs a minimum value of y in the z - direction. For any value of y there is no maximum or minimum value of x, other than at the end points of the range. The search for extrema must now be made in three ways. The rows and columns of the corresponding matrix must be individually tested and each point Z_{ij} ($= f(x_i, y_j)$) must be compared to its nearest non-diagonal neighbours, $Z(\pm i, j)$; $Z(i, \pm j)$ or to its nearest diagonal neighbours, $Z(\pm i, \pm j)$, excluding the peripheral elements. The latter method will reveal any global or local extrema for values of both x and y.

Functions of three variables (or more) can be handled in a similar manner; however, problems of visualization occur. For this reason, it is perhaps advantageous to decompose all higher order arrays into matricies and perform analysis at this level.

A specific statement or command to implement this technique will not be incorporated in Paltran due to the many possible variations of handling different arrays. It will be seen in succeeding chapters, however, that other Paltran statements can very easily be grouped in subroutine form to perform this analysis; as experience is gained, this technique may be incorporated as a single statement.

TECHNICAL CHARACTERISTICS OF PALTRAN

The technical characteristics of Paltran may now be examined in light of the proposed mathematical operations. These will include data organization, arithmetic expressions, the character set, and auxiliary statements.

Data Organization

As stated in Chapter One, the basic Paltran data structure will be the array. In general, any arithmetic operation that can be performed on a simple variable can also be performed automatically, on any size array, on an element-by-element basis.

Simple variables or scalars are the least structured data, being zero-dimensional. Vector variables can be of two types. The numeric vector holds only floating point (or integer, etc.) numbers and all arithmetic operations apply. Numeric vectors are always considered to be column vectors and print (on output) in this fashion. A character vector may hold alphanumeric data as well as any character in the Paltran set. This is literal data and has no numeric value, except where certain operations result in a null character vector, which has the value zero. Character vectors are always considered to be row vectors and print in this fashion.

Higher-order arrays are all restricted to numeric data. Special operations may be performed on vectors and matricies (i.e., matrix multiplication, inversion, etc.); operations on higher-order arrays (for example, vector multiplication of 3-d arrays) will not at this time be given explicit definition. They may be easily formulated with the basic statements in the language and called as subroutines.

Data may be generated by the user or incorporated directly in the program. In general, data will be written on some external medium (tapes, disks, etc.) and be processed by the program.

Arithmetic Expressions

Since arithmetic expressions form the basis of all the operations described under Mathematical Objectives, it is important to define what is meant by this term as well as describe the different types of arithmetic likely to be implemented.

An arithmetic expression will be defined as any set of variables or constants combined by the binary or didactic operators: (in order of precedence), (↑, *, /, +, -) that is, exponentiation, multiplication, division, addition and subtraction. Evaluation will take place left-to-right with exponentiation being performed first, followed by multiplication, etc. Parenthesis may be used to change the order of evaluation; square and angle brackets may be used interchangeably with parenthesis to improve readability. Functions may be included in arithmetic expressions and have precedence equal to or greater than that of exponentiation, hence, they are evaluated first. A function may have another arithmetic expression as its argument, since an expression can always be reduced to a single number (or group of numbers, in the case of arrays). The unary operators (+, -, !) may also appear in arithmetic expressions. The factorial operator (!) is really a function and treated as such. When applied to an array, the factorial of each element would be calculated. No two operators may appear together except for the factorial operator.

Numeric constants may be used in any arithmetic expression. While they are zero-dimensional, they may be combined with any array regardless of its size or dimensions.

Arithmetic expressions along with the equal sign (=) form the simple assignment statement: v=e where _v_ is a variable and _e_ is any legal expression. The variable _v_ and all variables present in _e_ must be of the same dimensions.

The type of variables used in arithmetic expressions will reflect the five possible means of performing arithmetic, namely, in integer, floating-point, complex, octal and logical format. Details as to naming variables and determining their type will be found in Chapter Three.

The possibility of performing arithmetic in five ways results in problems that can occur when different types of variables appear in the same expression. The handling of mixed modes is usually tackled in different ways by individual compilers for a given installation, but the following guidelines will be proposed.

A diagnostic is given to inform the user that a mixed mode expression has occurred. Each variable in the expression is then converted to an equivalent floating point quantity (e.g. absolute value of complex quantity found, logical Ø=0.000, etc.) and the expression is evaluated in floating-point format. The resulting value is then converted to the mode of the replacement (left-hand) variable.

The safest rule to follow when dealing with mixed mode expressions is simply to avoid them.

The mathematical operations in Paltran are fundamental to the intended usage in numeric computation, however, these must be supported by auxilliary commands to shape, test, modify and prepare the data. In addition, considerations must be given to the physical form in which the language will appear. These consist of the choice of a character set, punctuation characters employed, the manner in which key words or statements are delimited, the possible inclusion of subprograms or programs in other languages, execution sequence, etc. The remainder of this chapter is devoted to these considerations.

Before discussing the auxilliary statements, it is useful to examine the requirements for a character set, since all Paltran statements will be constructed from it.

The Character Set

The Paltran character set will be defined as that set of symbols needed to clearly and conveniently express the statements of the language. The definition is intentionally broad. Input/Output equipment is becoming sufficiently sophisticated, so that the user should not be limited to the capital alphabetic characters, the numeric characters and a few special symbols such as *, =, /, etc. A string of alphanumeric characters can be replaced by a corresponding special symbol whenever its usage is common and unambiguous. It is clearly better to use $\int$, $\sum$ and $\prod$ rather than "INTEGRATE", "SUM OVER", and "DETERMINE THE CONTINUED PRODUCT". On the other hand, it is more meaningful to use alphanumerics whenever there does not exist a special symbol to denote the process or operation. For example, s = MAX (v) is intuitively closer to "Set the scalar s, equal to the maximum element of vector v", than s =$\bigcirc$v or s =$\wedge$v.

The availability of lower case letters is a desirable feature. Variable, subroutine, or function names can be restricted to lower case letters (and numbers), while key words such as commands, in-line functions, etc., are capitalized. This permits immediate identification of non-variables and makes compilation (lexical and syntax analysis) somewhat less difficult. This scheme also alleviates the need of having reserved words in the language or deciphering key words from context. It is also desirable to provide the more common Greek letters, as many quantities in scientific and engineering work bear these designations.

Display generators, plotters, light-pens, etc., are special devices which can produce arbitrary characters and must be treated separately. The proposed Paltran Character Set and key words of the language are described and listed in Appendix I.

The elements of the character set may be classified into three groups: Delimiters, Identifiers and Key Words.

A delimiter is any character which is not a letter or a number. All key words, identifiers and numbers are terminated by delimiters.

An identifier (variable, subroutine, or function name) may consist of up to ten lower case letters or numbers, the first of which must be alphabetic.

Key words are the commands such as READ, SET, FOR, etc., of which statements are composed. In-line (or system defined) functions such as SIN, COS, etc., are also considered to be key words. Key words consist of upper case letters only.

Delimiters, identifiers and key words are combined along with labels to form statements.

Statements and Program structure

Paltran statements will consist of up to three fields. The first is a label field which consists of a STATEMENT NUMBER and is optional. In the batch mode, the label field may be omitted, however, statement numbers must be present in the on-line mode for indirect execution. A blank label field indicates the direct (or desk calculator) mode and results in immediate execution upon receipt of a statement delimiter.

The statement number consists of two parts: a group number x, and a number y denoting a line within the group. A period is used to separate these two parts; however, a statement number is not to be interpreted as a decimal or fixed point number, but rather as two integers. The lowest possible statement number is, therefore, 1.1, that is, line one in group one. There may be any number of lines in a given group. Execution starts at the lowest numbered line and continues with the next highest, regardless of physical order; in the batch mode, unnumbered lines are executed consecutively.

Field two is the command field which identifies the type of statement, e.g., READ, WRITE, SET, etc. If field two is blank, it is assumed to be an assignment statement.

Field three is the data (or operand) field which contains information (i.e. variable names) relating to the command. A statement is terminated by a carriage-return, line-feed, or semi-colon (;), and, in the case of punched card I/O, a semi-colon may be used to separate individual statements on one card. Statement fields are interpreted in context, and commas and blanks may be used freely to improve readability.

A program is composed of several Paltran statements with the possible inclusion of subroutines or user-defined functions. A subroutine is simply a separate set of statements which can be compiled independently and can accept or pass on more than one argument. A function is essentially a subroutine that has only one argument and is called by name rather than by an explicit statement.

Propositions for the auxilliary statements can now be examined.

Conditional Statements

The ability to test or compare the results of numeric calculations has manifested itself in most programming languages in the form of an "IF" statement. These come in two varieties: the arithmetic IF compares the results of some calculation to zero and the execution sequence subsequently changes, depending on whether the number tested was less than, equal to, or greater than zero; the logical IF compares two expressions by means of logical and relational operators, and execution sequence is dependent on the TRUE or FALSE outcome of the test.

In languages dealing with sequential processing, these tests always took place on simple variables or expressions of simple variables. The Paltran IF statement, must, however, make provision for arrays (or parts of arrays) to be tested; this will be effected by comparing arrays on an element-by-element basis. The arithmetic IF will be constructed so that simple variables or arrays may be tested for sign or, against zero, the proviso being that if all elements of an array are similar, the action of the IF statement is the same as that for simple variables. If any one or more elements of an array do not satisfy the requirements (as for simple variables), a default or alternative action occurs. As an example, consider the IF statement:

```
IF (e) n1, n2, n3, n4
```

The n_i refer to statement numbers and (e) is any arithmetic expression composed of simple variables or arrays of identical dimensions. The expression is evaluated and if it consists of arrays, all elements are tested: if all are negative, control passes to statement n1; if all zero, control passes to n2, and, if positive, to n3. If at least one element does not fall into the same categories, control passes to n4. Obviously, if the statement consists only of simple variables, control can never pass to n4.

If the case of the logical IF statement, all elements of an array must satisfy the comparison for the result to be TRUE. If any one corresponding element, of the expressions being compared, does not meet the criterion, the FALSE condition occurs.

Error Condition Detection

All of the high order Paltran operations are subject to errors of one sort or another. In general, each Paltran compiler will handle differently depending on the hardware of the target computer. A machine with many processors, but poor precision may not integrate as well as one with fewer processors but can accomodate high precision arithmetic. A matrix can be inverted by direct, iterative or random (Monte Carlo) methods, each of which can fail depending on the matrix or algorithm used. The user of high order Paltran commands must be aware of the algorithms his particular compiler uses and the limitations of such. Details pertaining to errors and applicability of data sets to the implementing algortithms would normally constitute a large part of the user's manual for a given compiler.

In practice, however, the user cannot always be trusted to take account of possible errors. The problem is compounded by errors occuring even when data sets and algorithms have been correctly matched and formulated. It is possible to detect many user errors at compilation time while certain others can be detected by hardware means at object time. In order to give the user some measure of success on the outcome of high order operations, the following is proposed. An internal vector variable called FLAG will be associated with each Paltran statement and its status may be tested by means of an IF statement. The elements of FLAG will be set equal to zero or FALSE if no error conditions are present, or set to one or TRUE if some error occurs. The individual elements can represent such conditions as incompatible dimensions, convergence failure, mixed modes, singular matrix on inversion, divide by zero, etc. Since compile time and object time are indistinguishable on most on-line systems, FLAG can be used to warn the user of errors detected during compilation, however for more elaborate systems, an actual error message is warranted. It should be noted that FLAG = 0 does not imply accurate or acceptible answers, but only that the possible error conditions have not been detected.

Composition and Decomposition

The majority of Paltran operations can be performed on vectors and matricies. Similar operations can usually be defined on three dimensional and greater arrays but no explicit definition of these will be given at this time. Operations on multi-dimensional arrays can always be formulated in terms of matricies or vectors and some means of breaking up (or building) large arrays are needed. Paltran provides two classes of statements for this use.

The first consists of only a single bi-directional statement (called the PROJECT statement) by which hyperplanes can be projected into planes, planes into rows or columns and so on. Conversely planes can be projected into hyperplanes and hyperplanes into the next highest dimension etc.

The second class consists of the PARTITION and BUILD statements which will operate only on matricies. The PARTITION statement is used to split large matricies into smaller submatricies, while the BUILD is used to construct large matricies from small ones. These statements find use in constructing augmented matricies which occur quite often in linear programming techniques or solutions of simultaneous equations. A second application is hardware and systems oriented. Normally, any size matrix comensurate with memory capacity can be operated on irregardless of the type of hardware. If partitioning occurs, this effect should be transparent to the user. Certain efficiencies can be gained however, if a network or array processor is being used and the user is aware of the number and dimensions of the processors available. He is then at liberty to use these statements to shape data to conform to the dimensions of the hardware.

Loop and Execution Control

There is still a need for indexing and repeated calculation in the parallel enviroment as defined by Paltran. The FOR and DO statements will be used to generate repeatable segements of code.

The DO is a form of execute statement which causes a line or group of lines in the program to be performed. A group of lines when called by a DO therefore form a type of subroutine.

The FOR statement is analgous to the ALGOL-FOR and FORTRAN-DO loops. Statements in the body of the FOR range are repeatedly executed, based on the value of an index variable which varies according to preset parameters. The mechanics of the Paltran FOR-DO statements will be detailed in the next chapter.

Input/Output

The input/output systems of most computers tend to be very complex, sometimes being computer subsystems themselves. The parallel computing system will undoubtedly incorporate very elaborate I/O equipment to maximize throughput and utilize resources efficiently. While means of accessing tapes,disks,mass memory, etc. are needed, these will not be discussed, instead only two I/O statements, geared to simplicity will be proposed. The Paltran I/O statements will be READ and WRITE and will handle simple variables and arrays in pre-determined formats in order to simplify data transmission. These statements will also be discussed in detail in the next chapter.

Miscellaneous Statements

Grouped in this category are statements and commands which are somewhat standard and found in many other languages. These include GOTO statements to change program flow, HALT or STOP statements to terminate execution, SUBROUTINE and FUNCTION statements to construct user-defined subprograms and functions and the various forms of declaratives such as dimension and type statements. Only these latter statements will be discussed further in detail, in the next chapter, which describes the proposed statement forms of the Paltran Language.

Chapter Three

THE PALTRAN PROGRAMMING LANGUAGE

<u>Introduction</u>

This section takes the concepts developed in Chapter Two and builds on them the various forms of proposed Paltran Statements. Extreme details as to the various forms of input/output statements, declarations, arithmetic performed and program structure will not be given as these do not directly relate to parallel processing and tend to be oriented to a particular machine. The language, as described, is considered to be the on-line version; however, text-editing and control statements will not be discussed since these are somewhat standard for any on-line language. A batch version would be a proper subset of on-line Paltran and would be somewhat less restrictive with regard to statement numbering and input/output methods.

The key words and functions normally to be incorporated into a Paltran system are listed in Appendix I.

The statements to be discussed in this chapter form the basic Paltran system and include:

The dimension statement and declaratives.

The Simple assignment.

The SET Statement.

The LOAD Statement.

The ROTATE Statement.

The PROJECT Statement.

The IF Statement.

The FOR and DO Statements.

The PARTITION and BUILD Statements.

The READ and WRITE Statements.

The Dimension Statement and Declaratives

The status of all variables must somehow be defined so that appropriate measures may be taken during compilation. The size and dimensions, as well as the precision and type of arithmetic to be performed on a variable, must be declared explicitly; otherwise, default values will be assumed.

All arrays in Paltran must be declared via a dimension statement:

```
*v(n), v(n,m), v(n,m,r), v(n,m,r,.....)
  or
**c(n),.....
```

The first statement defines a numeric vector variable with n elements, a matrix with n rows and m columns, a rectangular prism with r planes, etc. The second statement defines a character vector which may contain up to n characters.

The dimension statement is unnumbered and non-executable and may appear anywhere in the source program, any number of times, although some compilers may require that it appear before the first executable statement.

Variables which appear in the source program that have not been dimensioned are assumed to be simple numeric variables or scalars. Operations on arrays defined by the dimension statement may be global or local, and subscripts or partial subscripting may be used to access individual elements, rows (columns), planes, etc. of an array, as necessary.

In addition to the dimension statement, the following signal the compiler as to variable type:

REAL	v_i	v_i = v or v(m) or v(n,m) etc.
INTEGER	v_i	REAL refers to floating
COMPLEX	v_i	or fixed point variables.
OCTAL	v_i	
LOGICAL	v_i	

Dimensional information may be included in declarations, however, these statements must precede any executable statements.

The precision of all variable types is assumed to be single, but may be extended by including DOUBLE or TRIPLE with any of the above declaratives.

Examples:

```
    REAL a,b,c,mat(5,5)
    DOUBLE a1,v(10)
    TRIPLE COMPLEX z(4,5,6)
```

It is convenient to assign the following default modes for variables. All variables are assumed to be single precision and REAL, unless the name begins with an i,j,k,l,m,n, in which case an integer variable is assumed.

The Simple Assignment Statement

The simple assignment statement is the 'work-horse' of any numeric scientific programming language, as it conveniently and naturally expresses simple arithmetic operations. This statement consists of a left-hand or replacement variable which is set equal to some arithmetic expression as defined in Chapter Two. The form is quite straightforward: v=e where v is the replacement variable and e is any expression. The only restrictions that apply to the simple assignment statement are that dimensions of all variables must be the same and due consideration must be given to mixed modes. Operations are global when arrays are involved, although subscripts may be used when simple variables are present. Partial subscripts may not be used.

Limited operations on character strings are also a part of the simple assignment statement. Literals or alphanumeric data may be assigned only to 'Character Vectors' which hold non-numeric data and must be declared apart from the numeric vector variable. Operations include assignment, concatenation, extraction, expansion and intersection. The following examples illustrate their use:

ASSIGNMENT

x=abc y=123 KING ST. z=?'''* w=string v=trn

CONCATENATION

q=x+y .˙. q=abc123 KING ST.

EXTRACTION

p=w-v .˙. p=sig

EXPANSION (by a scalar factor)

r=2*x .˙. r=abcabc

s=2*(x+z) .˙. s=abc?'''*abc?'''*

INTERSECTION

Let s1=AEIOU and s2=COMPUTER

and s3=s1!s2

then s3=CUE or EUC or UEC etc.

The size of s3 must be at least that of one of the vectors on the right. The null character (printing as, but not equal to the character BLANK) would fill the remaining (if any) spaces in the vector. If the vectors are disjoint, the result would print as a blank vector and have the numeric value Ø. Note, this is the only numeric value a character vector can have; a vector full of blanks has no numeric value.

The extraction operation can also result in a null vector. All character vectors are considered row vectors and print (or plot, display, etc.) across the 'page' which makes use of these vectors valuable in I/O formatting. The hierarchy of operations are:

```
     *     Expansion
     +,-   Concatenation, Extraction
     !     Intersection
```

Further examples:

```
x=E       y=EACH       z=COMPUTE
     q=y!(2*x+y)       noting 2*x+z=EECOMPUTE
                       q=EAC or CAE or AEC, etc.
```

Other character handling facilities will be added to Paltran as need and use dictate.

<u>The SET Statement</u>

The SET statement is the most powerful of the Paltran statements, as integration, differentiation, sum and continued product operations, and the maxtrix operations (multiplication, inversion, etc.) are defined by it.

There are a large number of individual SET statements, and these are listed in Appendix II along with a brief explanation.

The SET statement is unique in that there are two operator or command fields. It has the general format:

```
     SET v=OPR,a,b,c.....
```

"SET" itself indicates that a high-order parallel operation takes place. The variable <u>v</u> is a general replacement variable and can be a scalar, vector, matrix, etc., depending upon the individual statement. The second command, OPR, consists of a symbol or alphanumeric string to indicate operations such as integrate, sum over, invert a matrix, etc. The remaining field consists of other variables or constants a,b,c,.....

Variables in any SET statement may be referenced globally by name only, or by partial or complete subscripts. As an example, the matrix a(n,m) may be used:

```
OPR a       - operate on a globally
OPR a(I,*)  - operate on row I only
OPR a(*,J)  - operate on column J only
OPR a(I,J)  - operate on element (i,j)
```

The LOAD Statement

It is often necessary to evaluate functions over some given range. This can be done by using the LOAD Statement.

```
LOAD a = (f(x,y,z,....)),x(xmax,xmin),y(ymax,ymin),......
```

The array a is filled with values of the function f(x,y,z,...) over the ranges in n steps (xmax,xmin), (ymax,ymin),.... in increments of Δx = (xmax-xmin)/n etc., hence the function is evaluated from xmin to xmax -Δx, etc.

As an example consider the function f(x) = x↑2 :

```
*v(5)
LOAD v = (x↑2),x(10,0)
```

Hence $\frac{\text{xmax-xmin}}{n} = \frac{10 - 0}{5} = 2$

n	x	v
1	0	0
2	2	4
3	4	16
4	6	36
5	8	64

The ROTATE Statement

The ROTATE statement may be used to shift data in arrays. This can be quite useful in generating displays and other graphical presentations. Rotation or shifting is always circular; the following examples illustrate:

```
*b(3,3), c(1,5)
ROTATE C,2,L
```

- that is, rotate the elements of C 2 places left.

e.g. C = 1,2,3,4,5

C' = 3,4,5,1,2

```
ROTATE b(*,3),2,U
```

- Rotate column 3 of b 2 places up.

b = 11 12 13
    21 22 23
    31 32 33

(the other possible directions are right (R) and down (D).)

b' = 11 12 33

```
b' =  21    22    13
      31    32    23
```

Rotations in 3-dimensional arrays are somewhat more complex since there are six possible directions to move rows, columns or planes. In 4 or greater dimensional arrays, rotations tend to be extremely difficult to visualize due to the number of units and directions.

For this reason, the ROTATE statement is defined only for vector or matrix variables. The PROJECT statement can be used to decompose multi-dimensional arrays into matricies and rotation performed at this level.

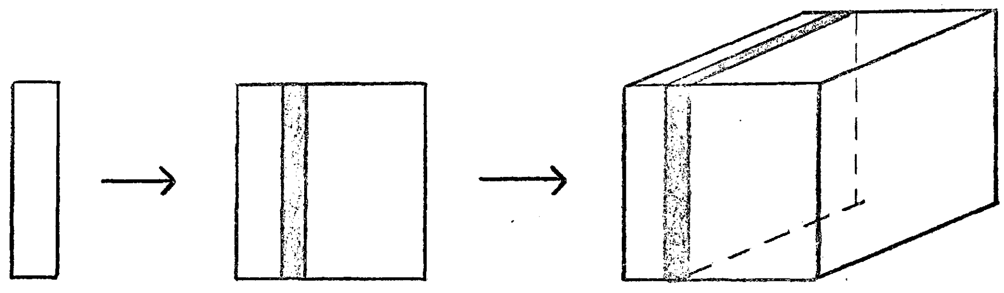
The PROJECT Statement

As discussed in Chapter Two, the PROJECT statement may be used for transferring data between arrays in one, two, or more dimensional blocks. The units to be transferred are denoted by asterisks in the variable names. The statement form is:

```
PROJECT v1(i,j,.....,*,*,.....) INTO v2(k,m,.....,*,*,.....)
```

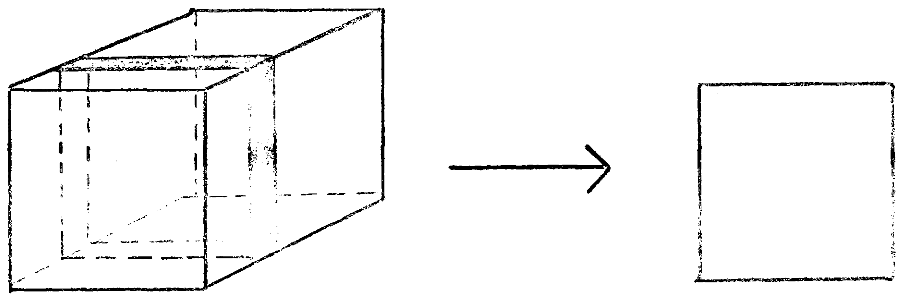
The following examples illustrate use of the PROJECT statement:

```
*a(5),b(5,5),c(5,5,5)
PROJECT  a  INTO  b(*,2)
PROJECT  b  INTO  c(*,2,*)
```



The narrow rectangle represents the elements of a which is projected into column 2 of the square representing matrix b. The previous contents of column 2 are lost. The matrix is then projected into plane 2 of the cube c.

```
*box(5,5,5), square(5,5)
PROJECT box(*,*,2) INTO square.
```



The diagram shows plane 2 of variable box which becomes the matrix square.

The IF Statement

There are two forms of IF statement in Paltran. The arithmetic IF statement has the form:

```
IF (e)n1,n2,n3,n4
```

and has been described in Chapter Two.

The logical IF statement has a slightly different format:

IF (e_1 op e_2)S1 ELSE S2

- e_1 and e_2 are any arithmetic or logical expressions.
- S1 and S2 are any executable Paltran statements, except for another IF.
- op stands for any one of the relational operators:

$=$	(equal to)
$\neq$	(not equal to)
$>$	(greater than)
$\geqq$	(greater than or equal to)
$<$	(less than)
$\leqq$	(less than or equal to)

or any one of the logical operators:

.A.	(And)
.O.	(Or)
.N.	(Not)

(This is one instance where alphanumerics are favoured over special symbols, in order to avoid ambiguity.)

In operation the two expressions e_1 and e_2 are evaluated and the resulting relational or logical expression is tested. If the result is TRUE statement S1 is performed, otherwise (ELSE) statement S2 is executed, indicating a FALSE result. Control passes to the next executable statement after S1 or S2 has been performed, unless a GOTO transfer is made.

If arrays are involved, the comparison takes place on an element by element basis and dimensional integrity must be maintained. If all elements in the evaluated expression result in a TRUE condition, statement S1 is performed. If any one element produces a FALSE outcome, Statement S2 is executed.

A second form of logical IF statement may be used to test portions of arrays by means of partial subscripting, on an element by element basis:

```
IF (a op c)S1 ELSE S2
```

- <u>a</u> is any array which may be referenced globally or by means of partial subscripts.
- op is any one of the relational operators
- <u>c</u> is a conditional expression which can be any arithmetic expression containing relational or logical operators whose dimensions are compatible with those of the array being tested.

<u>Examples</u>:

```
IF ( a(*,2) = 0.0) SET a(*,2) = -1.0 ELSE GOTO 1.1
```

The above statement tests column 2 of the matrix <u>a</u> against zero. If column 2 equals zero, it is changed to -1.0, if not, transfer is made to line 1.1.

```
*time(10,10,10), mat(10,10)
IF (time(*,*,10)> mat) HALT ELSE DO 5
```

In this case the tenth plane of array time is compared to the matrix mat. If all elements in plane 10 of time are greater than the corresponding elements in mat the program HALTS otherwise group 5 statements are performed.

The FOR and DO Statements

The FOR and DO statements may be used to generate repeatable segments of code.

The DO statement causes execution of a single line or a group of lines and has form:

```
DO     x   or


DO     x.y
```

- where x is a group number and x.y is a number of an individual statement. Upon completion of the line or group, control passes to the next highest-numbered statement after the DO, or the next sequential statement.

The DO statement must not reference its own line or group.

The FOR statement has the simple format:

```
FOR i = x,y,z; s1;s2;s3;.....
```

The index variable is the scalar i̲ and is initially set equal to x̲. The group of statements s1,s2,s3,..... are then executed at least once. The index variable is then incremented by y̲ and tested. If it is greater or equal to z̲, the loop is finished and control passes to the next statement; if not, s1,s2,s3,.....are again executed. The variables (or constants) i,x,y, and z may be integers or floating point quantities.

The body of the FOR loop is limited to one line; however, its range may be extended indefinitely by means of the DO statement. FOR loops may be nested, the innermost being performed first. Since all FOR loops are limited to one line, transfer into a FOR loop is impossible. Example of a FOR loop:

```
1.10 FOR i=1,1,10; FOR j = -100,25.9,-11; DO 2; WRITE a,b,c.
```

The PARTITION and BUILD Statements

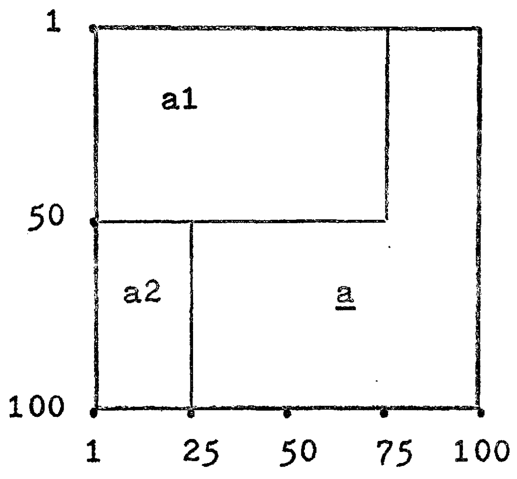
The PARTITION statement is used to split large matricies into smaller, submatricies. It has the form:

```
                                               ROWS
     PARTITION (x) INTO (x1,x2,x3,.....) BY
                                               COLUMNS
```

Examples:

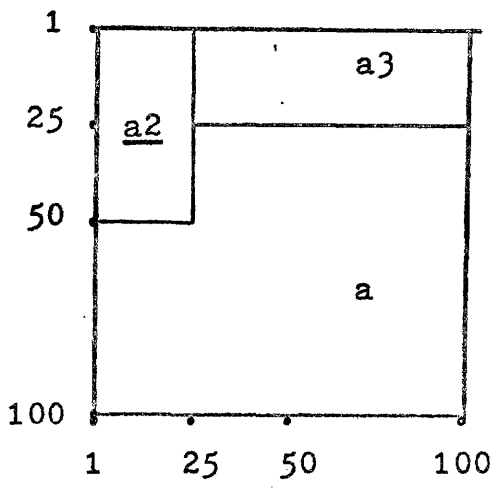
```
     *a(100,100), a1(50,75),a2(50,25),a3(25,75)
     PARTITION (a) INTO (a1,a2) BY ROWS
```

e.g.



```
     PARTITION (a) INTO (a2,a3) BY COLUMNS
```

e.g.



The BUILD statement is the converse of the PARTITION statement and is used to construct large matricies from smaller ones:

$$\text{BUILD (v) FROM (v1,v2,v3,.....) BY} \begin{matrix}\text{ROWS}\\ \text{COLUMNS}\end{matrix}$$

The PARTITION and BUILD statements may be used for matrix variables only.

The READ and WRITE Statements

The aim of Paltran input/output statements is simplicity and convenience for the user, hence only the READ and WRITE statements will be formally proposed. The format for these statements is:

READ/WRITE n VAR1, VAR2, !,←,"....", $, #,......

The appropriate I/O device is represented by the integer n. If n is omitted, the system device is selected. For purposes of discussion, this is assumed to be a teletype-like terminal. VAR1 refers to any numeric variable. Simple variables are written across the 'page' as usual, while vector variables are automatically written in column-wise fashion, down the page, along with an index denoting the elements of the vector. Matrix variables are also automatically printed. The numbers of rows in a matrix presents no problem, since the 'page' of most printers can be infinitely long. The width, however, is usually limited from 72 to 130 characters, and a large number of columns cannot be accommodated on a single page. As many columns as possible are printed across the page, taking into account field width, intra-column spacing and indicies. Remaining columns of the matrix variable are printed on succeeding pages. Higher-order arrays can be printed in terms of matricies. Suitable indicies are automatically supplied to identify rows, columns, planes, hyperplanes, etc. (Refer to the Paltran-8 Manual for examples.)

The input format is less restrictive. Vector and matrix variables are always read column-wise. Higher-order arrays are read in terms of matricies. An index is supplied to guide user input for on-line systems.

VAR2 is any collection of character vectors, which are always printed as rows. Text may also be enclosed in quotation marks, ("......") and any character except (") may be used. The (!) produces a carriage-return, line-feed (line-advance on a printer), and (⟵) produces a carriage-return only (reset left margin on a printer), and ($) produces a form feed (skip page). On input, these commands may be used for on-line systems; if card readers, paper tape readers, etc., are being used, they will have no meaning except as delimiters.

Numeric output format may be set by using (#). The format is initially set to "E" --format and the field width is the maximum number of significant digits that single precision allows. The occurrence of the first (#) changes this to:

#X.Y	-"F" - format with X digits and Y decimal places.
#X	-"I" - format, X integers.
#EX.Y	-"E" - format with X digits and Y decimals.
#	-"E" - format, full precision
#CX.Y	-complex output format, real and imaginary parts are written in succession; an i or j is automatically supplied to denote the imaginary part.
#OX.Y	-Octal output.

The format remains the same for succeeding numbers until the occurrence of another #.

On input, format is self-adjusting according to the variable type.

Chapter Four

PALTRAN IMPLEMENTATION

The basic Paltran statements have been developed in the last two chapters, however additional software is needed to make a truely effective programming system.

This can consist of up to four parts which can be assembled to put Paltran "on the air". The system includes the Basic Instruction Generator (BIG), the Macro Instruction Generator (MIG), the Paltran Operating System-EDitor (POSED), and the Parallel Task Analyzer (PTA).

The Basic Instruction Generator implements the statements of the Reference Language. The formidable problem of determinacy must be handled at this stage. A given hardware system may not be large enough to accommodate the size of data sets requested, nor may it be well suited to handle array arithmetic. Algorithms which ensure that a given parallel structure is determinate (executable) regardless of the number of processors, their speed, interconnection etc., must be normally included in the BIG. For some systems execution efficiency may be very low and Paltran should not be considered as a core language. The user must be aware of what the hardware limitations are and make suitable allowances to ensure efficiency, as in any language.

The BIG may be a sequential simulation consisting, for example, of a Fortran compiler, a subroutine library and an interpreter acting as a subroutine caller.

The Macro Instruction Generator is a mechanism to give the user access to the system hardware. High level macros can be incorporated in a program to implement parallel structures that are suitable for execution by the given hardware, but do not have a direct Paltran description. In this manner efficient hardware use can be achieved while retaining the convenience and power of the BIG. The MIG however, is optional and need not appear in all Paltran systems; it would not be useful for example, in sequential simulations.

The Paltran Operating System-Editor can be added to the BIG and MIG in order to permit on-line operation, most likely in a stand-alone mode. POSED consists of a text editor to generate and correct source programs, a load and go compiler-assembler and a dynamic error detection and handling routine.

Finally, if system resources are large enough (or Paltran programs small enough) programs may be subjected to a Parallel Task Analyzer (chapter two) and run in a doubly parallel mode. The PTA is optional and would be useful only in very large installations where high programming bandwidth is essential.

It is possible to write a Paltran simulator consisting only of the Basic Instruction Generator; this has been done and is discussed in the remainder of this chapter.

Paltran - 8

A Paltran system has been written for the Digital Equipment Corporation PDP - 8 series of minicomputers, and is called appropriately, Paltran - 8.

The configuration for which Paltran - 8 was written consists of only 4096, 12 - bit words of central memory, and a single teletype and high speed paper tape reader-punch for input/output equipment. The small amount of memory available necessitates that Paltran - 8 be a small subset of Paltran, however most of the major features are illustrated. The Statements in the subset are as follows.

The SET Statement

All arithmetic in Paltran - 8 is handled by the SET statement, including simple assignment. The standard arithmetic operators are available as well as eight basic functions (sine, cosine, etc.) and operations are done on arrays (vectors and matricies only) on an element by element basis. An extension to the SET statement permits matrix multiplication, inversion, and transposition to be performed. The determinant of a square matrix may be found, and the main diagonal extracted as well. There is also a provision for setting a matrix equal to the identity matrix.

The LIBRARY Statement

The Library statement simply groups together several vector variable operations and is reserved for future expansion. The five possible vector operations are:

- Plot a vector of co-ordinates
- Set a vector equal to a function evaluated over a given range
- Find the global minimum and maximum element in a vector
- Numerical Integration, which produces a vector of individual area calculations, as well as the accumulated sum
- Differentiation, which produces vectors containing the first forward differences, etc.

The IF Statement

The Paltran - 8 arithmetic IF statement has form and action identical to that described in chapters two and three.

The DO - CONTINUE Statements

The Paltran FOR - DO statements as used in looping, have been contracted to the DO - CONTINUE pair in Paltran - 8, and resemble the Fortran Do loop. There is no execute statement (in the form of the DO), nor is there a FOR statement.

Input/Output Statements

The READ and WRITE are the two I/O statements, and they work only with the teletype as a data transmission medium. They are similar to the READ/WRITE pair described in chapter three, however the notation has been altered somewhat to resemble other PDP - 8 software.

<u>Miscellaneous Statements</u>

These include the GOTO which causes unconditional program transfers and the HALT which terminates execution.

It is interesting to note the features that Paltran-8 does not have. Sum and Continued Product operations, character vector representation, and full or partial subscripting are not available. Likewise, the PARTITION, BUILD, and PROJECT statements have not been implemented. Considering the limited memory available, the Paltran - 8 system is quite remarkable for providing the operations it does.

The Paltran - 8 User's Manual in Appendix IV should be consulted for full operational details and several examples of parallel processing applications.

Chapter Five

THE MATRIX PROCESSOR

The Matrix Processor

One of the characteristics of high level programming languages is machine independence. In this regard, Paltran is not a language designed to be restricted to any one type of parallel processing scheme. It is useful, however, to relate Paltran commands to some particular hardware configuration in order to demonstrate the straightforward implementation of some high-level operations. A large class of Paltran computations take place on arrays of data and the MATRIX PROCESSOR is well-suited to handle these. This particular scheme takes its name both from the physical arrangement of processing elements comprising it, and the ease with which matrix operations may be implemented.

The heart of the system is a rectangular array of processing elements or cells (ref. Fig. 1). For purposes of discussion, the array will be considered square, containing n x n cells with the following properties:

- several words of local memory.
- the ability to ADD, SUBtract, MuLTiply and DIVide.

Other operations such as shifting, masking, logical operations on individual bits, etc. will not be considered, but would, in fact, be included in an actual

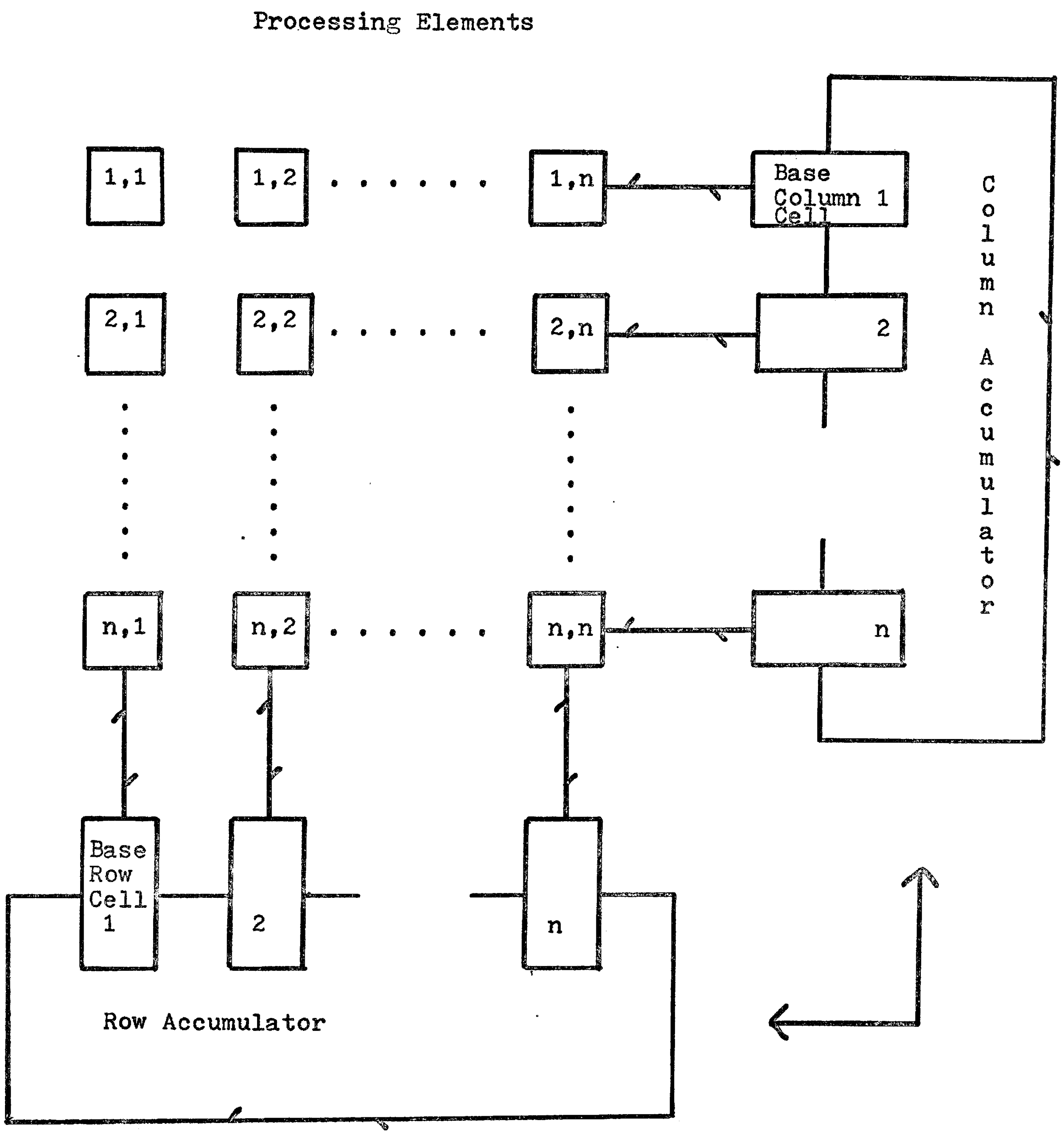


Fig. -1- Schematic Representation of the Matrix Processor hardware implementation.

In addition, a collection of cells which we will call a ROW ACCUMULATOR and a COLUMN ACCUMULATOR are added to the array; the term, 'accumulator', should not be taken too literally. This 'super-register' serves as a communications device to and from the array, as well as holding intermediate results. The accumulator (referring to either one or both) has properties which can be examined by looking at what might be a partial instruction set for this 'device'.

The first cell in either accumulator will be termed the base cell. It is assumed that data will be fetched (or stored) from central memory one word at a time. The instructions:

```
LOAD   X, R/C
UNLOAD X, R/C
```

will cause a transfer to or from central memory word X, to the base cell. R/C determines whether transfer is made to the ROW or COLUMN base cell.

```
SHIFT n, U/D/R/L
```

will shift the data n cells UP, DOWN, RIGHT or LEFT from the base cell. Shifts are assumed to be circular. U/D affects only the column accumulator; R/L affects only the row accumulator.

SWAP

will interchange ROW and COLUMN data.

The following instruction has both local and global action:

INSTRUCTION	
SET R=C	Set ROW=COL, COL unaffected
SET C=R	Set COL=ROW, ROW unaffected
SET R=Ø	Clear entire ROW
SET C=Ø	Clear entire COLUMN
SET R(i)=C(j)	Data interchange between
SET C(j)=R(i)	individual cells.
SET R(i=x,y)=C(j=W,s)	Set ROW cells x to y equal
SET C(j=w,s)=R(i=x,y)	to column cells w to s etc.

NOTE: An indexed, micro-programmed LOAD/UNLOAD-SHIFT instruction can be used to bring data into the entire ROW or COL accumulator, from central memory. In a more elaborate shceme, central memory can be re-organized in n-word blocks to make n-word transfers to the accumulators possible.

The actual physical location of these super-registers or accumulators poses an interesting question. Since the instructions mentioned apply to either one, the ROW and COLUMN accumulators can be one physical unit; it is only a matter of proper gating to obtain the results.

In fact, it will be convenient to let word (or location) Ø of each cell be called the accumulator. This effectively gives each cell capabilities for inter-cell communication as well as for central memory data transfer. Note, although the accumulator is accessible as word Ø of each cell, this does not imply there are n x n "accumulator cells".

Instructions that operate on a cell-to-cell basis can now be examined. The general format is:

OPR FIELD 1 FIELD 2

where

OPR = ADD/SUB/MLT/DIV

FIELD 1 = x,y,z

FIELD 2 = BLANK/(i,j)/(i,)/(,j)

OPR designates the arithmetic instruction to be performed. FIELD 1 defines the words in local memory on which OPR is done in 3-address format. FIELD 2 defines the scope of the operation; if it is blank, the operation is global; if two parameters are present, the operation is local to a single cell; if one parameter is present, the operation is performed on a row or column. Examples,

ADD 1,2,3	means add word #2 to word #1 and leave the result in word #3 of <u>all</u> cells.
SUB Ø,1,Ø(6,)	means subtract word #1 from the accumulator and leave the result in

the accumulator of all cells in row 6.

Note, operations involving the accumulator cannot be done globally, only on a row, column or single cell basis.

In addition, the transfer operation:

DEPOSIT SOURCE,DEST (i,j) is provided.

Example,

DEPOSIT Ø,3 (,2) - deposit the accumulator in word 3 of column #2 cells.

The accumulator and cell-to-cell instructions may be micro-programmed and indexed to form a set of high-level macros.

The LOAD, SHIFT, and DEPOSIT instructions can be combined to generate the macro:

LOAD W,(N,M),X

which can be interpreted as: Place in word W of each cell the N x M array, as read from central memory (by rows or columns) starting at word X (inCM).

Other useful macros are:

MACRO	ACTION
TRANSPOSE W	TRANSPOSE rows and columns, word W. $W \neq 0$
OPR (x,y,z)	Example: ADD word y to word x and leave result in word z of all cells. $x \neq y \neq z \neq 0$

OPR (x,y,z); $\overset{\text{ROW}}{\text{COL}}$ (i,j,k)	Example: ADD word y of ROW j to word x of ROW i and store in word z of ROW k
OPR w; $\overset{\text{ROW}}{\text{COL}}$ i.	Example: ADD together all words w in ROW i and leave result in word 0 of the base ROW cell
SHIFT w; $\overset{\text{ROW}}{\text{COL}}$ i,k	Shift word w from ROW/COL i to k

Some of the higher Paltran commands can now be related to these macros. The Paltran statement SET a=TRAN(b) is simply a combination of :

LOAD 1 (n,m),b	(bring the matrix b(n,m) into location 1)
TRANSPOSE 1	(transpose rows and columns)
UNLOAD 1 (m,n),a	(put the transpose into central memory as a(m,n))

Addition, subtraction, etc. of matricies is similar; the macros

```
LOAD  1 (n,m), a
LOAD  2 (n,m), b
ADD   1,2,3
UNLOAD  3 (n,m),c
```

simply mean *a(n,m),b(n,m),c(n,m)

c=a+b

Matrix multiplication involves bringing the two matricies into the array and transposing one of them. Row by column multiplication can now occur en masse and results added to form a column (or row) of the resultant matrix. One of the matricies is circularly rotated and the row by column multiplication and addition occurs again to form the second column of the product and so on.

Other matrix and vector operations can be handled in a similar manner. If some form of associative memory techniques are given to the matrix processor such as the ability to test the sign of a word in all cells simultaneously, the arithmetic IF statement becomes very easy to implement. The logical IF is an extension of these techniques.

The ROTATE statement is simply the SHIFT macro, while the PARTITION and BUILD as well as the PROJECT statements are combinations of LOAD-SHIFT-UNLOAD macros.

The matrix processor can also run in the default 'sequential' mode making use of only the base accumulator cell and one processor cell. Depending on the capabilities of the accumulators a Foreground-Background mode is possible using the row and column base cells plus two processors or it may be possible to run n processors simultaneously and independently.

The matrix processor is a complex hardware scheme but is certainly within the realm of possibility with LSI technology. In overall scope, some 4-th generation mult- and 'super' computers are every bit as complex if not more so. The advent of the matrix processor combined with a language such as Paltran will result in a very powerfull analytic tool.

Chapter Six

CONCLUSIONS

The conceptual basis for a new computer programming language for the parallel processing enviroment has been developed in this thesis. Various aspects of existing and proposed hardware schemes and forms of parallel processing have been investigated to determine the objectives for the language. Available hardware, in the form of array and associative processors and mathematical operations displaying inherent, array-oriented, parallelism, together with previous language experience, has suggested that the language be of the type suitable for mumeric, scientific calculations with an array-based data structure.

Several mathematical operations have been discussed in light of these requirements and statements for the language have been developed to accomodate them. The syntax and semantics of the language have been chosen to express high-order operations concisely and to maximize user convenience. An extensive character set has been proposed to incorporate common, well-used symbols and special characters in order to simplify the notation. The absence of special purpose mathematical operations also tend to keep the notation uncomplicated.

Various software schemes have been proposed to implement the language as it might appear on a large, fully parallel system. A small subset of the language has been written as a sequential simulation and very effectively demonstrates the major technical characteristics.

Finally, a possible hardware configuration in the form of an array processor is examined in the light of the language requirements.

A core language to communicate with parallel computers may be constructed from the concepts developed and form the basis of a powerful analytical tool.

APPENDIX I

The Paltran Character Set

and Key Words

The following is the proposed Paltran Character Set, which includes:

- The 8-bit ASCII (ANSCII, etc.) character set.

  (Upper and lower case letters, numerals, punctuation characters, and other "characters" such as carriage return, line feed, rubout, etc.)

- The special characters

  $\Sigma \int \pi \blacktriangle \Gamma \wedge \vee \ulcorner \leqq \geqq < >$ ǂ

- The upper and lower case Greek alphabet if possible.

The availability of special characters on I/O equipment makes possible the equivalence of the Reference, Publication, and Hardware descriptions of the Language. The following table lists the key words in Paltran:

KEY WORD	ABBREVIATION IF APPLICABLE	SYMBOL IF APPLICABLE
BUILD		
COLUMNS	COL	
COMPLEX		
DETERMINANT*	DET	
DIAGONAL*	DIAG	
DIFFERENTIATE*	DEL	▲
DO*		
DOUBLE		

EIGENVALUE		EVAL	
EIGENVECTOR		EVTR	
ELSE			
END			
FLAG			
FOR			
FROM			
GO *			
GOTO			
HALT *			
I *			
IF *			
INTEGER			
INTEGRATE	*	INT	$\int$
INTO			
INVERSE	*	INVR	
LOAD	*		
LOGICAL			
MAXIMUM	*	MAX	
MINIMUM	*	MIN	
OCTAL			
PARTITION		PART	
PRODUCT		RRD	Π
PROJECT		PRO	
READ	*		

REAL		
REVERSE	REV	$\leftarrow$
ROTATE	ROT	
ROWS		
SET *		
SORTDOWN	SRTDWN	$\vee$
SORTUP	SRTUP	$\wedge$
SUM		Σ
TRACE		
TRANSPOSE *		
TRIPLE		
WRITE *		

The standard functions:

SIN *	(sine)
COS *	(cosine)
ATN *	(arctangent)
EXP *	(exponential)
LGE *	(natural log)
LOG	(common log)
SQT *	(square root)
ABS *	(absolute value)
FLT	(float an integer)
ITR *	(fix a floating point quantity)

Functions such as tangent, hyperbolic cosine, arc hyperbolic cosecant, etc., can all be constructed from the above.

The following functions are all based on the definitions given in chapter two. These functions are also considered to be key words.

C(n,r)	(number of combinations)
P(n,r)	(number of permutations)
B(n)	(Bernoulli number)
Γ(z)	(Gamma function)
ERF(z)	(Error function)
T(n,x)	(Value of Chebyshev polynomial n, for argument x)
H(n,x)	(Hermite polynomial)
L(n,x)	(Laguerre polynomial)
G(n,x)	(Legendre polynomial)
J(n,x)	(Bessel function of the first kind)

In addition, functions that supply random numbers, time of day, etc. are usually provided, as defined by system parameters.

* Statements or functions marked with an asterisk are part of the Paltran - 8 subset and have been implemented.

APPENDIX II

The SET Statement

There are five catagories of operations which are defined in the SET statement. These are: Integration, Differentiation, Sum and Continued Product Operations, Matrix Operations and Miscellaneous Vector Variable Operations.

Integration

Statement Forms:

SET s= $\int$(f(x)),e

SET s= $\int$(f(x)),(*,b),e

SET s= $\int$(f(x)),(a,*),e

SET s= $\int$(f(x)),(a,b) *

SET v= $\int P_x$

SET s,v= $\int$(f(x)),(a,b,n)

where -s and x are scalars

-v is a vector of length n

-P_x is a vector of polynomial coefficients

-f(x) is some function that is to be integrated over the range (a,b); (a,*) denotes integration from a to plus infinity; (*,b) denotes integration from mimus infinity to b. Absence of explicit ranges denotes infinite integration from plus to mimus infinity. The scalar s is set equal to the value of the integral. The second last form of the statements defines polynomial integration as described in chapter two. If the individual area calculations are desired (for example, to plot integral curves) the last form may be used. Integration takes place over (a,b) with n intervals and the vector v is set equal to the individual areas. A third parameter e, may be specified for infinite integrals which are evaluated as described in chapter two. When two successive evaluations of the split range differ by less than e, the process is terminated. If e is absent, it is assumed to be 10^{-6}.

Differentiation

Paltran differentiation consists of forming vectors that contain the first, second, etc., divided differences of a set of values. If the interval spacing is constant the forward, backward or central differences are determined. Given the following:

```
*x(n),y(n),d1(n-1),d2(n-2), m(n-1,n-1)
SET d1 = ∆ x,y
```

generates a vector d1 containing the first divided differences of vector y with spacings given by vector x, where y=f(x).

```
SET d2 =∆x,d1
```

generates the second divided difference and so on.

```
SET m =∆x,y
```

generates the lower triangular matrix m containing all (n-1) divided differences.

If the interval spacing is constant the forward, backward and central differences are formed:

SET d1 = $\blacktriangle$ y *

SET d2 = $\blacktriangle$d1

•

•

SET m = $\blacktriangle$ y

SET y = $\blacktriangle$(x) is used for differentiating polynomials. The variable x represents the coefficient vector.

Sum and Product Operations

Given, s a scalar, v(n) a vector, m(n,m) a matrix, h(n,n,n) a rectangular prism, the following operations are possible:

SET s = $\sum$ v

results in the n elements of v being summed and set equal to s

SET s = $\sum$m(*,j)   SET s = $\sum$m(i,*)

results in either column j or row i of m being summed

SET s = $\sum$ h(i,j,*)

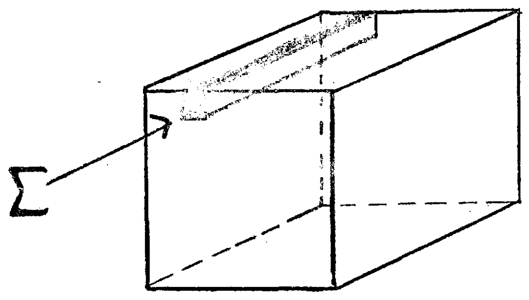
results in the summing of the elements in the remaining direction as located by the intersection of i and j.

The following graphical example is illustrative:

*h(5,5,5)

SET s= $\sum$h(1,2,*)

The cube represents the array h. The shaded portion represents the elements summed.



Sum reduction in higher-order arrays is treated in a similar manner. The sum operation is global if matricies and higher order arrays are named without subscripts. Thus,
SET $s = \sum a$ means $s = \sum a_{i,j}$ $i=1,2,\ldots n$ $j=1,2,\ldots.m$

A second form of summing operation can be performed on functions, without first producing a vector of values. The form of the statement is:
SET $s = \sum (f(x)), (a,b,c)$
The function f(x) is evaluated for x=a, incremented by b until the value c is reached or exceeded and the partial sums accumulated. The operation can be nested to any depth for functions of more than one variable: let q=f(x,y,z) we may write,
SET $s=\sum\sum\sum (f(x,y,z)),x(a_x,b_x,c_x),y(a_y,b_y,c_y),z(a_z,b_z,c_z)$

The continued product operation is carried out in exactly the same fashion as for summation except $\sum$ is replaced by $\prod$.

Matrix Operations

Matrix operations are also included under the SET statement. The following variaiables are used:

*a(n,m),b(n,r),c(r,m),d(m,n)	-four general rectangular matricies
*g(n,n),h(n,n)	-two square matricies
*u(m,1),v(n,1)	-column vectors
*w(1,m)	-a row vector
*z(n)	-a vector variable
s	-a scalar

It should be noted that column and row vectors are matricies and are distinct from vector variables. The following table lists the possible operations.

MATRIX OPERATIONS

IN PALTRAN

Operation		STATEMENT
Matrix multiplication	*	Set a=b*c
Matrix transposition	*	Set a=TRAN(d)
Matrix inversion	*	SET g=INVR(h)
Trace of a square matrix		SET s=TRACE(g)
Extraction of the main diagonal of a square matrix	*	SET z=DIAG(g)
Determinant of a square matrix	*	SET s=DET(g)
Eigenvalues		SET v=EVAL(g)
Eigenvectors		SET h=EVTR(g)
Set a square matrix equal to the identity matrix	*	SET g=I

The matrix norms:

Largest row sum of absolute values	SET s=ROW(a)
Largest column sum of absolute values	SET s=COL(a)
Premultiplication of a vector by a matrix	SET v=a*u
Premultiplication of a matrix by a vector	SET u=v*a

Vector multiplied by another vector (scalar or inner product)	SET s=w*u
	SET s=u*w

SET vector variable equal to column vector or vice versa	SET z=v
	SET v=z

Length of a row or column vector, ie. $(\sum v_i{}^2)^{\frac{1}{2}}$	SET s=(v)
	SET s=(w)

The didactic operators (↑,*,/,+,-) and functions can be used to preset elements in a matrix or higher-order array. The form is:

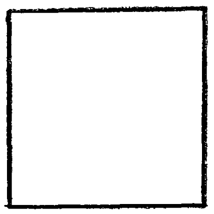
SET a(i,j,k,....,*,....) = e

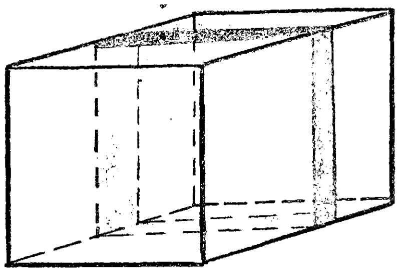
- e is any expression whose dimensions are compatible with those of a in the unspecified direction(s).

The following graphical examples are illustrative.

```
*a(5,5,5),b(5,5), theta(5,5),abc(5,5)

SET a(*,*,3) = b*2 + SIN(theta)
```

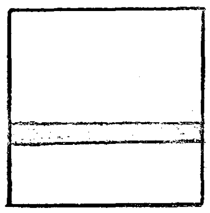


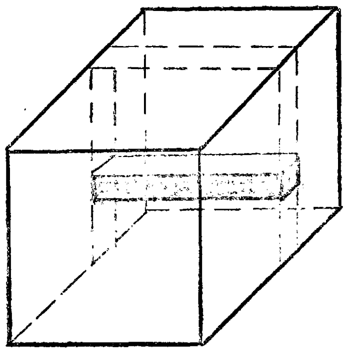


The square represents
the matrix corresponding
to b*2 + SIN(theta)

Plane 3 in the cube a
has elements set equal
to b*2 + SIN(theta)

SET a(3,*,3) = LOG(abc(3,*))





The square figure re-
presents the matrix
abc and the shaded
portion, the third row

The LOG of the elements
in row 3 of abc is taken
and transferred to the
third row in plane three
of the cube a

<u>Miscellaneous Vector Variable Operations</u>

STATEMENT	DESCRIPTION	EXAMPLE

SET v=$\leftarrow$w or
SET v=REV(w)
	Reverse order a vector	w=1 2 7 10 $\therefore$ v=10 7 2 1

SET v=$\wedge$w or
SET v=SRTUP(w)
	Sort elements of w in ascending order	w=62 31 10 99 $\therefore$ v=10 31 62 99

SET v=$\vee$w or
SET v=SRTDWN(w)
	Sort in descending order	w=62 31 10 99 $\therefore$ v=99 62 31 10

The following miscellaneous operations can be performed on any array.

SET s=MAX(a) or SET v=MAX(a)

SET s=MIN(a) or SET v=MIN(a)

The scalar <u>s</u> is set equal to the global maximum or minimum element of array <u>a</u>. If the replacement variable is a vector local minima or maxima will also be located. Should more extrema be found than elements in the vector, a diagnostic element in FLAG is set.

APPENDIX III

References & Bibliography

1-- Backus, J.W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference". Proc. Internat'l Conf. Information Processing, UNESCO, Paris,1959.

2-- USA Standard FORTRAN, United States of America Standards Institute,USAS X3.9-1966, New York, Mar., 1966.

3-- Bernstein, J. B., "Analysis of Programs for Parallel Processing", IEEE Transactions on Electronic Computers, EC-15 No. 5,(OCT. 1966)

Baer, J. L., "Compilation of Arithmetic Expressions for Parallel Computation", Proc. IFIPS, 1968,B4-B10.

Stone, H. S., "One=Pass Compilation of Arithmetic Expressions for a Parallel Processor",Communications of the ACM, 10:No. 4 (APR. 1967)

Ramamoorth and Gonzalez, "Recognition and Representation of Parallel Processable Streams in Computer Programs-II (Task/Process Parallelism)", Proc. ACM National Conference, 1969.

4-- Bingham and Reigel, "Parallelism Exposure and Exploitation in Digital Computing Systems", ECOM, 02463-F, June 1969.

5-- Murtha and Beadles, "Survey of the Highly Parallel Information Processing Systems", Westinghouse ONR Report No. 4755, Nov. 1964.

6-- Blakeney, G. R. et al., "IBM 9020 Multiprocessing System", IBM Systems Journal, 6, No. 2 (1967).

7-- Gall, R. G., "Hybrid Associative Computer Study", RADC-TR-65-445, Vol. 1, Final Report, July 1966.

8-- Illiac IV System Study Final Report, Burroughs Corporation, University of Illinois. Dec. 1966.

9-- Campeau, J. O., "The Block Oriented Computer", IEEE Computer Group Conference Digest, June 25-27, 1968.

10-- Thornton, J. E., "Parallel Operation in the Control Data 6600", Proc. AFIPS, FJCC, 1964, Part II.

11-- McGinn, L. C., "A Matrix Compiler for UNIVAC", Automatic Coding, Jour. Franklin Inst., Mono. No. 3,Apr. '57.

12-- Conway, M. E., "A Multi-Processor System Design", Proc. AFIPS, EJCC,1963.

Additional References

-Holland, J. H., "A universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously", PROC. AFIPS, EJCC, 1959. p 108.

-Gonzalez, R., "A Multi-Layer Iterative Circuit Computer", IEEE Transactions on Computers, Dec. 1963.

-Squire and Palais, "Programming and Design Considerations of a Highly Parallel Computer", PROC. AFIPS, SJCC, 1963.

-Katz, J. H., "Simulation of a Multi-Processor Computer System",PROC. AFIPS, SJCC, 1966.

-Karp and Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing", JSIAM Nov. 1966.

-Kuck, D. J., "ILLIAC IV Software and Applications Programming", IEEE Transactions on Electronic Computers, Aug. 68.

-Gosden, J. A., "Explicit Parallel Processing Description and Control in Programs for Multi-and Uni-Processor Computers",AFIPS, FJCC, 1966.

-Bernstein, J. A., "Analysis of Programs for Parallel Processing", IEEE Transactions on Computers, OCT. 66.

Bibliography

Ware, W. H., "The Ultimate Computer"
IEEE Spectrum, March 1972, p. 84.

Schwartz, J., "Large Parallel Computers"
Journal ACM, January 1966, p. 25

Sammet, J. E., "Programming Languages"
Prentice-Hall, 1969

Carnahan, Luther, et al., "Applied Numerical Methods"
John Wiley & Sons, 1969

APPENDIX IV

The Paltran-8 Users Handbook

INTRODUCTION

Paltran-8 is a small subset of the Paltran programming language designed for the PDP-8 family of computers. The Paltran-8 system consists of a compiler (PALX) and four operating systems (POS1, POS2, POS3, POS4). The PALX compiler operates on source input and generates an object tape (consisting of subroutine calls and data) which is subsequently passed on to the Paltran Operating System (POS). The operating system is a collection of subroutines which, when called by the object program, will execute the appropriate source code. In this sense the Paltran compiler and operating system form an interpretive system similar to PDP-8 FORTRAN.

Since Paltran is a very high order language, it is not possible to implement all of the main features in a basic 4K PDP-8. In order to implement as many statements as possible, four operating systems are available to deal with specific classes of instructions.

SCOPE OF PALTRAN-8

PALTRAN is designed to communicate with and take advantage of a multi-processer machine capable of executing many operations at the same time.

The concept of an array of processing elements capable of performing arithmetic (simultaneously) on large data structures is central to this idea. Since the PDP-8 is a single processer machine, PALTRAN-8 is necessarily a simulation; the only effect apparent to the user would be a decrease in execution time if the PDP-8 was a parallel (i.e. multi-processer) machine.*

MINIMUM SYSTEMS REQUIREMENTS

The Paltran compiler and Paltran operating systems each run in 4K of core, on a PDP-8 (I, L, etc.)

---

* While Paltran-8 can be used to solve problems and serve as a useful computational tool, it is intended only as a demonstration of Parallel processing capabilities, and, therefore, not "user-proof", as say FOCAL.

with teletype (keyboard and reader-punch). If a high speed reader-punch is available, it may be used. The current version of Paltran-8 does not take advantage of any additional memory.

PALTRAN ARITHMETIC OPERATIONS

Since it is assumed a parallel-processing machine is available, Paltran will do certain mathematical operations in parallel. These operations are performed on variables (and constants) which may be of three types, namely, zero, one, or two dimensional. They are the simple variable, vector variable and matrix variable, respectively.

An arithmetic expression may consist of variables, which must be of the same type, constants, operators and functions.

The Paltran operators and functions are:

Operator	Description	Precedence
(	Open parenthesis	0
)	Close parenthesis	0
+	Addition	1
-	Subtraction	1
*	Multiplication	2
/	Division	2
↑	Exponentiation	3
Function		
FSIN	Sine in radians	4
FCOS	Cosine in radians	4
FATN	Arctangent in radians	4
FEXP	Exponential	4
FLGE	Natural logarithm	4
FABS	Absolute value of	4
FINT	Integer Part of	4
FSQT	Square Root	4

Note: Zero precedence is the lowest and four is the highest.

The (+ - * / ↑) operators are all of the double operand type, that is, they always combine two numbers. The functions are of the single operand type since they operate on only one number or, an expression which can be reduced to a single number. Simple variables, vectors or matricies can be combined by these operators in the following manner:

<u>Simple Variables</u>

Any two simple variables combined by (+ - * /↑) yield another simple variable as a result. A function of a simple variable is another simple variable. Example,

```
C = A + B

BETA = FSIN (X)
```

BETA and C are single pieces of data if A, B, and X are simple variables.

<u>Vector Variables</u>

A vector is a linear collection of numbers. The length or size of a vector variable is defined in Paltran as the number of elements (or numbers) in that vector; thus, V(10) refers to a variable consisting of 10 elements and is uni-dimensional.

Any two vectors combined by (+ - * /↑) yield another vector as a result. These operations are done on an element by element basis, thus:

C = A + B

means $\underline{C}(i) = \underline{A}(i) + \underline{B}(i)$ $i = 1, 2, \ldots n$

where C, A, and B are all of length n.

A constant or scalar may appear in a vector expression, thus:

C = 2 results in every element in $\underline{C}$ being set equal to the number 2.

C = A * 4 results in a vector $\underline{C}$, whose elements consist of the elements of another vector $\underline{A}$, multiplied by 4.

Note: Constants or scalars when used in vector (or matrix) expressions must be numeric only. The following is illegal:

```
    Given  K   -  a simple variable
           B,C  -  vector variables
           .
           .
           .
           K = 100
           .
           .
           .
           C = B + K
```

All variables must be of the same type in any one expression.

A constant may be added (subtracted, etc.) to a vector (or matrix) in two ways only. Example:

```
           * V(10), X(10), K(10)
```

i.e. V, X and K are defined as vectors of length 10.

```
           .
           .
           .
      SET K=1000
           .
           .
           .
      SET X=V+K
           .
           .
           .
```

In this case, K becomes a constant vector whose elements are all equal to 1000; alternatively, the expression can be written directly as

```
           .
           .
           .
      SET X=V+1000
           .
           .
           .
```

A function when applied to a vector yields another vector as a result; the operation is done on an element by element basis, thus:

If A and B are vectors

```
      SET A=FSIN(B)  results in a vector A whose
                     elements are the sine of the
                     elements of vector B, or

         A(i)=SIN(B(i))        i = 1, 2, ... n
```

Matrix Variables

A matrix is a two-dimensional array of numbers. The size of a matrix is given by a pair of numbers which refer to the number of rows and columns in that array. Hence, A(5,10) refers to a matrix of 5 rows and 10 columns, containing (5 X 10) 50 elements.

A matrix may be regarded as a two-dimensional vector and, conversely, a vector is a matrix with only one row or one column.

Matrix variables are combined in exactly the same way as vector variables, that is, on an element by element basis.

If A, B, and C are matricies

```
      SET C=A+B  means  C(i,j) = A(i,j) + B(i,j)

                             i = 1, 2 ... n
                             j = 1, 2 ... m

      SET C=A*B  means  C(i,j) = A(i,j) x B(i,j)

                             i = 1, 2 ... n
                             j = 1, 2 ... m
```

Note, this is not matrix multiplication; similarly, C=A/B is not a form of matrix inversion; special instructions apply for these operations.

FURTHER NOTES ON ARITHMETIC

Arithmetic evaluation takes place left-right, with operations of highest precedence or priority being performed first. Hence, functions are evaluated first, followed by exponentiation, followed by multiplication and division, followed by addition and subtraction. Parenthesis may be used to change the order of evaluation as in the example below. Note, that multiplication and division have the same priority, hence:

```
If A=4
   B=2
   C=3

     A/B*C is 4/2 x 3 = 2 x 3 = 6
```

not

```
     A/(B*C)  which is  4/(2 x 3) = 4/6 = 2/3.
```

All Paltran arithmetic is done in 3-word floating point format, and, in fact, makes direct use of Floating Point Package number four.* The function FINT (Integer Part of) therefore returns a floating point number. Example:

```
SET X=3.3333
SET Y=FINT(X)
```

Hence   Y=3.0000

In actual operation, the operand X is fixed to a single precision integer (range -2047 x 2047) and then re-floated and normalized. Attempting to use FINT outside the range $\pm$ 2047 would produce erroneous results.

Special care must be taken when using the exponentiation operation,↑. In order to raise a number to a power, Paltran uses the relation:

$$A^B = EXP(B \cdot Ln(A)) \quad \text{or}$$

```
FEXP(B * FLGE (A))
```

Hence, there are no restrictions on the range of the exponent, B, but A must be positive and non-zero.

* DEC-08-YQ4A-PB

INTRODUCTION TO PALTRAN STATEMENTS

A Paltran statement may be numbered and can be up to 1 teletype-line (72 characters) long. A line can be terminated by a carriage return (CR) only. The statement numbers, if present, must lie in the range (1-2047). The line format is:

```
STATEMENT BLANK COMMAND BLANK REST OF LINE CARRIAGE
  NUMBER                                    RETURN
```

That is, the statement number must be terminated (or delimited) by the character BLANK or SPACE; the command must also be terminated by a BLANK or SPACE. These are the only two instances where the blank is important; it is ignored at all other times.

The Paltran commands are:

COMMAND	ABBREVIATION
READ	R
WRITE	W
SET	S
IF	I
GOTO	G
DO	D
LIBRARY	L
CONTINUE	C
HALT	H

Since the first character of each command is unique, Paltran commands may be abbreviated; in fact, the compiler checks the first character of a command and ignores the rest of the line until the terminator (BLANK or SPACE) is found. Thus,

```
READTHIS
READ
REA
RE
R
```

are all equivalent.

PALTRAN CHARACTER SET

The Paltran character consists of the standard 6-BIT (2 octal characters) ASCII code which includes:

The upper case letters ABCD......WXYZ
The 10 digits 0-1-2......9
The characters

!	Exclamation	
"	Quotes	
#	Number sign	
$	Dollar sign	
%	Per cent sign	
(	Open Parenthesis	
)	Close Parenthesis	
*	Asterisk	
,	Comma	
.	Period	
+	Plus sign	
-	Minus sign	
/	Slash	
=	Equal sign	
↑	Up arrow	
[	Open brackets	Square
]	Close brackets	Square

And the following non-printing characters

RUBOUT
CARRIAGE RETURN
SPACE or BLANK

In operation, the rubout key when struck will echo a back arrow (←); the return key will generate a carriage-return and line-feed (CRLF) and the space, of course, prints the character BLANK.

The following have no significance in PALTRAN-8, but may be used as delimiters:

<	Angle	}
>	Brackets	}
←	Back arrow	
:	Colon	
;	Semi-colon	
&	Ampersand	

Line feeds are ignored as well as blank tape and leader-trailer (Ø2ØØ-code) tape.

The following characters must not be used in any Paltran source statements,

?	Question mark
@	At sign

The 6-BIT ASCII for ? is 77, which the Paltran compiler senses as an end-of-tape symbol. A ? in a source statement would stop the compiler with no recovery possible. Similarly, the @ has code ØØ which is sensed as a line delimiter (the carriage return). The ØØ code would produce the action of a CRLF, but has no delimiting value.

A Paltran delimiter is any character other than a letter or a number. In order to make the carriage return a delimiter, the compiler generates two symbols for the single character CR. The first character is a Paltran delimiter (which is actually code for the & (ampersand)) and the second is ØØ which serves as a line terminator.

There is one special delimiter which serves as the end-of-tape, or actual physical end of the program. This is the $ (dollar sign). The compiler checks every input character to see if it is the $; when the $ occurs, the compiler halts.

PALTRAN STATEMENTS

The Dimension Statement

Unlike FORTRAN or FOCAL, all Paltran variables must be declared, even simple variables. This is done via the dimension statement which has the form:

```
*V1,V2,....Vn,....
```

Where Vn = VAR or VAR(I) or VAR(I,J)

VAR is a variable name up to four characters in length. The characters may be alphabetic or numeric, however, the first must be alphabetic and not the letter F.

VAR(I) defines a vector variable of length I and VAR(I,J) defines a matrix variable with I rows and J columns.

Commas or brackets serve as delimiters for the variable names.

The dimension statement is the only one different in structure from the other Paltran commands. It is not given a statement number and may occur anywhere in the source program any number of times. After loading the source program, the compiler looks for statements beginning with * and ignores all others. This is the first pass, during which the symbol table is generated. The symbol table can hold 30 entries, hence no more than 30 variable names can be defined in any one program. On the second pass, the compiler ignores all statements beginning with * and processes the rest. Since the source program is stored in core, the second pass does not involve any physical action of reloading tapes, etc.

Example:

```
PALTRAN-8 GO                  (This is an introductory
    .                          message typed by PALX)
    .
    .
*DOG,CAT,A123(10),BIRD
    .
    .
    .
*MAT(5,10),PI
    .
    .
    .
*ABCD
    .
    .
    .
$
```

The READ Statement

The READ statement is the basic mechanism for data input. It has the format:

```
n READ VAR,!,"..."
```

n is an optional statement number
VAR is a variable name
! generates a CRLF
"..." quotes may be used to insert text in the input format. Any ASCII character can be used between quotes except: $, ?, @ and ".

Note, vector or matrix variables need only be mentioned by name. The operating system will automatically read 1 or 2 dimensional variables according to a pre-arranged, indexed format.

Example:

```
*SIMP,VECT(10),MAT(2,2)
    .
    .
    .
READ "INPUT",!,SIMP,!,VECT,MAT
    .
    .
    .
100 R "MORE INPUT",!,ABCD,A,BC,BCD
    .
    .
    .
*ABCD,A,BC,BCD
```

The following would result from the first READ statement at execution time.

Let 3.14159 be a value we wish to assign to SIMP

```
Let      100
         200
         300
         400              be a vector we wish to be
         500              read as VECT
         600
         700
         800
         900
        1000

And       11    12        be a matrix we wish to
          21    22        read as MAT
```

Hence

```
INPUT                      (POS types INPUT)
:3.14159                   (POS types : indicating it is ready
VECTOR INPUT                to receive input)
(1)  :100                  (user enters 3.14159)
(2)  :200                  (POS then types VECTOR INPUT
 .                          and an index in parenthesis followed
 .                          by a colon. The user then enters
(10)  :1000                 the appropriate element of the
MATRIX INPUT                vector.)
ENTER ROW(1)               (The operating system then
(1)  :11 (2)  :12           types MATRIX INPUT as a
ENTER ROW(2)                guide indicating 2 dimensional
(1)  :21 (2)  :22           input. All matrix variables
                            are read by rows only.)
```

Input data is usually terminated by a carriage-return (with line-feed generated automatically) or a space. Proper use of these delimiters is also essential to input formating.

To change the input format slightly, a switch register option is available to the user. The index (in parenthesis) that is typed as a guide for entering vector or matrix variables may be omitted by raising bit Ø.

i.e.

BIT Ø = 1 (switch up) - index is not typed.

BIT Ø = Ø (switch down) - index is typed as above.

Note, the colon is always typed when input data is expected.

As an example, the following would occur if switch Ø was up:

```
INPUT
:3.14159
VECTOR INPUT
:100
:200
  .
  .
  .
:1000
MATRIX INPUT
ENTER ROW
:11 :12
ENTER ROW
:21 :22
```

The WRITE Statement

The WRITE statement is the basic mechanism for data output and has the following format:

```
     n WRITE VAR,!,#,%X/Y,"....."
```

n is an optional statement number
VAR is a variable name
! generates a carriage-return, line-feed
# generates only a carriage return
"....." text may be enclosed between quotes as in the READ statement
%X/Y affects the output format
X is an integer defining the total field width, i.e., number of digits in a number.
Y is an integer that sets the number of decimal places in the output number.
Example:

```
*ABC
 .
 .
 .
SET ABC=10.5
 .
 .
 .
WRITE %8/4,ABC,!
 .
 .
 .
```

Produces at execution time

```
            10.5000
```

The + sign (if the number is positive) is suppressed, as well as are all leading zeros. The same output format remains in effect until the occurrence of another %, which changes it.

The output format can be set to E-format by using %0/0, or, simply, %/,. Example:

```
*ABCD
  .
  .
  .
SET ABCD=5000000
  .
  .
  .
WRITE %0/0,"FIVE MILLION",!,ABCD,!
  .
  .
  .
```

Produces

```
            FIVE MILLION

            0.50000000E07
```

Vector or matrix variables are automatically typed out by the operating system. A vector variable is always typed as a column vector, along with an index. A matrix variable is typed out as a rectangular array. Example:

Assume that the following have been calculated previously:

```
V =     1                    11   12   13   14
        2          MAT =     21   22   23   24
        3                    31   32   33   34
        4                    41   42   43   44
```

And

```
*V(4),MAT(4,4)
   .
   .
   .
WRITE %5/2,V,!!!!MAT,!
   .
   .
   .
```

Produces

```
        (1)  :  1.00
        (2)  :  2.00
        (3)  :  3.00
        (4)  :  4.00
```

```
                1        2        3        4

    (1)  :   11.00    12.00    13.00    14.00
    (2)  :   21.00    22.00    23.00    24.00
    (3)  :   31.00    32.00    33.00    34.00
    (4)  :   41.00    42.00    43.00    44.00
```

The operating system prints the row and column index for matrix variables, taking into account the output format (i.e. %X/Y) that has been set. The number of digits in a number limits the number of columns that can be typed out in a 72 character line. Example:

- 6 spaces are required for the row index.
- 2 spaces are inserted between columns.
- If the total field width is 6 digits, then,

$$\frac{72-6}{6+2} = \frac{66}{8} = 8$$

columns can be typed out. With 4K PALTRAN, this is not a serious problem since core available for data storage limits the size of matrix variables.

The index accompanying vector output and the row index accompanying matrix output can be suppressed by raising BIT Ø.

The SET Statement

The SET or arithmetic replacement statement has the form:

n SET x=e

Where n is an optional statement number
x is a variable that is set equal to the arithmetic expression e, as defined previously.

Note, both x and e must be of the same type, that is, all zero (simple variable), one (vector) or two (matrix) dimensional. Only one level of forward replacement is allowed.

Examples:

```
*A,B,C,D,V1(10),V2(10),V3(10),MAT(5,5),ARRY(5,5)
 .
 .
 .
SET A=B+2.0*FSIN(C+FCOS(D))
 .
 .
 .
SET V1=V2↑3-FABS(V3)
 .
 .
 .
SET MAT=MAT+ARRY
```

<u>The IF Statement</u>

The IF statement can be used to change program flow depending on the sign of a calculated expression. It has four forms:

```
n IF [e] A,B,C,D

n IF [e] A,B,C

n IF [e] A,B

n IF [e] A
```

n is an optional statement number
e is a valid arithmetic expression as defined previously

If e is a vector or matrix expression, the first form of the IF statement <u>must</u> be used; the following action takes place:

e is evaluated, and a single vector or matrix result, is tested. If all of the elements of this result are negative, control transfers to the statement numbered A; if all elements are zero, control transfers to B; if all elements are positive, control transfers to C. If any one (or more) element does not fall into one of these categories, control passes to D. For example, if there is at least one negative or zero number in an array of positive numbers, control would pass to the default statement, D. If e is an expression comprised only of simple variables, control can never transfer to D. When only simple variables are involved, the abbreviated forms of the IF statement can be used.

If the second form is used, control transfers to A, B, or C, if e is negative, zero, or positive, respectively.

If the third form is used, control transfers to A if e is negative, to B if e is zero, and to the next executable statement following the IF, if e is positive.

Control transfers to A if e is negative, and to the next executable statement if e is zero or positive, when the fourth form is used.

Note: The expression e must be enclosed in square brackets.

The GOTO Statement

The GOTO statement transfers control directly and has the form:

```
    n GOTO x
```

Where	n	is an optional statement number
	x	is the statement number to which control is transferred.

USER'S NOTE:

GOTO and IF statements transfer control indirectly through links on Page zero. All Paltran Operating Systems have a maximum of 26 locations available for this purpose. Each GOTO statement uses one link, and an IF statement can require up to four links, if the first form is used. Since there is a limit on the number of links available, it is advisable to use the abbreviated forms of the IF statement whenever possible.

The DO Statement

The DO statement is used for generating repeatable segments of code or loops. It has the form:

```
     n DO m VAR=X,Y,Z
          .
          .
          .
          .
          .
     m CONTINUE
```

```
Where  n  is an optional statement number
       m  is the number of a CONTINUE statement which
          defines the range of the loop; m must be
          terminated by a blank. Every DO statement
          must be terminated by a CONTINUE statement;
          CONTINUE cannot be used as a dummy state-
          ment as in FORTRAN.
     VAR  is the DO index variable which is first
          set equal to X.
          The statements in the body of the loop
          are then executed, at least once. VAR
          is then incremented by Y and tested to
          see if this result is equal to or greater
          than Z.
          If VAR is less than Z, the body of the
          loop is executed again; if VAR is greater
          or equal to Z, control passes to the
          first statement after the CONTINUE
          statement.
```

Since all arithmetic is performed in floating point format, the DO indicies may be negative or decimal fractions. The DO statement cannot be shortened; all three indicies must be present.

DO loops may be nexted in standard fashion, with the following restrictions:

- Loops may be nested 6 deep, maximum.
- Control cannot pass from the body of one loop to another loop or the rest of the program until the loop is finished. If escape from a loop is desired before the loop finishes normally,

the loop variable, VAR, may be modified (set equal to or greater than Z) or repeated jumps to the CONTINUE statement may be made.

- Several loops cannot terminate on the same CONTINUE statement. Each DO must have its own CONTINUE statement.
- The compiler does not check for proper nesting.
- The DO indicies can be either simple variables or constants.

Example:

```
*I,J,K,N,M
 .
 .
 .
READ N,M
 .
 .
 .
DO 10 I=1,1,N
 .
 .
 .
DO 20 J=-0.5,0.01,0.0
 .
 .                    inner
 .                     loop
20 CONTINUE
 .                                   outer
 .                                   loop
 .
DO 30 K=2.5E-6,M,10.0E-5
 .
 .                    inner
 .                     loop
30 CONTINUE
10 CONTINUE
```

<u>The HALT Statement</u>

The HALT statement causes the Operating System to cease execution. It has the simple form:

```
n HALT
```

Where n is an optional statement number.

SPECIAL PALTRAN OPERATIONS

The statements discussed in the previous sections comprise the basic Paltran-8 system. The PALX compiler, along with Paltran-Operating-System Number One (POS1), will handle all statements in the basic system. In order to extend the range of Paltran operations, three more Operating Systems are available to process new commands. This extension is made at the expense of data storage, hence restricting the user to smaller arrays. (Refer to OPERATING PROCEDURES for details.)

MATRIX OPERATIONS

The following matrix operations are available when Paltran-Operating-System Number Two (POS2) is used:

- Matrix Multiplication
- Matrix Inversion
- Matrix Transposition

and commands which

- Compute the determinant of a square matrix.
- Set an array equal to the identity or unit matrix.
- Extract the main diagonal of a matrix.

The above operations are available as extensions of the SET statement; the basic format is:

```
SET *X BODY
```

where SET is delimited by a blank and the * signals the compiler to process the one or two character code X, as one of the above operations; X must be delimited by a blank; BODY defines variables used in the operation.

MATRIX MULTIPLICATION

If $\underline{C}$ is an (N,M), $\underline{A}$ an (N,R) and $\underline{B}$ an (R,M) matrix, then the matrix product $\underline{C}$, of $\underline{B}$ pre-multiplied by $\underline{A}$ is defined as:

$$\underline{C}(N,M) = \underline{A}(N,R) \times \underline{B}(R,M)$$

$$\underline{C}(i,j) = \sum_{K=1}^{R} \underline{A}(i,k) \times \underline{B}(k,j)$$

$$i = 1,2.....N$$
$$j = 1,2.....M$$

Note: The number of columns in $\underline{A}$ must equal the number of rows in $\underline{B}$.

The form of the matrix multiplication statement is:

```
SET *M A*B=C
```

where M denotes matrix multiplication, and $\underline{C}$ is the resultant matrix product. The PALX compiler does not check if A, B and C are of the proper dimensions; it is up to the user to ensure C(N,M) = A(N,R) * B(R,M).

Example:

```
*C(10,5),A(10,3),B(3,5)
 .
 .
 .
READ "FIRST MATRIX",!,A,"SECOND MATRIX",!,B,!
 .
 .
 .
SET *M A*B=C
 .
 .
 .
WRITE %6/3,"ANSWER, PRODUCT",!,C,!
 .
 .
 .
HALT
```

MATRIX INVERSION

The Inverse $\underline{A}^{-1}$ of a matrix $\underline{A}$ is defined as

$AA^{-1}=A^{-1}A=I$ where I is the identity or unit matrix.

$$I=\begin{matrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ \cdot \\ \cdot \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{matrix}$$

The main diagonal elements are unity, while all off-diagonal are zero.

Paltran computes the inverse of a matrix by solving a set of n-simultaneous equations, by a variation of the Gauss-Jordan method. (In a true parallel processing system, other techniques would of course be used. Since Paltran-8 is a simulation, only the final results are important.)

If A, B and X are square matricies, then the matrix equation

AX=B may be solved for X

where A is the coefficient matrix
X is the solution matrix
and B is the matrix of constant terms.

If B is set equal to the identity matrix, then

AX=I and, by definition, the solution matrix X must be the inverse of A.

The form of the matrix inversion statement is:

```
SET *V INVR/A
```

where V denotes matrix inversion
A is the matrix to be inverted
INVR is the inverse of A

INVR must be set initially to the identity matrix; this may be done in two ways:

1. By a READ statement; however, this is tedious.

2. By the statement

```
SET *I INVR
```

where I denotes the operation of setting INVR equal to the identity matrix.

Unfortunately, the original matrix, A, is destroyed in computation, It is, in fact, reduced to the identity matrix.

The following Program is an example of matrix inversion.

PROGRAM	COMMENTS

`*A(5,5),INVR(5,5)`
`*SAVA(5,5),TEST(5,5)`
	Declare variables.

`READ A`	Read A, the matrix to be inverted.
`SET SAVA=A`	Since A will be reduced to the identity matrix, SAVA is used to save the original coefficients.
`SET *I INVR`	INVR is set equal to the identity matrix.
`SET *V INVR/A`	INVR will be computed as $\underline{A}^{-1}$ and A is reduced to $\underline{I}$.

`WRITE %6/3,"INVERSE",INVR!`
`WRITE "A MATRIX",A,!`
	The inverse matrix is written, along with A as a check.

`SET *M INVR*SAVA=TEST`	The inverse INVR, is multiplied by the original matrix SAVA. By definition, if the inversion process has been carried out successfully, TEST should be the identity matrix.

```
WRITE "CHECK ON OPERATION", TEST,!
HALT
```

The results from this program for the matrix

$$\underline{A}= \begin{array}{ccccc} 10 & 5 & 0 & -5 & 10 \\ 1 & .01 & .7 & -.1 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ -1 & -9 & .5 & .5 & .01 \\ .3 & -.3 & 1.5 & 4.5 & 10 \end{array}$$

appear in Appendix 1.

The only restriction on this method is that all diagonal elements of the A matrix must be non-zero, since division by A(i,i) occurs in the process. This method is also prone to the usual pitfalls involved in finding inverses of ill-conditioned matricies.

Since this method consists of solving equations to find an inverse, it can be used to solve simultaneous equations directly.

Example:

```
    *A(5,5),B(5,5)

    READ "COEFFICIENT MATRIX",A,!
    READ "MATRIX OF CONSTANTS",B,!
    SET *V B/A
    WRITE "SOLUTION MATRIX",B,!
    HALT
```

Instead of initially setting B equal to the identity matrix, B is read as a matrix of constants, and after the SET *V B/A statement is executed, B is the solution to:

$$\begin{matrix} a_{1,1}\ldots\ldots a_{1,5} \\ \vdots \qquad\qquad \vdots \\ a_{5,1}\ldots\ldots a_{5,5} \end{matrix} \; * \; \begin{matrix} x_{1,1}\ldots\ldots x_{1,5} \\ \vdots \qquad\qquad \vdots \\ x_{5,1}\ldots\ldots x_{5,5} \end{matrix} \; = \; \begin{matrix} b_{1,1}\ldots\ldots b_{1,5} \\ \vdots \qquad\qquad \vdots \\ b_{5,1}\ldots\ldots b_{5,5} \end{matrix}$$

that is B = x.

MATRIX TRANSPOSITION

The transpose B of matrix A is defined as,

$\underline{B}(j,i)=\underline{A}(i,j) \quad i=1,\ldots N, \quad j=1,\ldots M$

The statement:

```
    SET *T TRAN/A
```

will set TRAN(N,M) to be the transpose of A(M,N). It is up to the user to ensure dimensional compatibility.

DIAGONAL EXTRACTION

The statement

```
SET *DI A/VECT
```

will set VECT equal to the main diagonal of A; that is,

```
VECT(i)=A(i,i)      i=1,2......n
```

A must be a square matrix, and VECT must be of vector of similar dimensions.

Note the two-character code following the asterisk in the SET statement.

DETERMINANT OF A SQUARE MATRIX

The statement

```
SET *DE DET/A
```

results in the simple variable, DET being set equal to the determinant of A. Since division by A(i,i) occurs in the process, all main diagonal elements of A must be non-zero. As is the case with matrix inversion, the A matrix is destroyed in computation.

PALTRAN LIBRARY OPERATIONS

The second class of extended operations include:

- Plotting
- Numerical Integration
- Numerical Differentiation
- Searching vectors for minima and maxima
- Loading functional values into a vector.

These operations are handled by Paltran Operating System Three (POS3) and Paltran Operating System Four (POS4). POS4 is basically identical to POS3, with only minor changes made in input-output structure.

The compiler handles these operations under the LIBRARY statement, which has the basic form:

```
     L OPCODE VAR,EXP
```

where L denotes the library class of operations.
OPCODE is one of the following:

P	for PLOT
S	for SET
M	for MINIMAX
I	for INTEGRATE
D	for DIFFERENTIATE

Note, both the Library command and opcode must be terminated by blanks.

VAR,EXP is the remainder of the statement consisting of variables or arithmetic expressions or both.

All of the above operations deal with vectors as the basic unit of data storage. Many cases arise where the values of some function are desired over a given range. The Library statement SET is used to generate a vector whose elements are values of some function, say F(x) over the range XMIN$\geq$X$\geq$XMAX. The form of this statement is:

```
     L S V,X[FX]
```

where V is the vector that will contain the calculated values.
FX is some function of X.

FX must be enclosed in square brackets and the variable X must appear as above.

At <u>object or execution</u> time of the SET statement, the Operating System (POS3) will type XMAX:, to which the user responds by typing in the maximum value of the functional range. The operating system then types XMIN:, to which the user makes a similar response. The operating system then computes:

```
V(I)=F((XMIN-ΔX)+ΔX*I)      I=1,......N
```

where - N is the size of the vector

$$- \Delta X = \frac{XMAX-XMIN}{N}$$

Thus V will then contain F(x) evaluated from XMIN to XMIN+ΔX*(N-1) in increments of ΔX.
Note XMIN+ΔX*(N-1)=XMAX-ΔX.

Example:

```
*X,V(100)
.
.
.
L SET V,X  FSIN(X)+FCOS(2.0*X)
WRITE %8/4,"SIN-COS FUNCTION",V,!
HALT
```

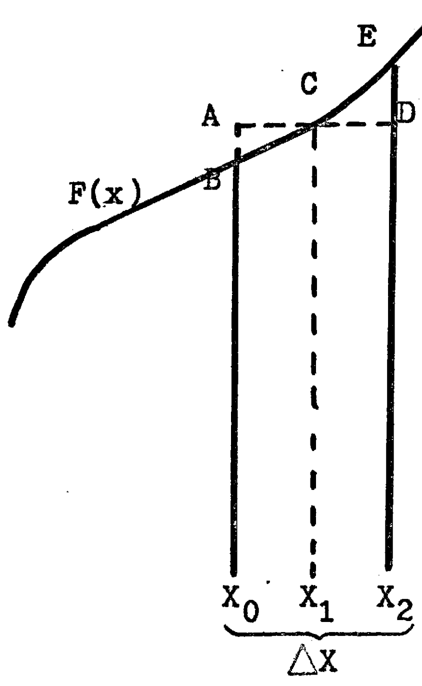
In this case, the vector V will consist of the function SIN(X)COS(2X) over a range determined by the user at execution time.

Integration and Differentiation are carried out by Paltran in a similar fashion.

The form of the Library statement INTEGRATE is

```
L I SUM,V,X[FX]
```

- SUM is a variable which will contain the value of the integral.
- V is a vector which contains the individual areas (see below).
- FX is a function of X.



If F(x) is evaluated at X_1 and ΔX is very small, $F(x_1).\Delta X$ is a good approximation of the area under curve of F(x) from X_0 to X_2, since the areas of triangles ABC and CDE are approximately equal. By making sufficient number of these area calculations and adding them, a good approximation to the true value of the integral can be made.

The range or interval of integration is again determined at execution time as was described under the SET statement. The size of the vector V determines the number of area evaluations; the larger V is, the more accurate the result. (Note, an integral curve can be plotted from the elements of V.)

Example:

```
*SUM,X,V(100)
.
.
.
L I SUM,V,X[X+1]
WRITE %6/3,"ANSWER",SUM,!
HALT
```

If the user entered the limits Ø, and 1 at execution time:

```
XMAX: 1
XMIN: Ø

ANSWER   1.498
```

Noting that $\int(X+1)dx = \left.\frac{X^2}{2} + X\right]_0^1$

$$= 1.5$$

In a similar fashion, a function may be numerically differentiated. The form of the Library statement DIFFERENTIATE is:

```
L D V,X[FX]
```

where FX is a function of X.

The function is divided into intervals as determined by the size of the vector V, and the slope of the function calculated, the individual slope calculations comprising the elements of V.

The differential curve of the function can now be plotted by means of the Library statement, PLOT, which has form:

```
L P V
```

where P is the abbreviation for PLOT and V is a vector of "Y" values. As before, an "X-value" range is determined at execution time from user input and the size of the vector V.

The X-scale is plotted down the teletype "Page" and the Y-scale plotted across. This routine is not very elaborate; however, plotting in all four quadrants is possible. The Y-range is limited to 50 places and the X-range by the length of the vector of plot points. To obtain Y-displacement, the value of the particular element in V is fixed and a number of spaces are typed as given by that value, followed by a period (.). Since this limits the range of values in the vector V from -25 to +25, the Operating system requests a third parameter: YMAG or Y-scale magnification. The values in the plot vector are multiplied by this parameter in order to place them in the -25 to +25 range. For convenience, the X and Y values are printed along the left-hand margin. If plotting will only take place in the positive Y-plane, the range of the plot vector may be changed to Ø to 5Ø by means of a switch register option. This is done by raising switch 11 (Bit 11 = 1).

Example:

```
*X,V(10)
WRITE %6/3
L SET V,X[X*X]
L PLOT V
HALT
```

At Execution time:

```
     XMAX: 10
     XMIN:  0
     YMAG: 0.25


  X        Y

0.000    0.000            .
1.000    0.250            .
2.000    1.000             .
3.000    2.250              .
4.000    4.000                 .
5.000    6.250                    .
6.000    9.000                        .
7.000   12.250                            .
8.000   16.000                                .
9.000   20.250                                    .
```

Note, the Y values printed are the scaled values.

Sometimes it is not apparent what the maximum or minimum value of a calculated set of points is. The minimum and maximum value, of the elements, of a given vector may be found by using the Library statement MINMAX. It has form:

```
     L M MAX,MIN,V
```

where - MAX is a variable name and is set equal to the largest element in vector V. Similarly, MIN is set equal to the smallest element.

At execution time, the operating system will also type out the minimum and maximum value.

Example:

```
     *X,MAX,MIN,V(25)
     100 L SET V,X[FEXP(X*FSIN(2.0*X))/X]
     200 L M MAX,MIN,V
     300 L P V
     HALT
```

Statement 100 describes a rather complicated function of X and it is not readily apparent what the maximum and minimum values would be for some given range.

Statement 200 results in a sort through the generated vector and prints the appropriate values. Knowing the range of "Y-values", the user can then enter a suitable value for YMAG in the plot routine.

POS4 MODIFICATIONS TO POS3

The printing and/or reading of scale or range values at execution time is convenient in some respects and not so in others. If the SET or MINMAX statement was in a loop, repeated entry of values from the keyboard would be troublesome.

Paltran Operating System Four (POS4) is a small modification of POS3 which changes the input-output mechanisms for the SET and MINMAX statements. At execution time, the operating system will read XMAX and XMIN, that is the function range, from core memory instead of from the keyboard. The values read are the first two variables declared. Similarly, the MINMAX statement obtains values for XMAX and XMIN from the first two variables dimensioned by the user. In addition, the MINMAX statement returns two new data items, which occupy the locations of the next two variables in core. The returned numbers are the values of X for which the maximum and minimum occur.

Example:

Suppose that some function has values, Y, for the given X-range:

X	Y
0	0
1	2
2	3
3	7
4	3
5	2
6	0
7	-2
8	-10
9	-2
10	0

Using POS4, the user must declare the first four variables appropriately:

```
*XMAX,XMIN,PMAX,PMIN,MAX,MIN,X,v(10)
READ XMAX,XMIN
L SET V,X  some F(x)
L M MAX,MIN,V
WRITE %6/3,"MINIMUM AND MAXIMUM VALUES ARE",
            MIN,"AND",MAX,!
WRITE "OCCURRING AT X=",PMIN,"AND",PMAX,"RESPECTIVELY",!
HALT
```

The program would return for the above values:

```
.
.
.
.
.
MINIMUM AND MAXIMUM VALUES ARE -10.000 AND 7.000
OCCURRING AT X=8.000 AND 3.000 RESPECTIVELY
```

Other than this, POS3 and POS4 are identical.

NOTE: The MINIMAX routine uses an internal calculation made by the SET, INTEGRATE OR DIFFERENTIATE routines; this necessitates using L M MAX,MIN,V in conjunction with L SET V,X FX ETC., when using POS4 only.

(Refer to Appendix II for a further example.)

OPERATING PROCEDURES

The PALX compiler is brought into core using the BIN loader, and starts in location Ø2ØØ. Before starting, the following switch register option should be set:

	Switch 11 UP (Bit 11=1)	Switch 11 DOWN (Bit 11=Ø)
ACTION	Source tape is read using the High Speed Reader (HSR). A $ is echoed every time a CR is read. Line-feeds, Ø2ØØ code and blank tape are ignored.	Source input is manual, through the keyboard. The Low Speed Reader (LSR) may be used.

Upon starting, PALX types

```
PALTRAN-8  GO
```

and reads a source tape or awaits keyboard entry, depending upon BIT 11. When a $ is encountered indicating the end of the source program, the compiler halts. Two switch register options are now available.

	BIT 11=1	BIT 11=∅
ACTION	Punch object tape on HSP (high speed punch).	Punch object tape on LSP.
	BIT ∅=∅	BIT ∅=1
	Type symbol table and program data after punching object tape.	Omit symbol print and program data.

After setting bits 11, and ∅ and turning on the appropriate punch, the user presses CONTINUE; PALX then punches the object tape. Error messages will be typed at this time if any errors are detected. After punching the object tape, PALX types

```
EOT (End of Tape)
```

to indicate a successful compilation. If the symbol print is selected, the names and dimensions of the variables will be printed. Example, if the following appear in the source program:

```
*ABCD,A,B,CAT(8),DOG(9),BIRD(4,5),MATR(7,7)
```

the symbol print will be:

```
SYMBOL  TABLE
NAME    DIMENSIONS
ABCD
A
B
CAT     008
DOG     009
BIRD    004    005
MATR    007    007
```

The compiler then prints the following program data:

- Number of lines in source program.
- Total data storage required (for variables only).
- Length of object program.
- Number of GOTO and IF links generated.

Note, all of the above numbers are converted to decimal (base 10) when typed by PALX. After printing this data, the compiler halts, and switch register options can be reset at this time; pressing CONTINUE restarts the compiler.

If the symbol print has not been selected, the compiler automatically restarts.

There is only one editing feature in PALX. When in the keyboard entry mode, the RUBOUT key may be used to delete the character just typed. Pressing RUBOUT again will erase the previous character, etc. A back arrow (<—) is echoed every time RUBOUT is used.

COMPILER DATA

The maximum source program length is 1280_{10} characters; use of abbreviated commands is therefore recommended to conserve space. Excessive use of RUBOUT is not advised since each RUBOUT adds to the source character count, and the deleted characters do not decrease this count. The source program may consist of a maximum of 50 lines (compatible with 1280_{10} characters, maximum). Each IF statement effectively generates GOTO statements, and these are included in the source count. The abbreviated form of the IF statement is therefore recommended wherever possible. In addition, the number of GOTO links will be correspondingly conserved.

Example:

```
IF [X+2.0*Y-ER]100,200,300,400
```

is considered as 5 source lines and requires 4 GOTO links.

```
IF [ALFA-EPS] 1000
```

counts as 2 source lines and uses only one GOTO link.

ERRORS DETECTED BY PALX

The Paltran compiler, PALX, checks the source program for syntax errors during compilation. In addition, source and object program overflow is detected and generates an error message. All Paltran-8 errors are fatal; the source program should be corrected before attempting execution of the object program.

Error messages are typed out in the following format:

```
     ERROR  XX
       AT   YY
```

where XX is a two-digit octal number indicating the error and YY is another octal number indicating the line in which the error occurred. The following table lists Paltran-8 errors detected by PALX:

Error Number		Description
ØØ	- - -	Source overflow. Source program too long. Compiler restarts.
Ø1	- - -	Missing line number, no GOTO transfer possible.
Ø2	- - -	Object program too long.
Ø3	- - -	Syntax error in a floating point number.
Ø4	- - -	Undefined variable name.
Ø5	- - -	Unrecognizable command; Invalid Paltran statement.
Ø7	- - -	Illegal terminator in a Dimension statement. ")" is missing.
1Ø	- - -	Too many characters in a variable name.
12	- - -	Unrecognizable extended SET command.
13	- - -	Mismatched parenthesis.
15	- - -	Unrecognizable operator or constituent in an arithmetic expression.
67	- - -	Unrecognizable command in a LIBRARY statement.
77	- - -	Same as error 1Ø. Too many characters in a variable name.

STATEMENT SUMMARY AND OPERATING SYSTEM DATA

The following statements can be handled by all operating systems (1-4)

```
*VAR,VAR(I),VAR(I,J),....

n SET X=e
n READ VAR,!,"...",....
n WRITE VAR,!,#,%x.y,"...",....
n IF [e] A,B,C,D
n GOTO r
n DO m VAR=I,J,K
m CONTINUE
n HALT

Where     VAR  is a variable name
          X    is a variable name
          e    is an arithmetic expression
          A,B,C,D,n,r,m   are statement numbers
          !    generates a CRLF
          #    generates a CR only
          %x.y changes output format
          "..."  denotes text
          I,J,K  are index parameters, either connstants
                 or variables
```

Paltran Operating System Number Two (POS2) will handle the following statements, as well as the basic commands:

n SET *M A*B=C	Matrix Multiplication
n SET *v AI=A	Matrix Inversion
n SET *I A	Identity Set
n SET *T TRAN/A	Transposition
n SET *DI A/VECT	Diagonal Extraction
n SET *DE DET/A	Determinant of a square matrix

Paltran Operating Systems Three and Four (POS3), (POS4) will handle the following Library statements as well as the basic commands:

L P V	Plot; Vector V
L S V,X [F(X)]	Set; V=F(X) over XMAX,XMIN
L I SUM,V,X [F(X)]	Integration; SUM is the value of the integral, V contains the area calculations.
L D V,X [F(X)]	Differentiation
L M MAX,MIN,V	Extraction of minimum and maximum

All four operating systems can accommodate 26_{10} GOTO and IF links, and a maximum of 380_{10} locations can be used to store the object program. The compiler will type the number of links and locations used by the object program at the end of each compilation.

The object tape is loaded (by the operating system) by placing the tape in the HSR and starting the computer in location Ø177. Execution begins immediately after loading. If the program is to be run again, it may be restarted in location Ø2ØØ.

If a high speed reader is not available, the following change must be made in order for the operating system to use the LSR:

LOCATION	OLD CONTENTS	NEW CONTENTS
2137	6014	6032
2133	6011	6031
2135	6016	6036

ERROR MESSAGES

The operating system will type "TILT" if the square root of a negative number, or divide by zero is attempted. The absolute value of the operand will be taken, in the case of the square root operation, and the quotient will be set to the highest positive number if division by zero occurs.

If the operating system receives an invalid operator in a SET statement, it will cease execution and 7777 will be displayed in the Accumulator. There is no recovery possible.

DATA STORAGE

The following table lists the memory available for data storage in each of the operating systems:

SYSTEM	LOCATIONS (MAXIMUM)	NUMBERS*
POS1	900	300
POS2	450	150
POS3	591	197
POS4	591	197

*A floating point number requires 3 locations for storage.

If location Ø141 is changed from 425Ø to 453Ø in POS3 or POS4, 600 locations (or 200 numbers) are available for data storage. However, this change limits DO loop nests to 4 deep, maximum.
The compiler types the number of locations needed for data storage at the end of each compilation.

PALTRAN BUGS

There are two Bugs in the PALX compiler. If a Dimension statement ends with a vector or matrix variable, a blank entry will be placed in the symbol. The symbol print routine stops printing when a blank entry is encountered; although the compilation is unaffected, variables declared after the initial statement will not appear in the symbol table print. If a simple variable terminates the Dimension statement, this does not happen.

The Paltran operators (+,-,*,/, ↑) require double operands. Problems occur when (-) is used to denote a negative number. Palx will treat the (+) or (-) operator as indicating a positive or negative number only if this is the <u>first</u> character in an arithmetic expression.

e.g. SET X=-2 is compiled as SET X=Ø-2

However, if a case such as:

SET X=FSIN(-2) arises, incorrect results occur, since the "(" and "-" are treated as two operators in succession. Writing SET X=FSIN(Ø-2) will correct this condition.

Due to round-off in the floating point format, "9" is returned as 8.9999..... . When the index accompanying vector or matrix variables is typed, (9) : will appear as (8) : due to truncation.

APPENDIX I

Sample Program
Matrix Inversion

```
MATRIX TO BE INVERTED

MATRIX INPUT
ENTER ROW( 1)
( 1)  :10 ( 2)  :5 ( 3)  :0 ( 4)  :-5 ( 5)  :10
ENTER ROW( 2)
( 1)  :1 ( 2)  :.01 ( 3)  :0.7 ( 4)  :-0.1 ( 5)  :4
ENTER ROW( 3)
( 1)  :1 ( 2)  :2 ( 3)  :3 ( 4)  :4 ( 5)  :5
ENTER ROW( 4)
( 1)  :-1 ( 2)  :-9 ( 3)  :.5 ( 4)  :.5 ( 5)  :.01
ENTER ROW( 5)
( 1)  :.3 ( 2)  :-.3 ( 3)  :1.5 ( 4)  :4.5 ( 5)  :10


INVERSE MATRIX

        1        2        3        4        5

( 1)     .168  -   .704     .117     .117     .054

( 2)  -   .020     .079     .009  -   .119  -   .015

( 3)  -   .099     .816     .334     .032  -   .393

( 4)     .071  -   .804     .072     .047     .213

( 5)  -   .022     .263  -   .086  -   .033     .060


ORIGINAL 'A' MATRIX

        1        2        3        4        5

( 1)    1.000     .000     .000     .000     .000

( 2)     .000    1.000     .000     .000     .000

( 3)     .000     .000    1.000     .000     .000

( 4)     .000     .000     .000    1.000     .000

( 5)     .000     .000     .000     .000    1.000


CHECK ON INVERSE, INVR*A=I

        1        2        3        4        5

( 1)    1.000  -   .000     .000  -   .000     .000

( 2)     .000    1.000     .000     .000     .000

( 3)  -   .000     .000     .999     .000  -   .000

( 4)     .000  -   .000     .000     .999     .000

( 5)  -   .000  -   .000  -   .000     .000     .999
```

APPENDIX II

Parallel Processing And Functional Analysis

The repertoire of Paltran - 8 Library commands may be used to demonstrate the application of a linearly configured array of processing elements or cells, to functional analysis. Each cell is assumed to have the following properties:

- arithmetic capability
- local storage or memory
- communication with a central control
- communication with central memory

The availability of many processing units makes possible a somewhat unusual technique of analyzing functions for the occurence of zeros, maxima or minima. Let f(x) = y be some function defined over the range (xmax,xmin) in which at least one zero, maximum or minimum is assumed to exist. The objective is then to find the value x* for which f(x*) = Q, where Q is the required condition. In operation, the function is evaluated over (xmax,xmin) in <u>n</u> steps (assuming <u>n</u> processors) and the resulting array searched for the required condition, noting the value of x for which this occurs. If <u>n</u> is sufficiently large, the answer may be found on the first pass, if not, new values of (xmax,xmin) are established around x* for which f(x*) $\cong$ Q and the process repeated until sufficient accuracy has been attained. In general there may be any number of zeros or extrema in the given range.

The speed at which the answer is found is quite considerable since only one effective function evaluation is performed per iteration. If large numbers of processors are available the value of x* approaches the final value by several orders of magnitude for each pass, thus necessitating only a few iterations. It should be noted however, that new ranges (xmax,xmin) cannot be set arbitrarily close to the approximation x*, since round-off and truncation error can accumulate and create problems of overshoot.

The case of f(x) = Ø will be chosen in the following example to demonstrate the equation solving ability of this particular method. There are many equations that arise in the scientific and engineering fields that cannot be solved by analytic means or require a disproportionate amount of work to do so. As an example, in certain microwave matching problems, the following equation must be solved for Z_1, given Z_0, Z_L and θ_z

$$\frac{Z_L - Z_0}{\tan^2\theta_z} = \frac{Z_1^2}{Z_0} + 2\left(\frac{Z_L}{Z_0}\right)^{\frac{1}{2}} \cdot Z_1 - \left(\frac{Z_L Z_0^2}{Z_1^2}\right) - 2\left(\frac{Z_L}{Z_0}\right)^{\frac{1}{2}} Z_1^{-1} \cdot Z_0^2$$

It is not immediately obvious which value(s) of Z_1 will satisfy the above equation. Solving for Z_1 directly is an excersize for the student.

To illustrate the method let us take the simpler case of the parabola: $(x-2)^2 - 10$. We may easily solve for x directly,

$$(x-2)^2-10 = 0$$
$$(x-2) = \pm\sqrt{10}$$

$$x = 2 \pm 3.162$$

$$x = 5.162 \quad \text{or} \ -1.162$$

Although it is now obvious in which interval the solution(s) lies, it is good practice to obtain a plot of the function to obtain a rough idea of its behaviour.

Fig. II is a plot of (x-2)*(x-2) - 10 over ($-5 \geq x \geq 10$) obtained from the following program:

```
*MAX,MIN,XV(30)
W %6/3
L SET V,X [(X-2)*(X-2)-10]
L M MAX,MIN,V
W "PLOT OF PARABOLA (X-2)↑2-10",!!!
L PLOT V
HALT
$
```

using POS3.

Fig. III is a source lising of the program used for the analysis. (POS4 is required).
The range values XMAX and XMIN are read in along with EPS, the value of f(x) at which we will consider a solution has been reached; the variable C is used as an iteration counter. (REF. statements # 1,2,3). The statement (#4)
L SET V, X [(X-2)*(X-2)-10] then fills vector V with 100 function evaluations over (XMAX,XMIN); the elements of V correspond to the cells of the array. The next statement (#5) SET V=FABS(V) will make all zero-crossings appear as minima, which can be found by means of statement number 6, L M MAX,MIN,V. (note, only global minima will be located; if two or more minima appear and are equal, only the last found will be returned by the MINIMAX routine.) The minimum found is then tested to determine if it is sufficiently close to zero. If it is not within the error limit new values of XMAX and XMIN are calculated and the process repeated.

New values of (XMAX,XMIN) are calculated as follows:
(REF. statements #8,9,10,11,12)

Let D1 = PMIN-XMIN

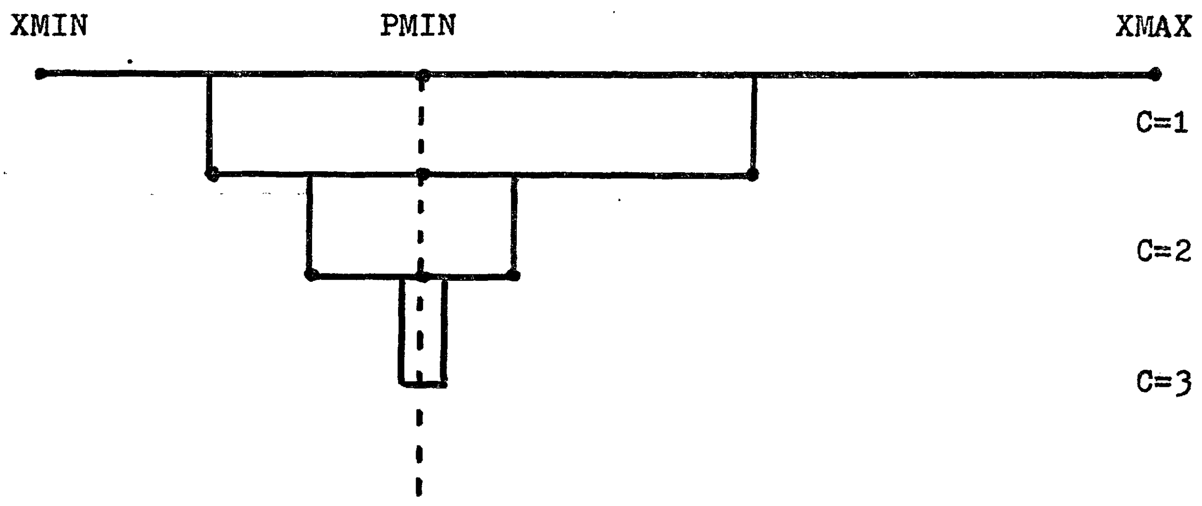
D2 = XMAX-PMIN

then

XMIN = PMIN-D1/(1+C)

XMAX = PMIN+D2/(1+C)

where C is the iteration counter and PMIN is the value of <u>x</u> for which the minimum occurs.
The following line diagram illustrates the convergence of (XMAX,XMIN) around PMIN with each successive iteration.



The following results were obtained from the program in Fig. III

INITIAL RANGE VALUES	ERROR EXIT LEVEL	SOLUTION X =	RESIDUAL VALUE OF FUNCTION	NUMBER OF ITERATIONS TAKEN
50, -50	.0001	5.162	.000099	6
0, -100	.0001	-1.162	.000032	7

The power of this method becomes even more apparent if the array of processing elements is extended to, two, three or more dimensions, enabling analysis of mult-variable functions.

This technique is a good example of how parallel processing can make possible methods which have been considered impracticle (or impossible) in the past.

```
     XMIN:-5 XMAX:10

MAX    46.250
MIN  - 10.000

PLOT OF THE PARABOLA (X-2)↑2 - 10

XMIN:-5
XMAX:10
YMAG:.5


  X        Y

-  5.000    19.499
-  4.499    16.124
-  4.000    12.999
-  3.500    10.125
-  3.000     7.500
-  2.500     5.124
-  2.000     3.000
-  1.500     1.124
-  1.000  -   .500
-   .500  -  1.875
    .000  -  3.000
    .500  -  3.874
   1.000  -  4.499
   1.500  -  4.875
   2.000  -  5.000
   2.500  -  4.875
   3.000  -  4.499
   3.500  -  3.874
   4.000  -  3.000
   4.499  -  1.875
   5.000  -   .500
   5.499    1.124
   6.000    3.000
   6.499    5.124
   7.000    7.500
   7.500   10.125
   8.000   12.999
   8.500   16.124
   8.999   19.499
   9.500   23.125


END
```

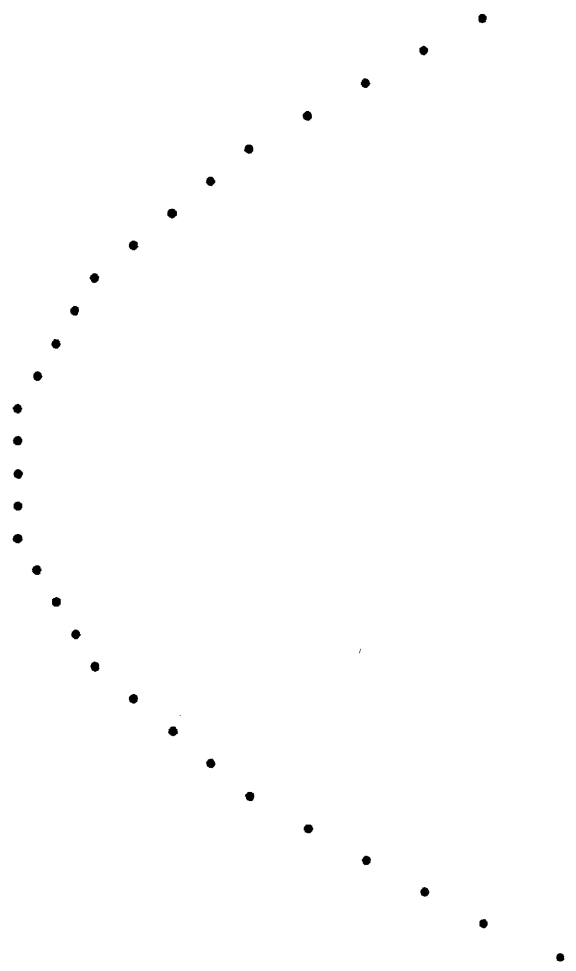


Fig. II -- Plot of Parabola $(x-2)^2 - 10$

```
PALTRAN-8  GO
*XMAX,XMIN,PMAX,PMIN,MAX,MIN,X,C,D1,D2,EPS,V(100)
1 R "ENTER INITIAL RANGE VALUES  ",!,XMAX,XMIN,!
2 R "ENTER ERROR EXIT LEVEL  ",EPS,!!
3 S C=0
4 L S V,X[(X-2)*(X-2)-10]
5 S V=FABS(V)
6 L M MAX,MIN,V
7 I [MIN-EPS]100,100
8 S C=C+1
9 S D1=PMIN-XMIN
10 S D2=XMAX-PMIN
11 S XMAX=PMIN+D2/(1+C)
12 S XMIN=PMIN-D1/(1+C)
13 W "."
14 G 4
100 W %8/4,!!!,"SOLUTION",!,"FUNCTION IS ZERO FOR X - ",PMIN,!!
101 W "ITERATIONS TAKEN  ",%3/0,C,!!
102 W %8/6,"RESIDUAL VALUE OF FUNCTION   ",MIN,!!!!!!
103 HALT
$

EOT


SYMBOL TABLE
NAME   DIMENSIONS
XMAX
XMIN
PMAX
PMIN
MAX
MIN
X
C
D1
D2
EPS
V      100


TOTAL DATA STORAGE   333

SOURCE LENGTH  021  LINES

OBJECT LENGTH  275  LOCATIONS

GOTO AND IF LINKS  003
```

Fig. III

Source Listing of Equation Solving Routine