Mini-Workshop on Data Directed Computation

12th & 13th June 1978

TIMETABLE

Monday

10.30	- 11.00	Coffee
11.45	- 12.30	Introduction to the Workshop (Brian Randell and Bob Hopgood)
12.30	- 13.45	Lunch
13.45	- 14.30	LLL presentation (George Michael)
14.30	- 15.15	LLL presentation (Chris Hendrickson)
15.15	- 16.00	Manchester presentation (John Gurd)
16.00	- 16.30	Tea
16.30	- 17.15	Newcastle presentation (Phil Treleaven).

Tuesday

09.00	- 09.45	Westfield presentation (Peter Osmon)
09.45	- 10.30	Manchester presentation (Greg Egan)
10.30	- 11.00	Coffee
11.00	- 11.45	Clarkson presentation (Susan Conry)
11.45	- 12.30	Discussion:range of applicability of data directed computation
12.30	- 13.45	Lunch
13.45	- 14.30	Warwick presentation (Bill Wadge)
14.30	- 15.15	Discussion:programmability - will users demand FORTRAN
15.15	- 16.00	Discussion:performance issues
16.00	- 16.30	Tea
16.30	- 17.15	General (final) discussion.

Mini-Workshop on Data Directed Computation

12th & 13th June 1978

TIMETABLE

Monday

10.30	11.00	Coffee
11.45	12.30	Introduction to the Workshop (Brian Randell and Bob Hopgood)
12.30	13.45	Lunch
13.45	14.30	LLL presentation (George Michael)
14.30	15.15	LLL presentation (Chris Hendrickson)
15.15	16.00	Manchester presentation (John Gurd)
16.00	16.30	Tea
16.30	17.15	Newcastle presentation (Phil Treleaven).

Tuesday

09.00	09.45	Westfield presentation (Peter Osmon)
09.45	10.30	Manchester presentation (Greg Egan)
10.30	11.00	Coffee
11.00	11.45	Clarkson presentation (Susan Conry)
11.45	12.30	Discussion:range of applicability of data directed computation
12.30	13.45	Lunch
13.45	14.30	Warwick presentation (Bill Wadge)
14.30	15.15	Discussion:programmability - will users demand FORTRAN
15.15	16.00	Discussion:performance issues
16.00	16.30	Tea
16.30	17.15	General (final) discussion.

Lucid and its dataflow semantics

W. Wadge
Warwick

Lucid is a vaguely Algol-like nonprocedural language based on equations. Its unusual feature is that it allows programmers to write natural looking programs which can be understood operationally as using iteration - thus disproving the widely held belief that iteration is incompatible with the nonprocedural (or applicative) approach to programming.

The semantics of a Lucid program is given mathematically, not operationally, so it cannot be said to be a dataflow language (because dataflow is an operational concept). There is, however, a very simple way of translating a Lucid program into a data flow net; but the net does not necessarily compute the output predicted by the mathematical semantics. The problem is that the net may deadlock, possibly in the course of computing unnecessary values. Fortunately there is a simple test (the cycle sum test) which rules out deadlock for a large class of "sensible" programs.

The lack of coincidence between the mathematical and the dataflow semantics of Lucid can be interpreted in two ways: as evidence that the purely mathematical approach is in some respects unrealistic; or as evidence that dataflow is in some respects inadequate. We argue for the latter.

WARWICK REPORT 21

WORKSHOP ON DATA DIRECTED COMPUTATION

This workshop, organised by the Computing Laboratory of the University of Newcastle upon Tyne, is sponsored by the Distributed Computing Systems project of the S.R.C. Computing Science Committee. It brings together representatives of several groups involved in the D.C.S. programme, and of other groups, all of whom are interested in one particular area of Distributed Computing Systems.

This is the area that we have, perhaps misleadingly termed 'Data-Directed Computation' - a term we have borrowed from our Livermore Laboratory colleagues. In fact the various groups represented here are tackling what in some cases are quite different problems, and have a range of quite different motivations underlying their work. These range from concentration on ultra-high performance computers for number crunching tasks, to interest in the provision of programs which can be formally validated. In some cases therefore the stress is on CPU architecture and design, in others on programming languages.

However there is a crucial common thread - namely the wish to break away from the conventional, largely sequential, form of programs, and of program execution, in favour of a form which facilitates the use of parallelism, perferably on a large scale.

Conventional programs, even on computers which are capable of parallel activity, are implicitly sequential, and require any desired parallelism to be indicated explicitly. It seems to me that the central theme of this workshop is a belief in the potential value of the opposite approach, i.e. of programs which are to be regarded as implicitly parallel, and where sequentiality has to be indicated explicitly. One motivation for the belief is the view that programs which are, so to speak, unnecessarily sequential, are an obfuscation of the programmer's real intention - another is that programs which have implicit parallelism and explicit sequentialism are a more appropriate method of controlling the sort of hardware with high replication factors that current technology makes feasible, and so of providing the means of a massive increase in processing speed. My own interest is not so much motivated by a concern for the speed of particular types of numerical computation, but rather by being intrigued about the possibility (and I can as yet put it no higher than that) of finding an entirely new approach to general purpose computing.

We are gradually realising that there are, not surprisingly, a number of different forms of implicitly parallel, explicitly sequential programming languages (both textual and graphical) and a variety of computer architectures that could be used to execute such programs. One language/architecture pairing, to date the most extensively studied, is that of the single-assignment language and the data flow computer architecture. This approach can be characterised by the abandonment of the notion of a variable, and hence of that of explicit storage. However other approaches retain these concepts.

I expect that this workshop will enable us all to obtain a better understanding of these various approaches, and to learn what each group is trying to achieve and has achieved, and will provide a major stimulus for friendly co-operation and competition between the groups. It may even enable us to reach some agreement on the definitions of the various terms we use to describe our work.

B. RANDELL

12th June 1978

DATA-DRIVEN COMPUTATION: Its Suitability in Control Systems

G.K. Egan and S. Wathanasin

Computer Science Department

University of Manchester

Manchester M13 9PL.

Abstract

Initial simulation studies have shown that the data-driven model of computation is suitable for expressing solutions to simple stimulus-response problems. Simulation becomes difficult for more realistic problems, and rarely produces convincing results. We are therefore developing a flexible hardware/software system to allow a more detailed study.

Introduction

We know from [Adams] that the data-driven computation model can represent any computable function. In theory, this means that all programs can be put in a "data-driven computation" (or data-flow for brevity) form; this is true even if the program has no obvious inputs. It follows from this that the model may be used as the basis of a general-purpose computing system, and this is the starting point of other data-flow project-groups (e.g. [Gurd], [Arvind], [Dennis]). We would agree with this premise, but feel that the control-systems application area with its high intrinsic parallelism should show the data-flow system at its best.

Not all control-systems applications are suited for implementation on a data-flow system. For example, a plant that has to perform a fixed sequence of operations may best be controlled by a conventional computer. The applications that we think are suitable are those in which the controlling system has to respond to external stimuli. A simple example is a plant that has to be maintained at a specific temperature .

In the following sections, we shall discuss the suitability of data-flow systems for handling stimulus-response systems, and describe an experimental system that we can use to study this problem. We will assume that the reader is familiar with the data-driven-computation model; the graph primitives are similar to those proposed by Gurd et al. [Gurd].

Stimulus-response Systems: Is data-flow suitable?

Characteristics of Stimulus-response Systems

We give here the characteristics of a stimulus-response system that are relevant to the discussion in the next section.

First, the controller has no direct control over either the time, or rate at which stimuli arrive. The controller must be able to cope with bursts of stimuli.

Second, the controller must respond in a certain time if it is to maintain control. This "critical period" depends on the application and may vary between micro-seconds and hours.

Third, the responses must be in the same order as the stimuli that caused them, i.e. the controller must be determinate.

Last, we expect the (controlling) computer system to be a small part of the system being controlled. The control system must work on its own once it has been set up. We do not rule out the use of other computers in setting up the system (cross-compiling the required programs , for example). In some applications the computer system may be physically embedded in the system being controlled (e.g. in controlling a robot); in these cases the controller must be small and light.

Because we are short of time, we shall not discuss learning (or

adaptive) control systems although we do not rule them out.

Suitability

We became interested in stimulus-response systems because of the strong resemblance to data-flow computation. In both cases, a function of the data is evaluated; this evaluation is triggered by the arrival of the data. No special programming (e.g. interrupt handling in a conventional system) is required to deal with uneven data-arrival rates; data that cannot be dealt with immediately is queued. Moreover, the data-driven computation model is determinate [Adams].

Finding the response time of systems where queueing is not maintained (e.g. [Gurd]) is difficult because pieces of data may "overtake" one another inside the system. We cannot, therefore, give an upper bound for the time a particular piece of data exists in the system. If we implement a "strictly queued" system (where no "overtaking" is allowed), then we can bound the time taken to evaluate the response function for a particular stimulus. If we also know (or can assume) a probability distribution for data arrival, then, using queueing theory, we can calculate the expected response time.

Another advantage of data-driven systems is the natural way in which unusual I/O devices, and specialised processors (e.g. array processors) may be interfaced. All we need to do is define them as new primitive functions or procedures and direct data appropriately.

Finally, the data-flow system can make use of inherent parallelism in the control program. There are two forms of exploitable parallelism:

1) Given one stimulus the program may have more then one executable node at any given time. We call this static parallelism.

2) Given more than one stimulus further gains may be made through a "pipe-line" effect in the program.

We call the overall parallelism dynamic parallelism.

We concluded that stimulus-response systems were a fruitful area for study, and we then chose a specific control application , hand-eye systems, to examine in detail. A hand-eye system can be studied on a small scale and, moreover, is of research interest in its own right. We shall now describe it in more detail.

Hand-eye Systems

Background

Industrial manipulators or robots have been in use for some time on production lines in Germany, Japan, and the United States though they are rare in this country. Although manipulator technology is quite advanced, the control systems and sensory devices are not. One of the more common manipulators is the Unimate [Shosan]; it is "blind" and has a fixed 200 step memory. However, the cost of transfer and indexing equipment may be ten times the cost of the manipulators themselves[Nitzan].

The most useful sense for finding and identifying randomly placed objects is sight. Optical sensors, or "eyes", in the past have usually been based on video cameras; these devices are passive in nature, and require the storage of large amounts of picture data. In addition, excessive computation is needed to extract depth information. Currently, one of the more interesting "eyes" is the laser tracker. The device is quite simple and initial research, carried out in Japan [Ishii], seems promising. A laser beam is deflected so that a spot in the field of interest is illuminated. An image scanner gives the position of the spot in the plane normal to the laser axis. It is then a simple matter, given the geometry of the system, to compute the cartesian co-ordinates of the spot relative to some reference frame. By suitable deflections of the beam, the size and shape of objects in the field of view may be determined. The laser tracker allows the control system to actively

interrogate its environment, whilst reducing the computational and storage overheads mentioned above.

Hand-eye systems (the combination of a manipulator and optical sensor) exhibit a high degree of parallelism. Parallelism exists in the control of the physical device (for example, control of the various motors in the manipulator) and the identification of objects by forming a number of hypotheses. The overall control task may be easily partitioned into quite simple sub-tasks well matched to the capabilities of current microprocessors. In the case of the Japanese team, the choice was to use two mini-computers, one for the tracker, and one for the manipulator.

Laser Tracker Control

We started by looking at this part of the problem. Figure 1 shows the tracker geometry. Computation of the laser spot co-ordinates is simple. The only complication is that the laser deflection angle must not be changed until the scanner generates its output (a simple feedback loop which gates new inputs solves the problem).

Although the tracker program (figure 2) did not seem to be highly parallel, simulation showed that the best possible speed-up is about 11 (40 processor modules); of this, 3.75 is due to static parallelism, and this speed-up is increased by a factor of 3 because of the pipe-line effect.

Limitations of Simulation

Initial simulation results are encouraging and a more detailed study is indicated. However extending the study by simulation leads to problems:

1) It is difficult to simulate, even at a simple level, a real environment with which the laser tracker can interact. Simulating a realistic visual environment for the laser tracker may prove impossible.

2) The cost of the simulation (in time and space) will be prohibitive with realistic programs.

We, therefore, decided to build an experimental system to study in more detail the problem of multi-processor control of the hand-eye systems. We describe this system in the next section.

The Experimental System

General

As the time available to us is limited, we are concentrating on the processing section of the facility. If time permits, we will build the "eye" section. The multi-processor is being implemented using both software on MU5 (MU5 is described in [Ibbett]) and actual hardware (figure 4). This approach has been chosen for the following reasons:

1) Full simulation of the processor and environment would have been excessively time consuming and not very convincing.

2) Emulation of the processor alone by MU5 is difficult in terms of interfacing real-time devices.

3) Construction of a complete system is expensive in time and money; answers to all the problems are not yet known.

Consequently we chose a mixture of hardware and software emulation. This will give us the following advantages:

1) Software can be debugged in parallel with machine construction.

2) Access to MU5 peripherals and software facilities.

3) Ease of interfacing with real-time devices.

Figure 4 shows the proposed system architecture. Four hardware modules are being built and will be interfaced with the MU5 system. Programs on MU5 will simulate another 12 modules (this number can be easily changed). Other support software will be made available on MU5 - an emulator for the processor chip on which we chose to base the hardware modules (the ZILOG Z80A), an assembler for the Z80A, and a high-level-language compiler for the data-flow system. A high-level simulator already exists for the hardware and this was used for initial evaluation.

Architecture of the Data-flow System

A block diagram of the data-flow system is given in figure 5. Basically, the system goes through the same processing cycle as the system proposed by Gurd et al [Gurd], and we will not describe it here. The main differences between their system and ours are :

1) The result queues in our system are true FIFO queues. This removes the need for iteration levels, and also allows us to estimate response times (as described earlier).

2) A processor module consists of an execution unit, a node

store and a result queue; data tokens do not enter a result queue unless they are destined for a node defined in the associated node store.

3) The details of the procedure calling mechanism are different .

4) Primitives are provided for manipulating streams [Weng]. This is a general mechanism for simulating most data-structures and for sequencing input and output.

5) Storage nodes are not provided. "Memory" is provided by the stream mechanism.

6) The data-tokens are typed; this provides for runtime checking and it also allows us to have variable-sized tokens.

Hardware

The hardware, while not optimal for data-flow, has a high degree of flexibility. This is essential as there is very little we are certain of.

Processor modules execute data-flow programs interpretively. To allow this the multiprocessor is configured as shown in figure 5.

Inter-module communication can only take place via the exchange and the destination module's result queue. Modules may communicate directly only with peripherals and store connected to their own bus. The use of an exchange is somewhat arbitrary, but it is simple to implement and can be replaced by other structures at a later date. The exchange has slots for 8 modules with an initial complement of 4. The modules are based on the ZILOG Z80A 8 bit microprocessor chip. The broad characteristics of the modules are as follows:

1) 1.0uSec. instruction time.

2) 4K X 8 bit bytes of dynamic RAM.

3) 2K X 8 bit bytes of EROM (expandable to 64K bytes).

4) 2 X 8 bit parallel I/O ports (expandable to 256 ports).

5) RS-232C serial interface.

The RS-232C interface of one module will be linked to the PDP-11 multiplexer for communications with MU5.

The language

We have designed (and are currently implementing a translator for) a high-level language called NEWSPEAK. The language is based on LISP [McCarthy] because of its suitability for expressing parallelism, its established use as a programming language and its

use as a formal mathematical notation. The functional form is well-suited to data-flow use and list structures are easily implemented using the stream mechanism. Some features of the language are :

1) the types of variables may be specified,

2) new data types and operations on them may be specified using the Class mechanism,

3) no side effects are permitted,

4) no iterations are allowed,

and 5) conditional expressions and function arguments may be evaluated in parallel.

We can best show the language in the space available by giving an example of its use. Figure 6 shows the NEWSPEAK version of the laser - tracker controller. For brevity, we assume that "readscanner" is available as a primitive function. Auxiliary definitions are introduced to make the program more readable. "Define" binds a function name to an expression; the type specifications should be self-explanatory. "Brackets" forms its arguments into a stream (a function may only return one argument). When the arguments for this function are available, the function can be evaluated; these arguments are obtained by evaluating the auxiliary functions X, Y, and Z, and they in turn need the results of other function

evaluations.

Figure 7 shows hand-compiled code from the program of figure 6; the compiler should produce similar code. In this example, type checking can be done at compile-time, and no extra code is needed. If a name occurs more than once, the appropriate number of "dup" instructions is generated.

We hope that this brief description gives the flavour of the language.

Current status

We are currently implementing the experimental system described above.

Conclusions

An application area has been described for which data-driven architectures seem well-suited. A particular problem in this area was studied in detail and we presented some simulation results for this problem. Finally, we described an experimental system that would allow us to study the problem in greater detail.

Acknowledgements

We would like to thank our supervisor, Prof. Morris, for his help and guidance, the members of the data-flow research-group for many

useful discussions, and S. Williamson for proof-reading.

We also thank the University of Manchester and the Computer Science Department for their financial support during the course of this project.

References

[Adams] Adams, D.A., A Model for Parallel Computations, Hobbs et al. (ed) Parallel Processing Systems, Technologies and Applications, Spartan Books 1970.

[Arvind] Arvind and Gostelow, K.P., A Computer Capable of Exchanging Processors for Time, Information Processing 77, North Holland '77, pp849-853.

[Dennis] Dennis, J.B., Misunas, D.P. and Leung, C.K., A Highly Parallel Processor Using a Data Flow Machine Language, CSG Memo, Lab. for Comp. Science, MIT, Jan 1977.

[Gurd] Gurd, J.G., Watson, I. and Glauert, J., A Multi-layered Data Flow Computer Architecture, Internal Report, Dept. of Comp. Science, University of Manchester, Jan 1978.

[Ibbett] Ibbett, R.N. and Capon, P.C., The Development of the MU5 Computer System, CACM Vol.21 No.1, Jan 1978.

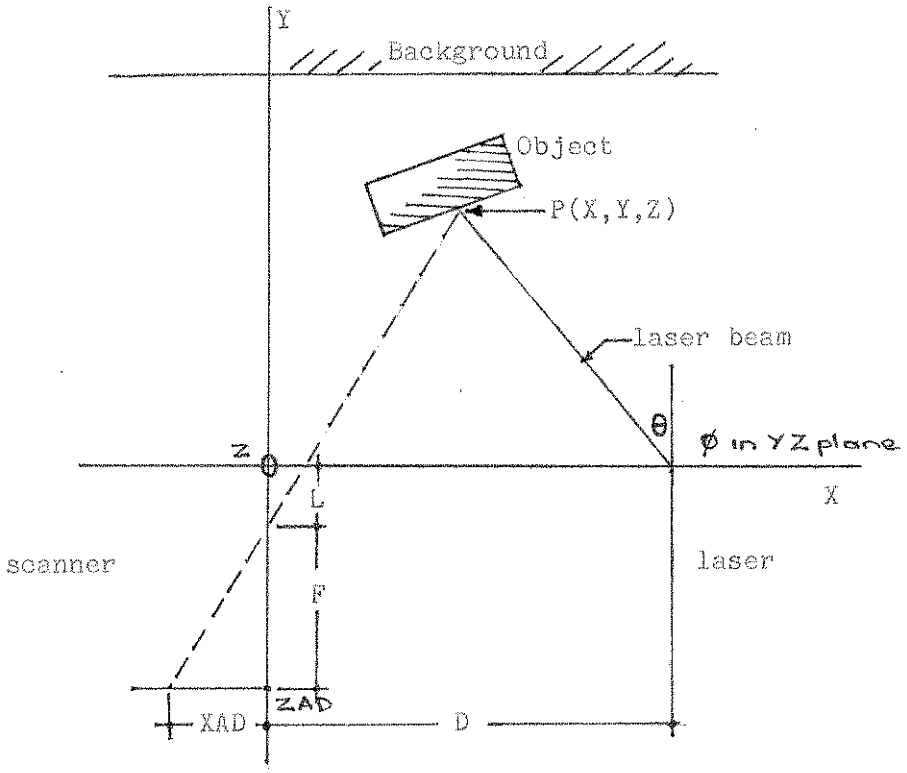
[Ishii] Ishii, M. and Nagata, T., Feature Extraction of Three-dimensional Objects and Visual Processing in a Hand-eye System, Pattern Recognition Vol.88 p229.

[McCarthy] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. LISP 1.5 Programmer's Manual , MIT Press (2nd Edn '66)

[Nitzan] Nitzan, D. and Rosen, C.A., Programmable Industrial Automation, IEEE Transactions on Computers, Vol.C-25 No.12, p1259.

[Shosan] Shosan, S., Industrial Robots: Their Potential in Today's Plant Operations, Plant Eng., July 1969.

[Weng] Weng, K.S., Stream-Oriented Computation in Recursive Data-flow Schemas, MIT Technical Memo. No. 68, Lab. for Computer Science, Oct. 1975



$$X = XAD.(D + L.\tan(\theta))/(XAD + F.\tan(\theta))$$

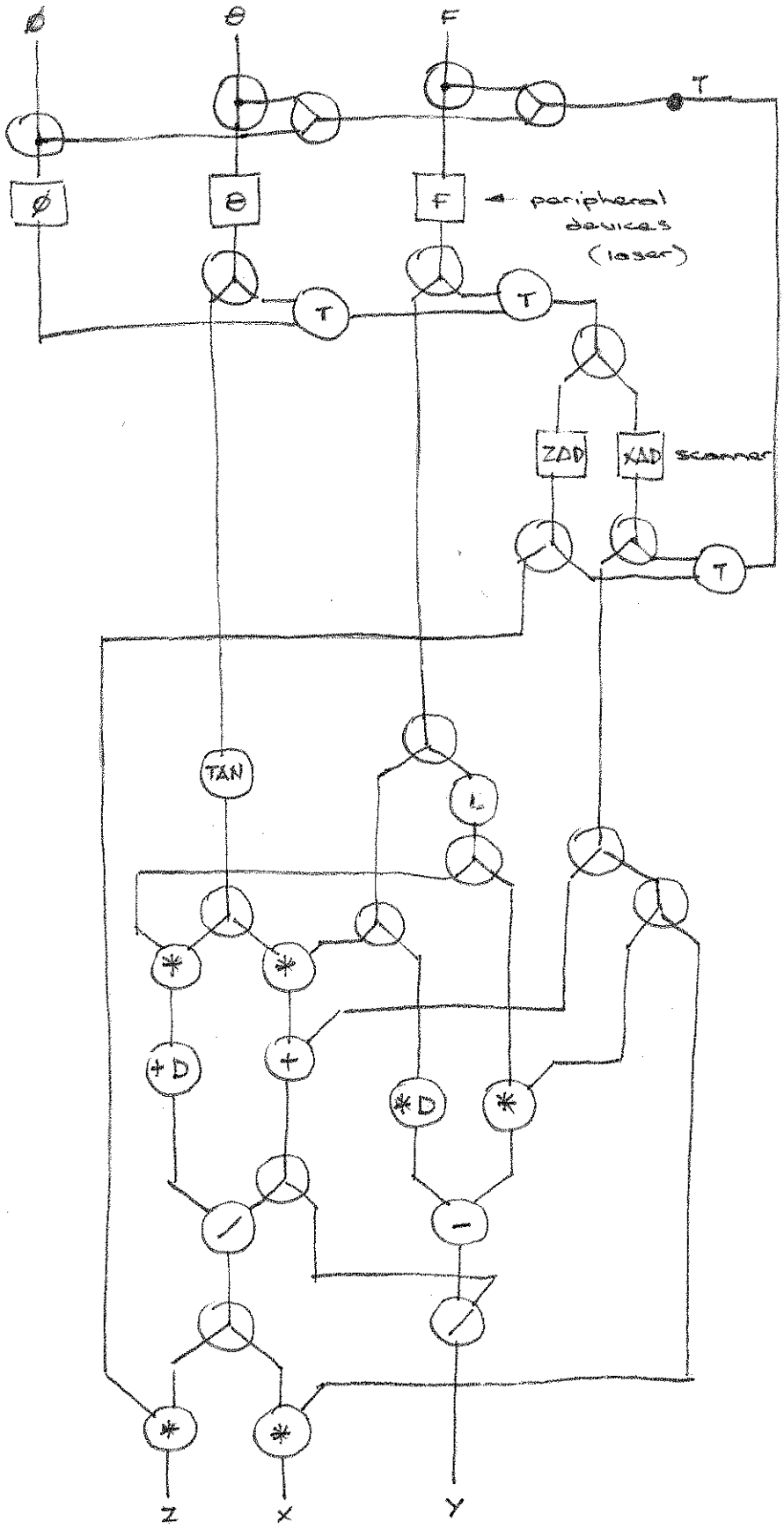
$$Y = (D.F - L.XAD)/(XAD + F.\tan(\theta))$$

$$Z = X.ZAD/XAD$$

Where:		
	X,Y,Z	laser spot co-ordinates
	XAD,ZAD	image scanner outputs
	θ,Ø	laser deflection angles
	L	y offset of scanner origin
	F	zoom lense coefficient
	D	x offset of laser origin

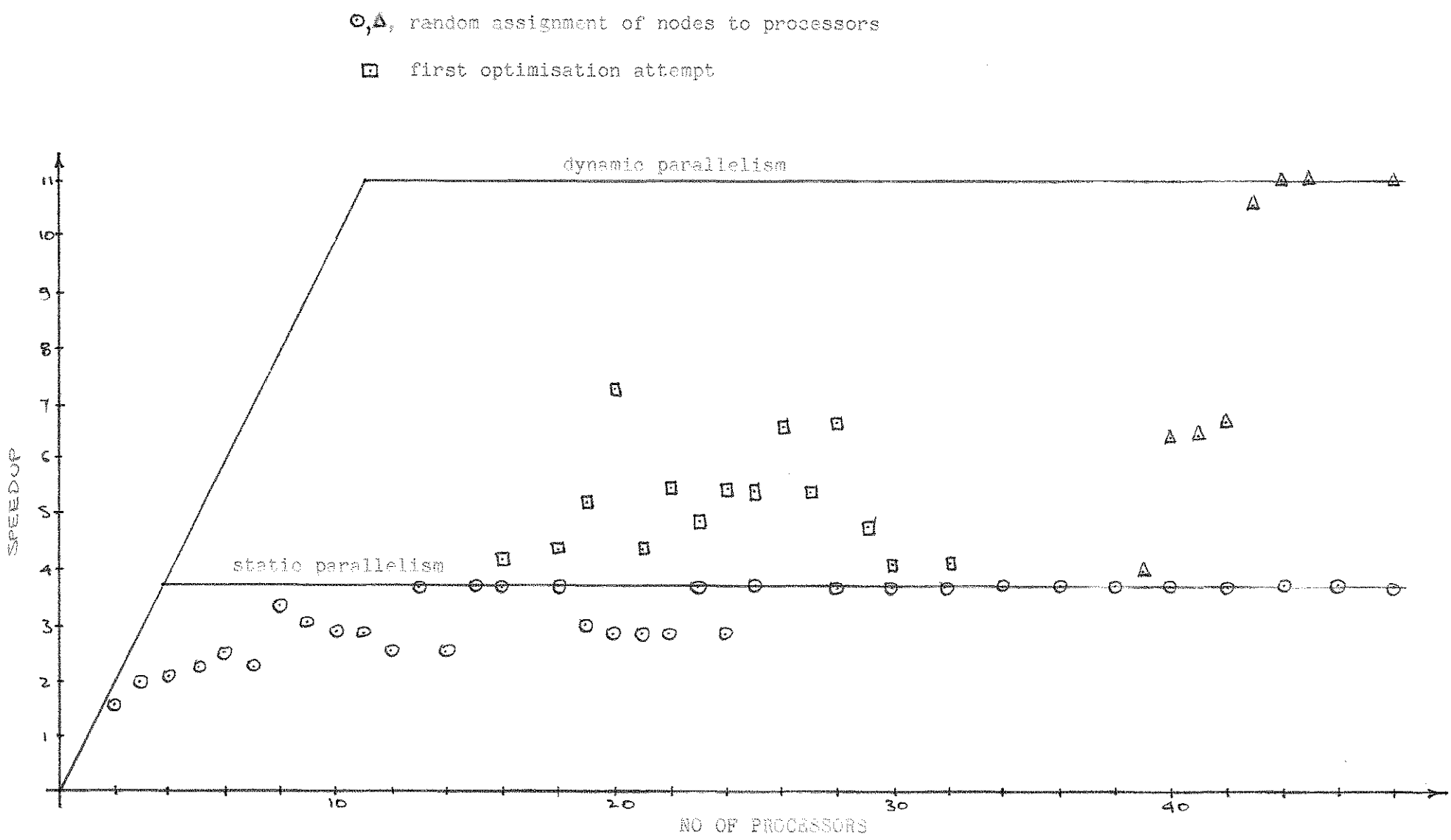
Laser Tracker Geometry

Figure 1.



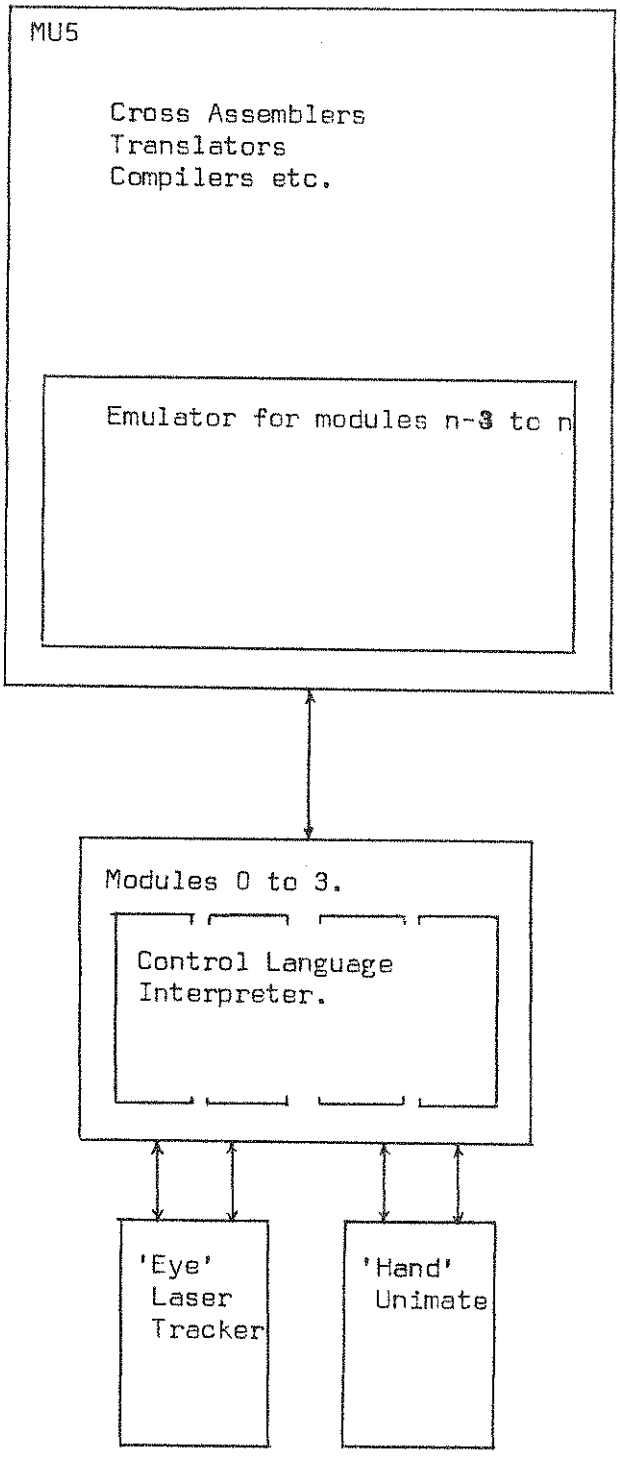
Laser Tracker Control Program

Figure 2



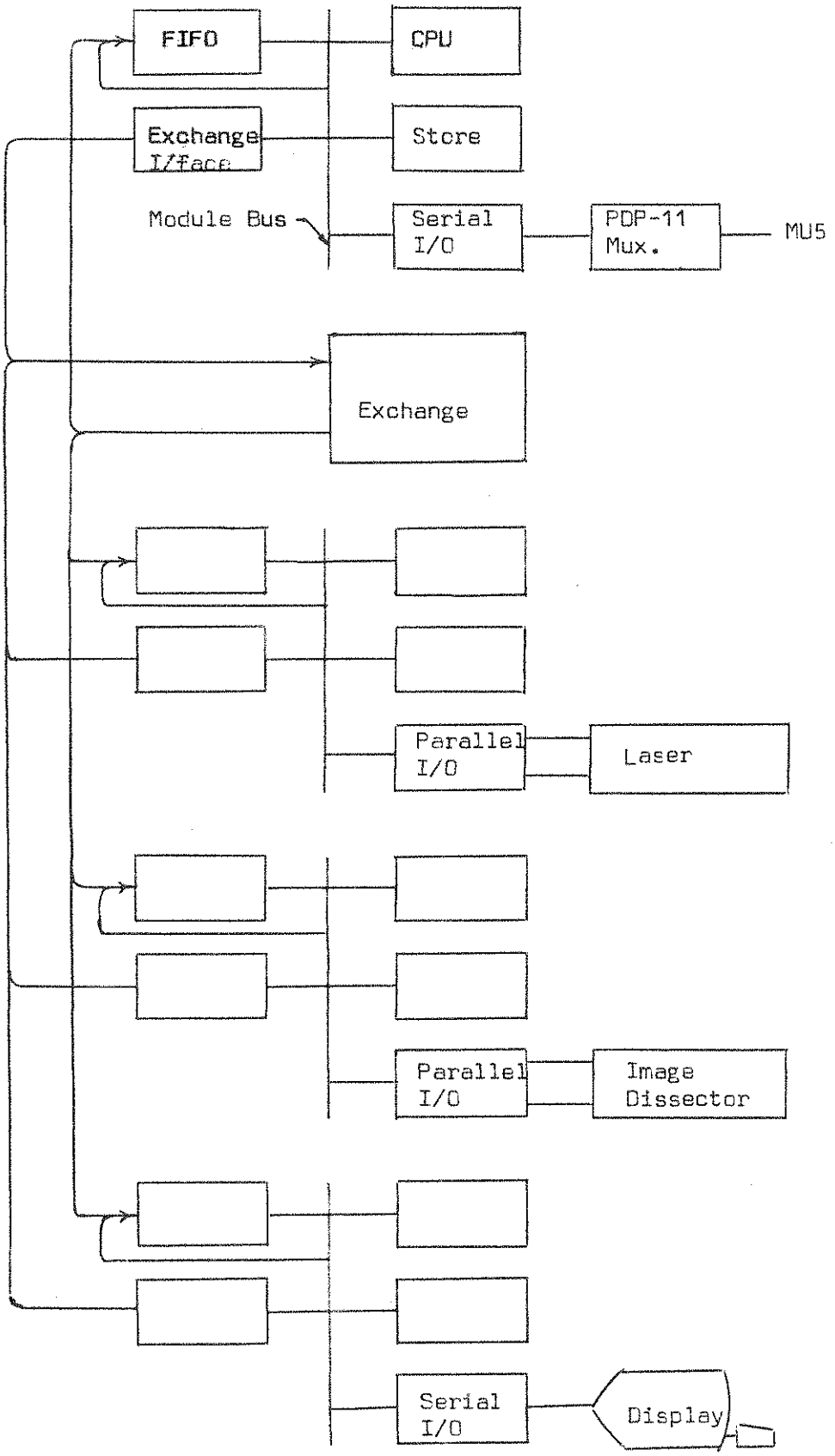
Parallelism in the Tracker Control Program

Figure 3



Organisation of the Experimental System

Figure 4



System Architecture

Figure 5

```
[ define [[
          [laser : [real; real; real] -> stream (real; real; real);
            ^[ [θ; F; Ø];
               [constant [[ [D; some value];[L;some value] ]];
                define
                  [[
                     [outscan : stream (real; real) ;
                           readscanner [θ; F; Ø]
                     ];
                     [XAD : real ; head [ outscan ] ];
                     [ZAD : real ; tail [ outscan ] ];
                     [X : real ; [L*tan[θ]+D*XAD/(F*tan[θ]+XAD)]];
                     [Y : real ; [D*F-(L*XAD)/(F*tan[θ]+XAD)]];
                     [Z : real ; [X * ZAD / ZAD]]
                  ]];

                bracket [ X; Y; Z]
               ]
            ]
          ]
         ]]
]
```

Notes

1) "Readscanner" is assumed to be a primitive function,

2) "Tan" is a system function;

3) Functions may only return one result;

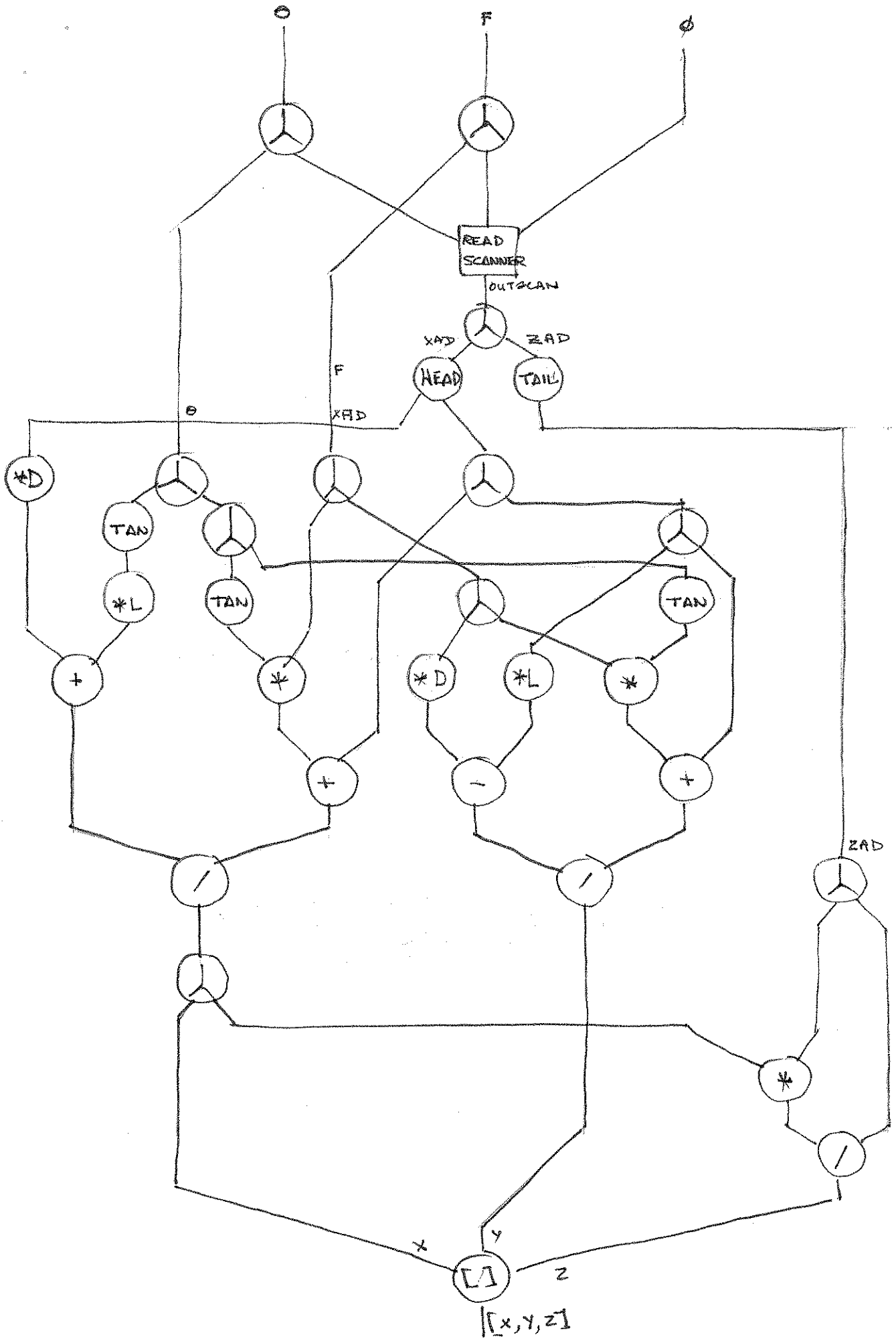
4) "Bracket" forms its arguments into a stream;

5) "Define" binds a name to an expression;

6) "Constant" binds a name to a value; the values of D and L are fixed for a particular laser tracker;

7) Arithmetic expresions are evaluated from left to right; precedence may be forced using brackets;

A NEWSPEAK function that controls the laser tracker
Figure 6



Hand-compiled Code for the Program of Figure 6

Figure 7

```
readscan : stream(real, real, real) -> stream(real, real, boolean)
    λ[ [0, ∅, F];
                    : boolean
        [define [[  [acc& writelaw [0] &
                    writelaw [∅] &
                    writelaw [F]
                ]
            ]];
    ]
    bracket [ XAD[acc]; ZAD[acc]; TRUE]
];

compute : real :5 -> stream (real : 3);

    λ[ [0, ∅, F, XAD, ZAD];

        [define [[ [x : real ; [L * tan [0] + D * XAD / (F * tan [0]
                                                    + XAD)];
                   [y : real ; [D * F - (L * XAD) / (F * tan [0] +
                                                    XAD)];
                   [z : real ; [X * ZAD / ZAD]
                ]];

        bracket [x, Y, z]
    ]
]
```

```
laser : stream[stream(real, real, num)] -> stream[stream(real, real, num)]
  λ [ [input] ] ;
  [null [ input]  => NIL;
    otherwise => [ define [[ [first : st (real, real, boolean);
                               readscanner [hh [input]; true]
                             ]
                          ]];
                   cons [ compute [ hh [input];
                                    hth [input];
                                    tth [input];
                                    h [first];
                                    ht [first]
                                  ];
                          laser2 [ t [input], tt [first] ]
                        ]
                 ]
  ],
]

laser2 : [stream [stream (real, real, real)] ; boolean ] => stream [stream (real, real, real)]
  , λ [ [input1, sync] ]
  [null [input] => NIL;
    otherwise => [ define [[ [outscan : stream (real, real, boolean);
                               , readscanner [ h [input]; sync]
                             ]];
                   cons [ compute [ hh [input];
                                    ht [ - ];
                                    tth [ :: ];
                                    h [outscan];
                                    t [outscan];
                                  ];
                          laser2 [ t [input], tt [outscan] ]
                        ]
                 ]
  ].
```