PROGRAMMING DECENTRALISED COMPUTERS

Isabel Gouveia Lima

Computing Laboratory, University of Newcastle upon Tyne

Ph.D. Thesis 1984

ABSTRACT

Programming, for the past thirty years, has been based on the sequential von Neumann computer model. However, there is a growing need to program decentralised computer systems ranging from mainframe computers that are geographically distributed, to miniature microcomputers on a single VLSI chip. Various pairings of programming languages and decentralised computers are being investigated: procedural languages with control flow, single-assignment languages with data flow, applicative languages with reduction, object-oriented languages with actor, and finally predicate logic languages with logic architectures.

This thesis investigates the programming of decentralised computers and consists of five major parts. Part 1 looks at four images of "future" computer systems: Fifth Generation Computers, Supercomputers, VLSI Processor Architectures and Integrated Communications & Computers. The former two images are "parallel machines", supporting logic and data flow programming, respectively. The latter two images are "decentralised computers" supporting control flow programming. Part 1 concludes that a "decentralised computer" image is most appropriate for future computers. Part 2 classifies and analyses the major categories of programming styles (i.e. procedural, object-oriented, functional, and logic). The analysis uses a Quicksort algorithm to contrast representative languages (i.e. PASCAL, MODULA-2, OCCAM, SMALLTALK, ID, FP, SASL, PROLOG and VISICALC) and relate their strength and weaknesses to their underlying computational mechanisms. It concludes that control flow (and procedural programming) is the most primitive and fundamental programming model.

Part 3 proposes the decentralised control flow programming model for future decentralised computers, which embodies the concepts underlying modern operating systems such as UNIX, and is a generalisation of the traditional sequential von Neumann control flow model. Part 4 presents two programming languages based on this model, called BASIX_1 and BASIX_2, which have been designed and implemented. The language BASIX_2 is analysed and assessed using two example applications, namely a banking system and an expert system. Part 5 presents the BASAL programming language, supporting a primitive form of decentralised control flow. For the assessment of the BASAL language two parallel sorts were chosen, using shared memory and message passing respectively. BASIX and BASAL are vehicles for exploring the decentralised control flow programming style, rather than new languages being proposed.

ACKNOWLEDGEMENTS

The author wishes to acknowledge:

- Dr. J. Eve, for supervising and encouraging the work carried out for this thesis.
- Drs. T. Anderson and P.C. Treleaven for providing support and useful advice.
- Mr. D. Mundy, for collaboration on the design of the BASIX programming languages, for writing the BASIX_2 interpreter, and for helping with the testing of the application programs.
- Dr. C. Gerrard, Mr. R.P. Mahon, and Mr. M. Wardley, for making available equipment and their expertise.
- The SERC of Great Britain, for funding the Research Associate post held by the author, and for the use of computer facilities.

CONTENTS

CHAPTER 1 - INTRODUCTION

Since the introduction of the sequential (von Neumann) control flow computer thirty years ago, the traditional control flow programming model has changed little. But, by the end of the past decade, a trend towards distributed information processing started to take shape and nowadays more and more emphasis is being given to systems that communicate and co-operatively process information - i.e. decentralised computers. This spectrum of decentralised computers ranges from mainframe computers that are geographically distributed, to miniature microcomputers on a single VLSI chip. To allow all these computer systems to be programmed, to co-operate in the communication of information and in the execution of a program, it is necessary for them to conform to a common decentralised programming model.

The various categories of programming models and associated languages that are being put forward as successors to the von Neumann control flow model are shown in Figure 1.1. They include control flow models and procedural languages, data flow models and single-assignment languages, reduction models and applicative languages, actor models and object-oriented languages, and logic models and predicate logic languages. Firstly, in a control flow model explicit flow(s) of control cause the execution of commands. In a procedural language (e.g. BASIC, FORTRAN) the basic concepts are: a global memory of cells, assignment as the basic action, and (sequential) control structures for the execution of statements.

Secondly, in a data flow model the availability of input operands triggers the execution of the command which consumes the inputs. In a single-assignment language (e.g. ID [5,6], LUCID [8], VAL [2], VALID [3]) the basic concepts are: data "flows" from one statement to another, execution of statements is data driven, and identifiers obey the so-called single-assignment rule.

Programming languages	procedural	single-assignment	applicative	object-oriented	predicate logic
Programming models	control flow	data flow	reduction	actor	logic

Figure 1.1: Categories of Programming Languages and Models

Thirdly, there are reduction models and applicative languages. In a reduction model [11] the requirement for a result triggers the execution of the command that will generate the value. In an applicative language (e.g. Pure LISP, SASL [55], FP [9]) the basic concepts are: application of functions to structures, and all structures are expressions in the mathematical sense.

Fourthly, there are the actor models and object-oriented languages. In an actor model the arrival of a message for a command causes the command to execute. In an object-oriented language (e.g. SMALLTALK [33]) the basic concepts are: objects are viewed as active, they may contain state, and objects communicate by sending messages.

Lastly, there are logic models and predicate logic languages. In a logic model a command is executed when it matches a target pattern and parallelism or backtracking is used to execute alternatives to the command. In a predicate logic language (e.g. PROLOG [23,36]) the basic concepts are: statements are relations of a restricted form, and execution is a suitably controlled logical deduction from the statements.

With the exception of control flow, the above categories of programming models and languages each propose a "revolutionary" approach to the succession of the traditional programming model. The actual choice of a successor to the von Neumann control flow model is made difficult by obstacles in measuring the benefits and drawbacks of any of these novel programming models (and their corresponding languages). Compared to control flow, each has a "higher-level" model of computation possibly constraining the range of algorithms the model can efficiently support.

There exists, though, an alternative, "evolutionary" approach, in the form of a decentralised control flow programming model. It involves the extension of control flow for parallel and distributed programming allowing the interconnection of heterogeneous processors in a decentralised system. An advantage of this approach is that it does not imply discarding the massive investment in traditional control flow computing.

This thesis investigates the programming of decentralised computers. It consists of five main parts: (i) an overview of decentralised computer systems, (ii) the classification and analysis of the various major programming styles, (iii) the description of the decentralised control flow model, (iv) the presentation of the BASIX language embodying this model, and (v) the presentation of the BASAL language, which embodies a primitive form of decentralised control flow.

Chapter 2 is an overview of decentralised computer systems, looking at four images of "future" computers: Fifth Generation Computers, Supercomputers, VLSI Processor Architecture and Integrated Communications & Computers. The former two images are "parallel machines", supporting logic and data flow programming, respectively. The latter two images are "decentralised computers" supporting control flow programming. It concludes that a decentralised computer image, capable of spanning heterogeneous processors, is most appropriate for future decentralised computers.

Chapter 3 presents the major styles of programming (i.e. procedural, object-oriented, functional, logic and application) that may be important in the future. This Chapter also classifies and analyses these programming styles, based on the computational mechanisms that underlie the corresponding programming models. The Quicksort algorithm is presented to be used as a common example.

Chapter 4 classifies and analyses Procedural and Object-Oriented programming styles, and Chapter 5 classifies and analyses Functional and Logic Programming styles, as well as Application Programming. The analysis is illustrated by programming languages that represent the individual characteristics of these programming styles. The semantics and syntaxes of these languages are described and their similarities and differences highlighted. The Quicksort algorithm is used as a common example, coded in the languages being described. The conclusion reached is that control flow, and procedural programming, is the most primitive and fundamental programming model.

Chapter 6 presents the decentralised control flow model of programming, which embodies the concepts underlying modern operating systems such as UNIX, and is a generalisation of the traditional sequential von Neumann control flow model. It is believed that this programming model should form the basis of future decentralised computer systems and their corresponding programming languages. The following chapters examine two languages BASIX and BASAL, which are meant to illustrate the style of decentralised control flow programming languages, not to propose new languages. Both these languages are primitive and are "low-level" system programming languages (cf. C) rather than high-level languages like PROLOG.

Chapter 7 describes the BASIX programming language, which embodies the decentralised control flow model. The BASIX language is based on the fundamental concepts of UNIX and LISP, and on the simplicity of BASIC's syntax. It is intended as a "total system", providing a complete, interactive, programming environment (cf. SMALLTALK, VISICALC, OCCAM, etc.). BASIX has implicit and explicit parallelism, in the sense that a new user command is executed as a parallel process, and that the user can specify the parallel execution of commands. Two versions of the BASIX language - BASIX_1 and BASIX_2 - are described. Chapter 8 is an analysis and assessment of the BASIX languages, and describes two application examples written in BASIX_2: a banking system and an expert system. These examples show the relationship of the language with the decentralised control flow model.

Chapter 9 describes the BASAL language, which was designed for programming a specific Multi-Microcomputer systems, based on a primitive form of decentralised control flow model. It extends the BASIC language in the sense that it allows parallelism and decentralised addressing. Chapter 10 is an analysis of BASAL, and describes two application examples: two parallel sorts, using shared memory and message passing respectively.

Lastly, Chapter 11, besides presenting the conclusions drawn from this work, describes the future work to be done in terms of the classification of programming styles, the decentralised control flow model, plus the BASIX and the BASAL programming languages.

CHAPTER 2 - DECENTRALISED COMPUTER SYSTEMS

This chapter examines the various possible organisations for "future" computer systems [54].

2.1. MOTIVATIONS

Many factors support the adoption of a radically new generation of general-purpose computers. Firstly, computing is moving from a sequential, centralised world to a parallel, decentralised world in which large numbers of computers are to be programmed to work together in computing systems. Secondly, the handling of non-numerical data such as sentences, symbols, speech, graphics and images is becoming increasingly important. Thirdly, the processing tasks performed by computers are becoming more "intelligent", moving from scientific calculations and data processing, to artificial intelligence applications. Lastly, today's computers are still based on the thirty-year-old von Neumann architecture; essentially all that has happened is that the software systems have been repeatedly extended to cope with increasingly sophisticated applications.

Important technological and social factors must also be considered. In technology, various separate areas of computing research are on the threshold of major advances [57]:

- <u>artificial intelligence</u> - methodologies to express "knowledge" and to infer from this knowledge, as seen in expert systems; and human-oriented input-output in natural languages, speech and pictures.

- <u>software engineering</u> - new higher level programming languages and computational models; and programming environments building upon systems such as the UNIX system.

- <u>computer architectures</u> - distributed architectures supporting computer networks; parallel architectures giving high-speed computers for numerical calculations; and VLSI architectures to make full use of the potential of VLSI technology.

- <u>VLSI technology</u> - VLSI computer aided design systems including new methods for semi-automatic design of logic circuits; and new devices such as those using Gallium Arsenide and Josephson Junctions.

For social factors notice: the evolution of computing from scientific applications in the 1950's, to include commercial and industrial applications in the 60's and 70's, and into consumer usage in the 80's and 90's. At the opposite ends of this application spectrum are the so-called Supercomputers and Fifth Generation Computers. Supercomputers handle the high-performance numerical applications. Fifth Generation Computers form the cornerstone of so-called intelligent consumer electronics - sophisticated televisions, video recorders, learning aids etc. - the next generation of wealth-creating consumer products [46]. The implication of all these factors taken together is that von Neumann (control flow) computers, originally designed in the 1950's for sequential computing, are no longer adequate for computation and that for the future a radical change in the computational concepts underlying computers is required.

There are at least four major areas of research involved in attempting to identify a future generation of computers, namely the investigation of:

1. Fifth Generation Computers which embody "knowledge" bases and support problem-solving and inference functions;

2. Supercomputers that utilise parallelism and support novel (very high level) forms of programming;

3. VLSI Processor Architectures, system architectures specifically aimed at exploiting very large scale integration through new VLSI components, encompassing both general-purpose and special-purpose processors.

4. Integrated Communications & Computers representing the fusion of wide-area networks, local area networks, and parallel computer architectures;

Any one of these four research areas could provide the new generation of computers. But each area's view of these computers seems very different, thus significantly affecting the style of future systems. The examination of these areas, and their likely impact, starts by presenting images of Fifth Generation Computers and of Supercomputers.

2.2. FIFTH GENERATION COMPUTERS

Fifth Generation Computers are knowledge processing systems designed to support knowledge-based expert systems [22,60]. Expert systems embody modules of organised knowledge concerning specific areas of human expertise. They also support sophisticated problem-solving and inference functions, for the purpose of rendering to users intelligent advice on one or other specialised topics. Future expert systems will also provide human-oriented input-output in the form of natural languages, speech and picture images. For example, an expert system for medical diagnosis could operate in a way analogous to the way a physician, a surgeon and a patient interact and use their knowledge to make a diagnosis.

In expert systems "knowledge" is often represented in terms of IF-THEN rules of the form [20]:

```
IF    condition_1  and
      condition_2  and
       . . .
      condition_n

THEN  implication  (with significance)
```

where if all the conditions are true then the implication is true, with an associated local significance factor. During the search of a set of rules an overall significance factor is maintained and when this significance becomes unacceptably low, then this search is abandoned and a new set of rules is searched.

As observed by Japanese researchers [39,53], this structure of expert systems is most closely matched by the structure of logic programming (its computational model). In a logic programming language, such as PROLOG, statements are relations of a restricted form called "clauses", and the execution of such a program is a suitably controlled logical deduction from the clauses forming the program. The following program [23] consists of four clauses:

```
father(bill, john).
father(john, tom).

grandfather(X, Z)  :-  father(X, Y), mother(Y, Z).
grandfather(X, Z)  :-  father(X, Y), father(Y, Z).
```

Figure 2.1: Logic Program for "family tree"

The first two clauses define that bill is the father of john, and john is the father of tom. The second two clauses use the variables X,Y,Z to express the rule that X is the grandfather of Z, if X is the father of Y and Y is either the mother or father of Z. Such a program can be asked a range of questions, from is john the father of tom: "father(john, tom)?" to is there any A who is the grandfather of any C: "grandfather(A, C)?".

The possible operation of a computer based on logic is illustrated below, using the program in Figure 2.1. Execution of, for example, "grandfather(bill, R)?" will match with each "grandfather( )" clause:

```
grandfather(X=bill, Z=R) :- father(bill, Y), mother(Y, R).

grandfather(X=bill, Z=R) :- father(bill, Y), father(Y, R).
```

both of which will attempt in parallel to satisfy their goals (called OR-parallelism). The first clause will fail, being unable to satisfy the "mother( )" goal from the program. The second clause has two goals "father( ), father( )" which it attempts to solve in parallel (called AND-parallelism). This involves pattern matching and substitution, to satisfy both the individual goals:

```
 grandfather(X=bill, Z=R) :- father(bill, Y), father(Y, R).
                          :- father(bill, Y=john), father(Y=bill, R=john).
```

and the overall consistency:

```
                          :- father(bill, Y=john), father(Y=john, R=tom).
```

Having illustrated the symbol manipulation operation of a computer based on logic, the possible organisation of such computers will be examined next. One possibility is a highly microprogrammed (control flow based) PROLOG machine [58,59], analogous to current LISP machines. Although a number of such designs can be expected in the near future, PROLOG machines are not true logic machines, just as LISP machines are not considered reduction machines. A logic organisation for Fifth Generation Computers is proposed in the Japanese FGCS Project plans [39]. Here Fifth Generation Computers are viewed as comprising three component machines, as shown in Figure 2.2. These machines though serving specialised roles will be linked by a common logic machine language and architecture.

```
 ----------------------------------------------------
|                                                    |
|              Logic Machine Language                |
|         ------------------------------             |
|        |                |              |            |
|  ---------------   ---------------   -------------  |
| |knowledge base|  |problem-solving| |"intelligent"| |
| |machine       |  |and inference  | |interface    | |
| |              |  |machine        | |machine      | |
|  ---------------   ---------------   -------------  |
 ----------------------------------------------------

  cf. filestore     cf. central       cf. input-output
  plus databases    processing unit   devices
```

Figure 2.2: Fifth Generation Computer

In summary, a Fifth Generation Computer is viewed as a parallel logic architecture supporting knowledge-based systems applications.

2.3. SUPERCOMPUTERS

Supercomputers are aimed at large-scale numerical calculations and attempt to achieve a high performance by exploiting parallelism. They may be envisaged as parallel "mainframe" (cf. CRAY 1) computers built from identical, powerful processors whose instruction execution is based on a concurrent alternative to the traditional sequential control flow architecture. For Supercomputers, the most prominent category of parallel architecture is data flow. In a data flow computer instruction execution is data driven; the availability of input operands triggers the execution of the instruction which consumes the inputs. The most important properties of data flow are that instructions pass their results directly to all the consuming instructions and that an instruction is executed when it has received all its inputs - properties that influence the general-purpose nature of data flow.

Data flow computers are most naturally programmed in a very high-level form of programming called single-assignment languages [1]. Single-assignment languages are based on a rule stating: a variable may appear on the left-hand side of only one statement in a program fragment. This allows the data dependencies in a program to be easily detectable and so statements may be specified in any order. As an illustration of single-assignment programming a procedure in ID [6] for inner-product (ai * bi) will be examined:

```
procedure inner-product (a, b, n)
   ( initial s <- 0
     for i from 1 to n do
          new s <- s + (a[i]*b[i])
     return s )
```

Figure 2.3: Single-Assignment Program for "inner-product"

This procedure takes as input two arrays "a" and "b", both of length "n", and returns their inner-product "s". These statements have the following interpretation:

```
s0 <- 0

s1 <- s0 + (a[1] * b[1])
s2 <- s1 + (a[2] * b[2])

     . . .

sn <- sn-1 + (a[n] * b[n])

return sn
```

and hence obey the single-assignment rule. Since execution is driven by the availability of data, all the multiplications can execute in parallel, after which the tree of partial results will be summed to produce the result "sn".

The possible operation of a computer based on data flow is illustrated by Figure 2.4, which represents the machine instructions corresponding to the inner-product example. In Figure 2.4 each data flow instruction consists of an operator, two input operands which are either literals or required data tokens, and a reference such as "i1/2" defining a consumer instruction "i1" and argument position "2" for the result data token. Data tokens are used to pass data from one instruction to another and they are also used to cause the execution of instructions. An instruction is enabled for execution when all its input arguments are available, i.e. when all its data tokens have arrived. The operator then consumes the data tokens, performs the required operation, and using the embedded reference stores a copy of the result data token into the consumer instruction(s).

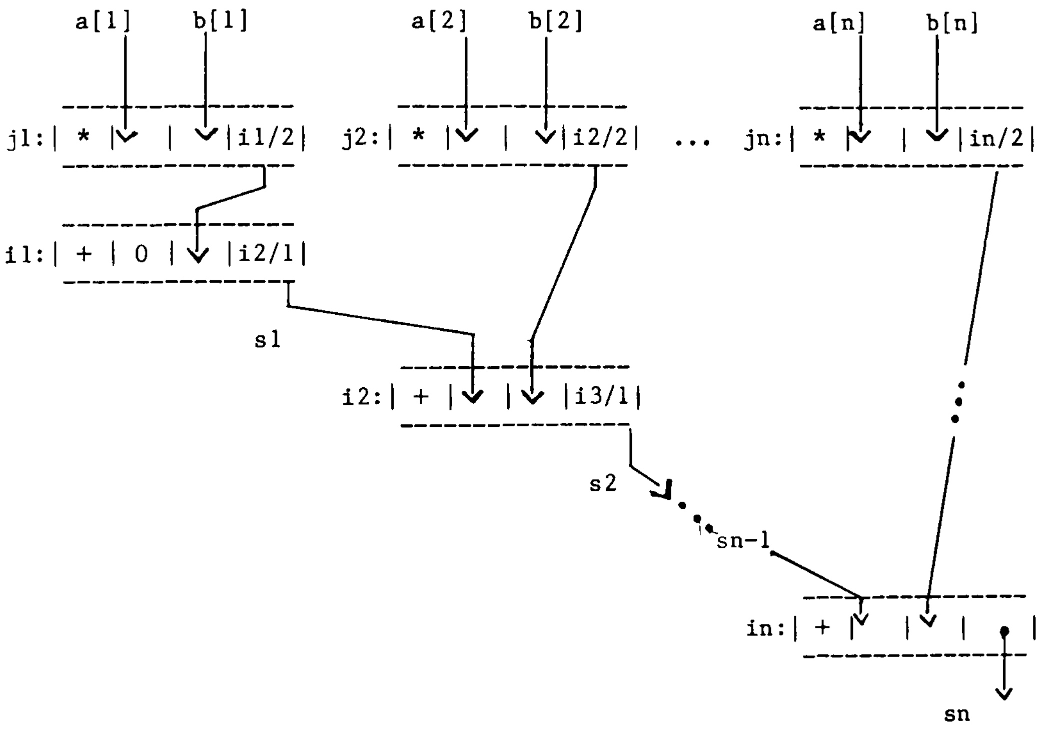


Figure 2.4: Data Flow Program for "inner-product"

Data flow computers are usually based on a packet communication machine organisation [18,50]. This organisation consists of a circular instruction execution pipeline of resources in which processors, communications and memories are interspersed with "pools of work", as shown in Figure 2.5.

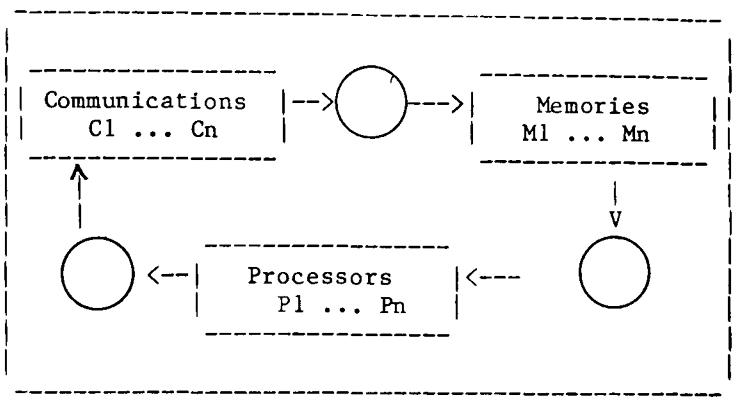


Figure 2.5: Packet Communication Computer

The organisation views an executing program as a number of independent information packets all of which are conceptually active, which split and merge. For a parallel computer, packet communication is a very simple strategy for allocating packets of work to resources. Each packet to be processed is placed with similar packets in one of the "pools of work". When a resource becomes idle it takes a packet from its input pool, processes it and places a modified packet in an output pool, returning then to the idle state. A number of data flow machines, based on packet communication, are already operational [50].

In summary, a future Supercomputer is viewed as a parallel (data flow) machine, supporting large-scale numerical calculations.

2.4. PARALLEL MACHINES

The above two areas of research, namely Fifth Generation Computers and Supercomputers, each view future computers as parallel machines, supporting a single form of very high level programming. In the former case, based on logic computation and in the latter, data flow computation.

There are, however, five main basic categories of computer architecture on which future computers could be based, as discussed in the Introduction (see Figure 1.1). They range from "low level" architectures, such as control flow, that specify <u>how</u> a computation is to be executed, to "high level" architectures, such as logic, that merely specify <u>what</u> is required. Associated with each category of computer architecture, is a corresponding category of programming languages.

Recall, these are: control flow computers and procedural languages, data flow computers and single-assignment languages; reduction computers and applicative languages; actor computers and object-oriented languages, and finally, logic computers and predicate logic languages.

For future computers, since each model (i.e. control flow, data flow, reduction, actor, logic) efficiently supports only a single corresponding programming style (i.e. procedural, single-assignment, applicative, object-oriented, predicate logic), a number of questions are raised:

1. which programming model has the most general-purpose computational concepts (e.g. is best able to support the other models)?

2. what programming style will be dominant in the 1990's (or will a number of styles be in use)?

3. can Fifth Generation Computers, Supercomputers, and von Neumann computers be based on different computational concepts (since it would seem essential for them to work together in the future)?

4. is it realistic to expect that the massive investment in traditional control flow computing will be discarded?

Since it seems naive to imagine that control flow computers will simply disappear (one only has to look at the longevity of FORTRAN), future computers may either continue to be dominated by control flow or will encompass various architectures.

Thus the most urgent challenge for future computers seems not to be the identification of the "parallel machine" but to identify the programming model for a "decentralised computer" that will allow dissimilar computers to work together. This is illustrated by Figure 2.6.

```
 ---------------------------------------------------------------
|                                                               |
|          DECENTRALISED  COMPUTER  ARCHITECTURE                |
|          ---------------------------------------------        |
|          |                  |                  |       . . .  |
|  --------------      ------------       ---------------       |
| | Supercomputer |   | von Neumann |   | Fifth Generation |    |
| |               |   | Computer    |   | Computer         |    |
|  --------------      ------------       ---------------       |
 ---------------------------------------------------------------
```

Figure 2.6: Future Decentralised Computer Architecture

Identification of such a "decentralised computer" programming model is assisted by examining the latter two areas of research listed in Section 2.1, namely VLSI Processor Architecture and Integrated Communications & Computers.

2.5. VLSI PROCESSOR ARCHITECTURES

Processor architectures to exploit very large scale integration (VLSI) are aimed at defining a new VLSI generation of components to succeed the conventional LSI microprocessor. Microprocessors containing over 100,000 transistors are starting to become commonplace. However, attempting to make larger-scale single processors in VLSI scaled to sub-micron dimensions becomes self-defeating, due to communication problems and the escalating costs of designing and testing such complex processors. One obvious solution (stimulated by the VLSI design philosophy of Mead and Conway [38]) is miniature microcomputers which can be replicated like memory cells and operate as a multiprocessor architecture. These novel general-purpose and special-purpose microcomputers are often implemented by only a few different types of simple cells, and use extensive pipelining and multiprocessing to achieve a high performance. Examples [52] range from special-purpose multiprocessors such as Kung's Systolic Arrays to general-purpose multiprocessors such as Caltech's Tree Machine built from 1024 identical chips.

For a semiconductor manufacturer to specify a new VLSI generation of components it is necessary to specify a system architecture defining communication and cooperation between both general-purpose and special-purpose microcomputers. The fundamental problem to be solved is how to orchestrate a single computation so that it can be distributed across the ensemble of processors [45]. Two elegant VLSI system architectures (the former special-purpose and the latter general-purpose) are: Kung's Programmable Systolic Chip [24,37], and INMOS' Transputer [10] and OCCAM programming language [49] based on communicating processes.

A more conventional approach is illustrated by the reduced instruction set multi-microcomputer system (RIMMS) [25,26] which is a network of primitive microcomputers. Figure 2.7 shows the organisation of RIMMS.

System

```
 -----------------------------------------------------------
|                                                           |
|          -----------------------------------------         |
|     1:|           2:|           3:|           4:|    . . .|
|   ---------    ---------    ---------    ---------        |
|  |simple   |  |simple   |  |simple   |  |simple   |       |
|  |processor|  |processor|  |processor|  |processor|       |
|  |---------|  |---------|  |---------|  |---------|       |
|  |256 word |  |256 word |  |256 word |  |256 word |       |
|  |memory   |  |memory   |  |memory   |  |memory   |       |
|   ---------    ---------    ---------    ---------        |
 -----------------------------------------------------------
```

Addressing

```
                  8 bit          8 bit
           --------------------------------
          | microcomputer | word in memory |
           --------------------------------
```

Registers

```
 C  -  Code pointer (program counter)
 D  -  Data pointer (base register)
```

Figure 2.7: Reduced Instruction Set Multi-Microcomputer System

The central idea in RIMMS, as illustrated by Figure 2.7, is that each microcomputer has its own 256 word local memory, but forms part of a global (two-level) address space. A microcomputer has a 16-bit word size, with each register, data element and address being 16 bits. Instructions, however, are 2 x 16 bits and use a 3-address format:

```
          M1  M2  M3        O1               O2               O3
 5 bits    1   1   1      8 bits           8 bits           8 bits

 ------------------------------------------------------------------------
|operator|mode bits|literal/address|literal/address|literal/address|
 ------------------------------------------------------------------------
                 0  -  literal
                 1  -  address (memory [D+ signed literal])
```

There are less than 20 operators. Each microcomputer in the multi-microcomputer system is addressable, and behaves as a combined memory and processor that is able to service load, store and execute operations. The design of the multi-microcomputer system centres around the 16-bit global address space. An address consists of two parts: the high 8 bits define a specific microcomputer, while the low 8 bits define a word in that microcomputer's memory. Although a microcomputer can access any word in the global address space, an attempt to execute alien code causes execution to transfer to the specified microcomputer. The processor implementation has a simple von Neumann data path, with the addition of the two-level address space and the FORK instruction for parallelism.

This design contains a number of key concepts. Firstly, although a microcomputer can make a data access to any word in the global address space, code is always executed by the local microcomputer. Secondly, a microcomputer has the ability, using a FORK instruction, to create a parallel flow of control in another (idle) microcomputer. Thirdly, a microcomputer executes a process to completion. (This atomic execution of local code removes many of the synchronisation problems typically found in control flow multi-processors.) Finally, to enable simple process migration, the amount of state information held in the processor's registers is minimised. This is achieved by a microcomputer using a three-address instruction format and having only two visible registers: the code pointer (i.e. program counter) "C" and the data pointer (i.e. base register) "D".

In summary, the aim of VLSI processors such as INMOS' Transputer, the Programmable Systolic Chip, and RIMMS is to define a system architecture for a new VLSI generation of general-purpose and special-purpose components.

2.6. INTEGRATED COMMUNICATIONS & COMPUTERS

Integrated data Communications & Computers represent the fusion of wide area computer networks, local area computer networks, and parallel computer architectures to form a fully integrated computer-communications network. Data communications and computers, specifically computer networks and parallel computers, have in the past developed independently from each other, with advances in both technologies being sustained by the rapid development of semiconductor devices. However, the importance of fully integrating the spectrum of decentralised systems shown in Figure 2.8 has long been advocated [35]. To achieve this, it is clearly necessary for all component computers to conform to a common decentralised system architecture (some harmonious interface) - allowing them to be programmed to cooperate in the communication of information and in the execution of a program.

Inter-computer distance	Computers located in		
1000 km	Continent	wide area network	Decentralised System Architecture
100 km	Country		
10 km	City		
1 km	Site	local area network	
100 m	Building		
10 m	Room		
1 m	Cabinet	parallel computer arch.	
100 mm	Circuit board		
1 mm	Chip		

Figure 2.8: Spectrum of Decentralised Systems

In these decentralised systems the most important related issues are communications and addressing of information, rather than parallelism and instruction execution. Thus the systems are usually based on control flow programming models enhanced with operating system concepts, as illustrated by the Newcastle Connection distributed UNIX system. The Newcastle Connection [16,47] is the name given to a novel software subsystem added to a set of standard UNIX systems [44] in order to connect them together as a distributed system, initially using just a single Cambridge Ring [61]. The resulting distributed system (which could employ a variety of wide and local area networks) is functionally indistinguishable at both the "Shell" command language level and at the system call level, from a conventional centralised UNIX system.

The secret of success of the Newcastle Connection is the hierarchical information and naming structure (for directories, files, devices, and commands) of UNIX. In the distributed system the structures of each component UNIX system are joined together as a single structure, in which each UNIX system behaves as a directory. This is illustrated by Figure 2.9.

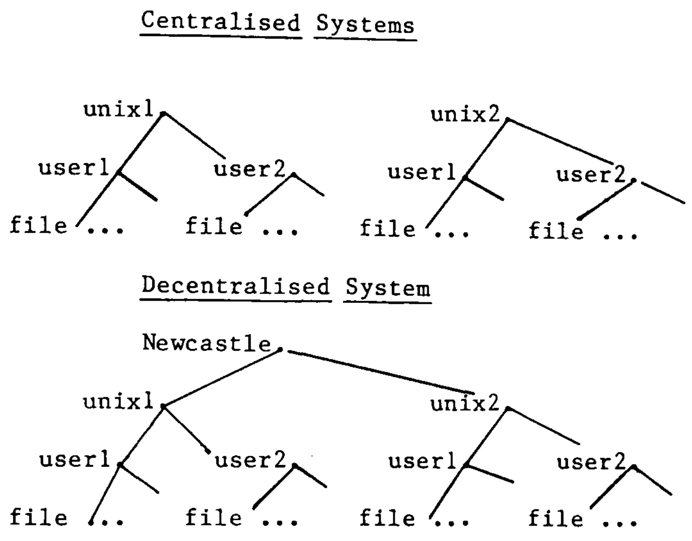


Figure 2.9: The Newcastle Connection of UNIXes

The result is that each user, on each UNIX system, can inspect any directory, read or write any file, use any device, or execute any command, regardless of on which physical system it belongs. For example if a user "user1" wishes to copy "cp" a file "file1" to another file "file2" on the same machine they type the command:

```
cp  file1  file2
```

and if file2 belongs to "user2" they type:

```
cp file1  /user2/file2
```

whereas on the decentralised system to copy the file "file1" to file "file2" of "user2" on machine "unix2" they type:

```
cp  file1  /../unix2/user2/file2
```

For those unfamiliar with UNIX, the initial "/" symbol indicates that a path name starts at the root directory, and the ".." symbol is used to indicate the parent directory. Perhaps the best analogy of the Newcastle Connection is with the naming structure of the international telephone network.

In summary, Integrated Communications & Computers is viewed as a decentralised computer representing the fusion of wide area network, local area networks and parallel computers.

2.7. DECENTRALISED COMPUTERS

Decentralised computers integrate distributed, parallel and sequential computers. Their system architecture defines a minimum set of principles that hardware and software components must obey so that they can be configured to work together in a system. It is also important for programming such systems, that these principles are mirrored in both the hardware and software just as FORTRAN and the von Neumann model embody the same principles.

A decentralised programming model provides a composite framework or image for future computers. This framework is even capable of spanning the four seemingly different views of: Fifth Generation Computers, Supercomputers, VLSI Processor Architectures, and Integrated Communications & Computers. For instance, future decentralised computers will be capable of specialisation, supporting a range of applications from the numerical calculations of Supercomputers to the symbol manipulation of Fifth Generation Computers. Programming languages will range from traditional procedural ones, to very high level languages such as PROLOG. Machine organisations will support concurrency, possibly utilising data driven and demand driven techniques. Implementations will employ the latest general-purpose and special-purpose VLSI technology. Systems will be highly decentralised at all levels with computers linked together in an integrated computer-communications network. An attempt to illustrate this is shown in Figure 2.10.

```
Wide Area Computer Networks
 ------------------------------------------------------------------------
   |                                |                              |
  ----                              |                             ----
                                    |
        Local Area Computer Networks|
      --------------------------------------------------------------
        |                              |                         |
       ---                             |                        ---
                                       |
                     Parallel Computers|
                     --------------------------------------
                       |            |              |
                  ------------  -----------   --------------
application:     |numerical   | |data       | |knowledge-based|  . . .
                 |calculations| |processing | |systems        |
                 |------------| |-----------| |---------------|
languages   :    |single-     | |procedural | |predicate      |
                 |assignment  | |           | |logic          |
                 |------------| |-----------| |---------------|
machine     :    |data        | |control    | |logic?         |
                 |flow?       | |flow       | |               |
                 |------------| |-----------| |---------------|
implementation:  | V L S I    | | V L S I   | | V L S I       |
                 |            | |           | |               |
                  ------------   -----------   ---------------

                 Supercomputer  von Neumann    Fifth Generation
                                Computer       Computer
```

Figure 2.10: Future Decentralised Computer Systems Architecture

Given the technological and social factors (discussed in Section 2.1 - Motivations) this "decentralised computer architecture" view of future computers would seem quite reasonable.

In conclusion, the three organisations for computers are: sequential computers, parallel computers, and decentralised computers (see Figure 2.11).

```
Sequential                                          von Neumann
                                                    Computer
     ----
    |proc|
    |----|
    |mem |
     ----
Parallel                                            Parallel
                                                    Computer
      -------------------
    | ----          ---- |
    ||proc|<=>...<=>|proc||
    ||----|         |----||
    ||mem |         |mem ||
    | ----          ---- |
      -------------------
Decentralised                                       Future
                                                    Computer
   ---------------------------------------------------------
  |   -------------------          -------------------      |
  | | ----          ---- |<=>...<=>| ----          ---- |   |
  | ||proc|<=>...<=>|proc||        ||proc|<=>...<=>|proc||  |
  | ||----|         |----||        ||----|         |----||  |
  | ||mem |         |mem ||        ||mem |         |mem ||  |
  | | ----          ---- |         | ----          ---- |   |
  |   -------------------          -------------------      |
   ---------------------------------------------------------
```

Figure 2.11: Contrasting von Neumann, Parallel and Future Computers

The von Neumann computer is sequential consisting of a single processor and memory. The parallel computer is, clearly, parallel being composed of sequential computers. Lastly, the future computer will be decentralised (distributed + parallel), and will consist of parallel and sequential computers. Therefore, as illustrated by Figure 2.11, a decentralised program model is capable of spanning distributed, parallel and sequential computers.

Thus for future computers a decentralised computer architecture is sought, analogous to that of the international telephone network, supporting the communication and cooperation of dissimilar hardware/software components. The essential properties are:

1. communication and cooperation of components

2. addressing of distributed information

3. extensible system of heterogeneous processors

4. many programming styles are supported

Properties 1-3 relate to system structuring and addressing issues, but property 4 relates to the choice of programming model (i.e. control flow, data flow, reduction, actor, logic)

Recall, in the discussion on Parallel Machines (Section 2.4), it was argued that future decentralised computers will either continue to be dominated by control flow or will encompass various programming models. In the next three chapters, this choice is discussed, classifying and presenting the major programming styles and analysing their advantages and disadvantages for computation.

CHAPTER 3 - CLASSIFICATION OF PROGRAMMING LANGUAGES

This chapter attempts to classify some major styles of programming that might become important in the future.

3.1. VERY HIGH LEVEL LANGUAGE PROGRAMMING

Computing is currently experiencing a veritable explosion of research into very high level programming notations. These include: procedural languages that aim to provide more effective programming environments such as ADA [43]; new languages based on novel models of computation such as PROLOG [36]; and application-oriented languages such as VISICALC [14] used for financial-modelling. In fact, in this latter area it is difficult to draw the boundary between application languages and packages, since today's packages may well be the programming languages of tomorrow.

How can styles of high level programming be classified so as to make useful observations about their advantages and disadvantages for computation? One approach is to group them by application area - thus having string processing languages, numerical languages, artificial intelligence languages and so forth. For the present purposes, however, this is not the best approach; firstly because there are so many potential application areas, and secondly because some of the most striking differences between programming styles and languages have absolutely nothing to do with the advertised differences in their application areas. For example, FORTRAN and APL are both "numerical" languages yet the differences between them are significant. These differences are closely bound up with different approaches to certain very basic questions such as how data is communicated in a program and the control structures supported.

Since new programming languages often try to present a model of computation that closely represents the underlying machine architecture, this discussion of programming will be strongly influenced by programming models (as shown in Figure 1.1). Figure 3.1 illustrates the various categories of programming and example languages, some of which will be examined. In the future any of these categories of programming may become "mainstream" programming styles, especially when novel decentralised computers (as discussed in the previous chapter) sympathetic to their support become available for use. Although most of these languages are termed general-purpose they rarely prove to be equally applicable to all classes of problems. It is therefore important to understand the strengths and weaknesses (and hence the potential for applications) of each category of programming.

Category	Examples
Procedural Programming	
conventional	BASIC, FORTRAN, PASCAL
concurrent	
shared memory	Concurrent PASCAL, MODULA
message passing	CSP, OCCAM
Object-Oriented Programming	SMALLTALK, ACT1
Functional Programming	
data flow	ID, LUCID, VAL, VALID
applicative	
function-level	FP
pattern-matching	PURE LISP, SASL, HOPE
Logic Programming	
Horn clauses	PROLOG
predicate logic	SETL
Application Programming	
"Electronic-sheet"	VISICALC

Figure 3.1: Categories of Programming

There are at least five major categories of programming: Procedural, Object-Oriented, Functional, Logic and Application programming. This examination of the various categories of programming is started by discussing the most dominant, procedural programming.

In procedural programming there are concepts which are almost taken for granted: a global memory of cells, assignment as the basic action, and implicitly sequential control structures for the execution of statements. In procedural programming there are two sub-classes of languages, namely the conventional sequential languages, and what is called concurrent languages (ex. Concurrent PASCAL) that have parallel control structures [48]. Most users of computers know of only one class of programming languages, what has therefore been called conventional languages. This class has developed for programming the traditional von

Neumann stored program computer. Hence the semantics of conventional languages reflect the von Neumann programming model: global memory, fixed-size memory cells, assignment and sequential execution.

Concurrent languages [15,48] extend this control flow programming model with parallel control structures based on processes, plus communication and synchronisation mechanisms. A process is an independent program consisting of a private data structure and sequential code that can operate on the data. Concurrently executing processes cannot operate on the private data of one another; they can only interact using the communication mechanism. The communication mechanism is the way processes communicate data among themselves. The most commonly employed mechanisms are: unprotected shared (global) memory, shared memory protected by modules or monitors, message passing and the rendezvous [48]. The synchronisation mechanism is the way processes enforce sequencing restrictions among themselves. The commonly employed mechanisms include: signals, synchronised sending, buffers, path expressions, events, conditions, queues and guarded regions etc. [48]. Other important distinguishing features of concurrent programming languages include: process creation, whether processes are created "statically" during compilation or "dynamically" at runtime from the execution of the calls; process topology, where the interconnection links either remain static during execution or may dynamically change; process scheduling, defining how processes are assigned to the processors; and process termination, the condition when a process has finished execution and can be deleted. Concurrent languages can be broadly classified, by the nature of their communication mechanism into: shared memory and message passing.

In object-oriented programming, computation is based upon active objects, sometimes called actors, which communicate by passing messages. Every object belongs to a <u>class</u> and is created as an <u>instance</u> of that class. The class defines the detailed representation of its instances, the messages to which they can respond, and the methods for computing the appropriate responses. Stored in an instance are the particular set of values that define its state. Ingalls [33] uses the following example to distinguish between object-oriented programming and procedural programming.

> to evaluate **<some object> + 4** means to present **+ 4** as a message to the object. The fundamental difference is that the object is in control, not the **+.** If **<some object>** is the integer **3** then the result will be the integer **7.** However, if **<some object>** were the string **META** the result might be **META4.**

In this way the conventional distinction between data and procedures is reduced, since the meaning rests with the objects of the system, and the code remains an abstract form, merely directing the flow of communication.

In functional programming, languages operate by the application of functions to values. Functional programming lives in the "clean" mathematical world of equations: expressions, function applications and structured data; a world excluding sequentiality, assignment statements, and side-effects. Firstly, non-sequentiality. A functional program usually consists of a series of equations which are viewed as unordered apart from their data dependencies. Each equation specifies a calculation but the programmer specifies no additional sequencing information over and above that implied by the data dependencies. Thus statements can appear in any order in a program. Secondly, absence of assignment. Functional programming does not contain the concept of assigning a value to a global memory as seen in the basis of procedural programming. Obviously, one still has to be able to associate a name with a value (as when one writes an equation "name=expression"); this is an essential feature of any usable language. The important difference is that a conventional assignment statement is used to "overwrite" a previously existing value, whereas here the destructive assignment concept is not allowed. This implies the absence of side-effects. These features of functional programming mean that in any one program, no two equations can have the same left-hand side for a statement of the form "name=expression". It also implies that all operations on data structures have to have a copying semantics, since a data structure cannot be overwritten or altered.

Two important classes of functional programming languages will be identified: data flow languages (i.e. single-assignment) and applicative languages. Data flow languages are designed to facilitate programming of data flow computers (which were discussed in the previous chapter), and are concerned with the easy expression and exploitation of parallelism. By data flow language one means any functional language based entirely upon the notion of data "flowing" from one function entity to another, or any language that directly supports such flowing semantics. This flow concept gives data flow languages the advantage of easily expressing programs either textually or by equivalent directed graphs [17]. There are a number of interesting data flow languages including ID [6], LUCID [8], VAL [2] and VALID [3]. Applicative languages are so-called because of the dominant role played by the applications of functions to structures. Quoting Henderson [30] "Intuitively, a function is a rule of correspondence whereby to each member of a certain class there corresponds a unique member of another class. That is to say, given two classes of individuals, respectively called the domain of the function and the range of the function, each member of the domain is made to correspond by the function to exactly one member of the range." Thus the important notion associated with applicative structure is that the value of an expression (its meaning) is determined solely by the values of its constituent parts. Thus, should the same expression occur twice in the same context, it denotes the same value at both occurrences. A language having this property for all its expressions is referred to as an applicative language.

In logic programming [19,36] a program consists of facts about a certain subject, stated as a collection of sentences which express information that can be used to solve problems or to answer questions. A sentence (i.e. clause) defines a relationship, and is either an <u>assertion</u>:

```
bill is the father of john
john is the father of tom
```

or an <u>implication</u>:

```
X is the grandfather of Z if X is the father of Y
                             and Y is the father of Z
```

where john, tom, etc. are atoms and X,Y etc. are variables. Basically, logic programming attempts to solve goals, which succeed or fail, when answering a question. For a given goal (initially the question), the system attempts to find any statements that can be made to match the goal. If the matching statement is an assertion then the system is successful but, otherwise, it proceeds to solve the subgoals. This execution can be viewed as pattern matching: selection of the statements, and substitution: solving of the goals.

Symbolic logic was first used as a formalisation of natural language and human reasoning. As a result it has long been appreciated in computing science that logic programming could yield very powerful languages, blurring not only the distinction between programs and databases but also the distinction between programs and specifications [36]. Information can be expressed and problems can be formulated without concern for specifying explicitly the details of execution or for efficiency. However, logic programs can be given an operational, machine intelligible interpretation.

Finally, in application programming, languages are being developed for specific application areas. One of the more interesting aspects of recent computing history is the explosive growth in the programming languages for specific application areas. Example areas include: financial-modelling [14], expert systems [60], and robotics [13]. However, in these application areas, the boundary between languages and certain software packages or utilities is cloudy. A prime example is VISICALC, and its derivatives, marketed as financial-modelling systems, but used in the additional fields of engineering, science, education and statistics. In fact, in any field where tabular reports of rows and columns of calculated numbers are required, the VISICALC language provides a very powerful tool. Cynics might say that there is currently more programming done in VISICALC-like languages than in all object-oriented, functional and logic programming languages together.

Next, the major programming styles are classified in terms of their underlying computational mechanisms, so as to analyse their advantages and disadvantages in terms of these mechanisms. The computational mechanisms presented here generalise the set of mechanisms originally proposed for a novel computer architecture [50].

3.2. COMPUTATIONAL MECHANISMS

Treleaven et al, in their survey paper [50], proposed a classification for data and demand driven computer architecture. This consisted of three ways in which an instruction could use an argument: "by literal", "by value", "by reference", and three control patterns: "sequential", "parallel", and "recursive". Although adequate for the purpose, this classification has a number of weaknesses, the most important of which is that it does not cover actor and logic architectures. Below, a more general classification is presented, oriented to programming models and languages. It will be used in classifying and analysing the styles of programming presented above.

"Programming model" is the term used in this Thesis to cover the way programs are represented and executed in a computer. For a programming model there are two basic computational mechanisms, which are referred to here as the data mechanism and the control mechanism. The **data mechanism** defines the way a particular argument is communicated (and shared) by a number of commands. There are two basic types of communication in computing:

1. **shared memory** - where a single copy of the argument is communicated via a shared memory, accessible to all commands.

2. **message passing** - where a unique copy of the argument is communicated, via a message, from the source to the destination command.

The **control mechanism** defines how one command causes the execution of one or more other commands. There are four basic types of execution in computing:

1. **control driven** - where a command is executed when it is selected by flow(s) of control.

2. **data driven** - where a command is executed when some combination of its arguments is available.

3. **demand driven** - where a command is executed when the result it produces is needed by another, already active command.

4. **pattern driven** - where a command is executed when some enabling pattern (or condition) is matched.

The relationship that is believed to exist between these data and control mechanisms and the major styles of programming is summarised in Figure 3.2.

		Data Mechanisms	
		shared memory	message passing
Control Mechanism	control driven	Procedural .conventional . concurrent (shared memory)	Procedural . concurrent (message passing)
	data driven		Functional . data flow
	demand driven	Functional . applicative (pattern-matching)	Functional . applicative (function-level)
	pattern driven	Logic	Object-Oriented

Figure 3.2: Classification of Programming Styles

It is believed that the properties of the programming styles and their associated languages relate directly to their choice of data mechanism and control mechanism.

Firstly, the properties related to the data mechanisms will be summarised. "Shared memory" has advantages for: the sharing of data structures, the taking of an unspecified number of copies of the data, and the ability, in certain models, to update the data. The disadvantages of "shared memory" relate to synchronising the reading and writing of data, not only in parallel systems, but also where a flow of control must be specified. "Message passing" has the advantages of synchronising communication of data, which is particularly useful between parallel processes, and the ability to be tied to the control mechanism as in data flow. Disadvantages of "message passing" include the need often to know all the consumers of the messages and, possibly, the need to explicitly delete unused messages.

Secondly, the properties related to the control mechanisms will be summarised. "Control driven" mechanisms have advantages such as the fact that they are very primitive and flexible, and provide maximum control over the execution of commands. This results in a separation of flows of control and flows of data in a program. Disadvantages also relate to this flexibility; as the sequence of execution of commands must be specified, this places an additional burden on the programmer, and it is easier to make mistakes. "Data driven" mechanisms have the advantage of specifying maximally parallel execution, but also the disadvantage of sometimes causing unnecessary computations. "Demand driven" mechanisms have the advantages of performing minimum work (since demands are only made when necessary), and of generating a hierarchical control pattern. A direct disadvantage is that the control pattern is restricted to such a tree structure. Lastly, "pattern driven" mechanisms have the advantage of being the highest level control mechanism requiring least control information to be specified by a programmer. Again, this leads to the disadvantage of the programmer sometimes having inadequate control over the execution of a program.

3.3. QUICKSORT

To facilitate comparison of Procedural, Object-oriented, Functional, Logic and Application programming, simple programs for the Quicksort algorithm are used. Quicksort, invented by Hoare [31], is one of the best sorting algorithms known. Although in the worst case its execution time can be proportional to n**2, its average time is nlogn.

The essential idea of Quicksort is to partition the original set to be sorted by rearranging it into two subsets: the first subset, all of whose elements are less than some arbitrary "pivot" value chosen from the set, and the second subset, all of whose elements are greater than or equal to the pivot. Then the partitioning process is applied to the two subsets, until each subset contains only one element. When all subsets have been partitioned, the original set has been sorted.

To illustrate the Quicksort algorithm, Figure 3.3 shows the series of comparisons and exchanges for an array of 16 elements. The elements being compared at each stage are indicated and square bracket symbols are used to delimit the subset. The sorting of a subset of the array "v" involves the "pivot" - the first element of the set - plus two pointers "i" and "j"; with "i = 1" and "j = 16" initially. Quicksort compares "v[i] <= pivot" and "v[j] >= pivot", exchanging "v[i]" and "v[j]" when it finds an out of order pair. This comparison is repeated until "i = j" at which point "v[i]" and the pivot are exchanged, inserting the pivot element into its correct position in the array. This is clearly indicated by the elements shown in Figure 3.3. Having partitioned the elements to be sorted, Quicksort can be reapplied to the two subsets.

```
Stage                        Array

 1  512  087  503  061  908  170  897  426  765  275  154  509  612  677  653  703
 2 [512  087                                                                 703]
 3 [512  087                                                             653     ]
 4 [512  087                                                        677          ]
 5 [512  087                                                   612               ]
 6 [512  087                                              509                    ]
 7 [512       503                                         509                    ]
 8 [512            061                                    509                    ]
 9 [512                 908                               509                    ]
10 [512                 509                               908                    ]
11 [512                 509                          154                         ]
12 [512                      170                     154                         ]
13 [512                           897                154                         ]
14 [512                           154                897                         ]
15 [512                           154           275                              ]
16 [512                                426      275                              ]
17 [512                                     765  275                              ]
18 [512                                     275  765                              ]
19 [275                                     512  765                              ]
20 [275  087  503  061  509  170  154  426]     [765  897  908  612  677  653  703]
21 [275  087                           426]     [765  897                     703]
22 [275  087                      154     ]     [765  703                     897]
23 [275       503                 154     ]     [765  703                653     ]
24 [275       154                 503     ]     [765       908           653     ]
25 [275       154            170          ]     [765       653           908     ]
26 [275            061       170          ]     [765       653       677          ]
27 [275                 509  170          ]     [765            612  677          ]
28 [275                 170  509          ]     [765                 677          ]
29 [170                 275  509          ]     [677                 765          ]
30 [170  087  154  061]     [509  503  426]     [677  703  653  612]     [908  897]
31 [170  087       061]     [509  503  426]     [677  703       612]     [908  897]
32 [170       154  061]     [509       426]     [677  612       703]     [897  908]
33 [170            061]     [426       509]     [677  612  653     ]
34 [061            170]     [426  503]          [677       653     ]     [897  908]
35 [061  087  154]                              [653       677     ]     [897  908]
36                                              [653  612]
37                                              [612  653]
38  061  087  154  170  275  426  503  509  512  612  653  677  703  765  897  908
```

Figure 3.3: Quicksort partition exchange sorting

In the example programs that follow in Chapters 4 and 5, Quicksort has been programmed in a relatively simplistic way attempting only to match the example languages to the structure of the algorithm. The array initialisation and input/output statements are usually omitted.

Using the above classification as a basis, the following two chapters present an examination of the advantages and disadvantages for program representation and execution of the major programming styles.

CHAPTER 4 - ANALYSIS OF PROCEDURAL AND OBJECT-ORIENTED PROGRAMMING

This chapter analyses Procedural and Object-Oriented styles of programming, identifying advantages and disadvantages.

4.1. PROCEDURAL PROGRAMMING

Procedural programming is based on a "shared memory" data mechanism and a "control driven" control mechanism. For the data mechanism, data is communicated via shared memory cells and the basic action in procedural programming is assignment to memory. In addition, a cell is accessible to a group of commands, any of which may take as many copies as it requires or update the contents of a cell. For the control mechanism, execution is "control driven" and implicitly sequential. Explicit control structures are also provided for: unconditional (e.g. GOTO), conditional (e.g. IF, CASE), and repetitive (e.g. FOR, REPEAT, WHILE) execution.

4.1.1. Conventional Languages

In conventional languages (e.g. PASCAL), besides the sequential "control driven" control mechanism, there are procedure and function CALLs, plus exception conditions such as ON in PL/I. These control structures could also be assumed to be "control driven", but seem more accurately to be "demand driven" and "pattern driven", respectively.

Although BASIC is perhaps the archetypal conventional language, PASCAL will be briefly examined. Since PASCAL contains both iterative and procedure call control structures, it also serves to introduce the programming of the Quicksort examples.

Figure 4.1 illustrates a conventional program in PASCAL based on a recursive algorithm, probably the simplest description of Quicksort. The heart of the program is the "sort" procedure, which partitions the elements of the array "v" to be sorted. To do this it is passed the indices "lo" and "hi", indicating the range of elements to be partitioned at this particular step. It chooses as the "pivot", "v[lo]". The inner loop of the algorithm (i.e. the iterative "repeat-until" statement), compares the elements and exchanges any pair that is out of order. When this loop exits, the elements have been partitioned, "pivot" is swapped with "v[i]", and the procedure recursively calls "sort" twice to partition the two subsets.

```
program Quicksort

var
   v: array[1..16] of integer;

procedure sort(lo,hi: integer);

var
    i,j:   integer;
  pivot:   integer;
  temp :   integer;
begin
    if (lo < hi) then
    begin
        i:= lo;
        j:= hi;
        pivot:= v[lo];
        repeat
             while (j > i) and (v[j] >= pivot) do j:= j - 1;
             while (i < j) and (v[i] <= pivot) do i:= i + 1;
             if (i < j) then     (* exchange out of order pair *)
                          begin
                              temp:= v[i];
                              v[i]:= v[j];
                              v[j]:= temp;
                          end;
        until (i >= j);
        (* move pivot to v[i] *)
        v[lo]:= v[i];
        v[i] := pivot;
        (* sort subsets *)
        sort(lo, i - 1);
        sort(i + 1, hi);
    end;
end;

begin

  sort(1, 16);

end.
```

Figure 4.1: Conventional (recursive) Program in PASCAL

Recursive algorithms are sometimes less efficient than the equivalent iterative ones, often because of "demand driven" procedure call overheads, but for Quicksort this is not a significant effect, since the recursion is not in the innermost loop. As a contrast, Figure 4.2 shows a PASCAL program based on an iterative algorithm but kept purposely close in structure to the previous example. The inner loop is identical. In Figure 4.2, a stack is used to store the pairs of indices "lo" and "hi" of the subsets to be partitioned. Thus instead of calling a procedure "sort" to partition a subset, the pair of indices are placed on the stack. The outer loop repeatedly removes a pair of indices from the stack and partitions the set. When the stack is empty the array "v" has been sorted.

```
program Quicksort

var

         v:  array [1..16] of integer;
     stack:  array [1..20, 1..2] of integer;
  stackptr:  0..20;
     pivot:  integer;
     lo,hi:  integer;
      i,j :  integer;
      temp:  integer;

begin
    stackptr:= 1;
    stack[stackptr,1]:= 1;
    stack[stackptr,2]:= 16;
    repeat
         lo:= stack[stackptr,1];
         hi:= stack[stackptr,2];
         stackptr:= stackptr - 1;
         if (lo < hi) then
         begin
             i:= lo;
             j:= hi;
             pivot:= v[lo];
             repeat
                  while (j > i) and (v[j] >= pivot) do j:= j - 1;
                  while (i < j) and (v[i] <= pivot) do i:= i + 1;
                  if (i < j) then   (* exchange out of order pairs *)
                  begin
                      temp:= v[i];
                      v[i]:= v[j];
                      v[j]:= temp;
                  end;
             until (i >= j);
             (* move pivot to v[i] *)
             v[lo]:= v[i];
             v[i] := pivot;
             (* sort subsets *)
             stackptr:= stackptr + 1;
             stack[stackptr,1]:= lo;
             stack[stackptr,2]:= i - 1;
             stackptr:= stackptr + 1;
             stack[stackptr,1]:= i + 1;
             stack[stackptr,2]:= hi;
         end;
    until stackptr < 1;
end.
```

Figure 4.2: Conventional (iterative) Program in PASCAL

Two of the principal concepts of the control flow model, namely "shared memory" communication of data and sequential "control driven" execution, are reflected in both programming examples examined. In both examples, the partitioning of the numbers is performed by rewriting the array. (An alternative strategy used in many of the programming languages, to be discussed, is to create two new arrays.) In addition, in the example in Figure 4.1, the two calls to "sort" clearly could be performed concurrently if parallel control driven structures were supported. Although almost no parallel control structures are found in conventional languages such extensions are found in the following procedural classes of languages examined.

4.1.2. Concurrent Languages

In concurrent languages (e.g. MODULA-2 and OCCAM), as well as the sequential "control driven" structures, there are parallel processes and structures to handle problems of process communication and synchronisation. In concurrent (shared memory) languages, communication is via the "shared memory" data mechanism, concurrent access to which is synchronised by "control driven" monitors (in MODULA-2) that guarantee mutual exclusion to accessing processes. In addition, MODULA-2 also provides a primitive "pattern driven" control mechanism, for synchronisation, in the form of a signal.

In contrast, in concurrent (message passing) languages, a "message passing" data mechanism is used to handle synchronised communication between parallel processes with the "shared memory" mechanism being used for communication within a process. In OCCAM, as well as the traditional control structures (e.g. IF, WHILE, FOR) there are also three "control driven" structures, for sequential "SEQ", parallel "PAR", and alternative "ALT" process execution.

Shared Memory

In shared memory communication, the synchronisation mechanism provides mutual exclusion on single bytes, words, or larger data structures. Examples of concurrent languages with shared memory communication include: MODULA-2, Concurrent PASCAL, and Path PASCAL. MODULA-2 will be examined.

MODULA-2 [63], one of the many languages designed by Wirth, extends Pascal with facilities for program structuring and concurrency. In MODULA-2, each program is declared as a module:

```
MODULE name;
<declarations>
BEGIN
<statements>
END name.
```

which encapsulates all of the data structures and procedures used by the program, and controls their usage by other programs.

Concurrency is based on: processes, shared variables, signals and monitors. Execution of a concurrent process is started by using the system call "StartProcess (P,n)" where "P" is the procedure to be executed and "n" is the size in words of the work space the process is to be allocated. Communication amongst processes occurs in two distinct ways, namely via common variables and so-called signals.

Using shared memory (i.e. common variables) to transfer data among processes raises the problem of asynchronous access to this data. The MODULA-2 solution is a "monitor"; a module which guarantees mutual exclusion of accessing processes and thereby ensures integrity of its local data. A module is designated to be a monitor by specifying a "[priority]" in its heading. Signals (pattern driven) serve to synchronise processes, but do not carry data. Only two operations are applicable to signals: a process may "SEND" a signal and it may "WAIT" for a signal from some other process. Execution of a WAIT suspends the process. Execution of a SEND reactivates at most one process. These concurrency mechanisms are illustrated by Figure 4.3.

Quicksort in MODULA-2 is perhaps best coded as a sequential recursive algorithm as in the PASCAL example in Figure 4.1. However, to illustrate the concurrency features of MODULA-2, an attempt has been made to code it using processes, signals and monitors. The example uses two modules called "Quicksort" and "monitor". "Quicksort" uses the process "sort" to recursively partition the array "v", and it uses the monitor "monitor[1]" to control the passing of the indices of the subsets (via a stack) to the sort processes. The algorithm thus combines the features of the recursive and iterative PASCAL examples.

Execution is started by Quicksort placing the indices "(1,16)" on the stack, and calling StartProcess to initiate "sort". The process sort partitions the array, places the limit of the two subsets on the stack, and then calls StartProcess twice. These concurrent "sort" processes may access the stack in any order, but only one may do so at a time, since the stack is within a monitor. Termination of Quicksort is controlled by the SIGNAL "finished" and the "count" of the processes executing. For each process started, "count" is incremented and when a "sort" finishes "count" is decremented, and if "count" equals zero, the signal "finished" is sent to the main program, so it may terminate. In addition, notice at the end of "sort" the signal "forever" - this is used to keep the process from being deleted.

```
MODULE Quicksort;
  FROM ProcessScheduler IMPORT INITSIGNAL,SIGNAL,WAIT,SEND,STARTPROCESS,
                               SENDDOWN;
  FROM SYSTEM IMPORT ADDRESS;
  FROM Storage IMPORT ALLOCATE,DEALLOCATE;
  FROM Input IMPORT ReadInt;
  VAR
    v        :  ARRAY [1..16] OF INTEGER;
    finished:  SIGNAL;                    (* signals a process has finished *)
    count    :  INTEGER;                  (* count of processes executed   *)
    wsp      :  ADDRESS;
    i        :  INTEGER;
    num      :  INTEGER;

  PROCEDURE sort;
  VAR
    lo,hi    :        INTEGER;
    i,j      :        INTEGER;
    pivot    :        INTEGER;
    temp     :        INTEGER;
    wsp1     :        ADDRESS;
    wsp2     :        ADDRESS;
    forever :         SIGNAL;
  BEGIN
      pop(lo, hi);  (* get limits of next subset *)

      IF (lo < hi)
      THEN
          i:= lo;
          j:= hi;
          pivot:= v[lo];
          REPEAT
               WHILE (j > i) AND (v[j] >= pivot) DO j:= j - 1 END;
               WHILE (i < j) AND (v[i] <= pivot) DO i:= i + 1 END;
               IF (i < j) THEN     (* exchange out of order pair *)
                                 temp:= v[i];
                                 v[i]:= v[j];
                                 v[j]:= temp;
                             END;
          UNTIL (i >= j);
          (* move pivot to v[i] *)
          v[lo]:= v[i];
          v[i] := pivot;
          (* store limits of subsets to be sorted *)
          push(lo, i-1);
          push(i+1, hi);
          count:= count + 2;
          ALLOCATE(wsp1,200);
          STARTPROCESS(sort,wsp1,200);
          ALLOCATE(wsp2,200);
          STARTPROCESS(sort,wsp2,200);
      END;
      count := count - 1;
      IF count = 0 THEN SEND(finished) ELSE
        INITSIGNAL(forever);
        WAIT(forever);
      END;
  END sort;


MODULE monitor[1];
  IMPORT SIGNAL,SEND,SENDDOWN,INITSIGNAL,WAIT;
  EXPORT push, pop;
  CONST N = 16;
  VAR
    stack   : ARRAY [1..N] OF ARRAY [1..2] OF INTEGER;
    stackptr: [0..N];
    NotEmpty, NotFull : SIGNAL;

  PROCEDURE push (lo, hi: INTEGER);
  BEGIN
    IF stackptr = N THEN WAIT(NotFull) END;
    stackptr := stackptr + 1;
    stack [stackptr, 1] := lo;
    stack [stackptr, 2] := hi;
    SENDDOWN(NotEmpty);
  END push;

  PROCEDURE pop (VAR lo, hi: INTEGER);
  BEGIN
    lo := stack [stackptr, 1];
    hi := stack [stackptr, 2];
    stackptr := stackptr - 1;
    SEND(NotFull);
  END pop;
BEGIN
   stackptr:= 0;
   INITSIGNAL(NotFull);
   INITSIGNAL(NotEmpty);
END monitor;

BEGIN
  (* initialisation *)
  INITSIGNAL(finished);
  FOR i:= 1 TO 16 DO
      ReadInt(num);
      v[i]:= num;
  END;
  push(1,16);
  count:= 1;
  ALLOCATE(wsp,200);
  STARTPROCESS(sort,wsp,200);
  WAIT(finished);
END Quicksort.
```

Figure 4.3: Concurrent (shared memory) Program in MODULA-2

It could be claimed that this form of concurrency (using "shared memory" data mechanism) is the most natural extension to conventional languages. In conventional languages data is communicated via variables; concurrent languages (such as MODULA-2) use the same mechanism, namely shared memory. In contrast, the next class of languages uses a "message passing" data mechanism for communicating data amongst concurrent statements.

Message Passing

In message passing communication, data is passed directly, using a channel or queue from the transmitting process to the receiving process, which stores the data locally in its private store. Examples of these types of programming languages include CSP [32] and OCCAM [49], as well as GYPSY, PARLANCE and PLITS [48]. The OCCAM programming language will be examined.

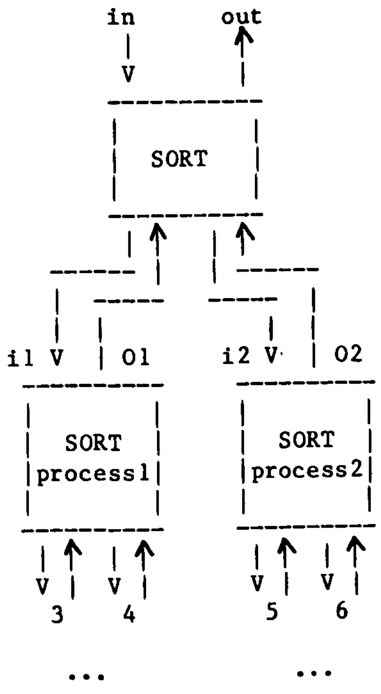
OCCAM [49], originating from Hoare's CSP, is based on processes which may be executed concurrently and may communicate using channels. The most direct implementation of an OCCAM program is a network of microcomputers each executing a process concurrently. However, the same program could also be implemented by a single time-shared processor.

A process - the fundamental working element in OCCAM - is a single statement, group of statements, or group of processes. Programs are constructed from three primitive processes: assignment, output and input. Assignment "x:=y" sets the value of a variable to an expression. Output "c!y" is used to output a value of an expression "y" to a channel "c". Input "c?x" sets the value of a variable "x" to a value input from a channel "c".

A channel is an unbuffered structure and allows information to pass in one direction only, synchronising the transfer of information. Thus a channel behaves as a read-only element to a receiving process and a write-only element to the transmitting processing. The transmitter can only write when the channel is empty, while the receiver can only read when the channel is full.

To control the order of execution of such processes OCCAM provides three "control driven" mechanisms: sequential (SEQ), parallel (PAR), and alternate (ALT), as well as the traditional IF and WHILE constructs. SEQ and PAR precede a list of processes, defining sequential and parallel execution, respectively. ALT causes exactly one of a list of processes to be executed, and will wait until at least one of the "guarding" conditions is true.

These control mechanisms are illustrated by the Quicksort example in Figure 4.4. This program consists of two processes: "sort", which partitions the array of numbers to be sorted and merges the sorted subsets; and "quicksort", which builds a tree of sort processes to perform the sorting:



The highest level sort process inputs the array of numbers to be sorted on "CHAN in". It partitions the array, sending numbers less than the pivot to sort process "1", and numbers greater than the pivot to sort process "2". Having partitioned the array, the highest level sort process merges the sorted subsets that it receives from processes "1" and "2", and outputs the result on "CHAN out". Subsidiary sort processes operate in a similar way, (sort processes "1" and "2" then become the highest level sort processes, and so forth). Each array of numbers to be sorted is terminated by "-1".

A sort process consists of six channels: "vin" - the numbers to be sorted; "lout" - the numbers to be sorted less than the pivot; "hout" - the numbers to be sorted higher than the pivot; "lin" - the sorted numbers less than the pivot; "hin" - the sorted numbers higher than the pivot, and "vout" - the sorted numbers resulting from the merge of "lin", "pivot", and "hin".

Since OCCAM's input/output only reads individual characters, to simplify the example, letters were included in the set to provide for 15 entities to be sorted. The set to be sorted here can be, for example:

7 3 1 0 2 5 4 6 B 9 8 A D C E

so as to guarantee a well balanced tree. In fact, this Quicksort will only work for such arrays, because processes must be statically (compile-time generated) in OCCAM.

```
DEF n = 15:
DEF term = -1:

PROC sort(CHAN vin,vout,lin,lout,hin,hout) =
  VAR  pivot, x:
  SEQ
    vin?pivot
    IF
     pivot <> term
       SEQ
         vin?x
         WHILE x <> term
           SEQ
             IF
              x < pivot
                 lout!x
              x >= pivot
                 hout!x
             vin?x
         PAR
           lout!term
           hout!term
         lin?x
         WHILE x <> term
           SEQ
             vout!x
             lin?x
         vout! pivot
         hin?x
         WHILE x <> term
           SEQ
             vout!x
             hin?x
  vout!term:

PROC quicksort(CHAN in,out) =
  CHAN i[(4 * n) + 5], o[(4 * n) + 5]:
  SEQ
    str.to.screen("QUICKSORT*C")
    PAR
      sort(in,out,o[1],i[1],o[2],i[2])
      PAR c = [1 FOR (2*n) + 1]
        sort(i[c],o[c],o[(2*c)+1],i[(2*c)+1],o[(2*c)+2],i[(2*c)+2]):
```

```
VAR input[n]:
CHAN in.out:
SEQ
  SEQ i = [0 FOR n]
    SEQ
      keyboard? input[i]
      screen! input[i]
  str.to.screen("*CDone Input*C")
  PAR
    SEQ
      SEQ i = [0 FOR n]
        in! input[i]
      in! -1
    quicksort(in,screen)
```

Figure 4.4: Concurrent (message passing) Program in OCCAM

OCCAM has two important, and interesting, features that should be noted. Firstly, it is a concurrent language specifically designed to facilitate the programming of a new generation of (networks of) micro-computers [10]; an essential requirement for exploiting VLSI. Secondly, unlike most other procedural languages, it has a formal basis which opens up the potential of formal reasoning and transformation as design techniques.

In summary, the main features of procedural programming related to the "shared memory" data mechanism are: shared memory cells, updatable cells, and assignment as the basic action; and related to the "control driven" control mechanism are: implicit sequential execution, plus explicit sequential and parallel control structures. Advantages of the "shared memory" data mechanisms include: its efficiency for supporting the sharing of data structures, the ability to take an unspecified number of copies of a cell's contents, and the updating of the contents. In fact, these features can be viewed as a simple scheme for memory management. A major disadvantage with this data mechanism, as discussed below, is synchronising access to a memory cell. An advantage of the "control driven" control mechanism for execution is that flows of data and control in a program are separate, and hence can be made identical or distinct. The related disadvantage of this control mechanism is that flows of control must be explicitly specified by the programmer.

Due to these data and control mechanisms, procedural programming is very flexible and most algorithms can be expressed with reasonable efficiency. But this flexibility also presents disadvantages: ensuring that the flow of control correctly synchronises the use of memory cells, and the difficulty, for modular programming, of encapsulating information due to the general accessibility of cells. Parallelism also presents major problems for the unconstrained use of the "shared memory" data mechanism, and is really only overcome by the introduction of "message passing".

Next, object-oriented programming is examined, which may be viewed as attempting to generalise the concurrent languages concepts of monitors and message passing.

4.2. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is based on a "message passing" data mechanism and a "pattern driven" control mechanism. For the data mechanism, data is communicated between the active objects (cf. processes) by "message passing", and within a process, by the use of "shared memory". These variables represent the state of an object, and in SMALLTALK, for example, there are six different kinds of variables. For the control mechanism, execution of objects is viewed as "pattern driven", but, in SMALLTALK, execution within an object is "control driven" and implicitly sequential. Each method (cf. entry point) within an object is identified by a message pattern consisting of a selector and names for the arguments. It is this pattern that is matched with that of the message. The remainder of the message is executed sequentially. As well as the implicit "control driven" execution of a method, there are explicit control structures that provide conditional "ifTrue" or "ifFalse", and iterative "whileTrue" or "whileFalse" execution.

SMALLTALK [4,33] illustrates the current state of object-oriented programming, and an examination of the format of classes, which is the natural unit of modularity in the language, follows. A class consists of three basic types of information: class name, variable declarations, and the "methods" describing the actions when a message arrives. Six kinds of variables may be used [33]: the instance variables, unique to each instance of the class; the pseudo-variable "self"; the actual message arguments; temporary variables, created when a message is received; class variables , shared by all instances of the class; and global variables.

The methods of a class specify what happens when its instances receive a particular message. These actions consist of sending other messages, assigning values to variables and returning a value to the original message. A method has three parts: a message pattern which is similar to a label, some temporary variable names, and expressions to process the received message (these three parts of a method are separated by vertical bars "|"). The message pattern consists of a selector and names for the arguments. Expressions are separated by dots ".", and the last one may be preceded by a vertical arrow indicating the value to be returned. These expressions contain conventional expressions, assignment statements, as well as message-sending expressions that serve a similar role to procedure calls.

A message-sending expression defines the receiver (cf. the procedure), the selector (cf. the entry point), and the arguments of the message. There are basically three types of message: firstly "unary" consisting of a single selector and no arguments, secondly "binary" consisting of a single selector and a single argument, and lastly "keyword" where an "identifier:" is prefixed to each argument. For example:

Message	Example
unary	name INC
binary	name + 1
keyword	name s1: 1 s2: "a"

Messages are evaluated left to right and, like with conventional expressions, parenthesis can be used to change the order of evaluation.

The two control structures in SMALLTALK described so far are the sequential "control driven" execution of expressions in a method and the "pattern driven" sending of messages that invoke other methods, that eventually return values. All other control structures are based on objects called blocks, each containing a sequence of expressions. Execution of blocks may themselves be controlled by conditional selectors "ifTrue" or "ifFalse", and by conditional iteration "whileTrue" or "whileFalse". Examples of the use of these can be seen in the Quicksort program, which follows.

Figure 4.5 shows a (restricted form of) class template [33] for sort. It contains a single instance variable called "result", and has a single method with selector "array". Sort is activated by a "pattern driven" keyword message containing two arguments: the array to be sorted and its size. These arguments are referred to as "v" and "n". Next, the declaration of the temporary variables "low", "high", "i", "j", "k", and "pivot" is seen. The array "v" will be partitioned into "low" and "high", while "j" and "k" provide indexes/counters for these two arrays. Sort starts by initialising these temporary variables, for instance setting "pivot" to the first element "at:0" of "v".

Next, array "v" is partitioned using the conditional iteration "[...] whileTrue [...]" which has the form of a WHILE-DO. Inside the iteration, the conditional selector IF-THEN-ELSE extracts elements from "v", inserting them in the array "low" if less than the "pivot", or into "high" if greater than the pivot. Having partitioned the array "v", sort is invoked (if necessary) for "low" and "high". Finally, the contents of "low", "pivot" and "high" are stored into the array "result" which is returned to the invoking message. Since "result" is declared as an instance variable, rather than as temporary, its value will be retained to the next call, which is illustrative but not particularly sensible in this example.

```
class name
sort

instance variable
result

methods

   array: v size: n | low high i j k pivot |
      "initialise temporary variables"
      j <- k <- 0.
      pivot <- v at: 0.
      i <- 1.
      "partition array v"
   [i < n] whileTrue: [ [(v at: i) < pivot]
                         ifTrue: [low at: j put:( v at: i).
                                  j <- j +1 ]
                         ifFalse: [high at: k put:(v at: i).
                                   k <- k + 1 ]
                    i <- i + 1 ].
      "sort both subsets if necessary"
      [j > 1] ifTrue: [low <- sort new array: low size: j].
      [k > 1] ifTrue: [high <- sort new array: high size: k].
      "store sorted subsets into result"
      result <- low.
      result at: j put: pivot.
      i <- 0
      k timesRepeat : [ result at: (i + j + 1) put: (high at: i) i <- i + 1].
      "return result array"
      ^ result.
```

Figure 4.5: Object-Oriented Program in SMALLTALK

SIMULA [12] was the first language to explore object-oriented programming for structuring information. It grafted the notion of objects onto an ALGOL-like base language, and provided for sharing amongst objects by organising them into classes. This work has been extended by incorporating message-passing in a number of languages, most notably the SMALLTALK series of languages. An important aspect of SMALLTALK is that it provides a "total" programming system unifying features, normally found in operating systems, with those of programming languages. In addition, object-oriented languages may be seen as attempting to combine concepts from procedural programming and functional programming.

In summary, the main features of object-oriented programming are: the use of active objects that retain state, the sharing of data between objects of a class using "shared memory", and the communication and execution of objects by "message passing". One of the major advantages of object-oriented programming and its associated concepts of class and instance is that it encourages modularity. Another advantage is that objects can act as templates for different types of data, and a further one, provided by the different types of "variables" as found in SMALLTALK, is that data can be shared by objects (of the same class) or can be only accessible within a class. A possible disadvantage of object-oriented programming is the "pattern driven" execution based on the arrival of messages, which is, arguably, harder to understand for the traditional programmer than either "control driven" execution, or "demand driven" execution associated with procedure calls. A final specific criticism of SMALLTALK is that the control mechanism is restricted to sequential execution.

In object-oriented and procedural programming, a computation is performed by executing a series of actions in a precisely specified order. Each statement in these programming styles represents only one step in an algorithm. This implies that correctness of an individual statement cannot be determined by solely examining the statement. Instead, the entire algorithm in which the statement occurs must be executed to determine if the statement is correct. In functional and logic programming, to be examined below, frequently it can be determined whether a statement is true by examining that statement only. The reason is that functional programming is based on expressions and logic programming is based on relations.

CHAPTER 5 - ANALYSIS OF FUNCTIONAL AND LOGIC PROGRAMMING

This chapter analyses Functional and Logic styles of programming, identifying advantages and disadvantages.

5.1. FUNCTIONAL PROGRAMMING

There has been a surge of interest in functional programming following the publication of John Backus' 1977 ACM Turing Award lecture [9]. After all, to quote Turner [56], "it is not every day that the inventor of FORTRAN gets up and says that he now thinks that the invention of the assignment statement was a serious error!" The language most people first think of when functional programming is mentioned is LISP. However LISP is a functional language only if RPLACA, RPLACD, and all other functions with side-effects are avoided; this subset of the language is often called "Pure" LISP. Unfortunately, a number of people find the syntax of LISP clumsy.

Functional programming is based on three different pairings of data and control mechanisms, as shown in Figure 3.2. (These, in fact, correspond to the programming models: data flow, string reduction and graph reduction [50].) The essential concepts shared by all categories of functional languages are: expressions, function applications and recursive data structures, as well as the absence of: sequentiality, the assignment statement, and side-effects. The two important sub-classes of functional languages, identified in Chapter 3, are data flow (i.e. single-assignment) languages and applicative languages.

5.1.1. Data Flow Languages

Recall, in the data flow programming model, a statement outputting (i.e. producing) a result, passes a separate copy by "message passing" to each statement wishing to input (i.e. consume) the value. There is no fundamental concept of variables - a "shared memory" data mechanism - in a data flow program; data is passed directly from one statement to another. Execution of a statement is "data driven", with a statement being executed as soon as all its input values are available. Statements in data flow languages are similar to statements in conventional languages, but follow a single assignment rule: a name may appear on the left side of an assignment only once within the area of the program in which it is active.

In the data flow languages, the data mechanism is "message passing", and the control mechanism is "data driven". For the data mechanism, an important property is the copying semantics, which means that any operation on a data structure always creates a new structure (it is most unusual to find any "shared memory" data mechanism in data flow languages). For the control mechanism, although execution is "data driven", statements in data flow language are superficially similar to statements in conventional languages, as can be seen from Figure 5.1. Explicit control structures are provided for conditional expressions (e.g. if-then-else), for iterative expressions (while-do, for-do) and for function calls.

The most important copying semantics of data flow languages means that, for instance, an array is not modified by a subscripted assignment statement "array[index]:=value", but is processed by an operator which creates a new array. In the ID language this operator appears as:

```
new_array<- array + [index]value
```

while in the VAL language it is written:

```
new_array:= array[index:value]
```

These single assignment statements are adequate for simple assignment of the form:

```
name:= expression
```

and even for conditional statements, as long as they are restricted to conditional expressions:

```
name:= if expression then expression
                          else expression
```

Where the single assignment rule might appear to cause trouble is for iterative statements which imply the updating of variables.

In data flow iteration, since there are no side-effects, the only state information in an iteration is the binding of loop variables, and the only activity that can occur is redefinition of these variables. An iteration therefore consists of [1]:

1. the definition of the initial values of the loop variables,

2. a termination test for the iteration,

3. the definitions of the new values of the loop variables, and

4. the results to be returned when the loop terminates.

As an illustration, an iteration to compute the ubiquitous factorial of "n" could be written in ID as:

```
answer <- (initial j <- n; k <- 1;
           while j <> 0 do
                          new j <- j - 1;
                          new k <- k * j;
           return k);
```

The last control structure is functions. To make functions as powerful as procedures in conventional languages (which can, in addition, exploit side-effects), data flow languages allow functions to return multiple values, or arrays, or even both.

For the Quicksort example, an examination of ID [5] designed by Arvind and Gostelow at the University of California at Irvine, was chosen. Figure 5.1 shows the ID program for Quicksort, based on the recursive procedure sort. This procedure consists of two main statements, the first "(low,j..." which partitions the array "v" into the subsets "low" and "high"; and the second "return..." that takes the two sorted subsets and the pivot "v[1]", creates a single array "t", and returns this sorted array. The first statement is iterative with the loop variables "low" and "high" which are arrays, "j" and "k" which are indexes/counts for the two arrays, and "pivot". Initially, the arrays are set undefined, the indexes are set to zero, and the pivot becomes "v[1]".

The body of the iteration is the for-statement. It compares each element "v[i]" against the pivot, placing the element in the array "low" and incrementing "j", if less than the pivot, else placing it in "high" and incrementing "k". When all elements of "v" have been partitioned, the arrays "low" and "high" are themselves sorted if they contain more than one element. The iteration then returns the sorted arrays "low" and "high", and "j" the number of elements in "low", which is used to merge the two arrays.

Although not obvious from the iteration syntax, each loop variable must be redefined on each iteration. Thus the conditional statement would be more clearly expressed as:

```
(new low[j+1], new j, new high[k+1], new k, new pivot) <-
                    (if v[i] < pivot then v[i], j+1, high[k+1], k, pivot
                                     else low[j+1], j, v[i], k+1, pivot)
```

reflecting the data driven nature of the execution.

The result of the iterative statement is two sorted arrays, but before these can be returned by the procedure sort, they must be concatenated to form a single sorted array. This task is performed by the return statement using an iterative expression. It initially sets the array "t" to the contents of "low" with the pivot "v[1]", appended on the end (i.e. the j+1 element). Next, each element of "high" is appended to "t". Finally, notice in Figure 5.1 that two ID array operation formats are used. The normal assignment format is "new_array <- array + [index]value"; however if the statement is a redefinition then the format is abbreviated to "new array[index] <- value".

```
procedure sort (v,n)
(low,j,high <-
       (initial  low   <- ^; j <- 0;
                 high  <- ^; k <- 0;
                 pivot <- v[1]
        for i from 2 to n do
              (if v[i] < pivot
                then new low[j+1] <- v[i];
                     new j <- j+1
                else new high[k+1] <- v[i];
                     new k <- k+1)
        return (if j>1 then sort(low,j) else low),
              j,
              (if k>1 then sort(high,k) else high)

return (initial t <- low+[j+1]v[1]
        for i from 1 to n-j-1 do
            new t <- t+[i+j+1]high[i]
        return t))
)
```

Figure 5.1: Data Flow Program in ID

Whereas ID, VAL and VALID were developed for programming data flow computers, LUCID (and FP discussed below) was developed for its attractive mathematical properties and its amenability to program verification, but it is nevertheless a suitable language for data flow computation.

The advantages of data flow languages like ID and VAL are that their single assignment syntax is similar to conventional languages, that parallelism is implicitly expressed, and they are natural for programming data flow computers. Programming languages for such computers must satisfy two criteria: it must be possible to deduce the data dependencies of the program operations; and the sequencing constraints must always be exactly the same as the data dependencies, so that the activation of statements can be based simply on the availability of data.

5.1.2. Applicative Languages

Here a distinction is made between two subsidiary classes of applicative languages, which have been termed "function-level" and "pattern-matching". Next, the two classes of applicative languages are examined. In the applicative (function-level) language, the data mechanism is "message passing" and the control mechanism is "demand driven". For the data mechanism, programs deal with structured data, and do not name their arguments. For the control mechanism, the role of control structures is handled by "combining operators" that directly manipulate their arguments. For example, FP is based on the use of so-called functional forms, namely operators such as Composition ".", Insert "/", and ApplytoAll "@". In addition, there are operators providing conditional expressions "(p -> f;g):x" and iterative expressions "(while p f):x".

In applicative (pattern-matching) languages the data mechanism is "shared memory" and the control mechanism is "demand driven". For the data mechanism there are four types of argument: number, string, list, and function. For the control mechanism there are "demand driven" operators, such as "CONS", "Concatenate", and "Subrange"; conditional expressions, in which a boolean guard is written in front of an expression; and the basic concept of function application. Although the execution semantics is usually referred to as "pattern-matching", the control mechanism has been classified as "demand driven", so as to distinguish the underlying reduction semantics from the semantics of logic and object-oriented programming.

Function-Level

Function-level languages denote a class of functional programming where the role of control structures is handled by "combining operators" that manipulate functions directly, without ever appearing to explicitly manipulate data. These functional programs deal with structured data, are often non-repetitive and non-recursive, are hierarchically structured and do not name their arguments.

The best known example of a function-level applicative language is Backus' FP [9] which has superficial similarities to APL. FP is founded on the use of a fixed set of combining forms called functional forms. The most important functional forms are Composition ".", Insert "/", and ApplytoAll "@" that combine existing functions to form new ones. If "f:x" is written for the result of applying "f" to the object "x", then "f.g" is the function obtained by applying first "g" and then "f" to the argument:

```
(f.g): <x1,x2,...,xn> is f: (g: <x1,x2,...,xn>)
```

"/f" is the function obtained by inserting "f" into the arguments;

```
/f: <x1,x2,...,xn> is f:<x1,f:<x2,...,f:<xn>>...>
```

and "@f" is the function obtained by applying "f" to every member of the argument:

```
@f: <x1,x2,...,xn> is <f:x1,f:x2,...,f:xn>
```

Functional forms plus simple definitions are the only means of building new functions from existing ones. (In addition, partial results cannot be given a name.) All the functions of FP are of one type: they map objects into objects, and always take a single arguments.

Figure 5.2 shows the FP version of Quicksort (produced by a colleague, D. Mundy) consisting of five definitions. Briefly, "sort" partitions the array; "merge" takes three lists and combines them; "strip" removes nulls from a list; "null" tests for null elements; and "lt" is a "less than" conditional operator.

Definition "sort" is a conditional expression with the following meaning:

```
(p -> f;g):x is if (p:x)= true  then f:x
                if (p:x)= false then g:x
```

The initial part of sort "null->id" tests for an empty list and returns it. The latter part "merge ..." performs the partition and divides into five parts. Working from right to left, the construction: "[id,1]" extracts the pivot element:

```
[id,1]:<x1,x2,...,xn> is <<x1,x2,...,xn>,x1>
```

Next, distribute right "distr" generates a new list where a copy of the pivot is paired with each element:

```
distr: <<x1,x2,...,xn>,x1> is <<x1,x1>,<x2,x1>,...,<nn,x1>>
```

Then, the comparison operators "lt", "eq", and "gt" are applied to the three copies of the list in parallel:

```
[@(...] : < > is  <lt: <x1,x1>,...,lt: <xn,x1>>,
                  <eq: <x1,x1>,...,eq: <xn,x1>>,
                  <gt: <x1,x1>,...,gt: <xn,x1>>
```

to produce three lists containing, respectively, all the values less than, equal, and greater than the pivot.

Definition "strip" is then applied, which uses the definition "null" to test for empty sequences, and removes them from the lists. Next, the construction "[sort...]" is used to sort the partitioned subsets. And finally "merge" is applied to concatenate the sorted lists.

```
def sort = null -> id;
           merge.[sort.1, 2, sort.3].@strip.[@(lt -> 1; []),
                                             @(eq -> 1; []),
                                             @(gt -> 1; [])].distr.[id,1];

def merge = \apndr.apndl.[1, \apndr.apndl.[2, 3]];

def strip = /(null.1 -> 2; apndl).apndr.[id, []];

def null = eq.[[], id];

def lt = ge -> '0; '1;

sort (5 2 1 7 9 4 3 6 10 8 11 12 13 14 15 16)
```

Notation:	@	apply-to-all
	.	composition
	\	insert left
	/	insert right
	'	constant

Figure 5.2: Applicative (function-level) Program in FP

The APL-like programming style of function-level applicative languages is clearly very concise and powerful. However, the fact that functions do not name their arguments implies that function level programs are sometimes difficult to understand. In contrast, in the second category of applicative languages, called "pattern-matching", naming of arguments by functions is an essential ingredient.

Pattern-Matching

This category of applicative programming languages denotes a class of languages whose functions use "pattern-matching" in the binding of formal parameters and actual parameters. (Recall, this control mechanism has been classified as "demand driven" so as to distinguish the underlying reduction semantics from the semantics of logic.) Typical of the many interesting pattern-matching languages is Turner's SASL [55]. The SASL system is interactive and includes built-in commands for: editing programs, and saving them in (and retrieving them from) files, etc. In addition the user can ask to have expressions evaluated (in the environment established by the program) and the result output at the terminal.

A SASL program is a collection of equations by means of which the user attaches names to various objects. There are four types of object: numbers, strings enclosed in double quotes, lists, and functions. Numbers and strings have the normal properties one could expect, with the usual kinds of operations defined on them. Lists are written using round brackets and commas:

```
number = ( 1,2,3,4,5,6,7,8,9,10)
```

and elements of a list are accessed by indexing. For example the expression "number 3" would here give the result "3".

Important list operators include ":" (corresponding to the LISP function "CONS") which adds a new element at the front:

```
0:(1,2,3,4,5,6,7,8,9,10) gives (0,1,2,3,4,5,6,7,8,9,10)
```

"++", which concatenates two lists:

```
(1,2,3,4,5) ++ (6,7,8,9,10) gives (1,2,3,4,5,6,7,8,9,10)
```

"--", which forms the difference of two lists:

```
(1,2,3,4,5,6,7,8,9,10) -- (1,3,5,7,9) gives (2,4,6,8,10)
```

and lastly "..", which denotes the list of numbers, such as:

```
(1..10) gives (1,2,3,4,5,6,7,8,9,10)
```

In addition, SASL supports infinite structures. For example "(1..)" is the list of all natural numbers starting at 1, and the equation "x=1:x" defines "x" to be the infinite list all of whose elements are "1".

Functions are denoted by writing down one or more equations with the name of the function (followed by some formal parameters) on the left and a value for the function on the right. For instance the obligatory factorial is expressed as:

```
fac 0 = 1
fac n = n > 0 -> n * fac(n - 1)
```

The order in which equations are written has no logical significance. Where order is important a boolean "guard", such as "n > 0" above, is placed in front of an expression. More sophisticated forms of pattern-matching involve the use of list structures in formal parameter positions as illustrated in the Quicksort example.

The SASL program in Figure 5.3 consists of four equations, two performing the sort and two handling the subsidiary partitioning operation. Sort differentiates between two types of parameter, the empty list "()" and non-empty lists "(a:x)". For an empty list, the first equation returns the empty list. For a non-empty list, the second equation uses the CONS operation in the formal parameter list "(a:x)" to give the name "a" to the first element of the list and "x" to the remainder of the list. The body of this equation consists of two parts, the subsidiary definition "m,n = split a x () ()" which partitions the list "x" using "a" as the pivot, then calls sort to partition the two subsidiary lists "m" and "n", and lastly concatenates the sorted lists. The meaning of "sort m ++ a : sort n" is

```
concatenate (sort(m), cons(a,sort(n))
```

Next, the two split functions will be examined. The split differentiates between two types of "x" parameters, the empty list and non-empty lists. The empty list occurs with a one element array, in which case the list corresponding to "m" and "n" in the call are returned. For non-empty lists, the second parameter CONS is used in calling the head "b" and the remainder "x". Split then compares the extracted element "b" against the pivot "a": if less than, then "b" is inserted into the list "m" using the CONS operator ":", else "b" is inserted into "n". Then in both cases split is recursively called to extract the next element of the list "x". Notice firstly that the body of the second split operates like an IF-THEN-ELSE, and secondly there are no side-effects - the names "a", "b" etc. are formal parameters and are thus distinct in the four equations.

```
DEF
sort () = ()
sort (a : x) = sort m ++ (a : sort n)
               WHERE m,n = split a x () ()
split a () m n = m,n
split a (b : x) m n =  b < a -> split a x (b : m) n
                       split a x m (b : n)
?
```

Figure 5.3: Applicative (pattern-matching) Program in SASL

Applicative languages have an additional powerful abstraction mechanism, called the <u>higher order function</u> [30], which is a function that returns another function as result. It works as follows. If a function is defined to have say "n" arguments, it can be applied to less than "n" (say "m") arguments. In this case the result is a function of (n-m) arguments in which the first "m" arguments are "frozen in". The advantage of this abstraction mechanism is that a large number of analogous functions can be built with little extra specification.

In summary, the main feature of functional programming is basically the "clean" mathematical world of equations. Advantages are the uniformity of the structures manipulated, implicitly expressed parallelism from the "data driven" and "demand driven" control mechanism, plus the absence of: explicit sequential execution, assignment and side-effects. Specific advantages for data flow languages are the similarity of their syntax to conventional languages and their obvious qualities for programming data flow computers. Specific advantages for applicative (function-level) languages are that they are often non-repetitive and non-recursive, hierarchically structured, and do not name their arguments. Specific advantages of applicative (pattern-matching) languages are the operators for manipulating lists, and higher-order functions. In functional programming the advantages and the disadvantages relate to the same concepts. Disadvantages of functional programming are, arguably, the absence of any "control driven" execution, assignment statements and side-effects. A specific disadvantage of data flow languages is the absence of a "shared memory" data mechanism, causing problems for the manipulation of data structures. Lastly, a specific disadvantage for applicative (function-level) languages is that they do not name their arguments, making programs terse and often difficult to understand.

5.2. LOGIC PROGRAMMING

Logic programming seems to be based on a "shared memory" data mechanism and a "pattern driven" control mechanism. For the data mechanism, data consists of sets of alternative values which can be numbers or strings. For the control mechanism, execution is based on pattern-matching and substitution. "Pattern driven" execution may select a number of alternative commands, which are executed in parallel (OR-parallelism). In turn a command may be executed by evaluating all the goals in parallel (AND-parallelism) and basically only succeeds if all goals succeed.

Next, an examination of a class of logic programming based on the Horn clause subset of logic is made.

5.2.1. Horn Clause Languages

For many applications of logic it is sufficient to restrict the form of clauses to those containing at most one conclusion. Clauses containing at most one conclusion are called Horn clauses, after the logician Alfred Horn. Each clause is either an assertion or an implication. In general, every assertion is an atom "A.", whereas every implication has the form "A if B1 and B2 ... and Bn." and all conclusions "A" and conditions "B1,B2, ... Bn" are atoms, expressing a simple relationship amongst individuals. Individuals can be named by constants such as numbers "1" and strings as "tom", or by variables such as "X". The "A" part of a clause is called the "head" and the "B1,B2,...,Bn" is called the body, and is expressed in a language such as PROLOG as:

```
head :- body.
```

whose form is illustrated by Figure 5.4.

Basically, PROLOG attempts to solve goals sequentially from left to right. For a given goal, PROLOG attempts to find a clause whose head can be made to match the goal. If the clause is an implication then it, in turn, attempts to solve the subgoals. The possible results of a goal will be failure or success, plus possible values associated with variables. To achieve success for a goal, all the subgoals must succeed. If one of the subgoals cannot be solved, PROLOG backtracks and tries to find another clause whose head matches the goal. If no untried clauses remain, then failure is returned for the goal.

More detail on how PROLOG works can be found by examining the Quicksort example. It is interesting to note, in addition, the similarities between this PROLOG program and the previous SASL example. In Figure 5.4 sort differentiates between two types of parameters, the empty list "[]" for which it returns an empty list "[]", and the non-empty list "[X|L]" for which it returns the sorted list "R". For non-empty lists, the second clause uses the CONS operator "|" in the formal parameter list "[X|L]" to set the first element to "X" and the remainder of the list to "L". The body of this clause consists of four subgoals: split using "X" as the pivot partitions the list "L" into the two subsidiary lists "L1" and "L2", sort takes a list and sets its result to the sorted list, and concat takes the two sorted lists plus the pivot and concatenates them to produce the result "R" of the sort clause.

Next split's clauses will be examined. Split differentiates between three types of parameters: the empty list, the list whose first element is less than or equal to the pivot, and the list whose first element is greater than the pivot. For an empty list, split returns two subsidiary empty lists. For non-empty lists, split uses CONS to set the first element of the list to "Y" and the remainder of the list to "L". This "Y" value is then compared against the pivot "X". If "Y =< X" then the second split cause proceeds to partition the remainder of the list "L" by the reinvoking split. When this list has been partitioned into "L1" and "L2", "Y" is CONS on the first list "[Y|L1]" and the two lists are returned by the clause. If "Y > X" the second split clause fails, and execution proceeds to the third clause. It operates in a similar way, reinvoking split to partition the list "L" and using CONS to append "Y" to the front of the second result list "[Y|L2]".

The final two clauses in Figure 5.4 show the specification of concatenate. In the SASL example their role was performed by the "++" operator. In fact, the sort clauses can be specified so as to remove the need for the concat clauses, but this employs a slightly less straight forward program.

```
sort([],[]).
sort([X|L],R) :-split(X,L,L1,L2),sort(L1,R1),sort(L2,R2),concat(R1,[X|R2],R).

split(_,[],[],[]).
split(X,[Y|L],[Y|L1],L2) :-Y =< X,split(X,L,L1,L2).
split(X,[Y|L],L1,[Y|L2]) :-Y > X,split(X,L,L1,L2).

concat([],L,L).
concat([X|T],L,[X|TL]) :-concat(T,L,TL).
```

Figure 5.4: Horn-Clause Program in PROLOG

A number of additional points concerning PROLOG, but not highlighted by the Quicksort example, are worth noting. Firstly, a PROLOG program may have more than one valid result, due to similar clauses. Once the first result is obtained, each additional result is obtained by typing "?" until failure is returned. Each causes PROLOG to search a further set of possible clauses. Secondly, there is great flexibility in specifying the question asked to a PROLOG program. Thus a program can be given (what may be viewed as) the input and asked to deduce the output. Alternatively, the output can be given to the program, and PROLOG can deduce the input. Lastly, there is clearly considerable potential for exploiting parallelism in the execution of PROLOG programs. This is pursued either by evaluating concurrently all of the heads that match a goal (this is referred to as OR-parallelism since any result is acceptable) or by evaluating all of the subgoals concurrently (this is referred to as AND-parallelism since all must succeed for the goal to succeed).

In summary, the main features of logic programming are pattern-matching (unification) and substitution. Advantages of logic programming include the fact that it is the most "high level" programming model, in specifying "what" rather than "how" a computation is to be executed, and is the closest programming style to knowledge-based systems. Disadvantages of logic programming are that the notation is very concise and therefore terse, and hence difficult to understand when seen in the form of a program. In addition, the "pattern driven" mechanism can lead to a lack of control over evaluation of commands.

5.3. APPLICATION PROGRAMMING

Application programming styles contain languages covering many different pairings of data mechanisms and control mechanisms. For example, expert systems building tools [60] seem to contain a "shared memory" data mechanism and a "pattern driven" control mechanism. However, the "electronic sheet" languages (discussed below) seem best classified as having a "shared memory" data mechanism and a "data driven" control mechanism. For the data mechanism, a memory location contains a value and possibly also an expression defining the value in terms of other memory cells. For the control mechanism, execution is "data driven" but not data flow; when the value of a location is changed, all other locations that use the value are notified and recalculate their values using this new information. Besides the "data driven" evaluation, a user may specify whether recalculation is to proceed down the columns or along the rows. (This is viewed as "control driven"). Note also the absence of conditional operators, which limits the scope of programming in certain of these languages.

5.3.1. Electronic Sheet Languages

VISICALC [14] was born out of the observation that many problems are commonly solved with a calculator, a pencil and a sheet of paper. With VISICALC the computer's screen becomes a "window" which looks upon a much larger "electronic sheet". The user can scroll this window in all four directions to look at any part of the sheet.

VISICALC's sheet is organised as a grid of columns and rows. As can be can seen below, rows are numbered 1, 2, 3, etc. and columns are labelled A, B, C, and so on. At each intersection of a row and column there is a variable with a coordinate (i.e. identifier) A1, B3, C17, and so forth. Into each variable the user can enter one of three types of data: a string, a number, or an arithmetic expression. When the contents of a variable is changed, the VISICALC system automatically recalculates all the other related variables on the sheet, changing their values and displaying them on the screen if within the window.

```
                                Entry contents

                    ------------------------------------------------
Entry line          | B3              +B2 -B1                      |
Prompt line         | VALUE                                        |
Edit line           | +B2-B1                                       |
column              |         A        B          C                |
and                 |                                              |
row labels          | 1    COST       600                          |
                    |                                              |
                    | 2    SALE       650                          |
                    |                -----                         |
cursor              | 3  PROFIT    |  50 |                         |
                    |                -----                         |
                    | 4                                            |
                    |                                              |
                    | 5                                            |
                    |                                              |
                    | 6                                            |
                    | ...                                          |
                    ------------------------------------------------
```

Figure 5.5: VISICALC Screen (abbreviated contents)

Figure 5.5 presents an abbreviated layout for the VISICALC screen. The screen consists of two basic areas: the "control panel" consisting of three lines at the top, and the "window" at the bottom. Making up the control panel are the entry line, the prompt line, and the edit line. Information displayed on the entry line gives a full explanation concerning the variable highlighted by the cursor, including its name (i.e. coordinates), its contents, and the type. On the prompt line is displayed the type of entry VISICALC thinks you are making, and on the edit line is the actual input typed by the user. VISICALC is "syntax-directed"; each time the user presses a key, VISICALC displays on the prompt line what can be typed next.

Operations in VISICALC are either editing commands that manipulate the contents of the screen, or built-in functions and operators that may be used in arithmetic expressions. Commands include operations for clearing a specific variable, row or column or the whole screen; for moving information between the screen and file; for replicating the contents of variables; and for printing. These commands are entered in the edit line. Built-in functions, as might be expected, provide generally useful operations such as minimum value "@MIN", sine "@SIN", and because it is a financial-modelling system, the net present value "@NPV(dr,range)" of the cash flows in "range", discounted at the rate specified by expression "dr". These built-in functions are used with the arithmetic operators (+,-,etc.) in the expressions stored in variables.

Execution, or recalculation as it is called in VISICALC, occurs each time a variable is changed. VISICALC recalculates by starting at the upper left-hand corner of the sheet and working its way downward and to the right until it reaches the lower right-hand corner of the sheet. However, the system allows the user to select either of two possible orders: "down the columns" or "across the rows" first.

As with previous programming languages, an attempt was made to program Quicksort in VISICALC but this seems impossible, which is not surprising since VISICALC does not provide comparison operators or conditional expressions. However, a reasonably respectable sort was coded, using the built-in functions "@MIN" and "@MAX", operating on a list of numbers, as shown in Figure 5.6. In this example, the numbers to be sorted are inserted into successive locations in column "A", providing the inputs for the expressions in column "B". Each expression compares two adjacent numbers and exchanges them if necessary. This process is repeated, using varying separations, in columns "C", "D", "E" etc., causing the results to move left to right on the sheet. By presetting the numbers in column "A" to be largest number, and placing the appropriate expressions in columns "B", "C" etc., a sort file can even be obtained to handle variable size arrays.

	A	B	C	D	E	
1	n1	@MIN(A1,A2)	B1	@MIN(C1,C2)	D1	
2	n2	@MAX(A1,A2)	@MIN(B2,B3)	@MAX(C1,C2)	@MIN(D2,D3)	...
3	n3	@MIN(A3,A4)	@MAX(B2,B3)	@MIN(C3,C4)	@MAX(D2,D3)	
4	n4	@MAX(A3,A4)	@MIN(B4,B5)	@MAX(C3,C4)	@MIN(D4,D5)	
5	n5	@MIN(A5,A6)	@MAX(B4,B5)	@MIN(C5,C6)	@MIN(D4,D5)	
6	n6	@MAX(A5,A6)	@MIN(B6,B7)	@MAX(C5,C6)	@MIN(D6,D7)	
7	n7	@MIN(A7,A8)	@MAX(B6,B7)	@MIN(C7,C8)	@MAX(D6,D7)	
8	n8	@MAX(A7,A8)	@MIN(B8,B9)	@MAX(C7,C8)	@MIN(D8,D9)	

• • •

Figure 5.6: "Electronic Sheet" Program in VISICALC

Advantages of electronic sheet languages include the simple "data driven" programming model, the "electronic sheet" user-friendly input/output, and the ability to specify if evaluation is by columns or rows. Disadvantages relate to the simplicity of the system, such as the limited range of operators, which restricts the scope of the language.

Having classified and analysed the major programming styles, a decentralised programming model will be chosen, in the next chapter, for future computers.

CHAPTER 6 - DECENTRALISED CONTROL FLOW MODEL

This chapter presents the decentralised control flow model, and justifies this choice by summarising the conclusions taken from: Chapter 2 (Decentralised Computer Systems), Chapter 3 (Classification of Programming Languages), Chapter 4 (Analysis of Procedural and Object-Oriented Programming), and Chapter 5 (Analysis of Functional and Logic Programming).

6.1. CHOOSING A PROGRAMMING MODEL

In Chapter 2, two possible "images" for future decentralised computers were presented, namely a parallel machine - consisting of identical powerful sequential processors, and a decentralised computer - consisting of the minimum principles that distributed, parallel and sequential computers must obey so that they can work together as a system. The conclusion taken from Chapter 2 was that future computers will require a decentralised computer image.

In Chapter 3, the fundamental computational (data and control) mechanisms that are believed to underlie programming languages were presented. In Chapters 4 and 5 these data and control mechanisms were used as a basis for analysing the major programming styles (i.e. procedural, object-oriented, functional and logic) and their underlying programming models (i.e. control flow, actor, data flow, reduction, and logic).

Significantly (as shown by Figure 3.2) each category of programming model regards the data mechanisms and the control mechanisms as largely incompatible sets of alternative concepts. (For example, control flow models use "shared memory" and are "control driven", whereas data flow models use "message passing" and are "data driven".) Hence each category of programming, although Universal in the sense of a Turing machine, has specific advantages and disadvantages for computation, related to its choice of mechanisms. Two additional observations should be made concerning the choice of mechanisms. Firstly, categories of programming models supporting "message passing" data mechanisms seem inevitably to also include a subsidiary mechanism for "shared memory". Secondly, categories of programming models supporting "data", "demand" and "pattern driven" control mechanisms frequently have a subsidiary "control driven" mechanism, arguably to alleviate control problems [7].

Given the above considerations, then for computation "shared memory" seems the fundamental data mechanism and "control driven" execution seems the most primitive control mechanism. Using these mechanisms, it is relatively easy to implement and support the other mechanisms; the reverse appears not to be true. Control flow (and procedural programming) would seem therefore to embody the most fundamental computational concepts.

The actual choice of programming model (see Figure 1.1) for future decentralised computers ranges from a low-level model, such as control flow, that specifies exactly <u>how</u> an algorithm is to be executed, to a higher-level model, such as logic, that merely specifies <u>what</u> algorithm is to be performed. Thus, for general-purpose computation, the essential choices are:

1. low-level model
   - flexibility of mechanisms
   - control over execution

   e.g. control flow

2. high-level model
   - powerful abstraction mechanisms
   - safeness of programs

   e.g. logic, reduction

This choice is analogous to that between the efficiency (but hazards) of assembly languages, and the power (but constraints) of high-level languages. High-level programming models are particularly attractive for languages, since they help manage software complexity. High-level (language) computers, on the other hand, have not been particularly successful, due to the spectrum of applications to which general-purpose computers are applied. The programming model for a computer, which is implicitly required to support an open ended set of programming styles has, as its main requirement, to be flexible and unobtrusive. This Thesis is concerned with the programming of decentralised computers, and therefore, is oriented to a low-level programming model for computers.

Thus, below, a control flow programming model is presented that embodies a "decentralised computer". This is referred to as the decentralised control flow model.

Next, a description of the principles of the decentralised control flow programming model is given, which is based on an earlier, highly recursive control flow model [51]. The recursive control flow model is closest in concept to reduction machines, having a highly recursive view of both computer and program structures. Operations are viewed as "editing" the program structure, causing programs to dynamically migrate between component machines. In contrast, the decentralised control flow model attempts to extend and generalise the von Neumann model for programming decentralised computers. These so-called principles should be viewed (and are described) as the "virtual machine" underlying decentralised control flow programming.

6.2. PRINCIPLES

A good way of starting is by contrasting the decentralised control flow model with the von Neumann model. In the von Neumann model the main principles are:

1. computer - a computer system comprising a processor and a memory;
2. memory - a linear organisation of fixed-sized memory cells;
3. addressing - a one-level address space of cells;
4. program - a low-level machine language;
5. communication - shared memory;
6. execution - sequential, centralised control of computation.

In the von Neumann model, a computer system comprises a vector of memory cells and a single processor. Each memory cell may contain just one elementary object (data or instruction) and has a unique address. The processor uses this address to perform a LOAD, STORE or EXECUTE operation on the contents of a memory cell.

For a future generation of decentralised computers it is clearly appropriate to transfer into the computer's architecture the fundamental mechanisms of high-level languages (e.g. structured memory) and operating systems (e.g. filestore, contextual addressing, processes). Transferring these mechanisms from software to hardware should lead to more efficient representation and execution of programs.

In the decentralised control flow model, the main principles are:

1. computer - a computer system is a decentralised computer (a hierarchy of distributed, parallel and sequential computers);

2. memory - a nested organisation of variable-size memory cells (like the file structure of an operating system);

3. addressing - contextual address space of cells (like telephone numbers);

4. program - a high-level machine language (as in LISP, where instructions may be recursively defined);

5. communication - shared memory and message passing;

6. execution - parallel, decentralised control of computation (as with UNIX commands).

An essential concept in the decentralised control flow model is the direct functional correspondence between the physical system and the logical information structure of the computer system:

Hardware

```
         1                2                3
   -----------      -----------      -----------
  | processor |<==>| processor |<==>| processor | ...
  |-----------|    |-----------|    |-----------|
  | memory    |    | memory    |    | memory    | ...
   -----------      -----------      -----------
```

Software

```
     -----------      -----------      -----------
  1:|           | 2:|           | 3:|           | ...
     -----------      -----------      -----------
```

Figure 6.1: Functional Correspondence of Hardware and Software

In the system, the memory of each computer is viewed as a memory cell whose address is the hardware address of the computer. Inside a memory, further memory cells are represented. (Thus each computer has a unique address and operates like a memory bank belonging to a global memory.)

The memory in such a computer allows each memory cell to contain a vector of subsidiary memory cells. For instance, an array of the numbers 0 to 9 can be represented by one memory cell containing the array, and subsidiary cells containing the individual numbers:

```
 ------------------------------------------------------------
|  ---   ---   ---   ---   ---   ---           ---  |
| | 0 | | 1 | | 2 | | 3 | | 4 | | 5 |   . . .  | 9 | |
|  ---   ---   ---   ---   ---   ---           ---  |
 ------------------------------------------------------------
```

Addressing in the model is based on each memory cell being considered a context and each subsidiary cell having a unique "selector" within this context:

```
 -      ----------------------------------------------      -
|    |        ---      ---      ---               ---   |      |
| 3: |   1:| 0 |  2:| 1 |  3:| 2 |     . . .   10:| 9 |  |  4:  |
|    |        ---      ---      ---               ---   |      |
 -      ----------------------------------------------      -
```

Thus, within the surrounding context, the address "3" may be used to access the whole array of numbers, while the address "3/2" allows access to the cell containing the number "1".

Lastly, there is communication of data. In traditional computers two operations may be performed on the contents of a memory cell: STORE and LOAD. STORE overwrites the content of the accessed cell, and LOAD takes a copy of the cell's content. These operations support the shared memory semantics. To support message passing, and to integrate it with shared memory, two additional operations may be performed on the contents of a memory cell: PUT and TAKE.

```
              shared memory                    message passing
              -------------                    ---------------

               STORE addr                         PUT addr
                    |                                 |
                 --------                          --------
         addr:  |        |                 addr:  |"empty" |
                 --------                          --------
                    |                                 |
               LOAD addr                          TAKE addr
```

Figure 6.2: Memory Operations - LOAD, STORE, TAKE and PUT

PUT may only store into an "empty" memory cell and TAKE may only remove a non-empty contents, replacing it with "empty". Both operations may be viewed as polling a memory cell until the cell is in the correct state. (Note STORE and LOAD operations are unaffected by a cell being "empty".)

Illustrations of the various possible levels of implementation of the decentralised control flow model are provided by the Newcastle Connection and the RIMMS multi-microcomputer, discussed in Chapter 2. For a more "idealised" computer implementation, each of the principles will be examined in turn. A good basis for such an implementation is provided by the Recursive Machine proposal of Barton and Wilner [62], which initially inspired the following "virtual" machine.

6.3. COMPUTER SYSTEM

A decentralised control flow computer system, as illustrated in Figure 6.3, has a hierarchical structure, with each computer system being composed of a network of computers. Each component computer system operates like a memory cell servicing a primitive set of operations (LOAD, STORE, ...). The contents of a memory cell are manipulated by the associated processor. This associated processor may be accessed using its address by any computer in the contextual address space.

Hardware

```
                               2:
 ----------------------------------------------------------
|                     P  R  O  C  E  S  S  O  R             |
|      1:              2:              3:           4:      |
|   ---------       ---------       ---------     --------- ... |
|  |processor|<=>|processor|<=>|processor|<=>|processor|    |
|  |---------|   |---------|   |---------|   |---------|    |
|  |memory   |   |memory   |   |memory   |   |memory   |    |
|   ---------     ---------     ---------     ---------     |
|                     M  E  M  O  R  Y                      |
 ----------------------------------------------------------
```

Software

```
2:(  1:(        )  2:(        )  3:(        )  4:(        )...  )
```

Figure 6.3: Decentralised Control Flow Computer

As illustrated in Figure 6.3 there is a direct correspondence (at the higher levels) between the physical system structure and the logical information structure of the computer system. This is particularly important for programming, since each computer system in the hierarchy may be programmed as a single computer, and accessed by other computers using its address as if it were a simple memory cell storing a single object. In addition, a component computer is not constrained to provide a general-purpose service; it might in fact be special-purpose even to the extent of being a traditional memory cell.

6.4. INFORMATION STRUCTURING

Memory in the model consists of a nested organisation of variable-size memory cells. Such a memory could be implemented by traditional LISP cells; however, delimited strings seem a more "idealised" implementation.

When information is represented as nested delimited strings, a delimited string is considered a recursively-defined, variable-size memory cell. All information stored in a decentralised control flow computer forms a single delimited string. Thus the computer's memory is logically like the hierarchical file structure of most multi-user operating systems. A string consists of two alphabets of characters, namely (i) characters that delimit strings, and (ii) data characters that form strings. For example, using brackets as delimiters, the array of the numbers 0 to 9 can be represented as:

((0) (1) (2) (3) (4) (5) (6) (7) (8) (9))

In the computer, the explicit delimiting characters would be left bracket "(" and right bracket ")", and the data characters are binary "0" and "1". Thus an array of the numbers 0 to 9 would be represented as:

((0) (1) (10) (11) (100) ... (1001))

It is unnecessary, however, to make all delimiters explicit. For example, if a particular machine implementation used conventional fixed word size memory cells, then implicit brackets may be viewed as occurring on word, byte and even bit boundaries. But the implementation would then restrict the usage of these lower level strings. Thus the allocation of memory cells is context dependent; depending on whether the cell corresponds to a "physical" computer or to a "virtual" delimited string. (A virtual cell is created by an access to an undefined structure, and is initially empty.)

6.5. ADDRESSING SCHEME

Addressing of information is based on the concept of context, which is the model for references in operating system filestores and for telephone numbers in the telephone network. As with telephone numbers in the international telephone network, an address is variable-length depending on the path between the point of reference and the target memory cell.

In the contextual address space, each memory cell (i.e. delimited string) in the information structure is considered a context relative to which a related cell is identified by a selector, such as an integer in the range 1..n. An address is a sequence of selectors specifying a "path" from the point of reference in the structure to the target memory cell. Each selector identifies a memory cell relative to the current context, and moves the remainder of the address to the new context for its further interpretation. Example classes of selectors provided by the model could be: _direct_ where "i" is the local address of a memory cell; _computed_ where the result of a command "fn" is "i" which is the local address of the cell, and _superior_ where ".." defines the surrounding memory cell.

For instance, to access the whole of the array shown on Section 6.4 above, from elsewhere in the surrounding context its selector, say "2", is used, whereas to access a subsidiary number "(100)" the address "2/5" is used.

```
          2:( 1:(0) 2:(1) 3:(10) 4:(11) 5:(100) ...   )
    2   =   (   (0)   (1)   (10)   (11)   (100) ...   )
    2/5 =                                 (100)
```

Using a memory cell's address any of five system-wide operations may be performed on its contents (i.e. LOAD, STORE, PUT, TAKE, EXECUTE). Thus both "shared memory" and "message passing" data mechanisms are supported by the model.

6.6. PROGRAM REPRESENTATION

Program representation in the decentralised control flow programming model is based on a single format. Each program object (i.e. executable delimited string) is stored in a memory cell and consists of a list of commands separated by controls:

```
(command  control  command  control  command  ... )
```

A control defines the order of execution of two adjacent commands which may be sequential or parallel, etc. A command consists of a list of arguments:

```
(arg0  arg1  arg2  arg3  arg4  arg5 ... )
```

The leftmost argument in each command defines the task or operation to be performed, and also the interpretation of the remaining arguments which are its parameters. A task or operation may be a simple operator such as "+":

```
(operator  arg1  arg2  arg3  arg4 ...)
```

or the address of a procedure to be called:

```
(address  arg1  arg2  arg3  arg4 ...)
```

Parameters are accessed like the contents of a memory cell by a command.

An argument in a command is either the actual object or its corresponding address:

```
        ( object )        or        address
```

Examples of how the interpretation of each instruction is defined systematically by the leftmost argument are shown in Figure 6.4 (where lower level delimiters and, in some cases, mode information have been omitted for clarity).

```
instruction:

     (arg0  arg1  arg2  arg3  arg4 ...)

examples:

     typed operand       (literal 26)

     expression          (+ a b)
   (returning its result)

    expression           (+ a b c)
   (storing its result)

    sub-program          (sqrt c)
```

Figure 6.4: Program Representation

The information provided by "arg0" includes the number of arguments, their order of evaluation (which may be in parallel), and whether they are used for input or output.

6.7. PROGRAM EXECUTION

Program execution in the decentralised control flow model is clearly "control driven", but is parallel. Each list of commands and each subsidiary command may be executed concurrently. The "process" managing the list of commands controls the invoking of its commands, and execution of a command is analogous to a function call.

```
"procedure call"  (arg0  arg1  arg2  arg3  arg4 ...)
"procedure body"  arg0:(    ...    )
```

The "control driven" execution of commands is specified by the controls. Once a command is invoked, its arguments are evaluated on demand under the control of the leftmost argument. The operation is executed as the results of the evaluated arguments become available.

6.8. OTHER PROGRAMMING MODELS

One of the aims of the decentralised control flow model is to introduce computational mechanisms from other programming models to make control flow more general-purpose. For instance, because of the five system-wide operations discussed in Section 6.5, the model is able to support the "shared memory" and "message passing" data mechanisms described in Chapter 3.

The actual data mechanisms supported by a particular programming model are closely tied to the types of argument that may be specified in its commands. Figure 6.5 illustrates an interpretation of the union of argument types that are found, for example, in control flow, data flow and reduction programming models. In Figure 6.5, commands are represented by delimited strings and the argument type is underlined.

Type	Meaning
literal	the literal type e.g. (+ 2 ...) is found in every programming model
unknown	the unknown type e.g. (+ ( ) ...) is used, logically, by data flow as a "place-holder" for dynamically generated values and delays evaluation until the value is available for the command
address	the address e.g. (+ a ...) is a reference used by control flow commands to load or store a value value
procedure call	the procedure call e.g. (p ...) is a reference used by reduction and causes the evaluation of the addressed operand
expression	the expression type e.g. (...(+2 a)...) is used by reduction for the nesting of commands

Figure 6.5: Spectrum of Argument Types

Similarly, the actual control mechanisms supported by a particular programming model are closely tied to the implicit control structures found in any model. Decentralised control flow has an implicit "control driven" execution mechanism. However, "data" and "demand driven" control mechanisms are also supported. "Data driven" execution is supported by using the empty memory cell "()" to delay execution until the argument is available. "Demand driven" execution is supported by the built-in procedure call, discussed above.

Finally, various developments at the University of Newcastle upon Tyne can be seen as based on a decentralised control flow programming model: the Newcastle Connection distributed UNIX system [16] (briefly described in Chapter 2); the LEGO recursive computer architecture [51], and the RIMMS Multi-Microcomputer System [25] (introduced in Chapter 2, and described in more detail in Chapter 9). In addition, the decentralised control flow programming model provides the foundation of the BASIX and the BASAL programming languages described in the ensuing chapters. BASIX and BASAL are vehicles for exploring the decentralised control flow programming style, rather than proposed new languages.

CHAPTER 7 - BASIX PROGRAMMING LANGUAGES

This chapter presents the BASIX programming languages, which are used to investigate languages embodying the full decentralised control flow model.

7.1. DESIGN PHILOSOPHY

The BASIX languages were designed to embody the decentralised control flow model, showing thus that the concepts of traditional computing could be effectively associated with more "revolutionary" concepts (such as those found in LISP and in the Shell of the UNIX operating system). This should be done by extending and generalising the traditional control flow programming style, which is a subset of the decentralised control flow model. The aim of the two BASIX languages, BASIX_1 and BASIX_2 described below, is to "mirror" the decentralised control flow model. Recall, these languages are primitive and are low-level system programming languages (cf. C) rather than high-level languages (cf. PROLOG). Thus the languages may appear rudimentary; for instance, they do not contain any data typing. In the syntax "{}" defines zero or one occurrence and "{}..." defines zero or more occurrences of the enclosed string.

7.2. BASIX_1 LANGUAGE

BASIX_1 represents the first attempt to design a language based on decentralised control flow. Its design is based on the BAS language supplied with UNIX.

BASIX_1's syntax (see Appendix A.1) may be viewed as a superset of BASIC, but it attempts to incorporate features from LISP and from UNIX. BASIX_1's commands can be simple statements as in BASIC, or can be delimited groups of statements or commands as in LISP. BASIX_1's environment attempts to be similar to UNIX: when BASIX_1 is invoked, the user program has access to any "files" - viewed as data structures by the program - previously created. If any "name" argument is provided when BASIX_1 is invoked, the structures associated with this "name" are used for input before reading commands from the terminal.

The description of BASIX_1 can be divided in four levels: i) commands; ii) statements; iii) expressions; and iv) names.

Commands in BASIX_1 are of four types:

```
statement
integer statement
(command{command}... )
integer(command{command}... )
```

"Statements" in BASIX_1 are immediately executed. "Integer statements" in BASIX_1 are known as internal commands stored for later execution, in sorted ascending order. The "(command{command}...)" commands are executed when the ")" is reached, and the "integer(command{command}...)", similarly to the integer statements, are stored for later execution.

Statements in BASIX_1 are very similar to BASIC, being either an expression or a command whose leftmost argument is a keyword:

```
comment
dim alphanumeric(integer {, integer}...)
done
dump
for name = expression expression statement
for name = expression expression
  ...
next
fork expression
join expression
goto expression
if expression statement
if expression
  ...
{ else
  ... }
fi
let name = expression
list {expression} {expression}
print list
prompt list
return {expression}
run
save {expression} {expression}
expression
```

The statement "comment" is ignored, being used only to interject commentary in a program.

The statement "dim alphanumeric (integer {, integer }...)" creates either temporary or semi-permanent data structures. When used in the form "dim alphanumeric (integer {, integer }...)" it creates a semi-permanent data structure ("file") which will not be deleted at the end of the program. When used in the form "integer dim alphanumeric (integer {, integer }...)" it creates a temporary data structure which will disappear at the end of the program.

The statement "done" returns control to system level, and in "dump" the name and current value of every variable is printed.

In "for name = expression expression statement" and "for name = expression expression ... next", the "for" statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression.

The statements, "fork" and "join", represent a new addition to a BASIC-like syntax. In the "fork expression", the expression is evaluated, truncated to an integer and a secondary thread of execution starts at the corresponding integer numbered command. The primary thread of execution continues to execute the statement following the "fork". If executed from immediate mode, the internal statements are compiled first. In the "join expression" statement the expression is evaluated and truncated to an integer. This positive integer defines the number of threads of control to be received by the "join" before sequential execution (of the following statement) is resumed.

In the "goto expression", the expression is evaluated, truncated to an integer and execution goes to the corresponding integer numbered statement. In "if expression statement" and "if expression ... { else ... } fi", the "if" statement (first form) or group of statements (second form) is executed if the expression evaluates to non-zero. In the second form, an optional else allows for a second group of statements to be executed when the expression evaluates to zero.

The statement "let name = expression" is the assignment statement. The left operand must be the name of a variable or an array element. The result is the right operand. Assignment binds right to left.

The statement "list { expression } { expression }" is used to print out the stored internal arguments. If no arguments are given, all internal statements are printed. If one argument is given, only that internal statement is listed. If two arguments are given, all internal statements inclusively between the arguments are printed. In "print list" the list of expressions and strings are concatenated and printed. (A string is delimited by " characters), and the "prompt list" statement is the same as print except that no newline character is printed.

In "return { expression }" the expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned. In the "run" statement control is passed to the lowest numbered internal statement.

The "save { expression } { expression }" statement is like "list", except that the output is written on the file argument specified in the call of BASIX_1 (i.e. "BASIX_1 {name}"). And, finally, "expression" is executed as described below.

<u>Expressions</u> in BASIX_1 can be of six different kinds:

```
number
name
(expression)
_expression
expression operator expression
name()
```

A "number" is used to represent a constant value, and is written in FORTRAN style, containing digits, an optional decimal point, and possibly a scale factor consisting of an "e" followed by a possibly signed exponent. The expression "name" is used to specify a variable.

In an "(expression)", parentheses are used to alter normal order of evaluation. In "_expression", the result is the negation of the expression. In "expression operator expression", common functions of two arguments are abbreviated by separating the two arguments by an operator denoting the function. A complete list of operators is given below. In "name()" procedures or functions can be called by an expression followed by parentheses. The name yields an integer which represents the line number of the entry of the function in the internally stored statements.

Names, in BASIX_1, have the following format:

```
0 (zero)
alphanumeric
expression
name|name{|name}...
```

When using the "0" the current context becomes the selected variable. An "alphanumeric" is used to specify a variable in the current context. Alphanumerics are composed of a letter followed by letters or digits. The "expression" is evaluated to an integer and used as a selector (i.e. index) for the name. The last form of "name" is "name|name{|name}...". It indicates a sequence of names separated by bars, which show the changes of context. This concept is based on the way the UNIX system [16] handles its directories and files. (Note: Due to the problem of distinguishing between a number such as "1" and selector such as "1", "0" and "expression" cannot occur as the only selector of a name.)

Operators in BASIX_1 contain two logical operators ("&", which is the logical AND, and "V", the logical OR); six relational operators (< <= > >= = <>), and five arithmetical operators (+ - * / **). The operator "&" has result "one" if both its arguments are non-zero. "V" has result "zero" if both of its arguments are zero. It has result "one" if either of its arguments are non-zero. The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, = equal to, <> not equal to) return "one" if their arguments are in the specified relation. They return "zero" otherwise.

As an illustration of the BASIC-like syntax of BASIX_1, Figure 7.1 shows a program for Quicksort. This program uses an iterative algorithm and a stack to store the pairs of indices "lo" and "hi" of the subsets to be partitioned. It is close in structure to the PASCAL program in Figure 4.1. The main point of interest in Figure 7.1 is the address selectors used to access the array "stack".

```
01 dim v(16)
02 dim stack(4,2)
03 let stackptr = 1
04 let stack|(stackptr)|1 = 1
05 let stack|(stackptr)|2 = 16
06 let lo = stack|(stackptr)|1
07 let hi = stack|(stackptr)|2
08 let stackptr = stackptr - 1
09 if lo < hi
10   let i = lo
11   let j = hi
12   let pivot = v|(lo)
13   if ((j <= i) V (v|(j) < pivot)) goto 16
14   let j = j - 1
15   goto 13
16   if ((i > j) V (i = j) V (V|(i) > pivot)) goto 19
17   let i = i + 1
18   goto 16
19   if (i >= j) goto 23
20   let temp = v|(i)
21   let v|(i) = v|(j)
22   let v|(j) = temp
23   if i < j goto 13
24   let v|(lo) = v|(i)
25   let v|(i) = pivot
26   let stackptr = stackptr + 1
27   let stack|(stackptr)|1 = lo
28   let stack|(stackptr)|2 = i - 1
29   let stackptr = stackptr + 1
30   let stack|(stackptr)|1 = i + 1
31   let stack|(stackptr)|2 = hi
32 fi
33 if (stackptr >= 1) goto 06
34 end
```

Figure 7.1: Quicksort Program in BASIX_1

Finally, BASIX_1 is implemented by a translator, programmed by the author, and an interpreter (supporting the virtual machine), programmed by David Mundy. The translator, written in PASCAL, is very "conventional", and does not warrant further description.

This initial version of BASIX is close to conventional languages. In addition, BASIX_1 contains a number of major problems, such as: it does not support a "message passing" data mechanism, the FORK/JOIN parallelism constructs proved difficult to use in practice, and "files" may only contain data not code. Improving on this initial version of BASIX, is BASIX_2, which is described in detail below.

7.3. BASIX_2 LANGUAGE

BASIX_2 [27,28], like BASIX_1, has a decentralised control flow programming model, and attempts to encompass more of the model. BASIX_2 attempts to be a more "sophisticated" language than BASIX_1. For instance, BASIX_2 has a single notion of object which serves the roles of variables, lists, messages, programs, files and directories. A number of long-term goals were set off for BASIX_2. Firstly it should be an interactive language providing a complete programming environment as with an object-oriented language such as SMALLTALK [4]. Secondly its semantics should aim to be as simple as BASIC. Thirdly BASIX_2 should aim to be as modular and extensible as LISP. Lastly, it should have control structures for processes such as those mechanisms found in the UNIX Shell. The complete syntax of BASIX_2 is given in Appendix A.2.

In the design of BASIX_2 it is envisaged that all users will share the same information structure and interact with the structure via their terminal screens. A user will have access to one or more current contexts, with the contents of each context being displayed as a "window" on the user's terminal (currently only a single window is supported). A window corresponds to a virtual computing system and displays the object stored by its memory cell. A window as shown in Figure 7.2 is divided into three areas defining the name of the current context, the context's information structures, and the commands being executed.

```
                 --------------------------------
                 | Context :                      |
          --------------------------------        |
          | Context :                     |       |
   --------------------------------       |       |
   | Context :                     |      |       |
   |                               |      |       |
   | name :  (    .    .    .    ) |      |       |
   | name :  (    .    .    .    ) |      |       |
   | name :  (    .    .    .    ) |      |       |
   |                               |      |-----
   |   .   .   .                   |      |
   |                               |-----
   | Command :                     |
   --------------------------------
```

Figure 7.2: Terminal "window" displaying a current context

Information in any of these three areas may be changed by positioning the cursor and typing the new information. A new context name changes the current context of the window and thus the contents displayed. New information changes the contents of the context, but does not cause execution. Lastly, a new user command is executed as a parallel process.

Information Structure

Information is represented as a single nested structure merging the concepts of directory, file, array, variable, message, and program etc. Each is a named object whose specific semantics is defined by which of the five system-wide operators (LOAD, STORE, ...) is performed on the object. A named object (i.e. the contents of a memory cell) may be accessed as "shared memory", as "message passing", or as "program".

These are distinguished in the language in the following ways:

Semantics	Operation	Usage of Name
shared memory	LOAD	... name ...
	STORE	name := ...
message passing	TAKE	... name[] ...
	PUT	name[] := ...
program	EXECUTE	name object ...
	EXECUTE	name(...)

Initially a named object is empty "()" and information is inserted either by a STORE or PUT operation. However should an empty memory cell have a TAKE or EXECUTE operation performed on it then the access is delayed until the information is inserted.

Names

A name consists of one or more selectors "{/}selector{/selector}" defining a path to the target object. Selectors are interpreted left to right, each selector moving the remainder of the name to an adjacent context. A selector may be: (i) an alphanumeric character string, (ii) a numeric character string, (iii) a bracketed object whose execution yields the selector, or (iv) a character defining one of the four accessible contexts:

Context	Character	Explanation
local	.	local objects of a program
parameters	$	parameters of a called program
non-local	..	non-local object of a program
current	/	current context i.e. the "directory" of the program; this character may optionally occur at the start of a name.

For example "$" is used to access standard input "$/I", standard output

"$/0", and the parameters "$/1", "$/2" ... of a process:

```
      $:( I:input arg0 1:arg1 2:arg2 3:arg3 4:arg4  ...  0:output )
```

A number of additional points should be noted. Firstly, a numeric string or a bracketed object may not be currently specified as the first or only selector of a name, due to the problem for example of parsing "10" the number and "10" the name. Secondly, that for the object:

```
                    "a:( ... i:(9) 9:(20) ... )"
```

the name "a/i" gives the "i" component "9", whereas the name "a/(i)" uses the contents of "i" as the selector to give "20". Lastly, as with most languages and operating systems, BASIX_2 automatically searches its four accessible contexts for a selector, in the order: local, parameters, current and non-local contexts [28].

Program Representation

Any program consists of "command { control command }...", a list of commands separated by control symbols. The "control" symbols, based on UNIX, define the order of execution of the two adjacent commands, which may be sequential ";", pipelined "|" or parallel "&". They also define how the standard inputs and standard outputs of the commands are connected together. BASIX_2 accepts commands of the form:

```
                    name : object
                    object
```

The first command is a declaration used to create and label an object relative to the current context. Only the "name" is evaluated before the assignment. The second command is immediately executed and either returns some value to the user's screen or makes some change to the information structure.

The description of BASIX_2 programs can be divided in three levels: i) objects; ii) expressions; and iii) statements.

Objects in BASIX_2 have the following syntax:

```
expression
statement
( object { object }... )
(command { control command }... )
```

An "expression" in BASIX_2 is a sequence of statements or objects separated by operators. A "statement" is a list whose leftmost object is a keyword. An "(object { object }...)" is a list of one or more objects, data or program, separated by spaces or commas. Lastly, a "(command { control command }...)" is a series of commands separated by controls, each control defining the order of execution of the two adjacent commands. Thus, an object may be any recognisable construct such as:

Construct	Example
expression	a + b - c
name	x/y/1
number	10
data structure	(a 10 (11 12))
function call	f(d, e) or f d e
program	(merge a1 a2 a3 a4 a; sort a b)

and an executable object is a list of objects separated by blanks where "blank" may be spaces or a comma. The leftmost object of the list defines the task to be performed. There are basically three types of executable objects:

	BASIX 2 Format	Example
procedure call	object object { object }...	sort infile outfile
statement	keyword { object }...	if a > b ...
expression	object { operator object }...	c + d

Expressions in BASIX_2 have the following syntax:

```
number
name
name[]
()
quote object
_object
object operator object
name( { object }... )
object object { object }...
```

In the expression "number", the object is an integer number. In "name" the object, synonymous with the name, is treated as a variable. In "name[]", the object, synonymous with the name, is treated as a list or a message. The undefined object is represented by "()", and an access to it is delayed until its contents are available. In "quote object", the result is the unevaluated object. In "_object", the result is the negation of the expression. In "object operator object", the objects are evaluated as operands for the operator, and the whole expression returns a value. In "name({ object }...)" a procedure or function with zero or more parameters may be specified in the traditional way as a name followed by the parameters in parentheses. The parameters may be separated by spaces or commas. Lastly, in "object object { object }...", a procedure or function with one or more arguments may be specified as an UNIX-like command.

Statements in BASIX_2 have the following syntax:

```
(* commentary *)
if { object -> object; }... { object } fi
do { object -> object; }... { object } do
for alphanumeric = object do object rof
goto name
cd name
rm name { name }...
```

The statement "(* commentary *)" is ignored, being only used to interject commentary in a program.

Conditional and repetitive statements centre on the conditional "object -> object" which specifies that the second object is only executed if the result of the "object ->" is true. The command "if...fi" consists of a list of commands which execute in turn until a conditional is true. This command may be used in the following ways:

Traditional Construct	BASIX 2 Format
IF ... THEN ...	if object -> object fi

IF ... THEN ...
ELSE ...
	if object -> object;
	object fi

IF ... THEN ...
ELIF ...
ELSE ...
	if object -> object;
	object -> object;
	object fi

The command "do...od" consists of a list of commands which execute repeatedly until no conditional is true. The statement may be used in the following ways:

Traditional Construct	BASIX 2 Format
WHILE ... DO ...	do object -> object od
REPEAT ... UNTIL ...	object ; do object -> object od

The command "for...rof" has the following format

```
for alphanumeric = object do object rof
```

and is intended to combine the traditional iterative command "for i = 1 to n do ..." with a command that replicates, such as the "SEQ i = [1 FOR n] ..." of OCCAM. This "for" command evaluates the first "object" and then replicates the second "object" replacing "alphanumeric" for each component of the resulting object. By using a quoted 'object list', which returns an unevaluated object, or "to" operator, that generates sequences, the statement may be used in the following ways:

Traditional Construct	BASIX 2 Format
FOR i := 1 TO n DO a[i]:=0;	for i = 1 to n do a/i:=0 rof
FOR i IN a b c d DO i:=0;	for i = 'a b c d' do i:=0 rof
WITH a.b.c DO j:=0;	for i = 'a/b/c' do i/j:=0 rof

The command "goto" has the format "goto name", and causes control to be transferred to the object defined by the local name. In order to change context to the object defined by name, the command "cd name" is used, and "rm name { name }..." removes objects created by the program.

Operators in BASIX_2 include an assignment (":=") operator, which updates a "name : object" pair relative to the local context, if necessary creating a pair. Both "name" and "object" are evaluated before the assignment. The arithmetical operators supported are those for addition, subtraction and multiplication (+ - *). Logical operators consist of "and", "or", and "not", and the relational operators are: = <> < <= > >=, returning true if their arguments are in the specified relation, otherwise returning false. Numeric sequences "1 2 3 4 ... " and alphabetic sequences "a b c d ... " are generated by the dyadic operator "to", and returned as an object, containing the sequence (see SASL, Section 5.1.2).

Program Execution

As a final illustration of the BASIX_2 language, a recursive Quicksort program "rquick" is shown in Figure 7.3. The Quicksort program in Figure 7.3 is divided into three parts: at the top is the declaration of the array "v" to be sorted, in the middle is the declaration of the program object "rquick", and at the bottom is the call to rquick. The array to be sorted is in fact the sixteen numbers, 512 ... 703. The corresponding implicit address selectors, from the left, are "1 2 3 ..."; alternatively the selectors could have been declared explicitly:

```
  v:( 1:(512) 2:(087) 3:(503) 4:(061) 5:(908) 6:(170) . . . 16:(703) )
```

as is necessary when alphanumeric selectors are used.

```
(*   the array to be sorted v[1] v[2] v[3] v[4]  . . .  v[n-1] v[n]    *)
v:(  512 087 503 061 908 170 897 426 765 275 154 509 612 677 653 703  )
(*   recursive Quicksort -----   rquick (lo, hi : integer )            *)
rquick:(
        lo := $/1 & hi := $/2 ;
        if lo<hi ->
                 (i := lo&
                  j := hi;
                  pivot := v/(j)  (*  pivot line  *)
                  do
                    (i<j) -> (
                              do (i<j) and ((v/(i) <= pivot)) -> i := i + 1 od
                              do (j>i) and ((v/(j) >= pivot)) -> j := j - 1 od
                              (* out of order pair *)
                              if (i<j) -> exchange(v/(i), v/(j)) fi
                             )
                  od;
                  exchange v/(i) v/(hi)  (*  move pivot to v(i)  *)
                  rquick lo i-1&
                  rquick i+1 hi
                 )
        fi
       )
(*   call Quicksort "rquick (1, n)"   v[1]  v[2]  . . .  v[n]          *)
 rquick   1   16   ;
```

Figure 7.3: Quicksort Program in BASIX_2

In the program object "rquick" storage for the variables "lo hi i j pivot" is created on demand. The first line of rquick initialises "lo" and "hi" from the first and second parameters in the call to rquick

```
lo := $/1  &  hi := $/2 ;
```

The control symbol "&" defines that the two commands are to be executed in parallel. Next comes the body of the Quicksort. It contains calls to two procedures: "exchange" which swaps two elements that are out of order, and the two "rquick"s that sort the subsets in parallel. Two formats for calls are illustrated, the traditional syntax "exchange(...)" and the list of objects "exchange ...", however the meaning is identical. Notice also that the array elements are accessed as "v/(i)" and not as "v/i". Finally the reader may find it interesting to compare this BASIX_2 version with the recursive PASCAL version given in Figure 4.1.

BASIX_2 was designed jointly by David Mundy and the author, and a translator, implementing the major parts [40] of the BASIX_2 language (written at first in C, and then rewritten in LISP), was produced by Mundy.

In the next chapter, two application programs will be discussed, written in BASIX_2, as the basis of an assessment of the language.

CHAPTER 8 - ANALYSIS OF BASIX

This chapter is an analysis of the BASIX programming languages, specifically BASIX_2, which is subsequently referred to as BASIX. Two applications are used for this analysis of BASIX: a simple Banking System and an Expert System. The current BASIX interpreter supports a subset of the language; for illustration the Banking System is programmed using "full" BASIX, and the Expert System is programmed using the "executable subset" of the language. Appendices A.4 and A.5 contain the listing of the Banking and Expert Systems.

8.1. BANKING SYSTEM

A simple banking system was chosen so as to demonstrate that BASIX can be used, successfully, for commercial applications. In addition, the banking application is meant to illustrate the uniform manipulation of files and variables, contextual addressing, etc.

8.1.1. Description of Application

This banking application system is a quite small and simple one: a current account system for a one-branch bank, which is called "Basbank". The clients of "Basbank" have only one type of account (current). Data maintained about the clients consist of name and address, as well as their balance, and the date of the last transaction.

The "Basbank"'s current account only allows two kinds of operations: deposits and withdrawals. All significant transactions data (e.g. client number, balance) are validated, and clients are then added to "Basbank"'s master file, ordered by "client number". Client's balances are altered according to deposit or withdrawals, but their personal data (name and address) can also be changed. Lastly, clients can only be excluded from the master file when their current balance is zero.

The "Basbank" system, as shown in Figure 8.1, consists of two main files (one containing the transactions, and the other the so-called "master" file or "old" file) and of one main program. The so-called "master" file holds all the data for the bank's clients. Each client has an individual record in the "master" file, composed of a client number, date of last update, name and address, and the current balance.

The banking application system program is composed of three routines, namely: i) "validate", ii) "sort", and iii) "update", as can be seen on Figure 8.1 below.

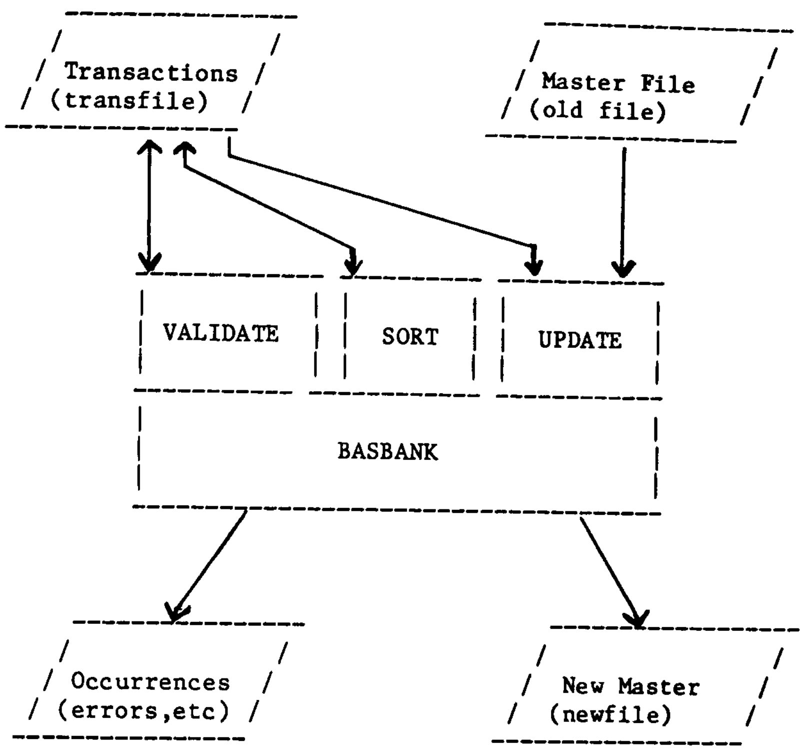


Figure 8.1: "Basbank" Banking System

8.1.2. Description of Program

The files used by the "Basbank" system are the "transactions" files, and the "master file". The "validate" routine (using the original "transactions" file as input) generates a "transactions" file with valid data, which is subsequently used as input by the "sort" routine to generate the sorted "transactions" file. The "transactions" file holds information of three basic kinds: inclusions, alterations, and exclusions. Firstly, those of type "1", which are inclusions of new clients, containing data such as the new client's number, the date of the inclusion, the name and address of the new client, and the pertaining present balance:

```
001  1  280852  martina w. felicitas  6 new happiness lane  606660
 |   |    |                |                   |                |
 |   |    |                |                   |                |
 |   |    |                |                   |                |
 |   |    |                |                   |             - balance
 |   |    |                |                - address
 |   |    |              - name
 |   |   - date inclusion
 |  - transaction code
- client's number
```

Secondly, those of type "2", which are <u>alterations</u> of client's data, where the value of deposits or withdrawals is conveyed, but also where information on changes to be made to existing data is supplied (e.g. correction of a client's name and/or address):

```
006  2  291283                        5678 ruddersville w.    000120
 |   |    |             |                   |                  |
 |   |    |       no correction for name    |               - credit
 |   |    |                                 |
 |   |    |                           - change of address
 |   |   - date transaction
 |  - transaction code
- client's number
```

Lastly, those of type "3", which are <u>exclusions</u> of clients, containing client's number and date of exclusion.

```
011  3  060683
 |   |    |
 |   |    |
 |   |   - exclusion date
 |  - transaction code
- client's number
```

An "inclusion" of a new client (transaction type "1") presupposes that the client is not yet in the "master" file, and therefore creates a client record containing information such as client number, date of creation of the record, name and address of the client, and current balance.

An "alteration" of a client record (transaction type "2") presupposes that the client exists in the "master" file. The data to be altered is optional, such as name and/or address. In the case of an withdrawal, the value of the debit is preceded by a minus sign ("-"), whereas for a deposit, only the value is specified.

Finally, an "exclusion" (transaction type "3") has the prerequisite that the balance of the client to be excluded must be zero. The client and all the respective data are removed from the "master" file.

The Banking program is divided in three main parts, namely: "validate", "sort", and "update".

The "validate" routine (seen in Figure 8.2) validates the information given in the transactions, such as client number, type and date of transaction, client balance, value of the credit or debit (deposit or withdrawal), etc. It discards those transactions where one or more errors have been found, besides listing them. Those records which were successfully validated, are kept and used in the routines "sort" and "update". The "validate" routine in BASIX can be seen in Figure 8.2.

```
(**********************************************************************)
(* validate - validates daily transactions input                     *)
(**********************************************************************)
validate :  (
                (* procedure okdate verifies if date is valid *)
                okdate: (
                          if
                           transrec/3/2 = 2 ->
                             if (transrec/3/1 < 1) or (transrec/3/1 > 29) ->
                                 errorflag:= 'true;
                             fi;
                           (transrec/3/2 = 4) or
                           (transrec/3/2 = 6) or
                           (transrec/3/2 = 9) or
                           (transrec/3/2 = 11) ->
                            if (transrec/3/1 < 1) or (transrec/3/1 > 30) ->
                                errorflag:= 'true;
                            fi;
                           (transrec/3/1 < 1) or (transrec/3/1 > 31) ->
                              errorflag:= 'true;
                          fi
                          if errorflag = 'true -> 'false; 'true; fi;
                        );
                (* procedure nameok verifies if name is alphabetic *)
                nameok: (
                          for i = 1 to 20 do
                              (if not ((transrec/4/(i) >= 'a) and
                                  (transrec/4/(i) <= 'z)) or
                                  (transrec/4/(i) = " " ) or
                                  (transrec/4/(i) = ".")) -> errorflag:= 'true;
                              fi);
                          rof
                          if errorflag = 'true -> 'false; 'true; fi;
                        );
              (* procedure addressok verifies if address is alphanumeric *)
              addressok: (
                          for i = 1 to 20 do
                          (if not (((transrec/5/(i) >= 'a) and
                                    (transrec/5/(i) <= 'z)) or
                                   ((transrec/5/(i) >= '0)  and
                                    (transrec/5/(i) <= '9))  or
                                    (transrec/4/(i)  = " ") or
                                    (transrec/4/(i)  = "."))-> errorflag:='true;
                          fi);
                          rof
                          if errorflag = 'true -> 'false; 'true; fi;
                        );
              (* main body of validate *)
                i:= 1;
                errorflag:= 'false;
                transindex:= 1;
                errorindex:= 1;
                temptrans := ();
                tempindex := 1;
                do
                 (transrec:= transfile/(transindex);
            transindex:= transindex + 1;
            transrec/1 <> 999) ->
            (if
             (transrec/1 >= 1) and (transrec/1 <= 100) and
             (okdate() = 'true) ->
              (if
               transrec/2 = 3 -> (temptrans/(tempindex):= transrec;
                                  tempindex:= tempindex + 1);
               transrec/2 = 1 ->
               (if (transrec/4 <> ())    and
                   (nameok() = 'true)    and
                   (transrec/5 <> ())    and
                  (addressok()='true)->(temptrans/(transindex):=transrec;
                                        tempindex;= tempindex + 1);
               fi);
               transrec/2 = 2 ->
               (if ((transrec/4 = ()      or
                     nameok() = 'true)    and
                    (transrec/5 = ()      or
                    addressok()='true)->(temptrans/(transindex):=transrec;
                                         tempindex:= tempindex + 1);
                fi);
              fi);
              (if errorflag = 'true -> (errorfile/(errorindex):= transrec;
                                        errorindex:= errorindex + 1;
              fi);
             fi);
           od
           temptrans/(tempindex):= transrec;  (* terminator 999 *)
           transfile:= temptrans;
       )
```

Figure 8.2: Validate - validates daily transactions

The "sort" routine reads the file which contains the validated transactions for the banking system, sorting them is ascending order by client number and transaction type. The code for the "sort" routines in BASIX can be seen in Figure 8.3. This example uses a simple linear selection with exchange sort algorithm to sort the records of the input file, based on the fields "transfile/()/1" and "transfile/()/2", outputting the resulting sorted records.

```
(***************************************************************)
(*  sort - sorts daily transactions input                      *)
(***************************************************************)
sort:  (i := 1;
        do
         transfile/(i)/1 <> 999 ->
        (j := i + 1;
         do
          transfile/(j)/1 <> 999 ->
           (if (transfile/(i)/1 > transfile/(j)/1) or
               ((transfile/(i)/1 = transfile/(j)/1) and
                (transfile/(i)/2 > transfile/(j)/2)) ->
               (
                 temp:= transfile/(i);
                 transfile/(i):= transfile/(j);
                 transfile/(j):= temp
                );
            fi;
          j:= j + 1);
         od;
        i:= i + 1);
       od;
      )
```

Figure 8.3: Sort - sorts daily transactions input

Lastly, the "update" routine (shown in Figure 8.4) updates the "master" file with the validated, sorted transactions. Updates in the "master" file are of three basic types: i) inclusion of a new client; ii) alteration of an existing client's data, and iii) exclusion of an existing client. The routine "update" in BASIX can be seen below.

```
(*********************************************************************)
(* update - updates Master File with validate, sorted daily transactions *)
(*********************************************************************)
update : (
          procupdate : ( if transrec/2 = 1 -> errorflag := 'true;
                            transrec/2 = 2 ->
                            (newrec/3 := transrec/3
                             if transrec/4 <> () -> newrec/4:= transrec/4 fi;
                             if transrec/5 <> () -> newrec/5:= transrec/5 fi;
                             newrec/6:= newrec/6 + transrec/6;
                            );
                            transrec/2 = 3 ->
                               (if exclflag = 'false -> exclflag:= 'true;
                                errorflag:= 'true;
                                fi);
                         fi
                       );
          transindex  := 1;
          oldindex    := 1;
          newindex    := 1;
          transrec    := transfile/(transindex);
          oldrec      := oldfile/(oldindex);
          newfile     := ();
          do
           (oldrec/1 <> 999) or (transrec/1 <> 999) ->
            if
               oldrec/1 < transrec/1 ->
                  (newfile/(newindex):= oldrec;
                   newindex:= newindex + 1;
                   oldindex:= oldindex + 1;
                   oldrec:= oldfile/(oldindex));
               oldrec/1 > transrec/1 ->
                   (if transrec/2 = 1 ->
                       (errorflag:= 'false;
                        exclflag  := 'false;
                        newrec    := transrec;
                       do
                        (transindex:= transindex + 1;
                         transrec:= transfile/(transindex);
                         newrec/1 = transrec/1 -> procupdate();
                       od;
                       if (errorflag = 'false) and (exclflag = 'false) ->
                              (newfile/(newindex):= newrec;
                               newindex:= newindex + 1)
                       fi;
                       errorflag:= 'true;
                    fi
                    if errorflag = 'true ->
                       (errorfile/(errorindex):= newrec;
                        errorindex:= errorindex + 1;
                       );
                    fi;
                   )
               oldrec/1 = transrec/1 ->
                   (  newrec:= oldrec;
                      oldindex:= oldindex + 1;
                    oldrec:= oldfile/(oldindex));
                    errorflag:= 'false;
                    exclflag:= 'false;
                    do newrec/1 = transrec/1 ->
                       (procupdate( );
                        transindex:= transindex + 1;
                        transrec:= transfile/(transindex);
                       );
                    od;
                    if (errorflag = 'false) and (exclflag = 'false) ->
                       (newfile/(newindex):= newrec;
                        newindex:= newindex + 1
                       )
                    errorflag = 'true ->
                        (errorfile/(errorindex):= newrec;
                        errorindex:= errorindex + 1;
                        );
                    fi
                 );
       fi;
      od
      newfile/(newindex):= oldrec;   (* terminator 999 *)
     )
```

Figure 8.4: Update - updates "Master File"

Appendix A.4 contains a complete listing of the Banking System, together with a sample run.

8.1.3. Assessment

In general, BASIX proved to be quite a reasonable language for programming the Banking System. This was helped by the fact that the von Neumann model is a subset of the decentralised control flow model, and the addressing scheme makes use of concepts similar to those of the UNIX operating system. The Banking System showed the addressing scheme to be a powerful tool in accessing the various contexts. Problems do arise with addressing, one is the impossibility of specifying a numeric string or expression as the only selector of a name such as differentiating between "1" the number and "1" the name. In order to solve this problem, "./1" is used to indicate the "name", and "1" remains as the "number". Another problem is that for the object:

```
a: ( ... i:(9) ... 9:(20) ... )
```

the name "a/i" gives the "i" component "9", whereas the name "a/(i)" uses the contents of "i" as the selector, giving "20". This might prove confusing for "traditional" programmers.

8.2. EXPERT SYSTEM

Next, an Expert Systems application is examined.

Recall, this application is coded in a (major) subset of the language which is supported by the current BASIX interpreter [40]. The restrictions of the current BASIX implementation include:

1. GOTOs are not supported, due to problems of implementing "name:object" pairs.

2. code declarations must be quoted "name: QUOTE code" to stop the right hand-side object being evaluated by the interpreter.

3. commands must be separated by explicit controls ";" and "&", because "newline" must be interpreted differently in code and data.

8.2.1. Description of Application

This Expert System application is a simple rule-based expert system for the identification of animals, taken from an article by Richard Duda and John Gaschnig [20], and re-coded in BASIX. The expert system written by Duda and Gaschnig implements a simple version of the backward-chaining procedure used in another (medical) expert system called MYCIN. It is based on a set of fifteen rules for the identification of animals [20].

Each rule has the form:

Format

```
(<name> (<a> ... <a>) (<c> ... <c>))
```

Example

```
(R6 ("HAS POINTED TEETH" "HAS CLAWS" "HAS FORWARD EYES") ("IS CARNIVORE"))
```

The name of the rule is not fixed (it can be any appropriate string). The antecedents <a> and the consequents <c> are delimited strings that correspond to propositions about the animal that may be either true or false. Should all antecedents be true, the program can use the rule to assert the truth of all consequents. Besides the rules, a set of hypotheses is also used (e.g. the animal is either a tiger, or a penguin, etc.). The aim of the program is to decide if one of the hypotheses is true, and a diagram of the way it works is shown below. In Figure 8.5, assertions are represented by boxes; ways of making combinations with assertions are the circles; and the rules are identified by R1, R2, etc.

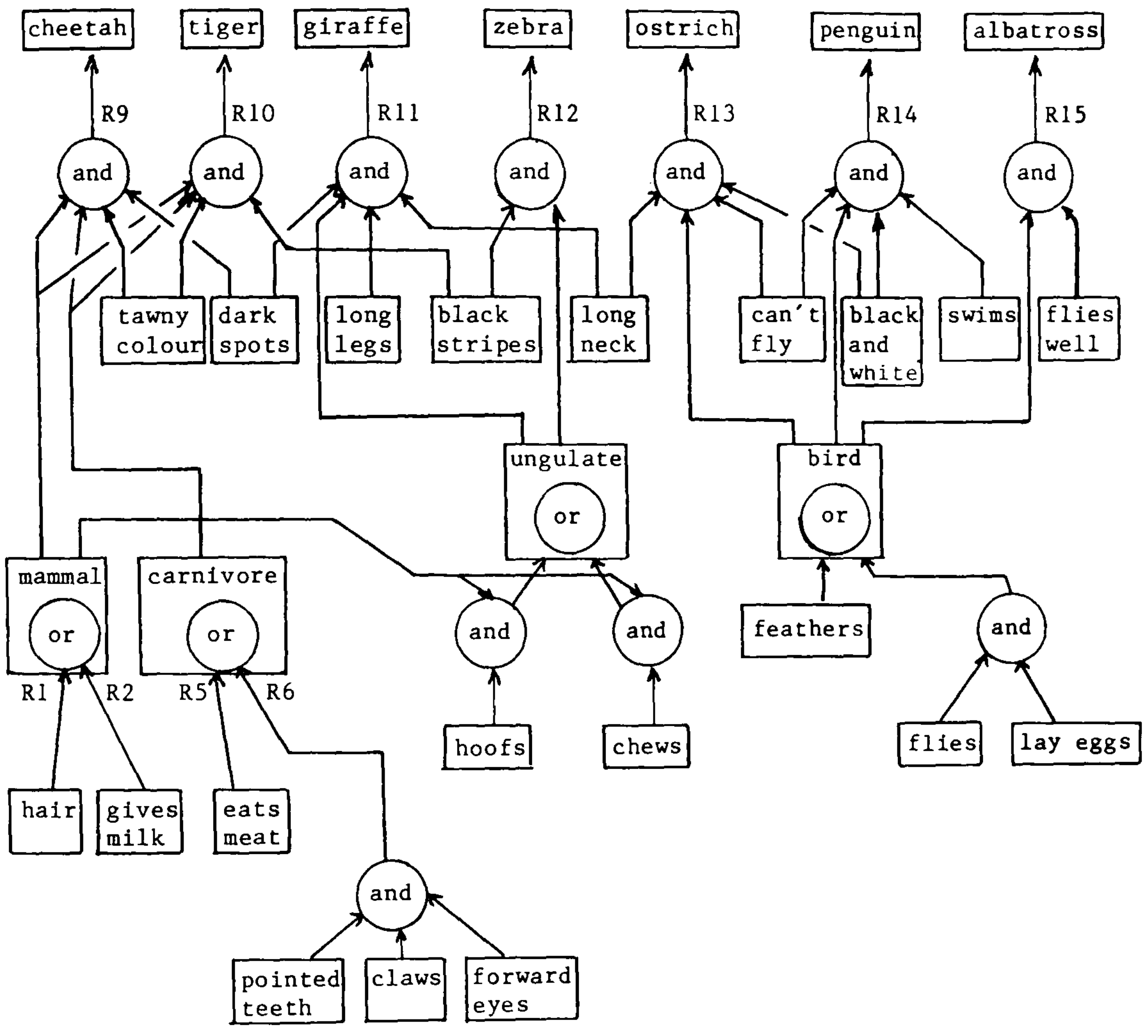


Figure 8.5: A diagram of the expert system for identifying "animals"
(reproduced from reference [20])

8.2.2. Description of Program

As in the Duda and Gaschnig [20] program, the BASIX version tries each hypothesis separately. For each hypothesis, the program consults the set of rules to see if the hypothesis can be deduced. If a deduction can be made, the antecedents for the relevant rules become new sub-hypotheses to be established, and the program looks for rules for deducing these antecedents. The descriptions of the variables used in the program can be seen below in Figure 8.6, followed by the program's main loop coded in the Executable Subset, which can be seen in Figure 8.7.

```
(*      queries = array of asked questions                              *)
(*      facts   = array for facts                                       *)
(*      hypo    = array for top-level hypothesis                        *)
(*      curhyp  = current top-level hypotheses                          *)
(*      q       = array of rule numbers for deducing a goal hypothesis  *)
(*      rules   = array for rules                                       *)
(*      currule = current rule index                                    *)
(*      curante = current antecedent                                    *)
```

Figure 8.6: Usage of Objects in Expert System

As it can be seen in Figure 8.6 in the Expert Systems program, the object "queries" stores the asked questions while "facts" keeps the facts. By using the built-in function of the BASIX interpreter called LIMIT, it is possible to establish how many facts have been recorded, and how many questions have been asked. The object "hypo" stores the top-level hypothesis in the expert system, and "curhyp" stores the current top-level hypotheses. The rule numbers for deducing a goal hypothesis are kept in "q", and the rules themselves are stored in "rules". The objects "currule" and "curante" store the current rule index and the current antecedent.

```
SYSOUT:= QUOTE "Hello!";
IF (LIMIT rules) = 0 -> SYSOUT:= QUOTE "No rules.";
   (LIMIT rules) > 0 ->
     (IF (LIMIT hypo) = 0 -> SYSOUT:= QUOTE "No hypotheses.";
         (LIMIT hypo) > 0 ->
           (SYSOUT:= QUOTE "I will use my ";
            SYSOUT:= LIMIT rules;
            SYSOUT:= QUOTE " rules to try to establish one of the following ";
            SYSOUT:= LIMIT hypo;
            SYSOUT:= QUOTE " hypotheses.";
            FOR i IN 1 TO LIMIT hypo DO
            ( SYSOUT:= name; SYSOUT:= hypo/(i));
            ROF;
            DO (facts: 0;
                queries: 0;
                done: FALSE;
                curhyp: 1;
                DO (NOT done) AND (curhyp <= LIMIT hypo) ->
                  (r: verify hypo/(curhyp) 1 1;
                   IF NOT r -> curhyp:= curhyp + 1;
                      r     -> (SYSOUT:= QUOTE "I conclude that ";
                                SYSOUT:= name;
                                SYSOUT:= hypo/(curhyp);
                                done:= TRUE);
                   FI);
                OD;
                IF NOT done ->
                     SYSOUT:= QUOTE "No hypothesis can be confirmed.";
                FI;
                SYSOUT:= QUOTE "r (restart) or q (quit) ?";
                DO (response: SYSIN;
                    (response <> QUOTE "r") AND (response <> QUOTE "q")) ->
                       TRUE;
                OD;
                response = QUOTE "r") -> TRUE;
           OD);
      FI);
```

Figure 8.7: Main Loop of Expert System

A search, chaining backwards through the rules, is made, and if no deductions can be achieved, the program asks the user if the sub-hypothesis it is working on is true.

As it can be seen on Figure 8.7 above, the main loop of the expert system sets up the arguments, and calls "verify" to establish the truth of "hypo/(curhyp)" and returns the answer, which is stored in "r". If no answer is found, a message saying that no hypothesis can be confirmed is printed. In case an answer is found, a conclusion is given, and the user is asked to restart or quit.

The main routine in the program is called "verify", and it can be seen in Figure 8.8.

```
verify: QUOTE (fact: ./1;
              currule: ./2;
              curante: ./3;
              q: 0;
              r: recall fact;
              IF NOT r -> (inthen fact;
                            IF (LIMIT q) = 0  -> r:= ask fact currule curante;
                               (LIMIT q) <> 0 ->
                                 (i: 1;
                                  DO (done: tryrule q/(i);
                                      IF NOT done -> i:= i + 1; FI;
                                      (NOT done) AND (i <= LIMIT q)) -> TRUE;
                                  OD;
                                  r:= done);
                            FI);
              FI;
              r);
```

Figure 8.8: Verify Facts in Expert System

Its function is to establish the truth of a hypothesis or sub-hypothesis, represented by the argument "fact". If the truth of "fact" has already been recorded, "verify" returns immediately. If there are no rules for deducing "fact", and if "verify" has not asked the user about "fact", it then asks. Otherwise, "verify" applies "tryrule" to each of the rules in turn, until it either finds a successful answer or there are no more rules left.

Other important subroutines are "inthen", which finds all the rules that have fact "fact" as a consequent, and subroutine "ask", which asks the user about "fact" and explains why it is asking. The subroutine "remember" records facts, and subroutine "testif" checks antecedents to see if rule "rule/(currule)" is applicable. "Rule" is applied by subroutine "usethen", which also prints new deductions.

A complete listing of the Expert Systems program, in the executable subset of BASIX, is given in Appendix A.5.

8.2.3. Assessment

Unlike in the Banking application program, where specific fields of a conventional record were being dealt with, in the Expert System program no urgent need to explicitly name fields was felt. In fact, most data was communicated using global variables as in BASIC. Access to stacks and arrays in BASIX is very similar to that in BASIC (e.g. s$(s1) = x$ in BASIC, and facts/(k) := fact in BASIX). The main difficulty was felt in "mimicking" the control structure of the original program [20], such as the extensive use of "gosub" made in the BASIC version. As Full BASIX does not have "gosub/return" statements, artificial labels would have had to be created and placed at the beginning of certain statements, to simulate control transfers and calls:

```
"artificial" labels ---
        |          |
        |          |
      0790 : goto 0820
           .
           .
           .
      0820 : $/O[]:= 'RESTART OR QUIT (R OR Q)'    (* print *)
```

For this reason the Expert System was coded to take advantage of some of the main features of BASIX (such as recursion) and to demonstrate the executable subset. The Executable Subset version, as would be expected, is considerably shorter and easier to understand than the original program [20] in BASIC.

8.3. ANALYSIS AND ASSESSMENT

Before presenting this assessment of BASIX, it should be again noted that BASIX is a low-level system language (cf. C - but without facilities such as data types), rather than a high-level language (cf. PROLOG), specifically designed to fully "mirror" the decentralised control flow programming model. In general, BASIX proved to be a reasonable language in which to program, even in such two diversified fields as Banking and Expert Systems. The fact that the von Neumann model is a subset of the decentralised control flow model was made clear by the example programs. Next, the main concepts of BASIX will be accessed in turn, starting with information representation.

The single concept of an object representing files and variables, etc. is powerful, combining attributes of the UNIX Shell with those of LISP and BASIC.

In terms of addressing, the contextual addressing concepts prove flexible in accessing the various contexts and are reasonably natural to work with. However, this contextual addressing could prove tricky to use, (at least at early stages) by traditional programmers (especially those working in a commercial environment). An example is the difference between "tr/i" and "tr/(i)"; the former accesses a subsidiary object with explicit selector "i:(...)", while the latter uses the contents of "i" as the selector. Other addressing problems relate to the different types of selectors. One of them is the impossibility of specifying a numeric string or an expression as the first or only selector of a name due to parsing problems. For example, it is difficult to differentiate "9" the number from "9" the name, unless some additional identification tag is applied. This is currently achieved in BASIX, by using the special selector "." to define a name "./9". Another area requiring further study is the relationship between control of contextual addressing in programming languages and in operating systems. In BASIX, this involves the interaction between the special selectors (i.e. "$", "..", ".", "/") that explicitly specify a contextual address, as opposed to some implicitly defined automatic search of surrounding contexts. This addressing problem is discussed in detail in [40].

Related to the addressing of objects is the support of both "shared memory" and "message passing" data mechanisms. If an object is accessed by "name", then it is treated as a variable, and if accessed by "name[]" it is treated as a message. This syntax and semantics have proved reasonable to work with in other examples, and is in practice quite powerful to use [27,28].

Next, the representation and execution of programs. Since BASIX embodies procedural programming, its syntax and semantics are relatively traditional. There are, however, two problem areas: the first is the conditional and repetitive commands, and the second is the control operators ";", "newline", "|" and "&". The conditional and repetitive commands, although having the traditional function have, in BASIX, been designed to span more conventional commands (e.g. IF-THEN, IF-THEN-ELSE, IF-THEN-ELSEIF-ELSE, WHILE-DO, REPEAT-UNTIL etc.) with the same basic set of constructs (e.g. IF-FI, DO-OD). In practice these constructs, in particular the "do-od", have proved difficult to use by those already acquainted with conventional languages. For the control operators, the problem relates to the "newline" which implicitly defines sequential execution. One of the aims of the BASIX language, like LISP, is to represent programs and data in the same way. Unfortunately, in the current BASIX system, "newline" is an operator in programs, but should be invisible in data. For this reason, explicit terminators are used in the executable subset of BASIX.

The major (and justifiable) criticism of BASIX_2 is that the designers have been over-ambitious, making the syntax too recursive, which has resulted in a somewhat confusing semantics. In addition, although "object" is a central concept in BASIX languages, this is not reflected in the BASIX_2 syntax.

In BASIX languages there are essentially four types of object:

```
expression
statement
({objects}...)
local_name:object
```

An expression consists of one or more simple objects, separated by operators. A statement is a list of objects whose leftmost object is a keyword or the name of a program object. A bracketed list of objects may be code or data. Lastly, comes the declaration of a "name:object" pair in the local context.

Control statements were identified as a problem area. For control statements, one simple strategy is to adopt Dijkstra's Guarded Commands [21]:

```
IF {expression -> command}... FI
DO {expression -> command}... OD
```

however a compromise is made with "GOTO", by restricting it to a local_context:

```
GOTO local_name
```

Names and selectors were also a problem area, both for syntax and semantics. A proposal is to define "names" as either:

```
local_name{.selector}...
${.selector}...
```

where "local_name" is an alphanumeric character string, "$" identifies parameters, and "selectors" consist of:

```
local_name
numeric
(expression)
```

This division should simplify both the syntax and semantics.

Finally, as illustration of the effect of these changes, Appendix A.3 contains the syntax of the improved BASIX.

CHAPTER 9 - BASAL PROGRAMMING LANGUAGE

This chapter presents the BASAL programming language, which is used to investigate languages (at the opposite end of the spectrum from BASIX) embodying a primitive form of decentralised control flow model.

9.1. DESIGN PHILOSOPHY

The programming model of the BASAL language [26], and of the RIMMS multi-microcomputer system [25], implements a subset of decentralised control flow. Its principles are:

1. computer - a network of microcomputers, each comprising a primitive processor and memory;
2. memory - a linear organisation of fixed-size memory cells;
3. addressing - a two-level address space, defining a micro and its local memory;
4. program - a low level machine language, where instructions consist of primitive operators and operands;
5. communication - shared memory and message passing;
6. execution - sequential and parallel control of computation.

Thus, the programming model can be seen as falling between that of the von Neumann model and the decentralised control flow model, described in Chapter 6.

RIMMS (as introduced in Chapter 2) consists of identical component microcomputers with a 16-bit word size: each register, data element and address is 16 bits. Instructions, however, are 2 x 16 bits and use a 3-address format. There are less than 20 operators. Each microcomputer in the multi-microcomputer system is addressable (has a unique address), and behaves as a combined memory and processor that is able to service load, store and execute operations. Design of the multi-microcomputer system centres around the 16-bit global address space. An address consists of two parts: the high 8 bits define a specific microcomputer, while the low 8 bits define a word in that microcomputer's memory. Although a microcomputer can access any word in the global address space, an attempt to execute alien code causes execution to transfer to the specified microcomputer.

BASAL is a parallel language (based on a subset of decentralised control flow) that extends BASIC and can be used for programming the RIMMS multi-microcomputer systems. It extends BASIC in four ways, firstly global or local identifiers may be used for names and labels; secondly both "shared memory" and "message passing" communication of data is supported; thirdly a new command "MICRO micro_name" causes all subsequent commands to be interpreted in microcomputer "micro_name"; and lastly, a program consists of a series of commands separated by controls: ";" and "newline" define sequential execution (of the two adjacent commands) while "&" defines parallel execution.

Before presenting the design and implementation of the BASAL programming language (produced by the author), the RIMMS design (of Foti et al [25]) is described.

9.2. RIMMS MULTI-MICROCOMPUTER SYSTEM

Traditionally, the trend in designing microprocessors and mainframe computers has been towards increasingly complex instruction sets and associated architectures [29]. In contrast, designs based on the so-called reduced instruction set [41,42] philosophy have a simple set of instructions, and a correspondingly simple machine organisation tailored to the efficient execution of these instructions. The aim of the ongoing RIMMS project is to design the simplest conventional microcomputer with primitive communications mechanisms that is able to form a component of a tightly-coupled multi-microcomputer system. The architecture of RIMMS is described in terms of two levels of machine: the multi-microcomputer level handles inter-process(or) communication supporting non-local load, store and execute operations; and the microcomputer level services these operations and handles the atomic execution of a single process.

9.2.1. Multi-Microcomputer

RIMMS consists of a linear array of up to 255 microcomputers that communicate via a shared bus, as shown in Figure 9.1. Each microcomputer has a simple processor and 256 words of local memory.

```
 -------------------------------------------------------------
|                                                             |
|                   8-bit global address                      |
|          ---------------------------------------------      |
|        1:|              2:|         . . . .      255:|      |
|   ------------    ------------               ------------   |
|  | processor  |  | processor  |             | processor  |  |
|  |------------|  |------------|             |------------|  |
|  |   memory   |  |   memory   |             |   memory   |  |
|  |(8-bit local|  |(8-bit local|             |(8-bit local|  |
|  |  address)  |  |  address)  |             |  address)  |  |
|   ------------    ------------               ------------   |
|                                                             |
 -------------------------------------------------------------
```

Figure 9.1: Multi-Microcomputer System

The system has a 16-bit address space:

```
                 address
        global (8 bits) local (8 bits)

        -------------------------------
       | microcomputer | memory cell   |
        -------------------------------
```

Figure 9.2: RIMMS Address

The top 8 bits is a global address (in the range 1-255) defining a microcomputer, while the bottom 8 bits is a local address (in the range 0-255) defining a word in its memory. (Global address 0 is the default for specifying the current local address space and is therefore not recognised at the Multi-Microcomputer level.)

When one microcomputer wishes to communicate with another, for example to access its local memory, the microcomputer generates a "packet". The format of a packet, as shown in Figure 9.3, consists of a 2-bit operation field, a 2x8-bit destination address, and a 16-bit operand. The 4 operations are: load from memory (LOAD), store into register (STORE_REG), store into memory (STORE_MEM), and execute instruction (EXECUTE).

```
                 global  local
  2 bits         8 bits  8 bits    16 bits

 ---------------------------------------
| operation |    address      |  operand |
 ---------------------------------------
```

Figure 9.3: Multi-microcomputer packet format

The packet operations are defined as follows:

LOAD - copies the contents of MEMORY[address] to the microcomputer's register defined by the 16-bit operand. This is implemented by the destination microcomputer generating a STORE_REG packet.

STORE_REG- places the operand in the microcomputer's register defined by the address.

STORE_MEM- places the operand into the MEMORY[address].

EXECUTE- starts a new process whose code is at MEMORY[address] and data environment is at MEMORY[operand].

For all these packets the global address defines the destination microcomputer.

Microcomputers take turns to send a packet on the bus. When a packet is sent the destination microcomputer may accept or reject the packet. In either case the source microcomputer relinquishes the bus. If rejected, the source microcomputer will re-attempt to send the packet at its next turn to use the bus. Whether a packet is accepted or rejected depends on the status of the processor and memory of the destination microcomputer. In simple terms, load and store operations may be serviced by the memory concurrently with the operation of the processor. However an execute packet may only be accepted when the processor is idle, having completed the execution of its previous process. Figure

9.4 lists the complete rules for processing packets.

	Packet Received			
	LOAD	STORE_REG	STORE_MEM	EXECUTE
Processor Status				
BUSY				
. EXECUTING	-	error	-	reject
. WAITING	-	accept	-	reject
IDLE	-	error	-	accept
Memory Status				
BUSY	reject	reject	reject	reject
IDLE	accept	accept	accept	accept

Figure 9.4: Microcomputer Status versus Packet Received

In Figure 9.4, the term BUSY EXECUTING specifies that the processor is executing instructions, and BUSY WAITING specifies that the processor is executing but temporarily waiting for an operand to be loaded from a memory.

Next, the architecture of a microcomputer is examined.

9.2.2. Microcomputer

A RIMMS microcomputer consists of three basic components: the local memory of up to 256x16-bit words, the memory controller, and the 16-bit processor for arithmetic, as illustrated by Figure 9.5.

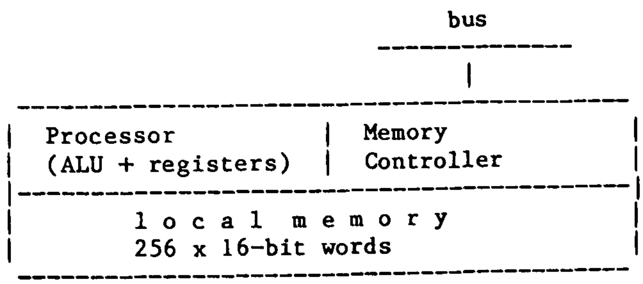


Figure 9.5: Microcomputer

The memory controller is connected to the global bus, and to the local processor and memory. It supports communication, in the form of packets, between these three units. To hold a packet, the memory controller has 3 registers: a 2-bit memory operation register, a 16-bit memory address register, and a 16-bit memory data register (see Figure 9.6).

memory operation register	MOP	(2 bits)
memory address register	MAR	(16 bits)
memory data register	MDR	' '

Figure 9.6: Memory Controller Registers

These registers correspond to the operation, address and operand fields, respectively, of a packet.

When a memory controller is idle it can receive a packet either from the local processor or from some other microcomputer. A packet from the processor can be destined for the local memory or for another microcomputer, whereas a packet from the bus can be destined for the local processor or memory. A packet's destination is specified by the top 8 bits of the address in MAR.

The processor, the last component of the microcomputer, consists of an arithmetic logical unit (ALU) and seven registers supporting a 16-bit word size. Each register, data element and address is 16 bits. Instructions, however, are 32 bits and use a 3-address format. Figure 9.7 shows the 7 registers of which only the first two are addressable.

program counter	C	(16 bits)
data register	D	(16 bits)
instruction registers	I1,I2	(2x16 bits)
ALU register 1	A1	(16 bits)
ALU register 2	A2	' '
ALU register 3	A3	' '

Figure 9.7: Processor Registers

C the program counter points to the local code currently being executed. D the data register points to the current data environment which may be anywhere in the address space. I1,I2 holds the current 32-bit instruction. A1,A2,A3 are the input registers to the ALU, holding the current instruction's operands. Their contents have no meaning from one instruction to the next.

An instruction's format, as illustrated by Figure 9.8, consists of: a 5-bit operator field, 3x1-bit mode (Mi) fields, and 3x8-bit operand (Oi) fields. Modes and arguments are interpreted as follows. If the value of mode bit Mi=0 then the corresponding 8-bit operand Oi is treated as a literal. Oi is sign extended to 16 bits and the resulting argument is placed in the corresponding ALU register Ai. If the mode bit Mi=1 then the 8-bit operand Oi is treated as a signed displacement relative to the data register D. The resulting address D+Oi is dereferenced (via the multi-microcomputer level if necessary) and the memory contents is placed in the ALU register Ai. Notice that the modes and operands are interpreted independently both of the operator and of whether they are to be used for input and output by the ALU. However, the operator does determine how many of the three arguments are used by the ALU.

```
            M1  M2  M3       O1              O2              O3
  5 bits    1   1   1       8 bits          8 bits          8 bits

 ------------------------------------------------------------------
 |operator|mode bits|literal/address|literal/address|literal/address|
 ------------------------------------------------------------------
                 0  -  literal
                 1  -  address (memory [D+ signed literal])
```

Figure 9.8: Microcomputer Instruction Format (32 bits)

The ALU supports only two information types: 16-bit integers (2's complement) and booleans (TRUE=FFFF, FALSE<>FFFF), and following the reduced instruction set philosophy only a minimal set of operators are provided. These operators are listed in Figure 9.9.

Operation	Mnemonic	Description
arithmetic	ADD	-
	SUB	-
logical	AND	-
	OR	-
	NOT	-
shift	LSHIFT	logical shift
	ASHIFT	arithmetic shift
compare	EQ	equals
	GT	greater than
control	IF	if TRUE jump
	FORK	fork flow of control
	HALT	halt processor
movement	MOVE	move argument to address
	STORE C	store program counter
	LOAD D	load data register
	STORE D	store data register

Figure 9.9 Processor Instruction Set

Finally note that the reason for choosing a 3-address instruction format and only two addressable registers is to minimise the state information that needs to be moved from one microcomputer to another, when control is transferred.

9.3. BASAL PROGRAMMING LANGUAGE

BASAL is a parallel language (based on a subset of decentralised control flow) that extends BASIC, and can be used for programming multi-microcomputer systems having the RIMMS philosophy. BASAL extends BASIC in four ways:

1. multiple processes/processors

2. a two-level address space for names and labels

3. shared memory and message passing communication of data

4. parallel execution from the "&" statement terminator

9.3.1. Description

In BASAL, a program consists of a series of commands separated by controls: ";" and newline define sequential execution (of the two adjacent commands), while "&" defines parallel execution. The description of BASAL can be divided into four levels: i) commands; ii) statements; iii) expressions; and iv) identifiers (i.e. labels and names). In the following description "{ }" defines zero or one occurrences, and "{ }..." defines zero or more occurrences, of the enclosed information.

Commands in BASAL are of three types:

```
MICRO micro_name
local_label statement
statement
```

For the "MICRO micro_name" command, all instructions following that keyword command are executed by the microcomputer "micro_name". The labelled statement "local_label statement" is similar to the BASIC one, being stored for later execution, with the "local_label" providing the reference in the corresponding micro. The "statement" is also similar to BASIC, in the sense that it is executed immediately.

Statements in BASAL have much in common with BASIC, being either an expression or a command whose leftmost argument is a keyword:

```
expression
DIM local_name (integer {, integer}...)
LET name = expression
IF expression THEN local_label
FOR local_name = expression TO expression
NEXT local_name
GOTO label
GOSUB label
RETURN
STOP
END
```

"Expression" returns the result in place. "DIM local_name (integer{,integer}...)" declares an array of the specified dimensions, designated by a name local to the current micro where it is being declared.

The statement "LET name = expression" is the assignment statement in BASAL, and can assign a value to a variable in any micro. The conditional statement "IF expression THEN local_label" provides the language with decision making ability, transferring control to the "local-label" if the the expression is true. In "FOR local_name = expression TO expression", repetitive execution of the enclosed expressions is provided. This statement is connected with the "NEXT local_name" statement, which indicates the end of the corresponding loop. Clearly, the "local_name" in "NEXT" must be exactly the one used after the "FOR" keyword.

The "GOTO label" statement causes unconditional control transfer to the specified label, while the "GOSUB label" statement is the BASAL equivalent of a procedure call. For "GOTO" and "GOSUB" a "label" may identify a statement in the calling micro or any other micro. The "GOSUB" statement is used in connection with the "RETURN" statement, which returns control to the statement immediately after the calling "GOSUB" statement. The two final statements are the "STOP" statement, that causes a microcomputer program to terminate and the "END" statement, which marks the end of a BASAL program.

<u>Expressions</u> in BASAL can be of seven different kinds:

```
number
name
"character"
_expression
(expression)
expression operator expression
?
```

Here, a "number" can be any integer number; "name" is an identifier (local or global) of a variable, message or array element; "character" can represent any ASCII character, but must be enclosed in double quotes; "_expression" represents the negation of the result of an expression (note that the sign is an underbar and not a minus, since there would be no way of telling apart an unlabelled statement composed of an arithmetic subtraction operation, from a labelled statement commencing by a negative expression):

```
              10   -2345        from         10   -2345
              ^    ^                         ^    ^
              |    |                         |    |
    number ---      --- arithmetic    label ---    --- negative expression
  (not label)           subtraction
```

"(expression)" represents a bracketed expression; "expression operator expression" allows the representation of arithmetical, logical and conditional expressions; and finally, "?" identifies an empty data location.

<u>Labels</u> in BASAL have the following format:

```
{micro_name.}local_label
<micro_name> ::= A..Z
<local_label>::= 01..79
```

where "{}" indicates an optional field. Thus, a global label consists of "micro_name.local_label" (e.g. "A.01"), and a local label consists of "local_label" (e.g. "01"). Notice (because of the implementation) that labelled statements may only be labelled with a local label in the range "01" to "79", whereas "GOTO" and "GOSUB" may specify either a global or local label.

Names in BASAL have the following format:

```
{micro_name.}local_name{(expression{,expression}...)}{[]}
<micro_name> ::= A..Z
<local_name> ::= alphanumeric
```

A variable can be local (used inside the current micro): "local_name", or global (used in another micro): "micro_name.local_name". A "{micro_name.}" is a letter in the range of "A" to "Z", equivalent to a micro (i.e. first micro is "A", second is "B", etc.), and "local_name" is any alphanumeric identifier.

To indicate access to an array element, the "{micro_name.}local_name" is followed by one or more expressions enclosed in brackets, and separated by commas:

```
{micro_name.}local_name(expression{,expression}...)
```

To distinguish between "shared memory" and "message passing", to indicate message passing, the "[]" symbol is used in the form:

```
{micro_name.}local_name{(expression{,expression}...)}[]
```

On the left of the assignment, it indicates a "PUT", and on the right side a "TAKE" operation. Recall, the empty memory cell used with "PUT" and "TAKE" is defined by the "?" symbol. Notice that PUT and TAKE operate on a single memory cell, polling a cell until it is in the correct state.

Operators in BASAL contain the four arithmetical operators (+ - * /), three logical operators (AND, OR, NOT), and six relational operators (<, <=, >, =>, =, <>). Finally, a complete syntax for the BASAL language is given in Appendix A.6.

As the reader may have noted from the above descriptions of the RIMMS multi-microcomputer and the BASAL language, the initial RIMMS does not fully support BASAL (i.e. "message passing" is not implemented in the hardware). Thus here the use of BASAL for programming "RIMMS-like" systems will be discussed.

In BASAL, microcomputers are allocated in a way analogous to the allocation of memory cells in a conventional computer. Thus two views are offered to a programmer or compiler. Due to the shared 16-bit address space, a system can be programmed as a single, sequential computer (e.g. with up to 255x256 words of memory) or, more interestingly, as a parallel computer (e.g. with up to 255 processors each with 256 words of memory). For instance a large sequential program, if allocated consecutive memory locations, will span a number of microcomputers. As control reaches the boundary of a microcomputer its program counter will contain a non-local address, causing control to migrate to the next processor.

For parallel execution, each process should be allocated a separate microcomputer. These processes are started in BASAL using either GOTO or GOSUB statements in conjunction with parallel controls "&":

```
GOTO B.11 &                    GOSUB B.11 &
GOTO C.22 &                    GOSUB C.22 &
  . . .                          . . .
```

These statements are implemented in RIMMS by "FORK" instructions; a FORK may be thought of as a GOTO that not only transfers control but also continues execution.

Having initiated a number of parallel processes, synchronisation of their execution centres on three mechanisms which loosely equate to the semantics associated with a memory location: (i) command, (ii) variable, and (iii) message.

Code in a microcomputer is executed atomically by a processor, thus behaving as an uninterruptable critical region which may be used to synchronise access to shared data. In contrast, "LOAD" and "STORE" operations (giving the "shared memory" semantics) are unsynchronised and compete for memory access. Lastly, "TAKE" and "PUT" operations support "message passing" semantics and may be used to pass a sequence of one or more values from a producer to a consumer process.

Since BASAL provides both types of data communication that are found in computing, then the two common forms of parallelism (namely shared memory and message passing) are supported. This is illustrated by examining BASAL programs for Sort/Merge, shown in Figures 9.10 and 9.11. Each program consists of 3 processes: "A" and "B" which sort their local arrays "V" into ascending order, and "C" which merges these two arrays and stores the results in its own array "V".

In Figure 9.10 communication between the 3 processes is by shared memory. Execution of each process is started by the unlabelled "GOTO 01" statement, which causes "A" and "B" to sort the contents of their arrays, and "C" to initialise "COUNT" and then "STOP" execution. When "A" and "B" finish execution each transfers control "GOTO C.03" to label "03" in process "C". This causes the decrementing of "COUNT". (The decrementing of "COUNT" works as a critical region because "C" is executed atomically.) When both sorts are finished, "C" merges the two arrays. In doing this "C" initially stores two terminators "32677", in parallel "&", into the arrays and then uses the "FOR" loop to merge the 200 values.

```
MICRO A
01 DIM V(101)
02 FOR I = 1 TO 99
03 FOR J = I+1 TO 100
04 IF V(I) <= V(J) THEN 08
05 LET TEMP = V(I)
06 LET V(I) = V(J)
07 LET V(J) = TEMP
08 NEXT J
09 NEXT I
10 GOTO C.03
GOTO 01

MICRO B
01 DIM V(101)
02 FOR I = 1 TO 99
03 FOR J = I+1 TO 100
04 IF V(I) <= V(J) THEN 08
05 LET TEMP = V(I)
06 LET V(I) = V(J)
07 LET V(J) = TEMP
08 NEXT J
09 NEXT I
10 GOTO C.03
GOTO 01

MICRO C
01 DIM V(200)
02 LET COUNT = 2
03 LET COUNT = COUNT - 1
04 IF COUNT <> 0 THEN 17
05 LET A.V(101) = 32677 &
06 LET B.V(101) = 32677 &
07 LET I = 1            &
08 LET J = 1
09 FOR K = 1 TO 200
10 IF A.V(I) > B.V(J) THEN 14
11 LET V(K) = A.V(I)
12 LET I = I + 1
13 GOTO 16
14 LET V(K) = B.V(J)
15 LET J = J + 1
16 NEXT K
17 STOP
GOTO 01
END
```

Figure 9.10: BASAL (shared memory) Sort-Merge

In the second Sort/Merge, in Figure 9.11, "A" and "B" are very similar to the corresponding processes in the previous example. The difference is statement "09" which passes each of the sorted values, as it becomes available, to "C" using message passing. Process "C", executing concurrently with "A" and "B", then merges these values. Notice statement "03" in "C", that locations "A" and "B" are accessed as variables (rather than messages) and repeatedly tested until both are non-empty. Then depending on which is the smallest, either "A[]" or "B[]" is accessed as a message and set to empty so as to receive the next value from its process.

```
MICRO A
01 DIM V(100)
02 FOR I = 1 TO 99
03 FOR J = I+1 TO 100
04 IF V(I) <= V(J) THEN 08
05 LET TEMP = V(I)
06 LET V(I) = V(J)
07 LET V(J) = TEMP
08 NEXT J
09 LET C.A[] = V(I)
10 NEXT I
11 LET C.A[] = V(100)
12 LET C.A[] = 32677
13 STOP
GOTO 01

MICRO B
01 DIM V(100)
02 FOR I = 1 TO 99
03 FOR J = I+1 TO 100
04 IF V(I) <= V(J) THEN 08
05 LET TEMP = V(I)
06 LET V(I) = V(J)
07 LET V(J) = TEMP
08 NEXT J
09 LET C.B[] = V(I)
10 NEXT I
11 LET C.B[] = V(100)
12 LET C.B[] = 32677
13 STOP
GOTO 01

MICRO C
01 DIM V(200)
02 FOR K = 1 TO 200
03 IF (A = ?) OR (B = ?) THEN 03
04 IF A > B THEN 07
05 LET V(K) = A[]
06 GOTO 08
07 LET V(K) = B[]
08 NEXT K
09 STOP
GOTO 01
END
```

Figure 9.11: BASAL (message passing) Sort-Merge

Next, the implementation of the BASAL programming language will be examined.

9.3.2. Implementation

BASAL is implemented by a translator that takes in a parallel BASAL program and outputs a sequential BASIC program. This BASIC program, when interpreted, simulates the execution of the BASAL program. The BASAL translator is written in PASCAL.

The translation of BASAL to BASIC poses a number of problems:

1. simulating the network of micros

2. handling the two-level address space

3. supporting the labelled and unlabelled statements

4 handling the "shared memory" and "message passing" data mechanisms, and finally

5 simulating parallel execution

To simulate the network of micros, "micro_name", from command "MICRO micro_name", is prefixed to all local_names e.g. "XYZ" becomes (inside MICRO A) "A.XYZ". In addition, local labels are transformed into global labels:

```
             MMLLS
            __| | |__
        ---     |     ---
  micro         |         system
              local
```

by being prefixed with the equivalent MICRO number (e.g. MICRO A => 01, etc.), and suffixed with a "system" label. Thus the two levels of address space of BASAL are supported by mapping all local names into global names, and local labels into global labels.

Thus each micro occupies a range of labels. For example, 01000 to 01999 is for MICRO A. To simulate the execution of multi-micros, certain ranges of labels are allocated for housekeeping functions, as shown in Figure 9.12.

```
01000          GOTO PC(01)
01010  )
 ...   }       labelled statements
01790  )

01799          GOTO 01998

01800          REM
01810  )
 ...   }       unlabelled statements
01990  )

01998          LET PC(01) = 01999
01999          REM
```

Figure 9.12: Allocation of Labels for MICRO A

The statement labelled with "MM000" is the "GOTO PC(MM)", where "PC(MM)" has the role of the program counter for micro "MM". This so-called program counter contains the address of the next statement to be executed whenever it is necessary to suspend execution.

Labelled statements are allocated labels in the range "MM010" to "MM790". At the end of the labelled statements, comes labelled statement "MM799", which is a "GOTO" that transfers control to the end of the statements for this micro. This will cause "PC(MM)" to be set to label "MM999", which will subsequently cause control to jump over this block of statements.

Unlabelled user statements are allocated labels in the range "MM810" to "MM900". Before execution the "PC" of each micro is initialised to "MM800", causing the execution of any unlabelled statements.

The handling of "shared memory" and "message passing" mechanisms will now be discussed. "Shared memory" semantics is directly supported by BASIC. For "message passing", however, it is necessary to test that a memory location is in the correct state, before the statement can execute. A BASAL statement of the following form should be considered:

```
22 LET X[] = Y[]
```

Here, the "X" defines a "PUT", and therefore the memory location must be empty, and "Y" defines a "TAKE", and therefore the location must be non-empty, before the statement can execute. This is achieved by generating the following code:

```
01810 LET A.X = -32768

01220 LET PC(01) = 01220
01221 IF A.X <> -32768 THEN 01999
01222 IF A.Y = -32768 THEN 01999
01226 LET A.X = A.Y
01227 LET A.Y = -32768
```

Figure 9.13: Code Generated to Support "Messages" Semantics

In the code in Figure 9.13, the "empty" state is represented by "-32786". For the "LET X[] = Y[]" to operate, memory location "X" must be initially set to "empty". This is achieved by generating an unlabelled "LET".

```
01810 LET A.X = -32768
```

as in Figure 9.13. Support of the actual assignment statement makes use of the "system" field in the label. The first statement at label 01220 sets "PC(01)" to the address of the block of code. The next two statements test that A.X is "empty", and that "A.Y" is "non-empty". If either of these tests fails, then execution branches to the end of this micro's code (when the code of this micro is next executed, control will be transferred to retest this block of code). If both tests succeed, then the assignment is made, and "Y" is set to "empty".

Next, the problems of simulating parallel execution must be discussed. This involves two considerations: firstly, that control cannot be transferred to an alien microcomputer if the microcomputer is already executing; and secondly, there is the support of "&" control. The first issue will be considered: when "GOTO", "GOSUB" or "RETURN" are to make a non-local transfer, it is necessary to test if the target microcomputer is already active. To support this, additional "housekeeping" information is generated. This makes use of the following: the array "PC" of program counters, and an array "STACK", which contains a stack for each microcomputer, and an array "STACKPOINTER", which points to the corresponding top of stack.

The code generated for "GOTO", "GOSUB", and "RETURN" will now be examined:

```
GOTO B.01

          01810 LET PC(01) = 01810
          01811 IF PC(02) <> 02999 THEN 01999
          01812 LET PC(01) = 01999
          01813 GOTO 02010
          01814 REM

GOSUB B.01

          01810 LET PC(01) = 01810
          01811 IF PC(02) <> 02999 THEN 01999
          01812 LET PC(01) = 01999
          01813 LET STACKPTR(02) = STACKPTR(02) + 1
          01814 LET STACK(02,STACKPTR(02)) = 01816
          01815 GOSUB 02010
          01816 REM

RETURN

          02810 LET PC(02) = 02810
          02811 LET I = STACK(02,STACKPTR(02))
          02812 IF I = 02 THEN 02814
          02813 IF PC(I) <> ((I * 1000) +  999) THEN 02999
          02814 LET PC(02) = 02999
          02815 LET STACKPTR(02) = STACKPTR(02) - 1
          02816 GOTO STACK(02,STACKPTR(02) + 1)
          02817 REM
```

Figure 9.14: Code Generated to Support "GOTO", "GOSUB", and "RETURN"

For a non-local "GOTO", as shown in Figure 9.14, the local program counter is initially set to the address of the block of code. Next, the alien micro is tested to see if it is inactive (i.e. PC(MM) = MM999). If not, then execution of the current micro is suspended. If control can be transferred, then the current micro is made inactive by setting its PC to "MM999". The "GOTO" is then performed.

"GOSUB" is supported in a similar way, as shown in Figure 9.14. So that it is possible to "RETURN" control, and in order to check the status of the micro, "GOSUB" places the "label" of the current micro on the top of the destination micros stack. This is shown as statements 01813-4 in the example. The use of this information is shown by the code for "RETURN" on Figure 9.14.

Lastly, the "&" will be examined. This is only important when it is used to terminate control statements. For example: "LET A = B & LET D = B" is implemented sequentially by the translator. However, if a statement like a "GOTO" is terminated by a "&", its semantics becomes similar to a "FORK". Therefore, when the code for "GOTO" is generated, and the statement is terminated by "&", the "PC" is set to the label of the statement following the "GOTO". For example, in Figure 9.14 if the statement had been "GOTO B.01 &", then the third line would have been:

```
01812 LET PC(01) = 01814
```

Whereas, if the "GOTO" is terminated by ";" or "newline", indicating sequential execution (which could transfer control out of the micro, and hence suspend its execution), then the "PC" is set to "01999" as above.

In the next chapter, an analysis and assessment of the programming language BASAL will be made.

CHAPTER 10 - ANALYSIS OF BASAL

This chapter is an analysis of the BASAL programming language. As a vehicle for this analysis, the Quicksort example is again used. However, because BASAL, like BASIC, does not support recursion, the examples are not very "flattering" to BASAL.

Four possible strategies in distributing code and data could be followed in programming the Quicksort algorithm in BASAL. Firstly, the sort code and the array of data could reside in the same microcomputer. Secondly, the sort code could be distributed, one process per microcomputer, but the array of data would reside in a single microcomputer. Thirdly, the sort code would reside in a single microcomputer, but the array of data could be distributed. Lastly, both the sort code and the array of data could be distributed across the microcomputers. In addition, since BASAL supports both a "shared memory" and a "message passing" data mechanism, then each of the algorithms may communicate data either via variables or by messages.

In the following analysis, two Quicksort programs are presented. In both programs the strategy is to distribute the code (for parallelism) but have the data resident in a single microcomputer. The first program is based on shared memory communication, and the second on message passing communication.

10.1. PARALLEL (SHARED MEMORY) SORT

The example of a parallel Quicksort using shared memory was chosen to demonstrate how BASAL deals with the communication and synchronisation associated with this form of parallelism.

10.1.1. Description of Application

The essential idea of Quicksort, as discussed in Chapter 3, is to partition the original set to be sorted by rearranging it into two subsets; the first contains those elements which are less than some arbitrary "pivot" value chosen from the set, and the second those elements which are greater or equal to the value. Then the partitioning process is applied, in turn, to the two subsets, until each subset contains only one element. When all subsets have been partitioned, the original set has been sorted. In this case of a parallel sort, various "micros" perform the sorting of a given array in parallel, making use of shared memory.

The example of parallel Quicksort using shared memory, written in BASAL, uses different "micros" for the comparisons, partitions and exchanges for an array of sixteen elements. "MICRO A" contains the array "V" and the main loop of the sort program. The remaining micros "B", "C", "D", etc. form a "tree" of processes, each containing a copy of a sort process. "MICRO A" invokes micros "B" and "C", in turn "MICRO B" invokes "D" and "E" etc. When the array is sorted, control is returned to "MICRO A". The sorting of a subset of the array "V" involves the "PIVOT" - the first element of the set - plus two pointers "I" and "J". "LO" and "HI" contain the lowest and highest numbers in the subset, and are stored into a micro before the micro's sort process is invoked.

Since BASAL does not have repetitive statements such as "REPEAT UNTIL" and "DO WHILE", the commands to partition and exchange subsets in the array "V" have to be simulated by a series of "IFs" and "GOTOs".

The program was written in BASAL, and run through the BASAL translator, producing BASIC code. This code, in turn, was then executed by a BASIC system. The BASAL translator, when generating BASIC, creates extra "housekeeping" code (as shown in Chapter 9) to drive the various separate "micros", such as the indication of where the instructions for each "micro" begin and finish, etc.

10.1.2. Description of Program

The first micro, "MICRO A", contains the main loop of the sort code and the array "V". The code for it in BASAL is summarised in Figure 10.1. In this example the main statements of interest are statements 33 to 38. These statements store the lowest and highest numbers of the two subsets to be sorted into micros "B" and "C", and then invoke the two micros using "GOSUB".

```
MICRO A
01 DIM V(16)
05 LET LO = 1&
06 LET HI = 16
10 IF LO >= HI THEN 39
12 LET I = LO&
13 LET J = HI
14 LET PIVOT = V(LO)
15 IF J <= I THEN 20
17 IF V(J) < PIVOT THEN 20
18 LET J = J - 1
19 GOTO 15
20 IF I => J THEN 25
22 IF V(I) > PIVOT THEN 25
23 LET I = I + 1
24 GOTO 20
25 IF I => J THEN 30
27 LET TEMP = V(I)
28 LET V(I) = V(J)
29 LET V(J) = TEMP
30 IF I < J THEN 15
31 LET V(LO) = V(I)
32 LET V(I) = PIVOT
33 LET B.LO = LO&
34 LET B.HI = I - 1
35 GOSUB B.10&
36 LET C.LO = I + 1&
37 LET C.HI = HI
38 GOSUB C.10
39 STOP
```

Figure 10.1: Quicksort (shared memory) Program in BASAL

The sort code for "MICRO B", "MICRO C", etc., is similar to the one seen above until about line 33 where, for instance, in "MICRO B" it reads:

```
MICRO B
10 IF LO >= HI THEN 39
12 LET I = LO&
13 LET J = HI
       ...
32 LET A.V(I) = PIVOT
33 LET D.LO = LO&
34 LET D.HI = I - 1
35 GOSUB D.10&
36 LET E.LO = I + 1&
37 LET E.HI = HI
38 GOSUB E.10
39 RETURN
```

The code for the other micros from line 33 onwards is almost identical, only the appropriate variables for the specific "micros" being changed (e.g. for "MICRO C": LET E.LO = LO, LET E.HI = I - 1, etc.)

After being processed by the BASAL interpreter, the code generated for the parallel Quicksort ("MICRO A") is illustrated in Figure 10.2. Its approximate version is:

```
10 DIM PC(26)
20 DIM STACKPTR(26)
30 DIM STACK(26,20)

        ...

1000 GOTO PC(01)
1010 DIM A.V(16)
1100 IF A.LO >= A.HI THEN 1390
1120 LET A.I = A.LO
1130 LET A.J = A.HI
1140 LET A.PIVOT = A.V(A.LO)

        ...

1330 LET B.LO = A.LO
1340 LET B.HI = A.I - 1
1350 LET PC(01) = 1350
1351 IF PC(02) <> 02999 THEN 01999
1352 LET PC(01) = 01999
1353 LET STACKPTR(02) = STACKPTR(02) + 1
1354 LET STACK(02,STACKPTR(02)) = 01356
1355 GOSUB 2100
1356 REM
        ...
```

Figure 10.2: BASIC code for Figure 10.1

The code generated by the BASAL translator for the other micros (B, C, etc.), is quite similar to the one above, the main difference being (again) for lines corresponding to those ranging from 1330 and beyond, where for "MICRO B" it becomes "LET D.LO = B.LO", "LET D.HI = B.I - 1", etc.

10.1.3. Assessment

This parallel version of Quicksort, using shared memory was slightly difficult to program, having in mind that BASAL does not have statements such as "REPEAT-UNTIL" and "DO-WHILE", and profuse use of "GOTOs" had to be made. The language proved simple to use for writing the commands for a single sort process. However, the manual replication of the code was very tedious. This is related to two problems, firstly the absence of recursion or a PAR-command (as in OCCAM) that replicates processes, and secondly that processes are allocated statically to microcomputers rather than dynamically. In fact, these problems clearly relate to the choice of BASIC as the basis of BASAL, rather than the programming model.

10.2. PARALLEL (MESSAGE PASSING) SORT

Next, the example of a parallel Quicksort using message passing is examined. It was chosen to demonstrate how BASAL deals with the communication and synchronisation associated with this form of parallelism.

10.2.1. Description of Application

This example of a parallel Quicksort exploits the "[]" capability of BASAL, specially intended for use in message passing communication. Like in the example above, an array of sixteen elements is to be sorted, using a "tree" of micros, each containing a sort process. "MICRO A", containing the main loop, partitions the array passing elements less than or equal to the value of the pivot to "MICRO B", and elements greater than the pivot to "MICRO C". In turn, "MICRO B" partitions its subset passing elements to "D" and "E" etc. When the array is fully partitioned, the elements are passed back up the "tree" of sort processes, until "MICRO A" reads and merges the final two sorted subsets.

10.2.2. Description of Program

The first micro to attempt to sort the set of numbers in this example is, again, "MICRO A". The BASAL code corresponding to the process "MICRO A" is summarised in Figure 10.3. In Figure 10.3, a number of points should be noted. When "MICRO A" starts executing, it initially "forks" control to micros "B" and "C" to start them executing:

```
GOTO B.01&
GOTO C.01&
```

Note, "MICRO A" partitions the array "V" putting elements into either "B.IN[]" or "C.IN[]". Lastly, a terminator "30999" is output to both messages.

```
MICRO A
01 DIM V(16)
05 GOTO B.01&
06 GOTO C.01&
10 LET PIVOT = V(1)
11 FOR I = 2 TO 16
12 IF V(I) > PIVOT THEN 15
13 LET B.IN[] = V(I)
14 GOTO 16
15 LET C.IN[] = V(I)
16 NEXT I
20 LET B.IN[] = 30999&
21 LET C.IN[] = 30999
30 LET I = 1
31 LET V(I) = B.OUT[]
32 LET I = I + 1
33 IF V(I - 1) <> 30999 THEN 31
34 LET V(I - 1) = PIVOT
36 LET V(I) = C.OUT[]
37 LET I = I + 1
38 IF V(I - 1) <> 30999 THEN 36
39 STOP
```

Figure 10.3: Quicksort (message passing) Program in BASAL

Having partitioned the array, "MICRO A" then attempts to take the two sorted subsets from "B.OUT[]" and "C.OUT[]" and merge them, placing the results back into the array "V".

The code for "MICRO B" etc. is different in this version of Quicksort using message passing, as can be seen in Figure 10.4. Its operation is fairly straightforward, and should not require further explanation.

```
MICRO B
01 LET PIVOT = B.IN[]
02 IF PIVOT = 30999 THEN 39
03 GOTO D.01&
04 GOTO E.01&
10 LET X = B.IN[]
11 IF X = 30999 THEN 20
12 IF X > PIVOT THEN 15
13 LET D.IN[] = X
14 GOTO 10
15 LET E.IN[] = X
16 GOTO 10
20 LET D.IN[] = 30999&
21 LET E.IN[] = 30999
30 LET X = D.OUT[]
31 IF X = 30999 THEN 34
32 LET B.OUT[] = X
33 GOTO 30
34 LET B.OUT[] = PIVOT
35 LET X = E.OUT[]
36 IF X = 30999 THEN 39
37 LET B.OUT[] = X
38 GOTO 35
39 LET B.OUT[] = 30999
40 STOP
```

Figure 10.4: Quicksort code for "MICRO B" (etc.)

Finally, the BASIC code generated for the BASAL commands in "MICRO A" is illustrated in Figure 10.5.

```
10 DIM PC(26)
20 DIM STACKPTR(26)
30 DIM STACK(26,20)
1000 GOTO PC(01)
1010 DIM A.V(16)
1050 LET PC(01) = 01050
1051 IF PC(02) <> 02999 THEN 01999
1052 LET PC(01) = 01054
1053 GOTO 02010
1054 REM

        ...

1100 LET A.PIVOT = A.V(1)
1110 FOR A.I = 2 TO 16
1120 IF A.V(A.I) > A.PIVOT THEN 1150
1130 LET PC(01) = 1130
1131 IF B.IN <> -32768 THEN 1999
1136 LET B.IN = A.V(A.I)
1140`GOTO 1160
1150 LET PC(01) = 1150
1151 IF C.IN <> -32768 THEN 01999
1156 LET C.IN = A.V(A.I)
1160 NEXT A.I

        ...
```

Figure 10.5: BASIC code for Figure 10.3

Listings of the BASAL Quicksort programs, both shared memory and message passing, are given in Appendix A.8, together with the corresponding outputs for the translator.

10.2.3. Assessment

This parallel version of Quicksort using message passing presented similar difficulties in programming to the previous Quicksort examples. These are: the absence of commands to replicate code (e.g. recursion or PAR) and the fact that processes must be statically rather than dynamically allocated. When comparing the two Quicksort algorithms, the message passing version has the advantage over the shared memory that all micros are not competing for access of the array "V" in "MICRO A's" memory.

10.3. ANALYSIS AND ASSESSMENT

In general, BASAL is a very simple language to program in (like BASIC) and has the advantage of being very close to the underlying RIMMS architecture. However, this flexibility is at the expense of "safeness" in the parallel programs written. The three main "problem" statements are:

```
LET name = expression
GOTO label
GOSUB label
```

The "LET" can assign values, non-deterministically, anywhere in the global address space. The "GOTO" can transfer control to any microcomputer (although, recall, control will only be transferred if the destination micro is halted). "GOSUB" can call any label in the global address space, as if it were an "entry-point". In addition, parameters must all be passed by global variables.

Besides this problem of encapsulating flows of data and control in BASAL programs, there is the previously mentioned problem of replicating processes and even of having dynamic process creation. This could be approached in two ways: either using recursion or a "PAR-statement" operating on "MICRO micro_name".

To improve encapsulation of information in a parallel language like BASAL, it is clearly necessary to restrict the flows of data and control. Flows of data can be restricted by introducing parameterised processes, as in OCCAM, and IMPORT/EXPORT statements, as in MODULA_2. For parameterised processes, the command "PROC micro_name{(local_name{,local_name}...)}" would replace "MICRO micro_name". For IMPORT/EXPORT, two new statements would be introduced into BASAL: "IMPORT micro_name.local_name" defining non-local access to a variable, and "EXPORT local_name", defining a variable that may be accessed non-locally by another process.

Flows of control can be restricted by only allowing GOTOs to transfer control locally, "GOTO local_label", and by replacing "GOSUB label" by "CALL micro_name{(local_name{,local_name}...)}. This restricts a process to a single "entry-point" namely "micro_name", and defines the local names that are to be common to the two processes. (This assumes a "call-by-reference" form of parameter passing.) Thus, the new syntax is:

```
PROC micro_name{(local_name{,local_name}...)}

IMPORT micro_name.local_name
EXPORT local_name
GOTO   local_label
CALL   micro_name{(local_name{,local_name}...)}
```

Next, the problems of replicating code and having dynamic creation of processes are examined.

To improve the replication of code (while keeping the static allocation of processes) the OCCAM "PAR i = [1 FOR n]" could be adopted. This effect could be achieved in BASAL by introducing an optional integer field in the process declaration:

```
PROC micro_name{(integer)}
```

defining the number of the copies of the process to be generated. It would then be necessary to introduce an optional field into micro_name "alphabetic{(integer)}" defining which process is being accessed. Although this change could be easily introduced into the BASAL syntax, it seems to be inelegant when compared, say, to recursion which achieves a similar effect.

Recursion seems the best approach to handle the dynamic creation of processes. When the processes are declared, each could be statically allocated to a microcomputer. However, should any process be called recursively, then the process' code would be dynamically copied into a new (unallocated) microcomputer. The microcomputers could be envisaged as being allocated as if forming a stack.

Finally, as an illustration of the effect of these changes, Appendix A.7 contains the syntax of the improved BASAL. The main merit of BASAL is in the fact that it allows the parallel programming of multi-microcomputers, but on the other hand still manages to have a simple syntax and semantics.

CHAPTER 11 - CONCLUSIONS

This chapter presents the conclusions drawn from this investigation of programming decentralised computers.

11.1. SUMMARY

A summary of the work presented in this Thesis is initially given.

In Chapter 2, images of various computer systems that could be in operation in the future were described. These were: Fifth Generation Computers, Supercomputers, VLSI Processor Architectures, and Integrated Communications & Computers. The former two images are of "parallel machines" supporting a "revolutionary" new programming model, namely logic and data flow, respectively. The latter two images are of "decentralised computers" supporting "evolutionary" control flow programming models. It concludes that a decentralised computer architecture capable of spanning distributed, parallel and sequential computers, is the most appropriate image for future computers.

In Chapter 3, the major programming styles that could be used to program these future decentralised computers were presented and classified. These styles cover procedural programming, including conventional and concurrent languages; object-oriented programming; functional programming, including data flow and applicative languages; and logic programming; as well as new forms of application programming, including electronic-sheet languages. The basis for this classification of programming styles was the computational (data and control) mechanisms that underlie their programming models.

In Chapters 4 and 5, each programming style was analysed, using a common Quicksort algorithm plus the data mechanisms and control mechanisms presented in Chapter 3. Chapter 4 analysed procedural programming including conventional and concurrent languages, and object-oriented programming, identifying their advantages and disadvantages. Chapter 5 analysed functional programming including data flow and applicative languages, and logic programming including Horn clause languages. These Chapters concluded that control flow (and procedural programming) was the most primitive and fundamental programming model.

In Chapter 6, based on the conclusions of Chapters 2 to 5, the so-called decentralised control flow programming model was presented. This programming model embodies a "decentralised computer" image of computers and is based on control flow. It was shown how this model generalises the traditional von Neumann model and, in fact, already provides the basic concepts underlying modern operating systems. It was argued that the decentralised control flow programming model should form the basis of future decentralised computer systems and their corresponding programming languages.

In Chapters 7-10, two programming languages called BASIX and BASAL embodying the decentralised control flow model were presented. BASIX and BASAL were used to investigate the style of decentralised control flow programming languages, and were not meant to propose new languages. Both these languages are primitive and are "low-level" system programming languages (cf. C) rather than "high-level" languages (cf. PROLOG). Chapter 7 presented the BASIX language which attempts to combine the fundamental concepts of the UNIX Shell, LISP and BASIC. It is intended as a "total system", providing a complete interactive programming environment (cf. SMALLTALK). Chapter 9 presented the BASAL language, based on a primitive form of decentralised control flow, and designed for programming the RIMMS multi-microcomputer system. BASAL is a superset of BASIC. BASIX and BASAL can be viewed as representing the opposite ends of the spectrum of languages based on the decentralised control flow model.

Below the four major areas and contributions of this Thesis are summarised. They are: the classification of programming styles, the decentralised control flow model, the BASIX languages, and the BASAL languages.

11.1.1. Classification of Programming Styles

The classification and analysis of the major styles of programming (presented in Chapters 3, 4 and 5) attempt to quantify the observable advantages and disadvantages of programming languages. The belief is that these advantages and disadvantages directly relate to the computational mechanisms underlying the particular programming model associated with these languages.

For a programming model there are two basic computational mechanisms referred to, in this Thesis, as the data mechanism and the control mechanism. The data mechanism defines the way a particular argument is communicated by a number of commands. There are two basic types, referred to as: "shared memory" and "message passing". The control mechanism defines how one command causes the execution of one or more other commands. There are four basic types referred to as: "control driven", "data driven", "demand driven", and "pattern driven".

In terms of the data mechanism, firstly "shared memory" has advantages for sharing data structures and for allowing an unspecified number of copies of data to be taken, but has the disadvantage of not supporting synchronised access to its contents, particularly by parallel commands. Secondly, "message passing" has the advantage of synchronised communication of data, but has the disadvantages of not supporting sharing of data structures and of often needing to know all consumer commands.

For the control mechanism: firstly "control driven" has the advantage of being very primitive and flexible, but the disadvantage of being relatively easy to misspecify in terms of the sequences of execution; secondly, "data driven" has the advantage of being "naturally" parallel, but the disadvantage of being unable to control unnecessary evaluation; thirdly, "demand driven" has the advantage of performing minimum execution, but has the disadvantage of restricting the control pattern to a tree structure; and lastly, "pattern driven" has the advantage of being the highest level control mechanism, but, in consequence, the disadvantage of sometimes not allowing sufficient control over the execution of a program.

It was noted in Section 6.1 that, significantly, each category of programming models regards the data mechanisms and the control mechanisms as largely incompatible sets of alternative concepts. Hence each category, although Universal (cf. Turing machine) has specific advantages and disadvantages for computation, related to its choice of data and control mechanisms. More significantly, the classification of programming styles seems to show that "shared memory" is the most important data mechanism and "control driven" execution is the most primitive control mechanism. Thus Section 6.1 concludes that control flow is the most fundamental programming model for computers.

11.1.2. Decentralised Control Flow

The decentralised control flow model, presented in Chapter 6, has the following principles:

1. computer - a computer system is a decentralised computer (hierarchy of distributed, parallel and sequential computers);
2. network - a nested organisation of variable-size memory cells (like the file structure of an operating system);
3. addressing - a contextual address space of cells (like telephone numbers);
4. program - a higher-level machine language (as in LISP, where instructions may be recursively defined);
5. communication - shared memory and message passing communication of data;
6. execution - parallel, decentralised control of computation (as with UNIX commands).

An essential concept in this decentralised control flow model is the direct functional correspondence between hardware and software. In addition, memory is recursively structured allowing any level of object to be accessed and this makes it a flexible and powerful memory model.

The use of contextual addressing, as in modern operating systems, is another advantage. Memory cells close to the current context can be addressed using short addresses, with the length of these addresses increasing with the different levels to be reached. The concept of memory cells in the decentralised control flow model supports both "shared memory" and "message passing" data mechanisms, which gives it more flexibility than other programming models. One point that could be improved, though, is that more addressing modes (such as content addressing) would be beneficial to the model.

Instructions in the decentralised control flow model have a single format, with a procedure call mechanism being intrinsically built-in. The single format is quite simple. The fact that the way programs and data are presented is not strictly identical (as in LISP) requires improvement.

One of the great strengths of the decentralised control flow model is to generalise sequential control flow. As a negative counterpart to that, the current program execution is conceptually quite complex, and the controls between commands are still a problem. Lastly, the support for other programming models is still not very effective at the moment.

11.1.3. BASIX Languages

The BASIX languages, mainly the BASIX_2 language presented in Section 7.3 has, like the decentralised control flow model it was designed to mirror, a recursive concept of "object". The "objects", very similar to LISP, are specified via the use of delimited strings. But, as it was pointed out above for the model (Section 11.1.2), unlike in LISP, programs and data are not represented in exactly the same way.

Addresses in BASIX_2 can be individual selectors or a sequence of selectors, but the addressing notation has its problems. One of them is the case where "a/i" means the component with the explicit selector "i:(...)", while name "a/(i)" is equivalent to the traditional "a[i]". Beside that, locations can have numeric selectors, leading to the impossibility of differentiating between numbers and selectors. This can be overcome by prefixing a single numeric selector with "./".

One of the main positive points of the BASIX_2 language is that it has a traditional syntax, even though it is based on a decentralised programming model. The current problems in the language's syntax can be found, though, in the statements "if-fi" and "do-od", as well as in "for-rof". Ideally, the language would have both iterative statements and statements that replicate other statements.

Finally, the fact that BASIX_2 attempts to generalise conventional languages makes it a powerful and "comprehensive" language, but this is perhaps offset by the fact that the current semantics tend to be somewhat complex.

11.1.4. BASAL Language

The BASAL language, presented in Chapter 9, is a simple, easy to understand language, but it suffers from the normal problems of its parent BASIC, such as having a primitive notion of nested object, unlike BASIX which allows access to variables and files alike.

The addressing scheme in BASAL is a two-level one, which is easy to understand, and the program representation is quite simple. The major problem of BASAL, as discussed in Section 10.3, is its inability to dynamically create processes, or even, as OCCAM does, of specifying the replication of a group of processes that are based on the same code.

Finally, BASAL is very close to traditional control flow (presenting no "traps" for users of conventional languages), but it has the important advantage of supporting both "shared memory" and "message passing" data mechanisms.

11.2. FUTURE WORK

Future work is clearly required in each of the four areas: the classification of programming styles, the decentralised control flow model, the BASIX languages, and the BASAL languages.

11.2.1. Classification of Programming Styles

The classification, and resulting analysis, of programming styles presented in Chapters 3, 4 and 5, make two possible contributions: firstly the data and control mechanisms may, in some sense, be fundamental to computation, contributing in the future to improvements in the design of programming languages, and secondly (even if this is not the case) the classification is believed to aid in understanding the various programming styles and the strengths and weaknesses of their associated programming languages.

Having said this, no classification is likely to be perfect. For instance it is arguable whether the current classification clearly differentiates between applicative (pattern-matching) languages, which are based on a graph reduction programming model, and logic languages, based on a logic model. So it will be necessary, in the future, to test the classification by using it to study other programming styles such as expert systems building languages [60].

11.2.2. Decentralised Control Flow

Future work on the Decentralised Control Flow model can be identified in the areas of addressing, program representation, program execution, and supporting of other programming models. For addressing, one point that could be improved is to increase the number of addressing modes (such as content addressing). For program representation it would be interesting to have a fully recursive format for commands as in LISP. This would allow any argument of a command to be a complete program fragment. However, this would probably prove very complex to implement in a computer architecture, and may complicate the semantics of the model. Concerning support of other models, the decentralised control flow model currently supports "data driven" and "demand driven" execution. This leaves only the support of "pattern driven" execution of commands to be investigated.

However the main problem to be tackled with the current decentralised control flow model is to provide a good implementation (which should include memory management) of the information structure and the addressing scheme, both of which need to be made efficient. This should also help refine the semantics of the programming model.

11.2.3. BASIX Language

In BASIX_2 perhaps the major improvement needed would be to make the language less recursive and thus more simple. The concept of "object" seems to cover too many semantic fields, and the language should benefit from the simplification.

Essentially four basic types of objects would be retained: "expression", "statement", "({object}...)", and "local_name:object". An expression consists of one or more simple objects, separated by operators. A statement is a list of objects whose leftmost object is a keyword or the name of a program object. A bracketed list of objects may be code or data, and lastly comes the declaration of a "name:object" pair in the local context.

Control statements, which had proved difficult to use in the current version of the BASIX, could be simplified by using Dijkstra's guarded commands: "IF {expression -> command}... FI" and "DO {expression -> command}... OD". The "GOTO" would be restricted to a local context, by using a "local_name". Names and selectors (which have been one of BASIX more serious problems) could, in the future, be defined as: "local_name{.selector}..." and "${.selector}...", "local_name" being an alphanumeric character string, and "selector" being redefined as either a "local_name", "numeric", or "(expression)", in the hope that this would bring about the simplifications of both syntax and semantics.

11.2.4. BASAL Language

In BASAL, it would be perhaps worthwhile to make the most primitive element not a word, but an object that may have structure. Also, in a parallel language such as BASAL, there is a need to encapsulate information. This could be done by restricting the flows of data and control. The introduction of parameterised processes and of IMPORT/EXPORT statements would deal with the restriction of flows of data, while the flows of control would be restricted by only allowing GOTOs to transfer control locally. It would, also, be very useful to have some ability to reference files as in, say, delimited strings.

However, the most important current limitation in BASAL is the inability to replicate code or dynamically allocate processes to micros. In order to improve this, an optional integer field could be introduced in the process declaration, which would define the number of copies of the process to be generated. So that these processes could be accessed, an optional field would need to be included in "micro_name", such as "alphabetic{(integer)}" defining which process is being accessed. Another, more elegant solution, would be the introduction of recursion. The processes, when declared, would be statically allocated to a microcomputer. In case of a process being called recursively, the code of that process would be copied (dynamically) into another (new, unallocated) microcomputer. This idea would be analogous to a stack, formed by allocated microcomputers.

11.3. FINAL CONCLUSIONS

In the computer science community there is a growing belief that the traditional von Neumann computer may be superseded over the next decade by a new decentralised computer programming model. Various categories of programming models (i.e. data flow, reduction, actor, logic) are being promoted as the von Neumann successor. The most prominent are logic for Fifth Generation Computers and data flow for Supercomputers. A problem with these novel parallel models is that they are largely unproven and represent a "revolutionary" solution, which discards the massive investment in traditional control flow computing.

However, it is felt that the evolutionary approach of a control flow model embodying a decentralised computer architecture - which is called decentralised control flow - is a more promising way of achieving many of the very ambitious goals characterising Japan's FGCS Project

[39,53]. This view, presented here, is partly conservative in recognising the greater practicality (in a world of existing systems and expertise) of persuading individuals and organisations to try out an evolutionary development. However, there is also the belief that control flow is a more fundamental model of computation (see Section 6.1) than the four other categories of models. In addition, the decentralised control flow principles (described in Chapter 6) already form the basis of modern operating systems such as UNIX. In effect, the basic programming model needed for future computers is already in use.

In choosing a low-level programming language, it is believed that the next generation of decentralised control flow programming language should fall somewhere between the current BASIX and BASAL languages, attempting to capture the simplicity of BASAL, and the sophistication of BASIX.

In conclusion, highly parallel and decentralised programming models will, and should, only supplant the traditional von Neumann model if they can match the latter's generality and flexibility, as exemplified by the large variety of both conventional and very novel programming languages and styles that it supports with reasonable effectiveness. The important aspect of the von Neumann model which gives this flexibility is that it is a control flow model allowing the programmer (or compiler/interpreter) direct control over the low level operation of the target machine when this is necessary. Thus, the key to the future generation of programming models would be identified as being some extended form of control flow which overcomes its deficiencies for decentralised concurrent systems, but retains its flexibility and generality.

REFERENCES

[1] Ackerman W.B.: "Data Flow Languages", IEEE COMPUTER, vol.15, no.2 (February 1982), pp. 15-25.

[2] Ackerman W.B. and Dennis J.B.: "VAL - A Value Oriented Algorithmic Language: Preliminary Reference Manual", Tech. Report TR-218, Laboratory for Computer Science, MIT (June 1979).

[3] Amamiya M.: "A Design Philosophy of High Level Language for Data Flow Machine Valid", Proc. Annual Conf. of IECE Japan (1981).

[4] Anon : "Special Issue on the Programming Language Smalltalk", Byte (August 1981).

[5] Arvind et al: "An Asynchronous Language and Computing Machine", Tech. Report 114a., Department of Information and Computer Science, University of California, Irvine (December 1978).

[6] Arvind and Gostelow K.P.: "The U-Interpreter", IEEE COMPUTER, vol.15, no.2 (February 1982), pp. 42-49.

[7] Arvind and Iannucci R.A.: "A Critique of Multiprocessing von Neumann Style", Proc. Tenth Int. Symp. on Computer Architecture (June 1983), pp. 426-436.

[8] Ashcroft E.A. and Wadge W.W.: "LUCID, a nonprocedural language with iteration", Comm. ACM, vol.20, no.7 (July 1977), pp. 519-526.

[9] Backus J: "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Comm. ACM, vol.21, no.8 (August 1978), pp. 613-641.

[10] Barron I. et al: "Transputer does 5 or more MIPS even when not used in parallel", Electronics, vol.56, no.23 (November 1983), pp. 109-115.

[11] Berkling K.: "Reduction Languages for Reduction Machines". Proc. Second Int. Symp. on Computer Architecture (January 1975), pp. 133-140.

[12] Birtwistle G.M. et al: "SIMULA BEGIN", 2nd. edition, van Nostrand Reinhold (1979).

[13] Bonner S. and Shin K.G.: "A Comparative Study of Robot Languages", IEEE COMPUTER, vol.15, no.12 (December 1982), pp. 82-96.

[14] Bricklin D. and Frankston B.: "VisiCalc Computer Software Program", Reference Manual, Personal Software Inc., Sunnyvale, California (1979).

[15] Brinch Hansen P: "The Programming Language Concurrent Pascal", IEEE Trans. on Software Eng., vol.SE-1, no.2 (June 1975), pp. 199-207.

[16] Brownbridge D. et al: "The Newcastle Connection or UNIXes of the World Unite!", Software - Practice and Experience, vol.12, (December 1982), pp. 1147-1162.

[17] Davis A.L. and Keller R.M.: "Data Flow Program Graphs", IEEE COMPUTER, vol.15, no.2 (February 1982), pp. 26-41.

[18] Dennis J.B.: "The Varieties of Data Flow Computers". Proc. First Int. Conf. on Distributed Computing Systems, (October 1979), pp. 430-439.

[19] Dewar R.K. et al: "Programming by Refinement, as Exemplified by the SETL Representation Sublanguages", ACM Trans. on Programming Languages and Systems, vol.1, no.1 (July 1979), pp. 27-49.

[20] Duda R.O. and Gaschnig J.G.: "Knowledge-based Expert Systems Come of Age", Byte (September 1981), pp. 238-281.

[21] Dijkstra E.W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Comm. ACM, vol.18, no.8 (August 1975), pp. 453-457.

[22] Feigenbaum E. A.: "Knowledge Engineering: The Applied Side of Artificial Intelligence", Memo HPP-80-21, Computer Science Dept., Stanford University, 1980.

[23] Ferguson R.: "PROLOG A Step Towards the Ultimate Computer Language", Byte (November 1981), pp. 384-399.

[24] Fisher A.L. et al: "Architecture of the PSC: A Programmable Systolic Chip", Proc. Tenth Int. Symp. on Computer Architecture, (June 1983), pp. 48-53.

[25] Foti L. et al: "Reduced Instruction set Multi-microcomputer System (RIMMS)", 1984 National Computer Conference (to be presented).

[26] Foti L. et al: "Programming the RIMMS Multi-Microcomputer", University of Newcastle upon Tyne, Computing Laboratory, Internal Report (September 1983).

[27] Gouveia Lima I. et al: "Decentralised Control Flow - BASed on unIX", Proc. ACM SIGPLAN 83 Conference, San Francisco, California, (June 1983), pp. 192-201.

[28] Gouveia Lima I. et al: "Decentralised Control Flow Programming", Proc. IFIP Congress, Paris (September 1983), pp. 487-492.

[29] Gupta A. and Toong H.D.: "An Architecture Comparison of 32-bit Microprocessors", IEEE MICRO, vol.3, no.1 (February 1983), pp. 9-22.

[30] Henderson P.: "Functional Programming - Application and Implementation", Prentice-Hall (1980).

[31] Hoare C.A.R.: "Quicksort", Computer Journal, vol.5, no.1, (April 1962), pp. 10-15.

[32] Hoare C.A.R.: "Communicating Sequential Processes", Comm. ACM, vol.21, no.8, (August 1978), pp. 666-677.

[33] Ingalls D.H.H.: "The Smalltalk-76 Programming System Design and Implementation", Proc. Fifth ACM Symp. on Principles of Programming Languages (January 1978), pp. 9-15.

[34] Kernighan B.W. and Plauger P.J.: "Software Tools in Pascal". Addison-Wesley Publishing Company (1981).

[35] Kobayashi K.: "Computer, Communications and Man: The Integration of Computer and Communications with Man as an Axis", COMPUTER NETWORKS - The Int. Journ. of Distributed Informatique, vol. 5, no. 4 (July 1981), pp. 237-250.

[36] Kowalski R.: "Logic for Problem Solving", Elsevier-North Holland Publishing Company (1979).

[37] Kung H.T.: "Why Systolic Arrays", IEEE COMPUTER, vol.15, no.1 (January 1982), pp. 37-46.

[38] Mead C.A. and Conway L.A.: "Introduction to VLSI Systems", Addison-Wesley Publishing Company (1980).

[39] Moto-oka T. et al: "Challenge for Knowledge Information Processing Systems (Preliminary Report on Fifth Generation Computer Systems)", Proc. Int. Conf. on Fifth Generation Computer Systems, North-Holland Publishing Company (1982).

[40] Mundy D.H.: "Decentralised Control Flow - A Computational Model for Distributed Systems", PhD. Thesis, Computing Laboratory, University of Newcastle upon Tyne (in preparation).

[41] Patterson D. and Sequin C.: "A VLSI RISC", IEEE COMPUTER, vol.15, no.9 (September 1982), pp. 8-21.

[42] Patterson D. and Ditzel D.: "The Case for the Reduced Instruction Set Computer", Computer Architecture News, vol.8, no.6 (October 1980), pp. 25-32.

[43] Pyle I.C.: "The ADA Programming Language: a Guide for Programmers", Prentice-Hall (1981).

[44] Richie D.M. and Thompson. K.: "The UNIX Time-Sharing System", Comm. ACM, vol.17, no.7 (1974), pp. 365-375.

[45] Seitz C.: "Ensemble Architectures for VLSI - A Survey and Taxonomy", Proc. 1982 Conf. on Advanced Research in VLSI, P. Penfield ed., MIT, (January 1982), pp. 33-45.

[46] Servan-Schreiber J.J.: "The World Challenge", Collins (1981).

[47] Shrivastava S.K. and Panzieri F.: "The Design of a Reliable Remote Procedure Call Mechanism" IEEE Trans. on Computers, vol. C-31, no.7 (July 1982), pp. 692-697.

[48] Stotts P.D.: "A Comparative Survey of Concurrent Programming Languages", ACM SIGPLAN Notices, vol.17, no.9 (September 1982), pp. 76-87.

[49] Taylor R. and Wilson P.: "OCCAM Process-oriented language meets demands of distributed processing", Electronics (November 1982), pp. 89-95.

[50] Treleaven P.C. et al: "Data Driven and Demand Driven Computer Architecture", ACM Computing Surveys, vol. 14, no. 1 (March 1982), pp. 93-143.

[51] Treleaven P.C. and Hopkins R.P.: "A Recursive Computer Architecture for VLSI", Proc. Ninth Int. Symp. on Computer Architecture (April 1982), pp. 229-238.

[52] Treleaven P.C.: "VLSI Processor Architectures", IEEE COMPUTER, vol.15, no.6 (June 1982), pp. 33-45.

[53] Treleaven P.C. and Gouveia Lima I.: "Japan's Fifth Generation Computer Systems", IEEE COMPUTER, vol.15, no.8 (August 1982), pp. 79-88.

[54] Treleaven P.C. and Gouveia Lima I.: "Future Computers - Logic, Data Flow,...,Control Flow", IEEE COMPUTER, vol.17, no.3 (March 1984), pp. 45-55.

[55] Turner D.A.: "A New Implementation Technique for Applicative Languages", Software - Practice and Experience, vol.9 (1979), pp. 31-49.

[56] Turner D.A.: "Programming Languages - Current and Future Developments", Proc. Software Development Techniques, Infotech State of the Art Conf. (1980), pp. 7/1-7/12.

[57] Uchida S.: "Towards a New Generation Computer Architecture", Tech. Report TR/A-001, Institute for New Generation Computer Technology (July 1982).

[58] Uchida S. et al: "The Personal Sequential Inference Machine", Tech. Report TM-002, Institute for New Generation Computer Technology (November 1982).

[59] Warren D.H.: "Logic Programming and Compiler Writing", Dept. of Artificial Intelligence, Univ. of Edinburgh, Research Report no.44 (September 1977).

[60] Waterman D.A. and Hayes-Roth F.: "An Investigation of Tools for Building Expert Systems", Memo R-2818-NSF, Rand Corp., Santa Monica, California (June 1982).

[61] Wilkes M.V. and Wheeler D.J.: "The Cambridge Communication Ring", Proc. of Local Area Network Symposium, Boston, National Bureau of Standards (May 1979).

[62] Wilner W.T.: "Recursive Machines", Xerox Palo Alto Research Center, Internal Report (1980).

[63] Wirth N.: "Programming in MODULA_2", Springer-Verlag, Berlin (1980).

APPENDIX A.1 - BASIX_1 Programming Language

NAME

BASIX_1

SYNOPSIS

BASIX_1 { name }

DESCRIPTION

BASIX_1 has a decentralised control flow operational model. Its syntax is a superset of BASIC, but it incorporates features both from LISP and from UNIX. **BASIX_1** commands can be simple statements as in BASIC, or can be delimited groups of statements or commands, as in LISP. **BASIX_1's** environment is similar to UNIX. When **BASIX_1** is invoked the user program input has access to any 'files' - viewed as data structures by the program - previously created. If any **'name'** argument is provided when **BASIX_1** is invoked, the associated structures are used for input before reading the terminal.

Commands have the following syntax:

statement
: The statement is immediately executed. The result of an immediate command is printed.

integer statement
: Integer numbered statements (known as internal commands) are stored for later execution. They are stored in sorted ascending order.

(command { command }...)
: The (command { command }...) is executed when the ')' is reached.

integer (command { command }...)
: Similarly to the 'integer numbered statements', these are stored for later execution.

Statements have the following syntax:

comment
: This statement is ignored. It is used to interject commentary in a program.

dim alphanumeric (integer {, integer }...)
: This statement is used to create either temporary or semi-permanent data structures. When used in the form

```
dim alphanumeric ( integer {, integer }... )
```

it creates a semi-permanent data structure ('file') which will not be deleted at the end of the program. When used in the form

```
integer dim alphanumeric ( integer {, integer }... )
```

it creates a temporary data structure which will disappear at the end of the program.

done
Return to system level.

dump
The name and current value of every variable is printed.

for name = expression expression statement

for name = expression expression

...

next
The for statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression.

fork expression
The expression is evaluated, truncated to an integer and a secondary thread of execution starts at the corresponding integer numbered command. The primary thread of execution continues to execute the statement following the 'fork'. Also see '**join**' statement.

join expression
The expression is evaluated and truncated to an integer. This positive integer defines the number of threads of control to be received by the '**join**' before sequential execution (of the following statement) is resumed.

goto expression
The expression is evaluated, truncated to an integer and execution goes to the corresponding integer numbered statement. If executed from immediate mode, the internal statements are compiled first.

if expression statement

if expression

 ...

{ **else**

 ... }

fi

The 'if' statement (first form) or group of statements (second form) is executed if the expression evaluates to non-zero. In the second form, an optional else allows for a second group of statements to be executed when the first group is not.

let name = expression

This is the assignment statement. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left.

list { expression } { expression }

Is used to print out the stored internal arguments. If no arguments are given, all internal statements are printed. If one argument is given, only that internal statement is listed. If two arguments are given, all internal statements inclusively between the arguments are printed.

print list

The list of expressions and strings are concatenated and printed. (A string is delimited by " characters.)

prompt list

Prompt is the same as print except that no newline character is printed.

return { expression }

The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

run

Control is passed to the lowest numbered internal statement.

save { expression } { expression }

Save is like list except that the output is written on the file argument.

expression

The expression is executed for its side effects or for printing as described above.

Expressions have the following syntax:

number
    A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an 'e' followed by a possibly signed exponent.

(expression)
    Parentheses are used to alter normal order of evaluation.

_expression
    The result is the negation of the expression.

expression operator expression
    Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. A complete list of operators is given below.

name
    A name is used to specify a variable.

name( )
    Procedures and functions can be called by an name followed by parentheses. The name evaluates to the line number of the entry of the procedure or function in the internally stored statements. This causes the internal statements to be compiled.

Names have the following syntax:

0   (zero)
    The current context becomes the selected variable.

alphanumeric
    An alphanumeric is used to specify a variable in the current context. Alphanumerics are composed of a letter followed by letters or digits.

expression
    The expression is truncated to an integer and used as a specifier for the name.

name|name{|name}...
    A name can also be a sequence of selectors, used to access structures such as arrays.

The following is the list of operators:

& V
    & (logical and) has result zero if either of its arguments are zero. It has result one if both its arguments are non-zero. V (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments are non-zero.

< <= > >= = <>

The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, = equal to, <> not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: a > b > c is the same as a > b & b > c.

+ - * / **

The arithmetic operators add, subtract, multiply, divide and exponentiation.

APPENDIX A.2 - BASIX_2 Programming Language

NAME

BASIX_2

SYNOPSIS

BASIX_2

DESCRIPTION

BASIX_2 has a decentralised control flow operational model. Its syntax attempts to combine some of the most important characteristics of BASIC, UNIX Shell, and LISP. For instance, BASIX_2 has a single notion of **object** which serves the roles of variables, lists, messages, programs, files and directories. BASIX_2 has a nested information structure and a contextual address space similar to the UNIX Shell - a hierarchy of "name : object" pairs. BASIX_2 commands can be simple expressions and statements as in BASIC, or can be delimited groups of commands as in LISP. Interaction is via terminal screens which display the information structures of active contexts as windows. (Shared contexts appear as identical windows.) A window is divided into three areas:

```
                   --------------------------------
                  | Context :                       |
            --------------------------------        |
           | Context :                      |       |
      --------------------------------      |       |
     | Context :                      |     |       |
     |                                |     |       |
     | name :  (     .   .   .   )    |     |       |
     | name :  (     .   .   .   )    |     |       |
     | name :  (     .   .   .   )    |     |       |
     |                                |     |-------
     |    .  .  .                     |     |
     |                                |-----
     | Command :                      |
      --------------------------------
```

defining the current context, the contents of the context, and the commands typed by the user. Information in any of the three areas may be changed by positioning the cursor and typing the new information. A new context name changes the current context. New information changes the contents of the context. Lastly, a new command is executed.

Commands have the following syntax:

name : object

Declares a "name : object" pair relative to the local context. Only the "name" is evaluated before the assignment.

object

The object is executed and either returns some value to the user's screen or makes some change to the information structure.

Objects have the following syntax:

expression
: An expression is a sequence of statements or objects separated by operators.

statement
: A list whose leftmost object is a keyword.

(object { object }...)
: A list of one or more objects, data or program, separated by spaces or commas.

(command { control command }...)
: A series of commands separated by controls; each control defines the order of execution of the two adjacent commands.

Expressions have the following syntax:

name
: The object, synonymous with the name, is treated as a variable.

name[]
: The object, synonymous with the name, is treated as a list or a message.

number
: The object is an integer number.

()
: This is the "undefined" object, and any access to it is delayed until its contents are available.

quote object
: The result is the unevaluated object.

_object
: The result is the negation of the expression.

object operator object
: The objects are evaluated as operands for the operator and the whole expression returns a value.

name({ object }...)
: A procedure or function with zero or more parameters may be specified in the traditional way as a name followed by the parameters in parentheses. The parameters may be separated by spaces or commas.

object object { object }...
: A procedure or function with one or more arguments may be specified as an UNIX-like command.

Statements have the following syntax:

(* commentary ***)**
    This statement is ignored. It is used to interject commentary in a program.

if { object -> object; }... { object } **fi**
    The list of commands is executed until an "object ->" evaluates to true.

do { object -> object; }... { object } **od**
    The list of commands is repeatedly executed until no "object ->" is true.

for alphanumeric = object **do** object **rof**
    The "for" statement evaluates the left "object" and then replicates the right "object" substituting "alphanumeric" for each component of the resulting object.

goto name
    Control is transferred to the object defined by the local name.

cd name
    Change context to the object defined by name.

rm name { name }...
    This statement is used to remove objects created by the program.

Names consist of sequences of selectors "{/}selector{/selector}...", where selector has the following syntax:

alphanumeric
    An alphanumeric character string is used to specify an object in the local context.

number
    A numeric character string is used to specify an object in the local context.

(object)
    The object is evaluated and its result used to specify an object in the local context.

$
    The parameters of a procedure or function is selected. It may be used to access the standard input "$/I", the standard output "$/O", and the parameters "$/1 $/2 ... ", any of which may be accessed as a an object.

.. (superior)
The calling context is selected; used for moving up through the "dynamic chain" of the information structure.

. (self)
The local context is selected.

/
When the first symbol of a name, the current context is selected. This is analogous to the concept of root in UNIX.

NOTE: "number" and "(object)" may not be used as the initial selector of a name.

Operators consist of the following:

:=
Assignment operator updates a "name : object" pair relative to the local context, if necessary creating the pair. Both the "name" and the "object" are evaluated before the assignment.

+ - *
Arithmetic operators add, subtract and multiply.

and or not
Logical operators.

= <> < <= > >=
The relational operators return <u>true</u> if their arguments are in the specified relation, otherwise they return <u>false</u>.

to
Numeric and alphabetic sequences are generated by the dyadic "to" operator, and returned as an object.

Controls consist of the following:

;
Execution of "command_1 ; command_2" is sequential.

|
Execution of "command_1 | command_2" is pipe-lined, with the output of "command_1" being passed as input to "command_2".

&
Execution of "command_1 & command_2" is in parallel.

APPENDIX A.3 - BASIX_3 Programming Language

NAME
BASIX_3

SYNOPSIS
BASIX_3

DESCRIPTION
BASIX_3 is a decentralised control flow programming language and attempts to combine some of the most important characteristics of BASIC, UNIX Shell, and LISP. In the syntax below "{ }" defines zero or one, and "{ }..." defines zero or more occurrences of the enclosed constructs.

Objects have the following syntax:

expression
An expression consists of one or more simple objects separated by operators.

statement
A statement is a list of objects whose leftmost object is a keyword or the name of a program object.

({object}...)
A list of zero or more objects, data or code.

local_name:object
Declares a name:object pair in the local context. The object is not evaluated.

Commands have the following syntax:

object
The object is evaluated and either returns some value to the user's screen or makes some change to the information structure.

(object {control object}...)
A series of commands separated by controls; each control defines the order of execution of the two adjacent commands.

Expressions have the following syntax:

( )
This is the empty object, and access to it using message passing semantics "name[]" is delayed until its contents are available.

TRUE|FALSE
The logical values true and false.

number
    The object is an integer number.

"string"
    The result is the unevaluated object.

name{[]}
    When used in the form "name" the associated object is treated as shared memory, whereas in the form "name[]" the object has message passing semantics. "name:=..." is a STORE; "...:= name" is a LOAD; "name[]:=..." is a PUT which may only overwrite an emtpy object; and "...:=name[]" is a TAKE which may only access a non-empty object, setting it to empty.

(expression)
    An expression delimited by brackets, which control the order of evaluation in the normal way.

_expression
    The result is the negation of the expression.

expression operator expression
    The expression is an arithmetic, logical, or conditional infix expression.

name({expression {, expression}...})
    The expression is a function or procedure call with zero or more arguments, separated by commas.

Statements have the following syntax:

(* comment *)
    This statement is ignored, being used to interject commentary in a program.

IF {expression -> command}... **FI**
    Each guard expression is evaluated in turn, until an expression is true. If all expressions are false then the statement aborts.

DO {expression -> command}... **OD**
    The guarded commands are repeatedly executed until none of the guards is true.

FOR local_name := expression **TO** expression **DO** command **ROF**
    The FOR-statement iteratively executes command for the series of numeric or alphabetic value.

GOTO local_name
    Control is transferred to the object defined by the local name.

CD name
    Change context to the object defined by name.

name object {object}...
    A procedure with one or more parameters may be specified as a Unix-like command.

Names have the following syntax:

local_name{.selector}...
    A name consists of a sequence of selectors preceded by an alphanumeric local name.

${.selector}...
    The parameters of a procedure or function are accessed as the local name "$", which may be followed by a selector identifying a specific parameter.

Selectors consist of the following:

local_name
    An alphanumeric character string is used to specify an object in the local (or surrounding) context. The object can be anywhere in the context (also see numeric selector).

numeric
    A numeric character string is used to specify an object in the local context. The specific object is found by counting from the left of the context "1:object 2:object 3:....". It may be helpful to view numeric selectors as implicit local names.

(expression)
    The expression is evaluated and its result is used to specify an object in the local context.

Local_name has the following syntax:

alphanumeric
    An alphanumeric character string.

Controls have the following syntax:

;
    Execution of "command_1; command_2" is sequential.

&
    Execution of "command_1 & command_2" is in parallel.

Operators have the following syntax:

:=
    Assignment operator updates a name:object pair relative to the local context. An object (if it does not already exist) is created automatically by the access and set empty. The only way to delete an object is by assignment.

+ - * /
    Arithmetic operators add, subtract, multiply and divide.

AND OR NOT
    Logical operators.

= <> > >= < <=
    The relational operators return TRUE if their arguments are in the specific relation, otherwise they return FALSE.

APPENDIX A.4 - Banking System Application

A.4.1 Banking System Program (in Full BASIX)

```
(*****************************************************************)
(* validate - validates daily transactions input                *)
(*****************************************************************)
validate :  (
              (* procedure okdate verifies if date is valid *)
              okdate: (
                       if
                        transrec/3/2 = 2 ->
                          if (transrec/3/1 < 1) or (transrec/3/1 > 29) ->
                              errorflag:= 'true;
                          fi;
                        (transrec/3/2 = 4) or
                        (transrec/3/2 = 6) or
                        (transrec/3/2 = 9) or
                        (transrec/3/2 = 11) ->
                         if (transrec/3/1 < 1) or (transrec/3/1 > 30) ->
                             errorflag:= 'true;
                         fi;
                        (transrec/3/1 < 1) or (transrec/3/1 > 31) ->
                           errorflag:= 'true;
                       fi
                       if errorflag = 'true -> 'false; 'true; fi;
                      );
              (* procedure nameok verifies if name is alphabetic *)
              nameok: (
                       for i = 1 to 20 do
                           (if not ((transrec/4/(i) >= 'a) and
                                (transrec/4/(i) <= 'z)) or
                                (transrec/4/(i) = " " ) or
                                (transrec/4/(i) = ".")) -> errorflag:= 'true;
                           fi);
                       rof
                       if errorflag = 'true -> 'false; 'true; fi;
                      );
            (* procedure addressok verifies if address is alphanumeric *)
            addressok: (
                        for i = 1 to 20 do
                        (if not (((transrec/5/(i) >= 'a) and
                                  (transrec/5/(i) <= 'z)) or
                                 ((transrec/5/(i) >= '0)  and
                                  (transrec/5/(i) <= '9))  or
                                  (transrec/4/(i)  = " ") or
                                  (transrec/4/(i)  = "."))-> errorflag:='true;
                        fi);
                        rof
                        if errorflag = 'true -> 'false; 'true; fi;
                       );
            (* main body of validate *)
               i:= 1;
               errorflag:= 'false;
               transindex:= 1;
               errorindex:= 1;
               temptrans := ();
```

```
      tempindex := 1;
      do
       (transrec:= transfile/(transindex);
        transindex:= transindex + 1;
        transrec/1 <> 999) ->
        (if
         (transrec/1 >= 1) and (transrec/1 <= 100) and
         (okdate() = 'true) ->
          (if
           transrec/2 = 3 -> (temptrans/(tempindex):= transrec;
                              tempindex:= tempindex + 1);
           transrec/2 = 1 ->
           (if (transrec/4 <> ())    and
               (nameok() = 'true)    and
               (transrec/5 <> ())    and
              (addressok()='true)->(temptrans/(transindex):=transrec;
                                    tempindex;= tempindex + 1);
           fi);
           transrec/2 = 2 ->
           (if ((transrec/4 = ()      or
                 nameok() = 'true)    and
                (transrec/5 = ()      or
                addressok()='true)->(temptrans/(transindex):=transrec;
                                     tempindex:= tempindex + 1);
            fi);
          fi);
          (if erroflag = 'true -> (errofile/(errorindex):= transrec;
                                   errorindex:= errorindex + 1;
          fi);
        fi);
      od
      temptrans/(tempindex):= transrec;  (* terminator 999 *)
      transfile:= temptrans;
   )
```

```
(****************************************************************)
(*  sort - sorts daily transactions input                       *)
(****************************************************************)
sort:  (i := 1;
        do
         transfile/(i)/1 <> 999 ->
        (j := i + 1;
         do
          transfile/(j)/1 <> 999 ->
           (if (transfile/(i)/1 > transfile/(j)/1) or
               ((transfile/(i)/1 = transfile/(j)/1) and
                (transfile/(i)/2 > transfile/(j)/2)) ->
               (
                 temp:= transfile/(i);
                 transfile/(i):= transfile/(j);
                 transfile/(j):= temp
                );
            fi;
          j:= j + 1);
         od;
        i:= i + 1);
       od;
      )
```

```
(***********************************************************************)
(* update - updates Master File with validate, sorted daily transactions *)
(***********************************************************************)
update : (
            procupdate : ( if transrec/2 = 1 -> erroflag := 'true;
                              transrec/2 = 2 ->
                              (newrec/3 := transrec/3
                               if transrec/4 <> () -> newrec/4:= transrec/4 fi;
                               if transrec/5 <> () -> newrec/5:= transrec/5 fi;
                               newrec/6:= newrec/6 + transrec/6;
                              );
                              transrec/2 = 3 ->
                                 (if exclflag = 'false -> exclflag:= 'true;
                                  erroflag:= 'true;
                                  fi);
                           fi
                         );
            transindex  := 1;
            oldindex    := 1;
            newindex    := 1;
            transrec    := transfile/(transindex);
            oldrec      := oldfile/(oldindex);
            newfile     := ();
            do
             (oldrec/1 <> 999) or (transrec/1 <> 999) ->
              if
                 oldrec/1 < transrec/1 ->
                    (newfile/(newindex):= oldrec;
                     newindex:= newindex + 1;
                     oldindex:= oldindex + 1;
                     oldrec:= oldfile/(oldindex));
                 oldrec/1 > transrec/1 ->
                     (if transrec/2 = 1 ->
                         (errorflag:= 'false;
                          exclflag := 'false;
                          newrec   := transrec;
                         do
                          (transindex:= transindex + 1;
                           transrec:= transfile/(transindex);
                           newrec/1 = transrec/1 -> procupdate();
                         od;
                         if (errorflag = 'false) and (exclflag = 'false) ->
                                (newfile/(newindex):= newrec;
                                 newindex:= newindex + 1)
                         fi;
                         errorflag:= 'true;
                      fi
                      if errorflag = 'true ->
                         (errofile/(errorindex):= newrec;
                          errorindex:= errorindex + 1;
                         );
                      fi;
                     )
                 oldrec/1 = transrec/1 ->
                     (  newrec:= oldrec;
                        oldindex:= oldindex + 1;
                        oldrec:= oldfile/(oldindex));
```

```
                errorflag:= 'false;
                exclflag:= 'false;
                do newrec/1 = transrec/1 ->
                   (procupdate( );
                    transindex:= transindex + 1;
                    transrec:= transfile/(transindex);
                   );
                od;
                if (errorflag = 'false) and (exclflag = 'false) ->
                   (newfile/(newindex):= newrec;
                    newindex:= newindex + 1
                   )
                errorflag = 'true ->
                    (errorfile/(errorindex):= newrec;
                    errorindex:= errorindex + 1;
                    );
                fi
              );
   fi;
  od
  newfile/(newindex):= oldrec;   (* terminator 999 *)
)
```

A.4.2 - Sample Run of Banking System

```
$comment - banking system - section 1 - validate the transactions


$comment list the transaction file.

$copy transfile
((005 3 (22 11 83))
 (006 3 (22 11 83))
 (014 1 (22 11 83) (MARYANNIE JOHN SMITH) (1 OLD OAKLAND AVENUE) 666777)
 (010 1 (22 11 83) (KATEANNIE JOSE SMITH) (2 NEW OAKLAND AVENUE) 333444)
 (001 2 (22 11 83) (MURIEL ALBERTA NEWTH))
 (002 2 (22 11 83) (ALBERT JOHN NEWMANNS) (66 OLD BARREL EMBANK))
 (999))

End of file



$comment - run the validate program.

$run *lisp
Execution begin 11:33:56
(RESTORE "BASIX.OBJ")
BASIX                                  11-24-83  RESTORED
(VALIDATE transfile)
(MTS)



$comment - list the validated transaction file

$copy transfile
((005 3 (22 11 83))
 (006 3 (22 11 83))
 (014 1 (22 11 83) (MARYANNIE JOHN SMITH) (1 OLD OAKLAND AVENUE) 666777)
 (010 1 (22 11 83) (KATEANNIE JOSE SMITH) (2 NEW OAKLAND AVENUE) 333444)
 (001 2 (22 11 83) (MURIEL ALBERTA NEWTH))
 (002 2 (22 11 83) (ALBERT JOHN NEWMANNS) (66 OLD BARREL EMBANK))
 (999))

End of file



$comment - banking system - section 2 - sort the transactions.



$comment - run the sort program.
```

```
$restart
(SORT transfile)
(MTS)



$comment - list the sorted transaction file.

$copy transfile
((001 2 (22 11 83) (MURIEL ALBERTA NEWTH))
 (002 2 (22 11 83) (ALBERT JOHN NEWMANNS) (66 OLD BARREL EMBANK))
 (005 3 (22 11 83))
 (006 3 (22 11 83))
 (010 1 (22 11 83) (KATEANNIE JOSE SMITH) (2 NEW OAKLAND AVENUE) 333444)
 (014 1 (22 11 83) (MARYANNIE JOHN SMITH) (1 OLD OAKLAND AVENUE) 666777)
 (999))

End of file



$comment - banking system - section 3 - apply the transactions.



$comment - list the old customer file.

$copy oldfile
((001 1 (22 06 83) (AAAAAAAAAAAAAAAAAAAA) (BBBBBBBBBBBBBBBBBBBB) 111222)
 (002 1 (22 06 83) (AAAAAAAAAAAAAAAAAAAA) (BBBBBBBBBBBBBBBBBBBB) 333444)
 (003 1 (22 06 83) (EEEEEEEEEEEEEEEEEEEE) (FFFFFFFFFFFFFFFFFFFF) 555666)
 (004 1 (22 06 83) (GGGGGGGGGGGGGGGGGGGG) (HHHHHHHHHHHHHHHHHHHH) 777888)
 (005 1 (22 06 83) (IIIIIIIIIIIIIIIIIIII) (JJJJJJJJJJJJJJJJJJJJ) 999111)
 (006 1 (22 06 83) (KKKKKKKKKKKKKKKKKKKK) (LLLLLLLLLLLLLLLLLLLL) 222333)
 (007 1 (22 06 83) (MMMMMMMMMMMMMMMMMMMM) (NNNNNNNNNNNNNNNNNNNN) 444555)
 (008 1 (22 06 83) (OOOOOOOOOOOOOOOOOOOO) (PPPPPPPPPPPPPPPPPPPP) 666777)
 (009 1 (22 06 83) (QQQQQQQQQQQQQQQQQQQQ) (RRRRRRRRRRRRRRRRRRRR) 888999)
 (010 1 (22 06 83) (SSSSSSSSSSSSSSSSSSSS) (TTTTTTTTTTTTTTTTTTTT) 000111)
 (011 1 (22 06 83) (UUUUUUUUUUUUUUUUUUUU) (VVVVVVVVVVVVVVVVVVVV) 222333)
 (012 1 (22 06 83) (WWWWWWWWWWWWWWWWWWWW) (XXXXXXXXXXXXXXXXXXXX) 444555)
 (013 1 (22 06 83) (YYYYYYYYYYYYYYYYYYYY) (ZZZZZZZZZZZZZZZZZZZZ) 666777)
 (999))

End of file



$comment - run the update program.

$restart
(UPDATE oldfile transfile newfile)
(altered
   (001 1 (22 11 83) (MURIEL ALBERTA NEWTH) (BBBBBBBBBBBBBBBBBBBB) 111222))
```

```
(altered
   (001 1 (22 11 83) (ALBERT JOHN NEWMANNS) (66 OLD BARREL EMBANK) 333444))
(deleted
   (005 3 (22 11 83))
(deleted
   (006 3 (22 11 83))
(error - new client
   (010 1 (22 11 83) (KATEANNIE JOSE SMITH) (2 NEW OAKLAND AVENUE) 333444))
(added
   (014 1 (22 11 83) (MARYANNIE JOHN SMITH) (1 OLD OAKLAND AVENUE) 666000))

(STOP)

Execution terminated  12:22:52  T=9.909  RC=0  $1.24



$comment - list the new customer file.

$copy newfile
((001 1 (22 11 83) (MURIEL ALBERTA NEWTH) (BBBBBBBBBBBBBBBBBBBB) 111222)
 (002 1 (22 11 83) (ALBERT JOHN NEWMANNS) (66 OLD BARREL EMBANK) 333444)
 (003 1 (22 06 83) (EEEEEEEEEEEEEEEEEEEE) (FFFFFFFFFFFFFFFFFFFF) 555666)
 (004 1 (22 06 83) (GGGGGGGGGGGGGGGGGGGG) (HHHHHHHHHHHHHHHHHHHH) 777888)
 (007 1 (22 06 83) (MMMMMMMMMMMMMMMMMMMM) (NNNNNNNNNNNNNNNNNNNN) 444555)
 (008 1 (22 06 83) (OOOOOOOOOOOOOOOOOOOO) (PPPPPPPPPPPPPPPPPPPP) 666777)
 (001 1 (22 06 83) (QQQQQQQQQQQQQQQQQQQQ) (RRRRRRRRRRRRRRRRRRRR) 888999)
 (011 1 (22 06 83) (UUUUUUUUUUUUUUUUUUUU) (VVVVVVVVVVVVVVVVVVVV) 222333)
 (012 1 (22 06 83) (WWWWWWWWWWWWWWWWWWWW) (XXXXXXXXXXXXXXXXXXXX) 444555)
 (013 1 (22 06 83) (YYYYYYYYYYYYYYYYYYYY) (ZZZZZZZZZZZZZZZZZZZZ) 666777)
 (014 1 (22 11 83) (MARYANNIE JOHN SMITH) (1 OLD OAKLAND AVENUE) 666000)
 (999))

End of file
```

APPENDIX A.5 - Expert System Application

A.5.1 Expert System Program (in executable subset of BASIX)

```
(rules: QUOTE ((r1  ("has hair")
                    ("is mammal"))
               (r2  ("gives milk")
                    ("is mammal"))
               (r3  ("has feathers")
                    ("is bird"))
               (r4  ("can fly" "lays eggs")
                    ("is bird"))
               (r5  ("eats meat")
                    ("is carnivore"))
               (r6  ("has pointed teeth" "has claws" "has forward eyes")
                    ("is carnivore"))
               (r7  ("is mammal" "has hoofs")
                    ("is ungulate"))
               (r8  ("is mammal" "chews cud")
                    ("is ungulate"))
               (r9  ("is mammal" "is carnivore" "is tawny colour"
                     "has dark spots")
                    ("is cheetah"))
               (r10 ("is mammal" "is carnivore" "is tawny colour"
                     "has black stripes")
                    ("is tiger"))
               (r11 ("is ungulate" "has long neck" "has long legs"
                     "has dark spots")
                    ("is giraffe"))
               (r12 ("is ungulate" "has black stripes")
                    ("is zebra"))
               (r13 ("is bird" "cannot fly" "has long legs"
                     "is black and white")
                    ("is ostrich"))
               (r14 ("is bird" "cannot fly" "can swim"
                     "is black and white")
                    ("is penguin"))
               (r15 ("is bird" "can fly well")
                    ("is albatross")));
```

```
hypo: QUOTE ("is albatross"
             "is penguin"
             "is ostrich"
             "is zebra"
             "is giraffe"
             "is tiger"
             "is cheetah");

name: QUOTE animal;

verify: QUOTE (fact: ./1;
               currule: ./2;
               curante: ./3;
               q: 0;
               r: recall fact;
               IF NOT r -> (inthen fact;
                            IF (LIMIT q) = 0  -> r:= ask fact currule curante;
                               (LIMIT q) <> 0 ->
                                 (i: 1;
                                  DO (done: tryrule q/(i);
                                      IF NOT done -> i:= i + 1; FI;
                                      (NOT done) AND (i <= LIMIT q)) -> TRUE;
                                  OD;
                                  r:= done);
                            FI);
               FI;
               r);

inthen: QUOTE (fact: ./1;
               q:= 0;
               FOR i IN 1 TO LIMIT rules DO
               ( rule: rules/(i);
                 FOR j IN 1 TO LIMIT rule/3 DO
                 ( IF fact = rule/3/(j) -> (k: 1 + LIMIT q;
                                            q/(k):= i);
                   FI);
                 ROF);
               ROF);

recall: QUOTE (fact: ./1;
               r: FALSE;
               FOR i IN 1 TO LIMIT facts DO
               ( IF fact = facts/(i) -> r:= TRUE; FI);
               ROF;
               r);

remember: QUOTE (fact: ./1;
                 r: recall fact;
                 IF NOT r -> (r:= TRUE;
                              k: 1 + LIMIT facts;
                              facts/(k):= fact);
                 FI;
                 r);

tryrule: QUOTE (currule: ./1;
                r: FALSE;
                IF testif currule -> r:= usethen currule; FI;
                r);

testif: QUOTE (currule: ./1;
               r: TRUE;
               done: FALSE;
               rule: rules/(currule);
               j: 1;
               DO (NOT done) AND (j <= LIMIT rule/2) ->
                  (r:= verify rule/2/(j) currule j;
                   IF NOT r -> (r:= FALSE; done:= TRUE);
                      r     -> j:= j + 1;
                   FI);
               OD;
               r);

usethen: QUOTE (currule: ./1;
                rule: rules/(currule);
                r: FALSE;
                FOR j IN 1 TO LIMIT rule/3 DO
                ( IF remember rule/3/(j) ->
                    (r:= TRUE;
                     SYSOUT:= QUOTE "Rule ";
                     SYSOUT:= rule/1;
                     SYSOUT:= QUOTE " deduces ";
                     SYSOUT:= name;
                     SYSOUT:= rule/3/(j));
                  FI);
                ROF;
                r);

ask: QUOTE (fact: ./1;
            currule: ./2;
            curante: ./3;
            done: FALSE;
            r: FALSE;
            FOR i IN 1 TO LIMIT queries DO
            ( IF fact = queries/(i) -> done:= TRUE; FI);
            ROF;
            IF NOT done ->
              (k: 1 + LIMIT queries;
               queries/(k):= fact;
               SYSOUT:= QUOTE "Is this true: ";
               SYSOUT:= name;
               SYSOUT:= fact;
               SYSOUT:= QUOTE " ? ";
               DO (response: SYSIN;
                   IF response = QUOTE "y" -> (done:= remember fact;
                                               r:= TRUE;
                                               done:= TRUE);

                      response = QUOTE "n" -> done:= TRUE;
                      response = QUOTE "w" -> why fact currule curante;
                   FI;
                   NOT done) -> TRUE;
               OD);
            FI;
            r);

why: QUOTE (fact: ./1;
            currule: ./2;
            curante: ./3;
            IF fact = hypo/(curhyp) ->
              (SYSOUT:= QUOTE "One of the possibilities is ";
               SYSOUT:= name;
               SYSOUT:= fact;
               SYSOUT:= QUOTE "I cannot deduce this except by asking you.");
               fact <> hypo/(curhyp) ->
              (rule: rules/(currule);
               SYSOUT:= QUOTE "I am trying to use rule ";
               SYSOUT:= rule/1;
               IF curante > 1 ->
                 (SYSOUT:= QUOTE "I already know that: ";
                  FOR j IN 1 TO curante - 1 DO
                  ( SYSOUT:= name; SYSOUT:= rule/2/(j));
                  ROF);
               FI;
               SYSOUT:= QUOTE "If: ";
               FOR j IN curante TO LIMIT rule/2 DO
               ( SYSOUT:= name; SYSOUT:= rule/2/(j));
               ROF;
               SYSOUT:= QUOTE "Then: ";
               FOR j IN 1 TO LIMIT rule/3 DO
               ( SYSOUT:= name; SYSOUT:= rule/3/(j));
               ROF);
            FI);

SYSOUT:= QUOTE "Hello!";
IF (LIMIT rules) = 0 -> SYSOUT:= QUOTE "No rules.";
   (LIMIT rules) > 0 ->
     (IF (LIMIT hypo) = 0 -> SYSOUT:= QUOTE "No hypotheses.";
         (LIMIT hypo) > 0 ->
           (SYSOUT:= QUOTE "I will use my ";
            SYSOUT:= LIMIT rules;
            SYSOUT:= QUOTE " rules to try to establish one of the following ";
            SYSOUT:= LIMIT hypo;
            SYSOUT:= QUOTE " hypotheses.";
            FOR i IN 1 TO LIMIT hypo DO
            ( SYSOUT:= name; SYSOUT:= hypo/(i));
            ROF;
            DO (facts: 0;
                queries: 0;
                done: FALSE;
                curhyp: 1;
                DO (NOT done) AND (curhyp <= LIMIT hypo) ->
                  (r: verify hypo/(curhyp) 1 1;
                   IF NOT r -> curhyp:= curhyp + 1;
                      r     -> (SYSOUT:= QUOTE "I conclude that ";
                                SYSOUT:= name;
                                SYSOUT:= hypo/(curhyp);
                                done:= TRUE);
                   FI);
                OD;
                IF NOT done ->
```

```
                        SYSOUT:= QUOTE "No hypothesis can be confirmed.";
                    FI;
                    SYSOUT:= QUOTE "r (restart) or q (quit) ?";
                    DO (response: SYSIN;
                        (response <> QUOTE "r") AND (response <> QUOTE "q")) ->
                           TRUE;
                    OD;
                    response = QUOTE "r") -> TRUE;
                OD);
        FI);
FI)
```

A.5.2 - Sample Run of Expert System

```
$comment - run the animals program

$run *lisp

Execution begins   17:21:36

(RESTORE "BASIX.OBJ")

BASIX          12-09-83 RESTORED

(EXEC)

(animals 0)

Hello!

I will use my 15 rules to try to establish one of the following 7 hypotheses.

animal is albatross
animal is penguin
animal is ostrich
animal is zebra
animal is giraffe
animal is tiger
animal is cheetah

Is this true: animal has feathers?
w

I am trying to use rule 3.
If: animal has feathers
Then: animal is bird.
n

Is this true: animal can fly?
w

I am trying to use rule r4.
If: animal can fly
    animal lays eggs
Then: animal is bird.
n

Is this true: animal has hair?
w

I am trying to use rule rl.
If: animal has hair
Then: animal is mammal.
y

Rule rl deduces animal is mammal.

Is this true: animal has hoofs?
w
```

```
I am trying to use rule r7.
I already know that: animal is mammal.
If: animal has hoofs
Then: animal is ungulate.
y

Rule r7 deduces animal is ungulate.

Is this true: animal has black stripes?
w

I am trying to use rule r12.
I already know that: animal is ungulate.
If: animal has black stripes
Then: animal is zebra.
n

Is this true: animal has long neck?
w

I am trying to use rule r11.

I already know that: animal is ungulate.
If: animal has long neck
    animal has long legs
    animal has dark spots
Then: animal is giraffe.
y

Is this true: animal has long legs?
w

I am trying to use rule r11.
I already know that: animal is ungulate
                     animal has long neck.
If: animal has long legs
    animal has dark spots
Then: animal is giraffe.
y

Is this true: animal has dark spots?
w

I am trying to use rule r11.
I already know that: animal is ungulate
                     animal has long neck
                     animal has long legs.
If: animal has dark spots
Then: animal is giraffe.
y

Rule r11 deduces animal is giraffe.

I conclude that animal is giraffe.

r (restar) or q (quit) ?
r
```

```
Is this true: animal has feathers?
w

I am trying to use rule r3.
If: animal has feathers
Then: animal is bird.
y

Rule r3 deduces animal is bird.

Is this true: animal can fly well?
w

I am trying to use rule r15.
I already know that: animal is bird.
If: animal can fly well
Then: animal is albatross
y

Rule r15 deduces animal is albatross.

I conclude that animal is albatross.

r (restart) or q (quit) ?
q

Execution terminated   17:27:427   RC=0   $3.04
```

APPENDIX A.6 - BASAL_1 Programming Language

NAME
    BASAL_1

SYNOPSYS
    BASAL_1

DESCRIPTION
    In **BASAL_1** a program consists of a series of commands separated by controls: ";" and newline define sequential execution (of the two adjacent commands) while "&" defines parallel execution.

Commands have the following syntax:

MICRO micro_name
    All subsequent commands are interpreted in microcomputer "micro_name"

local_label statement
    Stored for later execution

statement
    Executed immediately

Statements have the following syntax:

expression
    Returns result in place

DIM local_name (integer {, integer} ...)
    Declares an array of the specified dimensions

LET name = expression
    Assignment statement

IF expression **THEN** local_label
    Conditional statement

FOR local_name = expression **TO** expression
    Repetitive execution of enclosed statements

NEXT local_name
    End of corresponding loop

GOTO label
    Unconditional control transfer

GOSUB label
    Procedure call

RETURN
    Return from procedure call

STOP
    Stop execution

END
    End of program

Expressions have the following syntax:

number
    Integer number

name
    Identifier (local or non-local) of variable, message, or array element

"character"
    ASCII character

_ expression
    Negate result of expression

(expression)
    Bracketed expression

expression operator expression
    Arithmetical, logical and conditional expression

?
    Empty

Name
    {micro_name.}local_name {(expression {,expression} ...)}{[]}

Label
    {micro_name.}local_label

Operator

+ - * /
    Arithmetic operators

AND OR NOT
    Logical operators

< <= > >= = <>
    Conditional operators

Notes

<micro_name> ::= A..Z

<local_name> ::= alphanumeric

<local_label>::= 01..79

APPENDIX A.7 - BASAL_2 Programming Language

NAME
    BASAL_2

SYNOPSIS
    BASAL_2

DESCRIPTION
    In **BASAL_2** a program consists of a series of commands separated by controls: ";" and newline define sequential execution (of the two adjacent commands) while "&" defines parallel execution.

Commands have the following syntax:

PROC micro_name{(local_name{,local_name}...)}
    All subsequent commands belong to process micro_name, which is allocated to a separate microcomputer.

local_label statement
    Stored for later execution

statement
    Executed immediately

Statements have the following syntax:

expression
    Returns result in place

IMPORT micro_name.local_name
    Defines non-local access

EXPORT local_name
    Allows non-local access

DIM local_name (integer {, integer} ...)
    Declares an array of the specified dimensions

LET name = expression
    Assignment statement

IF expression **THEN** local_label
    Conditional statement

FOR local_name = expression **TO** expression
    Repetitive execution of enclosed statements

NEXT local_name
    End of corresponding loop

GOTO local_label
    Unconditional control transfer

CALL micro_name{(local_name{,local_name}...)}
    Procedure call

RETURN
    Return from procedure call

STOP
    Stop execution

END
    End of program

Expressions have the following syntax:

number
    Integer number

name
    Identifier (local or non-local) of variable, message, or array element

"character"
    ASCII character

_ expression
    Negate result of expression

(expression)
    Bracketed expression

expression operator expression
    Arithmetical, logical and conditional expression

?
    Empty

Name
    {micro_name.}local_name {(expression {,expression} ...)}{[]}

Label
    {micro_name.}local_label

Operator

+ - * /
    Arithmetic operators

AND OR NOT
    Logical operators

```
< <= > >= = <>
     Conditional operators
```

Notes

```
<micro_name> ::= a..z

<local_name> ::= alphanumeric

<local_label>::= 01..79
```

APPENDIX A.8 - Sorting Applications

A.8.1 Quicksort (Shared Memory) in BASAL

```
MICRO A
01 DIM V(16)
02 FOR I = 1 TO 16
03 READ V(I)
04 NEXT I
05 LET LO = 1&
06 LET HI = 16
10 IF LO > HI THEN 39
11 IF LO = HI THEN 39
12 LET I = LO
13 LET J = HI
14 LET PIVOT = V(LO)
15 IF J < I THEN 20
16 IF J = I THEN 20
17 IF V(J) < PIVOT THEN 20
18 LET J = J - 1
19 GOTO 15
20 IF I > J THEN 25
21 IF I = J THEN 25
22 IF V(I) > PIVOT THEN 25
23 LET I = I + 1
24 GOTO 20
25 IF I > J THEN 30
26 IF I = J THEN 30
27 LET TEMP = V(I)
28 LET V(I) = V(J)
29 LET V(J) = TEMP
30 IF I < J THEN 15
31 LET V(LO) = V(I)
32 LET V(I) = PIVOT
33 LET B.LO = LO
34 LET B.HI = I - 1
35 GOSUB B.10&
36 LET C.LO = I + 1&
37 LET C.HI = HI
38 GOSUB C.10
39 STOP
MICRO B
  ...

33 LET D.LO = LO&
34 LET D.HI = I - 1
35 GOSUB D.10&
36 LET E.LO = I + 1&
37 LET E.HI = HI
38 GOSUB E.10
39 RETURN
40 STOP
  ...
```

A.8.2 Quicksort (Shared Memory) translated into BASIC

```
10 DIM PC(26)
20 DIM STACKPTR(26)
30 DIM STACK(26,20)
40 FOR I = 1 TO 26
50 LET PC(I) = (I * 1000) + 800
60 NEXT I
70 FOR I = 1 TO 26
80 LET STACKPTR(I) = 0
90 NEXT I
100 REM
1000 GOTO PC(01)
1010 DIM A.V(16)
1020 FOR A.I = 1 TO 16
1030 READ A.V(A.I)
1040 NEXT A.I
1050 LET A.LO = 1
1060 LET A.HI = 16
1100 IF A.LO > A.HI THEN 1390
1110 IF A.LO = A.HI THEN 1390
1120 IF A.I = A.LO
1130 LET A.J = A.HI
1140 LET A.PIVOT = A.V(A.LO)
1150 IF A.J < A.I THEN 1200
1160 IF A.J = A.I THEN 1200
1170 IF A.V(A.J) < A.PIVOT THEN 1200
1180 LET A.J = A.J - 1
1190 GOTO 1150
1200 IF A.I > A.J THEN 1250
1210 IF A.I = A.J THEN 1250
1220 IF A.V(A.J) > A.PIVOT THEN 1250
1230 LET A.I = A.I + 1
1240 GOTO 1200
1250 IF A.I > A.J THEN 1300
1260 IF A.I = A.J THEN 1300
1270 LET A.TEMP = A.V(A.I)
1280 LET A.V(A.I) = A.V(A.J)
1290 LET A.V(A.J) = A.TEMP
1300 IF A.I < A.J THEN 1150
1310 LET A.V(A.LO) = A.V(A.I)
1320 LET A.V(A.I) = A.PIVOT
1330 LET B.LO = A.LO
1340 LET B.HI = A.I - 1
1350 LET PC(01) = 1350
1351 IF PC(02) <> 02999 THEN 01999
1352 LET PC(01) = 01999
1353 LET STACKPTR(02) = STACKPTR(02) + 1
1354 LET STACK(02,STACKPTR(02)) = 01356
1355 GOSUB 2100
1356 REM
1360 LET C.LO = A.I + 1
1370 LET C.HI = A.HI
1380 LET PC(01) = 1380
1381 IF PC(03) <> 03999 THEN 1999
1382 LET PC(01) = 01999
1383 LET STACKPTR(03) = STACKPTR(03) + 1
```

```
1384 LET STACK(03,STACKPTR,03)) = 01386
1385 GOSUB 3100
     ...
```

A.8.3 Quicksort (Message Passing) in BASAL

```
MICRO A
01 DIM V(16)
02 FOR I = 1 TO 16
03 READ V(I)
04 NEXT I
05 GOTO B.01&
06 GOTO C.01&
10 LET PIVOT = V(1)
11 FOR I = 2 TO 16
12 IF V(I) > PIVOT THEN 15
13 LET B.IN[] = V(I)
14 GOTO 16
15 LET C.IN[] = V(I)
16 NEXT I
20 LET B.IN[] = 30999&
21 LET C.IN[] = 30999
30 LET I = 1
31 LET V(I) = B.OUT[]
32 LET I = I + 1
33 IF V(I - 1) <> 30999 THEN 31
34 LET V(I - 1) = PIVOT
36 LET V(I) = C.OUT[]
37 LET I = I + 1
38 IF V(I - 1) <> 30999 THEN 36
39 STOP

MICRO B
01 LET PIVOT = B.IN[]
02 IF PIVOT = 30999 THEN 39
03 GOTO D.01&
04 GOTO E.01&
10 LET X = B.IN[]
11 IF X = 30999 THEN 20
12 IF X > PIVOT THEN 15
13 LET D.IN[] = X
14 GOTO 10
15 LET E.IN[] = X
16 GOTO 10
20 LET D.IN[] = 30999&
21 LET E.IN[] = 30999
30 LET X = D.OUT[]
31 IF X = 30999 THEN 34
32 LET B.OUT[] = X
33 GOTO 30
34 LET B.OUT[] = PIVOT
35 LET X = E.OUT[]
36 IF X = 30999 THEN 39
37 LET B.OUT[] = X
38 GOTO 35
39 LET B.OUT[] = 30999
40 STOP
   ...
```

A.8.4 Quicksort (Message Passing) translated into BASIC

```
10 DIM PC(26)
20 DIM STACKPTR(26)
30 DIM STACK(26,20)
40 FOR I = 1 TO 26
50 LET PC(I) = (I * 1000) + 800
60 NEXT I
70 FOR I = 1 TO 26
80 LET STACKPTR(I) = 0
90 NEXT I
100 REM
1000 GOTO PC(01)
1010 DIM A.V(16)
1020 FOR I = 1 TO 16
1030 READ A.V(A.I)
1040 NEXT A.I
1050 LET PC(01) = 01050
1051 IF PC(02) <> 02999 THEN 01999
1052 LET PC(01) = 01054
1053 GOTO 02010
1054 REM
1060 LET PC(01) = 01060
1061 IF PC(03) <> 03999 THEN 01999
1062 LET PC(01) = 01064
1063 GOTO 03010
1064 REM
1100 LET A.PIVOT = A.V(1)
1110 FOR A.I = 2 TO 10
1120 IF A.V(A.I) > A.PIVOT THEN 1150
1130 LET PC(01) = 1130
1131 IF B.IN <> -32768 THEN 01999
1136 LET B.IN = A.V(A.I)
1140 GOTO 1160
1150 LET PC(01) = 01150
1151 IF C.IN <> -32768 THEN 01999
1156 LET C.IN = A.V(A.I)
1160 NEXT A.I
1200 LET PC(01) = 1200
1201 IF B.IN <> -32768 THEN 01999
1206 LET B.IN = 30999
1210 LET PC(01) = 01210
1211 IF C.IN <> -32768 THEN 01999
1216 LET C.IN = 30999
1300 LET A.I = 1
1310 LET PC(01) = 01310
1312 IF B.OUT = -32768 THEN 01999
1316 LET A.V(A.I) = B.OUT
1317 LET B.OUT = -32768
1320 LET A.I = A.I + 1
1330 IF A.V(A.I - 1) <> 30999 THEN 1310
1340 LET A.V(A.I - 1) = PIVOT
1360 LET PC(01) = 01360
1362 IF C.OUT = -32768 THEN 01999
1366 LET A.V(A.I) = C.OUT
1367 LET C.OUT = -32768
1370 LET A.I = A.I + 1
```

```
1380 IF A.V(A.I - 1) <> 30999 THEN 1360
      ...

2000 GOTO PC(02)
2010 LET PC(02) = 02010
2012 IF B.IN = -32768 THEN 02999
2016 LET B.PIVOT = B.IN
2017 LET B.IN = -32768
2020 IF B.PIVOT = 30999 THEN 02390
2030 LET PC(02) = 02030
2031 IF PC(04) <> 04999 THEN 02999
2032 LET PC(02) = 02034
2033 GOTO 4010
2034 REM
2040 LET PC(02) = 02040
2041 IF PC(05) <> 05999 THEN 02999
2042 LET PC(02) = 02044
2043 GOTO 05010
2044 REM
2100 LET PC(02) = 02100
2102 IF B.IN = -32768 THE 02999
2106 LET B.X = B.IN
2107 LET B.IN = -32768
2110 IF B.X = 30999 THEN 2200
2120 IF B.X > B.PIVOT THEN 2150
2130 LET PC(02) = 02130
2131 IF D.IN <> -32768 THEN 02999
2136 LET D.IN = B.X
2140 GOTO 02100
2150 LET PC(02) = 02150
2151 IF E.IN <> -32768 THEN 02999
2156 LET E.IN = B.X
2160 GOTO 2100
2200 LET PC(02) = 02200
2201 IF D.IN <> -32768 THEN 02999
2206 LET D.IN = 30999
2210 LET PC(02) = 02210
2211 IF E.IN <> -32768 THEN 02999
2216 LET E.IN = 30999
2300 LET PC(02) = 02300
2302 IF D.OUT = -32768 THEN 02999
2306 LET B.X = D.OUT
2307 LET D.OUT = -32768
2310 IF B.X = 30999 THEN 02340
2320 LET PC(02) = 02320
2321 IF B.OUT <> -32768 THEN 02999
2326 LET B.OUT = B.X
2330 GOTO 02300
2340 LET PC(02) = 02340
2341 IF B.OUT <> -32768 THEN 02999
2346 LET B.OUT = B.PIVOT
2350 LET PC(02) = 02350
2352 IF E.OUT = -32768 THEN 02999
2356 LET B.X = E.OUT
2367 LET E.OUT = -32768
2360 IF B.X = 30999 THEN 02390
2370 LET PC(02) = 02370
```

```
2371 IF B.OUT <> -32768 THEN 02999
2376 LET B.OUT = B.X
2380 GOTO 2350
2390 LET PC(02) = 02390
2391 IF B.OUT <> -32768 THEN 02999
2396 LET B.OUT = 30999
2400 GOTO 02998
2800 REM
     ...
```