ASPECTS OF PROGRAMMING LANGUAGE DESIGN

Peter Grogono

A Thesis in the

Department of Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Québec, Canada

October, 1979

ABSTRACT

ASPECTS OF PROGRAMMING LANGUAGE DESIGN

Peter Grogono

This thesis starts by surveying the relationship between contemporary programming languages and contemporary computer architecture, using as a reference point the von Neumann machine. It demonstrates that the complexity and unreliability of modern software is in part due to the different design goals of architects and programmers. This thesis argues in favour of a more unified approach to computer system design, and presents a simple language that supports some of the claims made.

Preface

The original goal of the research that led to this thesis was the design of a programming language suitable for the interactive manipulation of dynamic data structures. Early investigation revealed that the tradition of designing a procedural language by abstracting from machine language is a major obstacle to the designer who wishes to make a significant extension to the power of a typical contemporary language.

This thesis consists of two parts. The first part, sections 1 through 3, contains an analysis of the factors that have led to complexity and unreliability in system software, especially in compilers, and makes some suggestions as to how these problems might be solved by programming language designers. The second part, section 4, presents a simple language, the design of which employs some of the ideas developed in the first part.

I have received much help and encouragement during the preparation of this thesis. I wish to thank those with whom I have discussed the ideas presented herein. My deepest debt, however, is to V.S. Alagar, who patiently allowed my research to meander in strange and unexpected directions, but was able to guide me at the critical corners. Professor Alagar also read and criticized many versions of this thesis. The final version owes much to him; the mistakes are my own.

Contents

<u>Diagrams</u>

0 Introduction

The object of this thesis is to demonstrate that there is a widening gap between processor design and programming language design, and that this gap has harmful consequences for software development.

The boundary between hardware and software design is the machine language. The problem of the hardware designer is to construct a machine, or perhaps a range of machines, that will interpret the machine language as rapidly as possible, subject to constraints of hardware technology and economics. The problem of the software designer is to construct operating systems, interpreters, and compilers that will enable users of the computer system to work efficiently at a higher, less machine dependend, level; the software designer is subject to the restraints of software technology and economics.

Section 1 is a critical survey of the consequences of this division of design at the arbitrary level of the machine language. It demonstrates that the independent development of hardware and software has lead to unecessarily increased complexity in both, but not to comparable increases in efficiency and performance. The limiting factor is always the machine language interface, which has changed little during the last twenty years.

If substantial progress is to be achieved in computer design, machine languages must be radically changed. Proposals for change at the machine language level should come from software designers, because software designers have (or, at least, ought to have) a deeper understanding of the problems which must be solved by the machine language.

Although a unified top-down approach is ostensibly desirable, it is not without dangers. Section 2 discusses ways in which machine languages can be modified to make high-level languages easier to implement. This approach cannot be expected to lead to radical innovations, because most high-level languages are merely abstractions of contemporary machine languages.

The traditional computer is designed around a single processing unit executing a single stream of sequential instructions. The computing power of a single processor is limited, and it is therefore inevitable that in the future large computers will have many processors. The recently developed techniques of very large scale integration, and in particular the introduction of cheap, mass-produced microprocessors, will hasten this trend. Section 3 discusses some of the issues raised for language designers by these developments.

Sections 1, 2, and 3 take a broad view of the future of computer design. Taken together, they point to the need for a greater unification between hardware and software design. Section 4 shows the form which such a unification might assume. A simple language, based on Hoare's [1978] language CSP, is described, together with an implementation.

1 Survey

This section surveys the current scene in terms of both hardware and software. It is not an unprejudiced survey; it is intended to demonstrate that, despite much activity, there have been few fundamental advances during the last ten years. This stagnation is shown to be due to the paucity of communication between architects of hardware and designers of software.

1.1 Contemporary Machines

The designer of a programming language which is to be widely used can assume only very general properties of the machines on which it is to be implemented. In principle, a language designer will follow Dijkstra's [1976] dictum, that the processor should be regarded as a slave that executes programs written in a programming language according to predefined semantics, rather than regarding the semantics of the language as being defined by an implementation on a particular processor. Despite this, most languages have been designed for contemporary processors, and to this extent processor technology can be perceived as a limiting factor in language design. The general properties of contemporary processors are those of a von Neumann machine. The basic von Neumann machine is described in section 1.1.1 below. The von Neumann machine has been extended in a number of ways that do not affect this argument; these extensions are discussed in section 1.1.2. There are other extensions to the basic von Neumann machine which are more significant but which cannot yet be called 'general properties'; these are discussed in section 2.1.1.

1.1.1 The von Neumann machine

The term von Neumann machine is used to denote a computer with the following general properties [von Neumann, 1946]:

1. The machine consists of a processor P and a memory M.

2. The unit of memory is a string of binary digits called a word. The memory stores the values of a number of words. Each word has the same number of bits.

3. The processor can read from or write to the memory. During a single transfer, the value of one word is moved from M to P (read) or from P to M (write).

4. Having read a word from memory, the processor can interpret it as an instruction or as data. It is not in general possible to determine whether a word is destined to be used as an instruction or as data by examining its value.

The instructions executed by the processor are of two main kinds: one kind alters the flow of control, and the other does not. Instructions of the first kind are typically called <u>conditional jumps</u>: the processor evaluates a simple boolean expression, using data in memory as operands, and executes either the next instruction in sequence or another one depending on the result of the evaluation. The unconditional jump is merely the special case in which the expression evaluated is the constant <u>true</u>. Instructions of the second kind are either <u>computational</u> or <u>motional</u>. A computation instruction applies an operator to one or more operands, and produces one or more results. The operators may be arithmetic or boolean, or special operators such as shift, mask, or extract. A motional instruction moves data from one place to another; in particular, input and output instructions may be regarded as motional.

The implementation of these instructions varies considerably from one actual computer to another. For example, a conditional jump instruction may be implemented by simpler instructions, such as test and jump, or test and skip. Sometimes a single instruction may perform several logical tasks: for example, decrement and jump if zero. The underlying principles, however, are always the same.

The von Neumann machine is important for three reasons. First, almost all of the computers currently in use are, in essential respects, von Neumann machines. Second, many contemporary programming languages were designed to be implemented efficiently on von Neumann machines, and were therefore strongly influenced by von Neumann machine characteristics. Third, even those languages that were not designed to be implemented on von Neumann machines are in practice executed by von Neumann machines, because these are in most instances the only machines available.

1.1.2 Extensions to the von Neumann Machine

More than thirty years have elapsed since the first von Neumann machines were built. The purpose of this section is to show that, although many improvements have been made to the initial design during that period, modern computers still have the basic characteristics of von Neumann machines.

1. The earliest computers had a register, sometimes called the <u>accumulator</u> (AC), which had the same number of bits as a word of memory. Later a second register of the same size, the <u>multiplier/quotient</u> (MQ) was provided to store the other half of the double length quantities that arose during multiplication and division of integers. Later still, <u>index registers</u> were added to remove some of the burdens of address calculations from the accumulator. Eventually, several <u>general purpose registers</u> were provided instead of an accumulator and index registers.

   Registers are considered to be part of the processor.

They provide short-term storage, thereby reducing the number of data transfers between the processor and the memory.

2. The simplest form of memory controller allows only one memory access to be in progress at a particular time. A memory which is divided into banks allows concurrent access to several locations, provided that no two locations are in the same bank. The addresses within a bank are usually not contiguous, and so memories of this type are called <u>interleaved memories</u>.

3. The processor can use an interleaved memory effectively by executing several instructions concurrently, provided that the instructions do not interact with each other. A processor that executes several instructions concurrently is usually designed in such a way that the programmer is not aware of instruction overlap.

4. The processor does not in general access the memory randomly. The distribution of access within a short time will often be confined to a small interval in the address space. This fact can be exploited by constructing the memory in two parts, one of small capacity and high speed, and the other of large capacity and lower speed. The smaller memory is called a <u>cache</u>.

5. The principle of the cache is extended in virtual memory machines which provide a large virtual address space by using disks to back-up a solid-state memory.

6. Input and output operations are usually muoh slower than processor operations. Input and output may proceed in parallel with processing if the processor can respond to an interrupt sent by a peripheral device which is ready for more data. Greater efficiency can be obtained if the peripheral is allowed direct memory access.

7. Input and output functions were originally performed by the processor, but additional processors can be added to perform input and output. The processor is sometimes called the central processor to distinguish it from ancillary processors.

There are three things to note about these extensions to the von Neumann machine:

1. Although some of them have not been widely implemented until recently, they are all old ideas. Index registers were used in the MU1 (1949) and general purpose registers in the Ferranti Pegasus (1956). Interrupts and asynchronous input and output date from the Univac 1103 (1954) and the Remington Rand LARC (1956). The IBM 360/87 (1966) used a cache memory, and

the Ferranti Atlas (1958) had virtual memory.

2. The machine language of a computer possessing these extensions is essentially the same as the machine language of the earliest von Neumann machines. The power of individual instructions has increased only slightly, and the improvements that have occurred help the assembly language programmer more than the compiler writer. The reason for this is often that computer manufacturers want to provide their customers with 'upward compatibility'. They will therefore go to great lengths to enhance the hardware in a manner that is transparent to the programmer. The cache memory of the IBM 360/91 and the pipelined instructions of the CDC 6600 are examples of this philosophy. The result is that programming language development suffers, and attempts to enhance performance are made at the wrong level of abstraction.

3. All of the extensions described are intended to reduce the number of accesses to the memory. This suggests that a fundamental limitation of the von Neumann machine is the single path between the processor and the memory. Backus [1978] calls this the <u>von Neumann bottleneck</u>.

1.2 Contemporary Programming Languages

We can associate with the von Neumann machine a class of programming languages called von Neumann languages. A program in a von Neumann language has the following properties:

1. Two kinds of language construction are used; one describes data and the other describes actions.

2. The actions described by the program are those that can easily be carried out by a von Neumann machine: expressions are evaluated, data is moved about, and the flow of control is influenced by values of the data.

The class of widely used von Neumann languages includes COBOL, FORTRAN, ALGOL-60, PL/I, ALGOL-68, and Pascal. Languages such as LISP, SNOBOL, and APL are not von Neumann languages in the sense defined here, although they may be, and in fact usually are, implemented on a von Neumann machine.

It is an important part of this thesis that, despite the advances in hardware technology described in section 2.1.3 below, modern computers do not adequately support von Neumann languages. In order to establish this point,it is necessary to examine in more detail the sense in which the languages mentioned above are von Neumann languages. We consider Pascal, because it is a reasonably modern, general purpose, and cleanly designed language. The argument developed in the two sections below could be made almost as easily in terms of Algol, Fortran, Basic, Cobol, or any other von Neumann language.

1.2.1 Pascal as a von Neumann Language

The programming language Pascal was defined about 10 years ago, although its first published description dates from 1971 [Wirth, 1971]. It was originally intended to be a language for teaching programming principles, and consequently it is a relatively 'sparse' language, in which elaborate data and control structures are built from simpler units. One specific objective of Pascal was that it should be possible to compile and execute Pascal programs efficiently on typical computers of the late 1960's. As these computers are essentially von Neumann machines with the superficial enhancement described above, Pascal may be called a von Neumann language.

1. The primitive types of Pascal, _integer_, _real_, _boolean_, and _char_ correspond to the types of data that are usually stored as logical units in memory. The methods for constructing arrays and records are abstracted from the indexed and offset addressing techniques of assembly language programming. Sets are simply strings in an abstract form.

2. Pascal expressions consist of operands and operators. The operands are data with the types described above, and the operators correspond to the machine language instructions for performing computations. Pascal assignment statements map to instructions that alter the content of the memory of the von Neumann machine.

3. Pascal has a relatively small number of control structures. The <u>sequence</u> (a series of statements separated by semicolons) corresponds to the convention of the von Neumann machine that instructions are executed sequentially unless they are jumps. The selection statements (<u>if</u>, <u>case</u>) and looping statements (<u>while</u>, <u>repeat</u>, <u>for</u>) are abstractions of machine instructions that control the flow of execution.

1.2.2 Inadequacies of the von Neumann Machine

Although the preceding section demonstrates that Pascal is a von Neumann language in the sense defined at the beginning of section 1.2, there are many features of Pascal that have no counterpart in the von Neumann machine. These inadequacies of the von Neumann machine are reflected in the complexity of the compiler, object code, loader, and run-time system that are needed to support Pascal programs in a typical environment.

1. Code and data in a Pascal program are separated; code cannot be accessed or altered, and data cannot be executed. The von Neumann machine, on the other hand, does not distinguish code and data.

2. The type of a variable in a Pascal program can be determined from its declaration in the program text. The function of an operator is determined by its operands. For example, the operator '*' may denote integer multiplication, real multiplication, or set intersection. In the von Neumann machine, the situation is reversed: the type of a variable cannot be determined from its representation in memory, and different operations are implemented by different instructions. The compiler therefore carries a twofold burden: it must maintain an elaborate symbol table containing the types of variables, and its code generator must emit instructions appropriate to the types of their operands.

3. Arrays and records are natural ways of structuring data in a von Neumann machine, but the machine provides very little assistance in accessing or altering them. The index registers and the instructions that use them help, but it is rare that an array or record component is accessed by a single instruction in the machine code. Furthermore, the machine provides no assistance

at all in checking the legality of access to an array or record component. Bounds checking for arrays and type checking for record components must be performed by additional code emitted by the compiler. This inadequacy of the machine leads to overhead during both compilation and execution.

4. The control structures of Pascal can be implemented using the conditional jump instruction, but the von Neumann machine provides no further assistance. It does not, for example, recognize nested statements or block structure. The raw code emitted by the compiler will contain many jump instructions, and often the targets of these jumps will themselves be jumps. The implementor has the choice of accepting the inefficiency of the object code, or of increasing the complexity of the compiler by adding 'optimizing' features. Moreover, modern operating systems require the compiler to generate code that is at least relocatable, and ideally position independent. Structured source languages tend to generate code with many jumps, thereby placing an extra burden on the relocating loader. Machines which use 'short' and 'long' addresses for jump operands place an additional burden on the compiler, since it must determine when a 'short' branch can safely be used. The high degree of structuring present in the source program does not help

these low-level 'optimizations' algorithms.

5. The principal abstraction mechanism of Pascal is the procedure. Most computers provide no help at all in the implementation of procedure calls and exits. At most, there is a 'subroutine call' instruction, which is often not usable because it leaves the link in the wrong place.

6. In many implementations of Pascal, a stack is used to store local variables, parameters, and links. Although this use of stacks has been known since ALGOL was first implemented in 1958, very few modern computers have instructions suited to the manipulation of stacks. Some computers have 'increment address and store' and 'load and decrement address' instructions, or their equivalent, and these may be helpful in evaluating expressions which the compiler has translated into reverse polish. They do not help the ALGOL or Pascal implementor, however, because he needs an efficient means of creating a stack frame, and of addressing variables with respect to the base of the current stack frame.

1.2.3 Other von Neumann Languages

Although Pascal was chosen to demonstrate that contemporary von Neumann machines do not adequately support contemporary von Neumann languages, similar arguments can be applied to other languages. In fact, to the extent that Pascal was designed for efficient implementation on contemporary processors, it is actually a poor choice for demonstrating the inadequacy of these processors. PL/I and Algol-68 both have many features which cannot be implemented efficiently on a von Neumann machine.

1. PL/I and ALGOL-68 both allow programs to define structures whose size may change during execution of the program. This is a common requirement of string processing algorithms, for example. The task of storing dynamically varying structures, and of checking the legality of access to them, must be performed by code emitted by the compiler and the run-time system, because the machine certainly does not help.

2. PL/I provides many data representations, in particular, binary and decimal fixed point. Operations on fixed point data must be simulated on a host machine that does not possess instructions that operate on data in these formats.

3. PL/I has an <u>on</u> statement that defines a response to an

interrupt caused by a program error, such as an overflow or a range error. Some of these errors may be detected by the machine, but others are not. The PL/I run-time system must detect all of the exception conditions, with or without help from the machine and operating system, and provide the appropriate response.

1.2.4 Other Kinds of Language

There are a number of languages which are in widespread use, but which cannot be called von Neumann languages. These languages are usually implemented on von Neumann machines for the simple reason that other kinds of machine are for the most part unavailable. The implementation usually requires an interpreter, or a compiler supported by an elaborate run-time system.

One of the essential components of the von Neumann machine is the memory. The value of a datum stored in the memory of a von Neumann machine may subsequently be retrieved from the memory. The hardware concept of storage is associated with the linguistic concept of assignment. In _applicative languages_, there is no assignment operation, and in principle memory is not required. A program in an applicative language is an expression composed of operands, operators, and function invocations. Execution of the program consists of applying operators and function definitions until the expression achieves an irreducible form.

If the functions RPLACA and RPLACD, and the functions that depend on them, are removed from LISP, the resulting language is applicative. Operators and functions have no side-effects; the only consequence of their application is a result; and the value of the result depends only on the value of the operands. A <u>LISP machine</u> accepts as input an expression satisfying the rules of LISP syntax (a symbolic expression or <u>S-expr</u>), evaluates it, and displays the result. During the evaluation, structures binding variable names to values, and formal parameters to actual parameters, are created. Some of these are part of the final result, others are not. When the LISP machine is simulated by a von Neumann machine, the structures will occupy space in memory. Since the amount of memory available is finite, the space occupied by structures that are no longer required must be reclaimed. The machine has no way of deciding whether a memory cell is active or inactive, and consequently garbage collection must be performed by the interpreter. The garbage collector is an expensive piece of machinery which is required by the implementation, not the language: this is a strong indication that LISP is not a von Neumann language.

A LISP implementation is, nonetheless, usually supported by a von Neumann machine, and the language is usually extended in ways that reflect the von Neumann machine. Typically, assignment and goto statements are added (as part of the prog feature); more recent LISP systems provide higher level control structures. These extensions are symptoms both of the capabilities of the host processor and the programmers' preference for sequential execution.

The memory of a von Neumann machine consists of a number of addressable cells of uniform size. A data object whose size changes during its lifetime cannot be stored in a simple way. Similarly, the operations provided by the von Neumann machine have fixed length operands. Languages such as SNOBOL, in which operands are strings, or APL, in which operands are vectors, are not von Neumann languages, although, like LISP, they can be simulated by an interpreter running on a von Neumann machine.

The language LUCID [Ashcroft and Wadge, 1975, 1977, 1978] is a language in which statements, as opposed to commands, are written. The designers of LUCID have demonstrated that a language amenable to conventional mathematical techniques of proof can nonetheless incorporate control structures, such as iteration, that are normally associated with command languages.

Backus [1978] proposes a functional style of programming which is a kind of high-level LISP. He perceives the von Neumann bottleneck as a limitation of von Neumann architecture, and suggests that it is the excessive detail required in contemporary programming languages that leads to the inefficiency of their implementation.

The language SASL [Turner, 1975] is also akin to LISP. Turner [1979] has demonstrated a novel implementation technique for applicative languages in which the source program is mapped into a tree structure, and the tree is then transformed into a canonical form which is equivalent to the result that would be obtained by executing the program. An interesting feature of the technique is that local variables disappear during the transformation.

1.3 Recent Developments in Program Language Design

Although many people are interested in programming language design, radical innovations have been few and far between. The field of programming language design is, in fact, stagnant. This stagnation is due to intrinsic incompatibilities between the principal areas of active interest -- program verification, abstract data structures, and concurrent programming -- and von Neumann architecture.

1.3.1 Program Verification

The idea of formally demonstrating that a program is correct, instead of relying on tests, is not new. Although program verification has been the subject of intensive research, there are at present few non-trivial programs that have been convincingly proved correct. There are several reasons for this.

First, the early proponents of formal programming methodology recommended that correctness be achieved by the strict application of a formal program development technique. The proof should be developed in parallel with the program, so that when the program is complete, so is the proof. As Dijkstra [1976, page 216] says: '... therefore, instead of first designing the program and then trying to prove its correctness, we develop correctness proof and program hand in hand'. The alternative approach is to start with the text of a complete program, and prove that it does what it is supposed to do; this is a much more difficult task.

Second, a traditional mathematical proof is a set of assertions that are simultaneously true. A program in a von Neumann language describes a process. The proof of a program must take as its starting point the text of the program, and yet it must recognize the changes of state that occur as the program is executed. Thus a statement as innocuous as

```
x := x + 1
```

with informal semantics 'increase the value of x by 1' leads to problems in formalizing the meaning of the program text using traditional mathematical techniques. A further difficulty in this area is that computer manufacturers claim that their processors can 'add', 'multiply', and so on. In fact, the operations that a typical processor performs bear only a slight resemblance to the operations known to mathematicians as 'addition' and 'multiplication'. Thus the axioms for a programming language describe the implementation on an idealized machine whose performance is not even approached in practice.

Exploring the techniques of program verification more deeply, we find that two problems recur: the difficulty of capturing the essential properties of assignment, and the difficulty of dealing with arrays. Both of these problems can be traced back to the architecture of the von Neumann machine, the memory of which is essentially an array to which assignments are made.

The third reason for the lack of success of verification techniques is the complexity of the proofs. Although a proof is often considered to be a chain of logical deductions whose truth depends only on axioms, premises, and rules of inference, in practice a proof is successful only to the extent that it is convincing. Unfortunately, the proofs of programs are often less convincing than the programs that they claim to prove. There is no more justification for believing a proof to be free of errors than believing a program to be free of errors. In fact, we should probably be more suspicious of the proof, because it is both longer and more difficult to construct [DeMillo et al, 1977].

The fourth difficulty with program verification stems from the dual nature of useful computer programs. The correctness of a program depends on two things: the correct representation of the external world within the program, and the internal logic of the program itself. Floyd [1967b] demonstrated the connection between theorem proving and the correctness of a program: we can specify correctness in terms of theorems about programs. Given a suitable axiomatic system, for example, Hoare's [1969], we can prove these theorems rigorously. Moreover, we can devise programs that construct the proofs, and hence automatic program verifiers.

An automatic program verifier constructs a proof that is formal in the sense of Lakatos [1976, pp124-5]. The validity of the proof does not depend on the meaning of the specific (problem dependent) terms. In other words, the proof can say nothing about the correctness of the representation within the program of facts from the external world. This is what Pratt [1977] means when he says: 'Proofs are just computations for convincing mere machines limited by Church's Thesis of the truth of propositions'.

Proofs of programs will perhaps become manageable when these two aspects of programming (external facts and internal logic) can be separated. Pratt [1977] calls this the 'fact/heuristic dichotomy' (the term is derived from Chomsky's 'competence/performance dichotomy'), and he has demonstrated that in a language in which fact and heuristic are separated, correctness proofs are empty, and the program is automatically correct if the facts on which it is based are true. An application of the fact/heuristic principle to code generation in a compiler is given by Johnson [1978].

1.3.2 Abstract Data Types

The second area of active research concerns abstract data types. The type of a variable determines both the set of values that it may assume, and the set of operators which accept it as an operand.

For simple types, these two properties are usually built into the language. For example, the declaration

```
        VAR flag : boolean
```

determines the values that flag may assume (true and false), and the operators which accept it as an operand (AND, OR, and NOT). These concepts extend naturally to structured types: for example, the properties of an array can be derived from the properties of each of its components. An operation on an entire array is either illegal (~~as in~~ Pascal for all operations other than assignment), or is interpreted as the application of the specified (scalar) operation to each component of the array in turn (as in PL/I).

This concept of type, which already goes far beyond the von Neumann machine (as shown in Section 1.2.2), is inadequate for the higher levels of abstraction found in recent programming languages such as Alphard [Shaw et al, 1977] and CLU [Liskov et al, 1977]. These languages provide mechanisms not only for implementing more elaborate data types, but also for *encapsulating* them. Data abstraction in this sense requires three features not provided by earlier languages:

1. The compiler must detect and reject attempts to employ knowledge of the implementation of an abstract data type. This is what is meant by encapsulation: only those operators that are meaningful for the type may be performed upon it. The operations permitted on a variable of type *stack* might be: procedures *push* and *pop*; and boolean valued functions *empty* and *full*.

2. The instantiation of a variable may involve appropriate initialization. For example, a stack should be initially empty.

3. The data types must be parameterized. For example, once the type <u>stack</u> and the associated operations have been defined, it must be possible to instantiate a 'stack of 10 integers' or a 'stack of 100 reals'.

The first two of these features are more important than the third. We can make statements about variables of a given type which are true when the variable is initialized, and whose truth is preserved by the permitted operations. These statements are then <u>invariant properties</u> of the variables of the type, and this invariance can be deduced from the defining capsule of the type only.

The von Neumann machine does not provide any support for abstract data types. Substantial checking is required during both compilation and execution. The cost of implementing abstract data types is high, because of the complexity of the compiler and the run-time support system.

1.3.3 Concurrent Programming

The third area of active research considered here is concurrent programming. The art of writing code that supports concurrent programs and responds to asynchronous events has been perceived as difficult, and this perception has been substantiated by unreliable software. Most techniques of concurrent programming are based on the assumption that sequential programs are easier to understand than parallel programs, and that parallel programs should therefore consist mostly of sequential code, with the addition of a small number of synchronizing primitives. The synchronizing primitives that have been proposed include semaphores [Dijkstra, 1968], monitors [Hoare, 1974], and mailboxes [Atwood et al, 1972].

A <u>process</u> consists of a sequential program (body of code), which can be shared with other processes, and a <u>state vector</u> which is unique to the process. From time to time a processor is allocated to a process, and steps in the process are executed. Processors are regarded as <u>resources</u>, and, like most resources, are limited in number. A layer of software, consisting of an interrupt handler, synchronizing and communication primitives, and a scheduler, is required to maintain the illusion of concurrent sequential processes. Processes therefore run on virtual machines.

The earliest machines did not have hardware interrupts. Interrupts were introduced to provide a means of overlapping processing and peripheral operations. This solution, conceived and executed at the engineering level, resulted in a machine that had undesirable properties for the programmer, such as non-determinacy. The task of disguising these undesirable properties was left, as usual, to the programmer.

Floyd [1979] has introduced the phrase <u>paradigms of</u>

programming, in which the word 'paradigm' is used in the sense of Kuhn [1970]. The dominant paradigm of programming at the present time is probably 'top-down design with step-wise refinement', or perhaps, more loosely, 'structured programming'. Floyd comments on his own introduction of a programming paradigm: the use of non-deterministic algorithms, originally intended for parsing, and later adopted by the artificial intelligence community [Floyd, 1967a].

The contemporary paradigm for concurrent programming is something like this: 'do as much as you can sequentially, and, when you have no alternative, use one of the currently fashionable devices for communicating or synchronizing'. The purpose of this paradigm is to ensure that the confidence that we have in the correct, deterministic behaviour of sequential programs can be extended to concurrent programs. The methodology arose from two considerations: first, sequential programs were better understood; and, second, most concurrent programs were actually executed by one, or perhaps two, processors. Thus the concept of concurrent processes was introduced to disguise the existence of non-deterministic events, such as interrupts, rather than as an abstraction of a machine with many processors. The need for concurrency was recognized, but the object was to minimize its impact on the practice of programming.

It is a central argument of this thesis that this paradigm of concurrent programming is obsolete. It must be replaced by a paradigm that leads to programs with much more parallelism. Such a paradigm has been proposed by Hewitt and Atkinson [1977]. Section 4 presents a simple programming language for which a paradigm of this kind is appropriate.

2 The Semantic Gap

The 'distance' between the concepts underlying computer architecture and the concepts underlying high level languages has been called the semantic gap [Gagliardi, 1973]. It has been demonstrated in the preceding sections that the semantic gap is wide. Although many programming languages are based on von Neumann principles, and most computers have a von Neumann architecture, in practice the machines only provide rudimentary support for the languages. The width of the semantic gap is responsible for the complexity of modern compilers and operating systems, and this complexity is in turn responsible for the expense and unreliability of software.

Some people perceive the semantic gap as a serious problem. For example, Myers [1978] finds that modern processors are quite inadequate for the support of modern programming languages. He considers the problem of implementing PL/I on the IBM 360, and so his concern is perhaps not surprising. On the other hand, there are those who are not worried by the semantic gap, and are even proposing that it should be allowed to widen. Feldman [1979] works from the premise that compilers as complex as the best contemporary artificial intelligence programs will be constructed in the near future.

The point of view adopted in this thesis is that the semantic gap should be narrowed. As it is unlikely that programmers will accept a drastic reduction in the power of programming languages, this can be achieved only by designing machines that come closer to providing the features required by high level languages.

2.1 Narrowing the Semantic Gap

There have been many reports of computers designed for the efficient execution of high level languages. These projects can be graded according to how close the machine is to a particular language.

At one extreme, there are machines that interpret a program written in a high level language directly from its source text. An APL machine has been described by Thurber and Myna [1970], and an ALGOL-60 machine has been described by Bloom [1975]. Machines of this type, which perform no encoding of the program at all, are unlikely to be efficient. These systems might be well-suited to educational environments, where programs are short and source-level debugging aids are essential.

A less radical solution is to tailor the instruction set of the machine to the high level language that it is intended to implement. Then it is only necessary to provide an 'assembler' to translate the source text into the machine language. The assembler may be implemented in hardware or software. Machines incorporating hardware assemblers also include an APL machine [Robinet, 1975], and an ALGOL machine [Haynes, 1977]. The SYMBOL machine [Chesley and Smith, 1971; Rice and Smith, 1971; Cowart, Rice, and Lundstrom, 1971] is of particular interest.

The SYMBOL computer was built by Fairchild Inc for Iowa State University. The SYMBOL programming language is typeless, has a syntax and block structure similar to that of to PL/I, and provides data structures in the form of lists. There is no software; one part of the processor reads and translates the source program, and another part executes the compiled code. The translator is very fast: the existing version compiles 1250 statements per second, and a proposed (but not implemented) version could compile 5000 statements per second. (Note also that the SYMBOL machine was built in 1970, and these performance figures are based on a 4 microsecond memory cycle time.) The speed of execution of the compiled code is comparable to that of more orthodox processors, although the high level architecture allows certain complex operations to be performed very rapidly.

Machines with software interpreters include an Algol-like machine [de la Guardia and Field, 1976] and an ISPL machine [Balzer, 1973].

Obvious limitations of computers which implement a specific language are that they cannot easily be adapted when the language changes, and that they cannot be used at all for languages other than the one for which they were designed. McKeeman [1967] introduced the term language oriented architecture to denote an architecture that is not designed specifically for a particular high level language but which provides instructions and hardware which facilitate the implementation of a language or a family of languages.

Examples of language directed architectures include the English Electric KDF9, directed towards ALGOL, and various computers marketed by Burroughs Corporation, directed towards ALGOL (B5500, B6500, B6700, B7600) and COBOL (B3500).

Tanenbaum [1978] has described a computer architecture suitable for languages of the ALGOL family, such as ALGOL-60, ALGOL-68, Pascal, XPL, BCPL, and SAL. Tanenbaum's architecture is strongly influenced by the stack-based implementation of these languages. Addressing modes, procedure calls, expression evaluation, and array component address calculation are all designed in terms of a hardware stack.

2.2 Extensions to the von Neumann Architecture

In this section, we consider some extensions to the von Neumann machine. The von Neumann architecture -- a single processor and a randomly addressable memory -- remains unchanged. These extensions allow von Neumann machines to support contemporary programming languages more efficiently, both by reducing the complexity of the compiler and by simplifying the object code.

2.2.1 Procedure Invocation

The following is a fairly general schema for invoking a procedure in a high level sequential programming language:

1. Create an environment in which the procedure will execute;
2. Compute the addresses or values of parameters which are to be passed to the procedure;
3. Save the status of the caller in its own local environment;
4. Transfer control to the procedure;
5. Execute the procedure;
6. Return values of parameters to the caller;
7. Destroy the local environment;

8. Return to the caller.

Some languages do not require all of these steps; for example, FORTRAN environments are static, and so steps 1 and 7 may be omitted.

The only contribution that a typical processor makes to this operation is a save-link-and-jump instruction of some kind, that may suffice (but often does not) for steps 3 and 4. More powerful instructions can be envisioned: ideally, a CALL instruction with an appropriate operand would implement steps 1 through 4, and a RETURN instruction would implement steps 6 through 8.

2.2.2 Addressing Mechanisms

The addressing modes provided by most processors are machine oriented rather than language oriented. Some instructions address part of the instruction itself, some address registers, and some address memory. These are the simple addressing modes: it is usually possible to compute an address which is the sum of the contents of a register and part of the instuction.

These addressing mechanisms are usually adequate in the sense that there is usually an instruction, or a small group of instructions, that will access the required operand. A compiler, however, will usually employ only a few of the available addressing modes, and there may be no efficient ways of accomplishing the most common tasks. Many machines have general purpose registers which may be used to store an operand or an address. The optimal use of these registers is beyond the ability of most compilers. In fact, the intelligent use of registers is one of the factors that distinguishes code produced by a good assembly language programmer from code produced by a compiler.

The following objects must be addressed in a typical high level language:

1. Constants (note that constants are often stored in the instruction space rather than in the data space);

2. Local variables;

3. Non-local variables;

4. Array components (indexed addressing);

5. Record components (offset addressing).

As stated above, it is always possible to realize these addressing modes in terms of an instruction set. Nevertheless, a machine that provided the addressing modes listed would simplify both compiler design and object code.

A large proportion of the assignment statements in high level language programs have very simple right hand sides

[Knuth, 1971; Tanenbaum, 1978]. This suggests that it is more useful to provide instructions that can move data around in memory and perform simple operations on data in memory, than it is to provide only instructions to load and store registers. The provision of instructions of this type probably accounts for some of the success of the DEC PDP11 architecture.

2.2.3 Evaluation Stacks

It is much easier to generate code for a stack machine than for a multi-register machine. This is because expression evaluation is inefficient for a one-register machine, and requires register optimization on a multi-register machine. Stack machines may also be able to evaluate expressions faster than machines with a small number of hardware registers, because fewer memory accesses are required. (This argument is not applicable to machines such as the DEC PDP11 which use memory for the stack; it is, however, applicable to machines such as the English-Electric KDF9 and the Burroughs 6500, in which high-speed registers are used for the stack.)

The advantages of stack machines are most fully realized when the expressions to be evaluated have several operators. The studies of Knuth and Tanenbaum cited above show that such expressions are comparatively rare. The stack code for

```
x := y
```

is

```
PUSH y
POP x
```

and the stack code for

```
n := n + 1
```

is

```
PUSH n
PUSH 1
ADD
POP n
```

Both of these statements can be implemented in one instruction by a machine that has memory to memory move and memory increment instructions. Stack evaluation is desirable, but it must be supported by additional instructions for special cases such as these.

2.2.4 Variable Size Memory Cells

The memory of a von Neumann machine is used to store objects of many different types. Some of the more common types are instructions, integers, characters, booleans, reals, and structured combinations of these. Early computers could only address a word of memory at a time, and all operations involving more than one word, or a part-word, had to be simulated by software. More recent computers can address units of several lengths, such as 8, 16, 32, and 64 bits in the case of the larger IBM 370 computers. The word length of the Burroughs B1700 computer is determined by the microprogram.

2.2.5 Tagged Data

One of the more striking differences between a high level language and the von Neumann machine on which it is implemented is the way in which operators and operands are treated. In the source text, the effect of an operator is determined by its operands. In the machine, the effect of an operator is determined by the instruction chosen.

There is no reason, however, why an item of data in memory should not be tagged with its type. Myers [1978] points out several advantages of this approach:

1. Fewer operation codes are required, and so instructions can be shorter. It might seem that this advantage would be offset by the extra storage required for the data, but this is not necessarily so. Usually, a datum will be stored once only and accessed more than once, and so storing type information in the datum rather than in the instruction may lead to a net saving of space. This is especially true of arrays, because only one type descriptor is needed for an entire array.

2. The processor can perform type compatibility checks

during execution. This is not in fact a particularly significant advantage, since type compatibility errors should probably be detected during compilation anyway.

3. The processor can perform automatic type conversion. For example, an integer may be converted to a real before being added to another real.

The concept of tagged data can be extended to structures: an array tag specifies the number of components and their type, and a record tag specifies the type of each component. The processor can then perform bounds checking for arrays and type checking for record components.

2.2.6 Associative Memory

A von Neumann machine has a memory that is accessed by address. A large part of the complexity of von Neumann systems is due to the address translations performed during compilation, linking, and execution. In certain circumstances, address translation can be eliminated, or at least reduced, by the use of associative memory.

An associative memory is capable of retrieving a datum of which the value (or part of the value) is known. The retrieval of an entry from a symbol table, which requires an algorithm in a von Neumann machine, is a basic operation in an associative processor. Although the idea of associative memory is not new, it is only recently that the technology to produce it economically has become available. A detailed discussion of associative memory is beyond the scope of this thesis. Yau and Fung [1977] have surveyed recent work in this field.

2.3 Microprogramming

Hardware is petrified software. The advantage of software is that it can be easily and cheaply altered; the advantage of hardware is that it can be cheaply mass-produced.

The boundary between hardware and software used to be fixed at the level of the conventional machine language. We can now 'lower' this boundary by writing a program, at a lower level of abstraction, that interprets the instructions of the machine language. Such a program is called a microprogram [Wilkes and Stringer, 1953]. We can also 'raise' the boundary by executing a program that is stored in read-only memory in machine language form;

Microprogramming has already been exploited in a number of different ways:

1. The machine language may be implemented by a microprogram to reduce the cost of the processor. In this case, the microprogram cannot be altered by the user. The ICL 1907 and DEC PDP11/45 computers are microprogrammed in this way.

2. Microprogramming may be used to provide compatibility of machine language instructions over a wide range of processor architectures. This is the approach used by IBM in the 360 and 370 series of computers.

3. The user may be allowed to extend the instruction set of the computer by microprogramming. The Hewlett-Packard 2100 computer is microprogrammed; the manufacturer provides the microprogram for the basic instruction set, and also for certain standard extensions, such as floating-point instructions. The user may add to the microprogram, extend the instruction repertoire by adding to the microprogram, or create his own instruction set.

4. Different microprograms may be used to implement different languages. The Burroughs B1700 computer provides instruction sets tailored for different languages: the switch from one instruction set to another is achieved by a jump in the microprogram.

The application of microprogramming can be extended from programming languages to the programs themselves. The declaration of an abstract data type, for example, could be compiled into a microprogram that provides the instructions necessary for operating on the defined data type.

2.4 Conclusion

The preceding sections have demonstrated stagnation in the development of both hardware and software concepts, and have identified von Neumann architecture as a principal cause of this stagnation. This situation is due not to weaknesses in von Neumann's work; rather, it is due to the phenomenal brilliance of his original design.

The von Neumann machine was ahead of its time; in 1946 the available technology was not adequate for a digital processor of reasonable power. EDVAC had 30,000 tubes, and each time it was switched on, several of them burned out. By the early 1950's, the von Neumann design was well matched to the available technology. The processor was constructed with complex, high-speed switching circuits, and the memory consisted of delay lines, storage tubes, and magnetic cores. The processor contained a few expensive devices, the memory contained many cheap devices, and costs were balanced.

The von Neumann architecture was so successful that it is still the most common architecture in use after 33 years of research and development. During this time the speed and complexity of electronic devices has increased by several orders of magnitude, and the cost and size of these devices has diminished by several orders of magnitude. No other design has survived such a radical change in the underlying technology.

The technological background against which a computer should be designed today is utterly different from what it was in 1946. The most important change is the size of the basic component. For about 20 years the logical component from which a computer was built was the <u>gate</u>, a simple electronic switch. Gates were realized physically by relays, tubes, transistors, and eventually by integrated circuits. The unit of fabrication of computers constructed in the future will be a chip constructed by LSI (large scale integration) techniques. A typical unit might contain a processor, 8K bits of ROM, and 32K bits of RAM. The cost per chip decreases rapidly with the volume manufactured, and so there will be a strong incentive to restrict the variety of LSI chips, and to match them to specific functions by programming them.

Manufacturers are currently producing microprocessors in large quantities. These microprocessors have von Neumann architectures, and most of the undesirable features of mainframe computers designed in the early 1960's. There is a serious danger that the proliferation of microprocessors will be a fatal blow to the ailing science of software theory.

3 Multiprocessor Architecture

There are a number of indications that the use of Multiprocessor systems will increase rapidly during the next few years. This section examines the reasons for these increases; the forms that multiprocessor systems may take; and the implications for programming language design.

3.1 Motivations for Multiprocessor Systems

Several processors are better than one: this is the first and most important explanation for the interest in multiprocessor configurations. A second explanation is provided by the technological developments of the last few years.

3.1.1 Limitations of Uniprocessors

A single processor is limited, as we have seen, by the von Neumann bottleneck: the bandwidth of the channel between the processor and the memory. This bandwidth can be increased by enlarging the word size; by interleaving memory banks; by providing cache memory; or by providing more registers. All of these procedures add to the complexity of the processor, and hence to its cost. Moreover, measures such as providing more registers may increase the complexity, and hence the cost, of software.

3.1.2 Very Large Scale Integration

The logical unit of processor design is the gate; the physical unit is the manufactured component. In the early days of computers, using tubes, and later transistors, the ratio of logical to physical units was of the order of 1:10. The introduction of integrated circuits in the 'third generation' of computers, starting with the IBM 360 series, increased this ratio to about 10:1. This change did not lead to a revolution in processor design because processors were still conceived as a collection of gates, although it did lead to an increase in the use of modular design concepts. More recently, the development of VLSI (very large scale integration) technology has increased the logical/physical ratio to 10^4:1 or even 10^5:1. The design units of the future will therefore be of much greater complexity than single gates: they will be processors, peripheral controllers, memory access controllers, memory blocks, and so on.

The design of a VLSI module is expensive (typically hundreds of thousands of dollars), but the circuits can be manufactured very cheaply (often for less than a dollar per unit), provided that production runs are large. It is therefore economical to use VLSI techniques only if a very large number of units can be sold. This implies that general purpose devices are more amenable to VLSI techniques than special purpose devices.

Once a VLSI circuit has been designed and put into production, it cannot easily be altered. Moreover, the complexity of the circuits is such that the logical design cannot be adequately evaluated until the device has been extensively tested.

These two considerations -- the need for large production runs and the complexity of individual devices -- point to the same conclusion: VLSI circuits should be programmable. In fact, of course, the programmable microprocessor is already the most common example of VLSI technology.

Although microprocessors are not yet as powerful as mainframe processors, their potential is much greater. A CDC 7600 mainframe costs about $5,000,000; for the same cost, using today's prices, it would be possible to construct a machine containing 10^4 microprocessors and 10^9 bytes of memory. Such a project is not feasible at present because the software necessary to make such a machine usable does not exist. The example, however, serves as an indication of the need for such software.

3.2 Aspects of Multiprocessor Design

Most contemporary multiprocessor systems fall into one of two categories: identical processors sharing a common memory; and distributed systems consisting of autonomous computers exchanging messages. In this section we consider an intermediate architecture, in which the coupling between processors is tight enough to allow a simple computational task to be shared between them, but at the same time loose enough to allow extensive concurrency.

It should be noted that the multiprocessor architecture proposed here, and developed in more detail in section 4, is not a multiprocessor in the sense in which the term is often used in the literature. For example, Enslow [1977] states the following criteria for a multiprocessor:

1. A multiprocessor contains two or more processors of approximately comparable capabilities.

2. All processors share access to common memory.

3. All processors share access to input/output channels, control units, and devices.

4. The entire system is controlled by one operating system providing interactions between processors and their programs at the job, task, step, data set, and data element level.

The multiprocessor systems discussed in this thesis do not satisfy any of these requirements. In Enslow's terminology, the systems described here are multicomputer systems.

3.2.1 Common Clock

Most electronic logic systems are controlled by one or more clocks. A clock, in this context, is a logic signal that alternates between false and true; the oscillation is regular and is usually between 10^6 and 10^8 cycles per second. All state transitions in the system take place at the same time as clock transitions; between clock transitions the system is quiescent.

It is a premise of logic design that state transitions are instantaneous, and that propagation and recovery times are infinitesimal. In practice, of course, this is not so, and it is a problem of logic engineering to construct a system that will run fast and reliably. Reliability depends on many factors, but a necessary condition for reliability is that the interval between clock transitions is long compared with the propagation and recovery times within the system. In a large system, the clock rate is limited by the complexity or even the physical size of the system.

This is one reason for abandoning the concept of a common clock in a computer with several ~~processors~~ processor. Another reason is that in a large multiprocessing network, processors will have specialized functions. A clock rate suitable for a peripheral controller might be quite unsuitable for a floating-point processor.

3.2.2 Shared Memory

It is possible to connect several processors to a single memory. The obvious advantage of such a configuration is that very little new software is needed. Application programs can run in one processor of a multiprocessor environment just as they ran in a single processor environment, and the operating system software needs to be changed only slightly so that it can schedule several processors. For this reason, many commercial computer systems are offered with two or more processors sharing memory: computer manufacturers must usually make a commitment to existing software. The disadvantages of this approach are described in section 1.1.2 above.

For the same reason, a configuration of this kind is not of great interest to researchers; it offers little potential for interesting software development. Processes running in different processors communicate by altering shared data structures, and secure ways of accessing shared data structures have been described by Hoare [1974] and Brinch Hansen [1975].

A more serious objection to systems of this type is that the performance of the processors may be degraded because they are competing for memory. Since memory response time is a limiting factor in some single processor systems, this is an important problem.

3.2.3 Connections Between Processors

There are a number of ways in which the processors of a multiprocessor system may be interconnected:

1. Each processor is connected to a common bus.

2. Each processor is connected to every other processor.

3. Processors are at the nodes of a rooted tree whose edges define the connections.

4. Processors are placed at the 2^N vertices of an N-dimensional hypercube whose edges define the connections.

5. Processors are placed at the nodes of an arbitrary graph whose edges define the connections.

Note that configuration 5 includes configurations 2, 3, and 4. Configuration 1 is a special case: any processor can communicate directly with any other processor, but only one communication can take place at a time.

Configuration 2 requires N(N-1) connections between N processors; it is too expensive to implement unless N is very small. Configuration 3 is an elegant implementation of a hierarchical system; the processor at the root of the tree handles the most abstract features of the task, and delegates subtasks to its subtrees. The DDM1 machine [Davis, 1979] has a tree structure in which any processor that is not a leaf of the tree may delegate tasks to a processor immediately below it in the tree. It is unlikely that efficient processor utilization will be achieved with a fixed tree structure.

The hypercube (configuration 3) is a generalized tree in which every processor can be regarded as a root. If there are $N = 2^k$ processors, the longest path between any two processors has $k = \log_2 N$ edges and $N.\log_2 N$ connections are required altogether. IMS Associates manufacture computers with processors connected in this way: configurations with 16 and 256 processors are available.

The configuration may change dynamically. The first configuration is actually a dynamic configuration because at a given time, any two processors may be connected. Tree and hypercube configurations usually have a fixed structure, and this may limit the degree of parallelism that can be obtained with them. The general graph (configuration 5) is not very useful unless either the system is used for solving one problem only or the topology can be altered to suit the problem. In systems of this type some processors may be dedicated to communications; many edges would emanate from them, but only a few edges would emanate from a processing node.

3.2.4 Instructing the Processors

Regardless of the way in which the processors are connected, they must be told what to do. There are two possibilities: either a processor may be provided with code for all of the actions that it will ever be required to perform, or it may be loaded prior to each task. In the first case, the messages passed between processors will consist of pure data, and in the second case they will consist of a mixture of instructions and data. The distinction may be blurred in some cases: if a processor contains a LISP interpreter, and is sent LISP programs, it can be placed in either of the above categories according as to whether LISP programs are considered to be instructions or data. In dataflow systems, a message consists of an operator and operands; a processor may apply some operations and pass others to a lower level processor. At the other extreme, an operating system loads a large program, such as a compiler, into a processor before activating it.

The important point is that loading time should be small compared to processing time; otherwise the processors will be utilized inefficiently.

In a procedural language, code and data are distinct, and code may be re-entrant. Re-entrant code is useful for a single processor supporting several processes which require the same code. (Recursion is a special case of this.) The concept of re-entrant code and multiple ~~activ~~ation records seems to be less useful in an environment in which the number of processors is comparable to the number of processes.

3.3 Languages for Multiprocessors

It is possible to implement a von Neumann language on a multiprocessor system. If the language is not adapted for multiprocessing, the programmer has gained nothing and the semantic gap has been widened. If the language is extended by the addition of a few synchronization and communication facilities, it will still not be closely matched to the system architecture, and the problems of inefficiency, unreliablility, and complex compiler design will not have been solved. It follows that new languages will be required for multiprocessor systems.

3.3.1 Static Program Architecture

The structure of all but the most simple systems is hierarchical. In computer science, the hierarchy is usually represented by a tree: the root of the tree denotes an abstract concept, and the leaves of the tree denote concrete instantiations. The intermediate nodes of the tree are abstractions of the subtrees below them, and concretizations of the nodes above them. (This tree, like most trees in computer science, has its root at the top.)

The hierarchical nature of complex systems has long been recognized by programmers and designers of programming languages. Almost all programming languages have some form of procedure, and in a program, procedures are the nodes of the hierarchical tree. We use the term arborization to denote the extent to which a program has a tree-like structure.

In a fully arborized program, the only channels of communication are the edges of the graph describing the tree. For example, the procedures G and K of the program depicted in the Fig. 3.1 can only communicate along the darkened edges GFEAIJK. In a practical situation, this path is so indirect that G and K are effectively isolated from one another. In many instances, this is a desirable situation: G and K can be coded by different programmers working in different places, and subsequently one can be altered without reference to the other. Furthermore, provided that G and K do what is expected of them by F and J respectively, they need have no knowledge of the rest of the program.

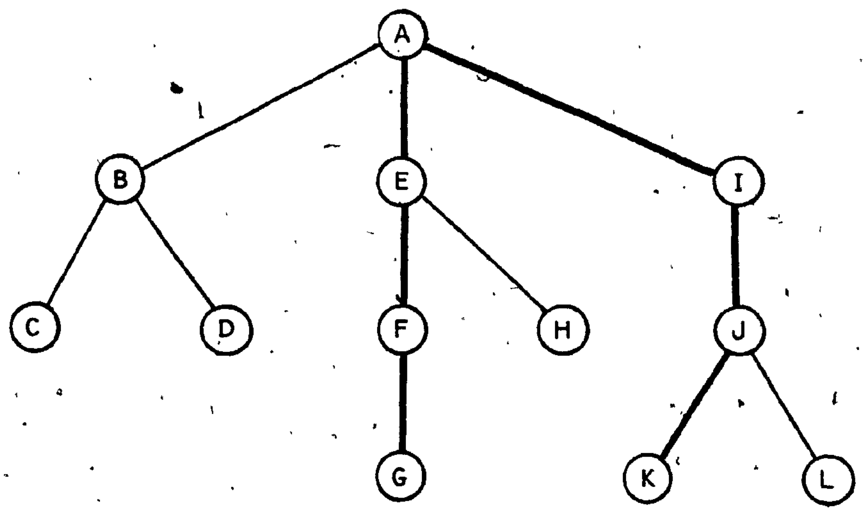

Fig. 3.1: A Program with Tree Structure

Although this property of a fully arborized program is attractive in principle, it is unworkable in practice. This is evident from the fact that there is no widely used programming language in widespread use which lacks a mechanism for subverting the arboreal structure. FORTRAN has COMMON, and in BASIC and COBOL control may move freely at any level of the tree and data is global. Languages of the ALGOL family use nested scopes: in terms of the diagram, procedures G and H can only communicate by means of variables declared in or above procedure F. In practice, most programs written in these languages employ global data, declared in the root, A. This is widely recognized to be too restrictive: there are times when G should be allowed to communicate with H without F knowing about it.

It is therefore necessary to provide a means of converting the tree into a network without destroying its basically tree-like nature. We use the term reticulation to denote the degree to which a program possesses a network structure.

The contribution of structured programming has been to emphasize the arboreal nature of programs. The need for reticulation has been realized more recently, and has lead to increased interest in abstract data types. Languages of the future must enable both arboreal and reticular structures to be implemented.

3.3.2 Dynamic Program Structure

A program written in a language of the ALGOL family consists of a 'main program' and procedures. The main program may be considered as a procedure with a special privilege -- it is called by, and returns to, the operating system. Call the main program P_0 and the procedures $P_1, P_2, \ldots, P_n$, and write down all these names.
If P_i calls P_k, draw the line P_iP_k. If both P_i and P_j call P_k, then create a new node $P_{k'}$ and

join P_iP_k and $P_jP_{k'}$. The diagram is now a tree of which P_0 is the root. Select a node and label it 'A'. Label all the nodes on the path between the selected node and the root, including the root, 'S' (there will not be any if the selected node was P_0). Label all other nodes 'I'. Suppose that each node represents a process and that the only active process is the one labelled 'A' and that the others are either suspended at a procedure call ('S') or are inactive ('I'). This is a model for a possible, though unusual, execution of an ALGOL program: the active node moves about the tree; all processes on the path to the root are suspended; and all processes not on the path to the root are inactive. This suggests a question: why are all processes except one suspended or inactive?

We are accustomed to writing programs that do one thing at a time because we use processors that can only do one thing at a time. Yet this is an unnecessary and arbitrary limitation. There are many programming situations in which a multiprocessing model is more natural than a tree-structured sequential model.

Consider this sequential program:

```
repeat
    read
    do something
    write
until end of file
```

This is a common abstraction for many problems. It is satisfactory if each 'read' produces exactly the right amount of data for each 'do something', but it breaks down if this is not so. We then have to choose one component of the program and use it to 'drive' the other two. It is much more natural to define the abstract solution to a problem of this kind as three communicating processes:

read ---> do something ---> write

In this model, each unit is a process, receiving data from a process or file on its left, and sending data to a process or file on its right. No process dominates, controls, or drives.

In many compilers, the parser is a procedure which, when called, returns part of the parse tree. The parser, in turn, calls the scanner, which returns a token. It is simpler to think of the parser as a process which receives tokens from the scanning process and sends subtrees to the code-generation process. Multi-pass compilers are often written in this way, but the 'passes' are executed sequentially rather than concurrently. The intermediate files used to convey the partially-compiled program from one pass to the next can be eliminated by parallel processing.

In ALGOL-type languages, it is considered undesirable to allow functions to have side-effects. Sometimes side-effects cannot be avoided, however; consider, for example, the case of a function which is required to return a pseudo-random number. It is more consistent to think of a pseudo-random number generator as a process emitting random numbers as they are required, rather than as a function.

These arguments, considered in conjunction with the arguments developed in section 3.1 in favour of a multiprocessor architecture, suggest that a language should be developed that embodies parallel processing in its fundamental design. Some of the obvious advantages of such a language are:

1. Properties of a multiprocessing system are inherent in the language, not grafted onto it.

2. The present situation, in which physical machines and abstract machines are quite different in nature and are only related by elaborate compilers and operating systems, can be avoided.

3. A language which supports parallel processing can be used to write operating system software as well as application software; thus entire systems can be written in one language.

3.3.3 Granularity

The work done by a multiprocessor system can be divided into two parts: first, the part performed by operations within individual processors, and, second, the messages exchanged between processors. A complete definition of one of these parts fully determines the other part; we will compare systems by comparing the messages that move between processors.

Some of the important characteristics of messages within a system are: their mean length (L), their mean information content (I), and their mean frequency (F). We can say that there are constants k_1 and k_2 such that, approximately:

$$L = k_1 I$$

and, even more approximately:

$$F = k_2 I/L$$

assuming that the system has a message carrying ability FL and is operated close to the limit of this capacity. If in fact FL is close to the maximum message carrying capacity, we can call the system <u>message bound</u>. The system may also be limited by the power of its processors, in which case we call it <u>processor bound</u>. If neither of these situations prevails, the system is probably inefficient or under-utilized.

A <u>fine-grained</u> system is characterized by messages that are very short and very frequent. In a <u>coarse-grained</u> system the messages are longer and less frequent. Most contemporary processors employ parallelism at the fine-grain level. For example, the bits of a word are processed simultaneously rather than sequentially. Powerful processors, such as the CDC 6600, IBM 360/91, and Amdahl 470 V/6 can overlap the execution of instructions. This kind of parallelism is not visible to the programmer and is therefore not discussed further in this thesis.

3.3.4 Dataflow Languages

A number of researchers have proposed dataflow languages and machines of various kinds [Ackerman, 1979; Davis, 1979; Gostelow and Thomas, 1979; Ruggiero, 1979; Watson and Gurd, 1979].

Consider the evaluation of the expressions:

$$x' = x.\cos(t) + y.\sin(t)$$
$$y' = y.\cos(t) - x.\sin(t)$$

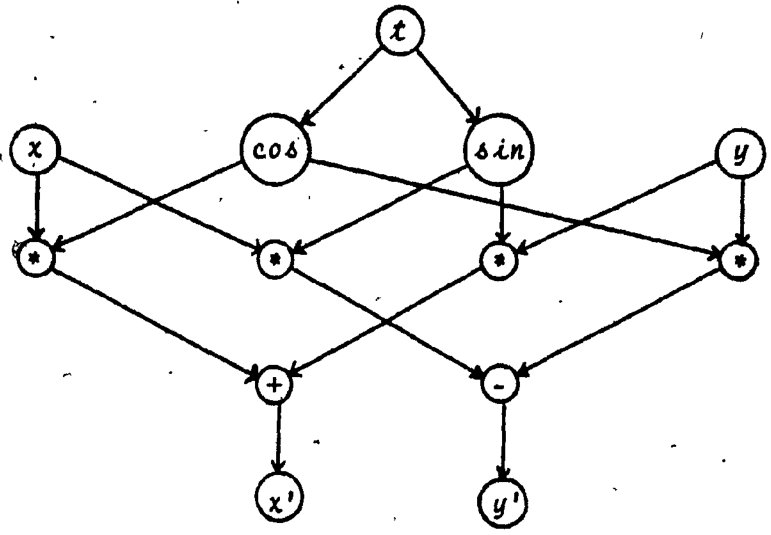

Fig. 3.2: A Dataflow Diagram

Fig. 3.2 is a representation of the evaluation of these expressions in the form of a dataflow diagram. Edges on this diagram denote operands and nodes denote operations. A node is conceived as a cell which 'fires' when it has an operand on each of its inputs. The 'firing' initiates the operation, and when the operation is complete, the resulting operand is placed on the output edge. Operands flow from the top to the bottom of the diagram, and the only synchronization constraint is that a cell cannot fire until all of its operands are available. Assuming that enough processors are available, the computation above could be completed in time

$$\max(T_{sin}, T_{cos}) + T_{mul} + T_{add}$$

in which T_{sin}, T_{cos}, T_{mul}, and T_{add} are the times required to compute a sine, compute a cosine, multiply, and add, respectively. There are some problems with the dataflow model. It is essentially an applicative model, in which there is no history. The symbols in the example above are names of values, not variables, as we can see by attempting to construct a dataflow diagram for

$$x = x + 1$$

These problems can be resolved. An important advantage of dataflow systems is that program verification is more natural, because properties of programs can be derived without a definition of 'process'. Dataflow concepts are very appropriate to the implementation of applicative languages. However, it is well known that applicative languages are more suited to parallel processing than are conventional algorithmic languages, and a number of proposals for concurrent applicative systems have been made [Kahn, 1974; Keller, 1979; Williams, 1978].

The disadvantage with dataflow from the point of view of the present discussion, however, is that it is too fine-grained. A natural implementation of the graph above requires an 'add' processor that waits for two operands, adds them, and passes on the result. Assuming that operands travel in the system as messages, it is likely that the time required to receive two messages and transmit a third would exceed the time required to perform the add operation.

3.4 Conclusion

Most proposed and implemented multi-processor systems have either a loosely-coupled architecture, in which several computers occasionally exchange messages, or a tightly-coupled architecture, in which memory is shared. An intermediate architecture, in which messages are frequent and memory is not shared is described in the next section.

4 A Case Study

This section describes a simple language and an implementation of it. The language is such that it cannot be implemented satisfactorily on a von Neumann machine, and so it is an example of the kind of language discussed in this thesis.

The language is called CSP. It is based on a notation developed by Hoare in his paper Communicating Sequential Processes [Hoare, 1978].

CSP, in the form in which it is described here, is a 'toy' language. It is intended to be a medium for illustrating concepts of programming language design and implementation; CSP is not a language in which large and complex programs should be written.

4.1 Description of CSP

The design of CSP is based on the premise that a program can be constructed from a collection of concurrent processes which can communicate with one another but which do not share data. Thus the primitive concepts of CSP are concurrency, input, and output. CSP is simple because certain rules have been strictly followed. One of these rules is that processes cannot be created dynamically, and so the number of processes can be determined by the compiler. This means that recursive processes can only be implemented in a very restricted way. Another rule is that an output command must specify a target process and an input command must specify a source process.

The control structures of CSP are based on Dijkstra's [1975,1976] guarded commands. If more than one guard in a structured command succeeds, then only one of the corresponding guards is executed. The choice of the command selected for execution is arbitrary, and to this extent the language is non-deterministic.

The syntax of CSP has a strong flavour of Pascal [Wirth, 1971]. CSP syntax, like Pascal syntax, overloads the keyword <u>end</u>. Those who prefer to do so may read <u>trats</u>, <u>nigeb</u>, <u>esoohc</u>, or <u>taeper</u> when they see <u>end</u>.

4.1.1 Syntax and Informal Semantics

CSP programs are written in free format. Blanks may be used freely between terminal symbols, and there must be at least one blank between two keywords or between a keyword and an identifier. A line break is equivalent to a blank. A comment, consisting of '$', a string not containing '$', and a closing '$', is also equivalent to a blank. Keywords are reserved. Upper case and lower case letters are not distinguished in the language, and the use of upper case letters for reserved words herein is merely a convention.

The syntax is defined by productions in Wirth's [1977] notation. Appendix 1 contains a description of this notation. Appendix 2 contains the complete grammar of CSP.

```
   program = global-declaration parallel-command .
```

A program consists of global declarations followed by a command defining one or more processes that will be executed concurrently. Processes may contain parallel commands.

```
   parallel-command = "START" process { "AND" process }
     "END" .
```

The processes are started simultaneously, and continue to execute concurrently.

```
   process = process-identifier range local-declaration
     "BEGIN" command-list "END" .

   process-identifier = identifier .

   range = "[" ( integer-constant | identifier ":" subrange
     ) "]" | empty .

   subrange = integer-constant ".." integer-constant .
```

A <u>range</u> introduces an array of processes or a numbered process. For example:

```
START proc[0]
  ....
AND proc[p : 1..99]
  ....
AND proc[100]
  ....
END
```

There are 101 processes; the processes proc[1], proc[2],...,proc[99] all have the same code, and within that code p is a bound variable. Proc[0] and proc[100] have different code, and might be used for initialization and termination.

```
global-declaration = "STRUCT" { type-clause } | empty .

type-clause = identifier { "," identifier }
  ":" type-descriptor { "," type-descriptor } ";" .

type-descriptor = [ "[" subrange "]" ] simple-type |
  empty .

simple-type = "INTEGER" | subrange | "CHARACTER" .
```

Processes may exchange messages. The content of a message is specified by a structure name. This is a global declaration:

```
STRUCT
  int5 : [1..5] INTEGER;
  card : INTEGER, [1..80] CHARACTER;
  ready : ;
```

Three structures are defined: the structure int5 describes a message containing 5 integers; the structure card describes a message containing an integer followed by 80 characters;

and the structure ready describes an empty message, which is called a signal. A message that contains a single integer or a single character need not be declared globally.

```
   local-declaration = "VAR" { var-clause } | empty .

   var-clause = identifier { "," identifier }
     ":" ( simple-type | array-type ) ";" .

   array-type = "[" subrange "]" simple-type .
```

Local declarations appear at the beginning of a process definition and contain declarations of variables that are local to that process. This is a local declaration:

```
     VAR
       count : INTEGER;
       card  : [1..80] CHARACTER;
       column : 1..80;
```

```
   command-list = command { ";" command } .
```

Commands in a command list are executed sequentially in the order in which they appear in the program text. When a command is executed, it either succeeds or fails.

```
   command = null-command | assignment-command
     | input-command | output-command | choose-command .

   null-command = "SKIP" | "WAIT" .
```

Null commands have no semantic effect and never fail. The WAIT command requires an unspecified amount of time to be executed. It is not used in production programs; in demonstration programs it is used to mean 'some time consuming action which does something irrelevant to this example'.

```
   assignment-command = target ":=" expression .
```

The expression is evaluated, and its value is given to the target. The target and expression must match, in the sense defined below. The assignment command fails if the target and expression do not match.

```
   target = simple-target | structured-target .

   simple-target = variable .

   variable = identifier | component .

   component = identifier "[" simple-expression "]" .

   structured-target = constructor
     "(" variable { "," variable } ")" .

   expression = simple-expression | structured-expression .

   structured-expression = constructor
     "(" simple-expression { "," simple-expression } ")" .

   constructor = identifier .
```

Simple targets and variables have a conventional form, and their types must conform. Array components may appear on either side of an assignment. These are assignment commands:

```
x := 2*n + 1
card[i] := '*'
```

An assignment command with a structured target must have a structured expression with the same constructor, as in this command:

```
buffer(line,length) := buffer(card,80)
```

```
input-command = "RECEIVE" target "FROM"
  process-descriptor .

output-command = "SEND" expression "TO"
  process-descriptor .

process-descriptor = process-identifier
  [ "[" simple-expression "]" ] .
```

Input and output commands enable processes to communicate with one another. Since there are no shared variables, these commands are in fact the only way by which processes can communicate. If process _a_ contains a command of the form

```
SEND e TO b
```

then process _b_ must contain a command of the form

```
RECEIVE v FROM a
```

These two commands are executed simultaneously, and their joint effect is that of the assignment statement

```
v := e
```

Since corresponding input and output commands are executed simultaneously, either may have to wait for the other, and there is no implicit queueing. The command

```
SEND e TO b
```

fails if either (1) process _b_ is no longer active, or (2) process _b_ cannot accept a message of the type _e_. The command

```
RECEIVE v FROM a
```

fails if either (1) process _a_ is no longer active, or (2) process _a_ sends a message of a different type.

When an input or output command has the subscripted form

```
SEND e TO p[i]
```

the subscript must be an expression containing only constants and variables bound by range descriptors. For example, the processes introduced by

```
START sieve [i:1..100]
```

may contain commands of the form

```
RECEIVE p FROM sieve[i-1]
```

for $2 \leq i \leq 101$. This rule may be expressed in terms of an implementation in this way: subscripts in input and output commands are evaluated by the compiler.

```
choose-command = ( "CHOOSE" | "REPEAT" )
   guarded-command { "OR" guarded-command } "END" .

guarded-command = range guard "->" command-list .

guard = boolean-expression | input-command
   | boolean-expression ";" input-command .
```

The general form of the choose command is

```
CHOOSE G1 -> C1
    OR G2 -> C2
    ....
    OR Gn -> Cn
END
```

in which the G_i are guards and the C_i are command lists. When a guard is evaluated, it either succeeds or fails. A boolean expression succeeds if its value is true, otherwise it fails. The effect of the choose command above is as follows:

If no guard succeeds, then the choose command fails;

If exactly one guard succeeds, then the corresponding command list is executed;

If more than one guard succeeds, then exactly one of the corresponding command lists is executed.

The choose command is therefore non-deterministic; this is the only instance of non-determinism in the language.

The repeat form of the choose command is similar, except that if one or more guards are true, the command list is executed, and the whole command is repeated. When all the guards fail, the repeat command terminates without failing. The following commands compute the sum S = 1+2+...+n:

```
s := 0;
m := 0;
REPEAT
  m < n -> m := m + 1; s := s + m
END
```

The command

```
REPEAT
  RECEIVE v FROM a -> SEND v TO c
END
```

terminates when the input command fails; this could be either because process a terminates or because ~~it~~ sends an unacceptable message. The command

```
REPEAT
  RECEIVE cat FROM a -> SEND miaow TO c
OR
  RECEIVE dog FROM a -> SEND bark TO c
END
```

will continue to execute until process a either terminates or sends a message other than cat or dog. The input commands need not specify the same process (although they do in this example).

When a guard contains a boolean expression followed by an input command, the input command is attempted only if the boolean expression evaluates to <u>true</u>. The command

```
CHOOSE
  n > 0; RECEIVE v FROM a -> C
END
```

can only receive <u>v</u> when <u>n</u> is positive.

The range part of a guarded command is a shorthand notation with no other semantic significance. The command

```
CHOOSE
  [i:1..3] RECEIVE v FROM p[i-1] ->
    SEND v TO p[i]
END
```

is an abbreviated form of the equivalent command

```
CHOOSE
  RECEIVE v FROM p[0] ->
    SEND v TO p[1]
OR
  RECEIVE v FROM p[1] ->
    SEND v TO p[2]
OR
  RECEIVE v FROM p[2] ->
    SEND v TO p[3]
END
```

The remaining productions of the grammar describe expressions. They are conventional in nature, and are described briefly here.

```
boolean-expression = comparison { "&" comparison } .
```

A [illegible]lean-expression can only occur as part of a guard. The [illegible]nction operator only is required, because disjunction can be achieved by multiple guards.

comparison = simple-expression comp-op simple-expression .

```
simple-expression = [ add-op ] term { add-op term }
   | boolean-constant | character-constant .

term = factor { mult-op factor } .

factor = variable | integer-constant
   | "(" simple-expression ")"
   | function-identifier "(" simple-expression ")" .
```

The syntax specifies the priority of multiplicative over additive operators, as in Pascal. The language provides some built-in functions, but functions cannot be defined by the user.

```
boolean-constant = "TRUE" | "FALSE" .
```

FALSE and TRUE are keywords; they cannot be redefined by the user.

```
character-constant = "'" character "'" .
```

This version of CSP does not support strings.

```
integer-constant = digit { digit } .

comp-op = "<" | "≤" | "=" | "≠" | "≥" | ">" .

add-op = "+" | "-" .
```

```
mult-op = "*" | "/" | "\" .

function-identifier = identifier .

identifier = letter { ( letter | digit | "_" ) } .
```

Operators and identifiers have conventional syntax. If the character set is limited, "≤", "≠", and "≥" may be represented by "<=", "<>", and ">=" respectively.

4.1.2 Standard Processes

There are two standard processes which do not need to be defined by the user: input and output.

The process input is connected to an input device, such as the keyboard of a terminal, or a disk file, from which it reads a stream of characters. One of the characters that may be read is eol, signifying the end of a line of text. When an end of file condition is signalled by the input device, the process input terminates.

Any user process can receive messages from the process input by using the command

```
RECEIVE v FROM input
```

If the type of v is character then exactly one character is sent (it may be eol). If the type of v is integer then a number is returned. In this case, the process input skips over blanks and line endings until it finds a sign or a digit, and reads the digits following according to this syntax:

```
number = { blank | eol } [ "+" | "-" ] digit { digit } .
```

If input cannot find a number, it fails.

Arrays and constructors may also be read by input; the concepts above are extended in the obvious way. If line is declared by

```
VAR line : [1..100] CHARACTER;
```

then the command

```
RECEIVE line FROM input
```

has the following effect: characters are read from the input device and copied into line until either 100 characters have been copied or eol is read; in the letter case, eol is stored in line and reading ceases.

The process output is connected to an output device such as the screen of a terminal or a printer, to which it sends a stream of characters. The process output terminates only when all user processes have terminated.

Any user process can send messages to output. Output will accept characters, integers, arrays, and structures. If an array of characters containing eol is sent to output, the characters following eol will not be printed.

In this implementation of CSP, input and output are the only processes that can communicate with the external environment.

4.2 Example Programs

The examples that follow are simple CSP programs. They are intended to illustrate the application of CSP to simple problems in concurrent programming, and also to demonstrate that some traditionally sequential algorithms (such as Eratosthenes' sieve) can be expressed in a natural way using concurrency.

Example 1: Direct Copy

This program copies characters from input to output. When the last character has been read from the input file, the procedure input terminates; the receive command then fails, terminating the repeat command, and thereby the process copy.

```
START copy
  VAR
    ch : character;
  BEGIN
    REPEAT
      RECEIVE ch FROM input ->
        SEND ch TO output
    END
  END
END
```

Example 2: Change Line Structure

This program demonstrates process synchronization. It consists of two concurrent processes: *read* and *write*. *Read* removes *eol* characters from the input stream and sends all other characters to *write*. *Write* sends characters to the output stream, inserting an *eol* character after each group of 100 characters.

Read terminates when its *receive* command fails. When *read* has terminated, the *receive* command in *write* fails, terminating the *repeat* command. The *choose* command in *write* appends a final *eol* to the output stream if necessary.

```
START read
  VAR
    ch : CHARACTER;
  BEGIN
    REPEAT
      RECEIVE ch FROM input ->
        CHOOSE ch = eol -> SKIP
        OR ch ≠ eol -> SEND ch TO write
        END
    END
  END

AND write
  VAR
    ch : CHARACTER;
    count : 0..100;
  BEGIN
    count := 0;
    REPEAT
      count < 100; RECEIVE ch FROM read ->
        SEND ch TO output;
        count := count + 1
    OR
      count = 100 ->
        SEND eol TO output;
        count := 0
    END;
    CHOOSE
      count > 0 -> SEND eol TO output
    OR
      count = 0 -> SKIP
    END
  END
END
```

Example 3: Bounded Buffer

The processes read and write in the previous example were closely synchronized. Sometimes the processing rates of two processes fluctuate, and if this is the case, the throughput may be improved by interposing a buffer between them. The process buffer is a simple example of the implementation of an abstract data type in CSP.

The following program is based on Hoare's [1978] example 5.1. It assumes the existence of two processes: producer which sends characters, and consumer which receives the characters sent by producer. Instead of sending characters directly to consumer, producer sends them to the process buffer, and buffer sends them on to consumer. Buffer can store up to 100 characters in the buffer buf.

The number of active characters in the buffer at any one time is N = in - out. The guards ensure that characters can only be put into the buffer if N < 100, and that they can only be removed from the buffer if N > 0. Consumer must issue the command

```
SEND ready TO buffer
```

when it is ready for another character. The program assumes that the characters sent by the producer can be counted without causing integer overflow. The operator "\" is equivalent to mod in Pascal.

```
STRUCT
  ready : ;

START producer ...

AND consumer ...

AND buffer
  VAR
    in, out : INTEGER;
    buf : [0..99] CHARACTER;
  BEGIN
    in := 0; out := 0;
    REPEAT
      in - out < 100;
      RECEIVE buf[in\100] FROM producer ->
        in := in + 1
    OR
      in - out > 0;
      RECEIVE ready FROM consumer ->
        SEND buf[out\100] TO consumer;
        out := out + 1
    END
  END
END
```

Example 4: Text Scan

The next program reads a stream of characters from input and counts the number of occurrences of each of the 26 letters of the alphabet. One process is used for each letter. When the end of the stream is encountered, the signal print is used to synchronize printing of the counters in alphabetical order. This particular program is trivial, but it demonstrates the way in which parallelism could be achieved in more elaborate programs. A cross-reference generator, for example, could keep 26 tables, one for each initial letter, and update them concurrently. The function ord is equivalent to the Pascal function ord.

```
STRUCT
  print : ;

START count[0]
  VAR
    ch : CHARACTER;
    c : INTEGER;
  BEGIN
    REPEAT
      RECEIVE ch FROM input ->
        c := ord(ch) - ord('A') + 1;
        CHOOSE
          [p:1..26] c = p ->
            SEND ch TO count[p]
        END
    END;
    SEND print TO count[1]
  END

AND count[p:1..26]
  VAR
    n : integer;
  BEGIN
    n := 0;
    REPEAT
      RECEIVE ch FROM count[0] ->
        n := n + 1
    END;
    CHOOSE
      RECEIVE print FROM count[p-1] ->
        SEND n TO output;
        SEND print TO count[p+1]
    END
  END

AND count[27]
  BEGIN
    RECEIVE print FROM p[26]
  END
END
```

Example 5: Eratosthenes' Sieve

This elegant program, adapted from Gries's solution in Hoare's notation, uses N+2 concurrent processes to print the prime numbers less then N^2. In the version shown, N = 100. Process sieve[i] receives a string of primes from process sieve[i-1]; it prints the first prime p that it receives, and sends the others on to process sieve[p+1], unless they are multiples of p.

```
START sieve[0]
  VAR
    n : INTEGER;
  BEGIN
    SEND 2 TO output;
    n := 3;
    REPEAT
      n < 10000 ->
        SEND n TO sieve[1];
        n := n + 2
    END

AND sieve[i:1..100]
  VAR
    p, m, mp : INTEGER;
  BEGIN
    RECEIVE p FROM sieve[i-1];
    SEND p TO output;
    mp := p;
    REPEAT
      RECEIVE m FROM sieve[i-1] ->
        REPEAT
          m > mp -> mp := mp + p
        END;
        CHOOSE
          m = mp -> SKIP
        OR
          m < mp -> SEND m TO sieve[i+1]
        END
    END
  END

AND sieve[101]
  VAR
    p : integer;
  BEGIN
    REPEAT
      RECEIVE p FROM sieve[100] ->
        SEND p TO output
    END
  END
END
```

Example 6: Hungry Philosophers

The next program is a solution to the following problem posed by Dijkstra:

> Five philosophers alternately think and eat. In order to eat, they enter a dining-room containing a circular table with 5 chairs around it. In the centre of the table there is a bowl of spaghetti, and around its periphery there are 5 plates, and between each two plates there is a fork. In order to eat, a philosopher sits down at his chair and takes a fork in each hand. The problem is to program the philosophers so that none starves.

If each philosopher goes into the dining-room, sits down, and picks up the fork on his left, and waits for the fork on his right to become available, all will starve. Dijkstra suggests limiting the number of philosophers in the dining-room at any one time to four in order to prevent this unfortunate occurrence.

The program that follows is based on Hoare's [1978] solution; it models the behaviour of the philosophers, forks, and room, by 11 processes. The processes communicate by means of signals. A philosopher is not allowed to enter the dining-room if the other four philosophers are already in it. When a philosopher wants to think, he reads a character from the process input to think about. The command WAIT indicates an arbitrary time delay with no side-effects. Although each fork process sends the signal <u>putdown</u> immediately after receiving the signal <u>pickup</u>, the philosopher does not actually put the fork down until he has finished eating.

```
STRUCT
  enter, exit, pickup, putdown : ;

START philosopher[p:0..4]
  VAR
    ch : CHARACTER;
  BEGIN
    REPEAT
      RECEIVE ch FROM input ->
        WAIT; $ Thinking about ch $
        SEND enter TO room;
        SEND pickup TO fork[p];
        SEND pickup TO fork[(p+1)\5];
        WAIT; $ Eating $
        RECEIVE putdown FROM fork[p];
        RECEIVE putdown FROM fork[(p+1)\5];
        SEND exit TO room
    END
  END

AND fork[f:0..4]
  BEGIN
    REPEAT
      RECEIVE pickup FROM philosopher[f] ->
        SEND putdown TO philosopher[f]
    OR
      RECEIVE pickup FROM philosopher[(f-1)\5] ->
        SEND putdown TO philosopher[(f-1)\5]
    END
  END

AND room
  VAR
    occupants : 0..4;
  BEGIN
    REPEAT
      [p:0..4] occupants < 4;
      RECEIVE enter FROM philosopher[p] ->
        occupants := occupants + 1
    OR
      [p:0..4] RECEIVE exit FROM philospher[p] ->
        occupants := occupants - 1
    END
  END
END
```

Example 7: A Non-Terminating Program

The semantics of CSP do not enforce termination of the following program, because when it is executed, the process slave may never select the second option of the repeat command (Hoare [1978], section 7.6). The program is therefore considered to be incorrect, although any reasonable implementation would eventually choose the second option.

```
STRUCT
  stop : ;

START master
  BEGIN
    SEND stop TO slave
  END

AND slave
  VAR
    run  : boolean;
  BEGIN
    run := true;
    REPEAT
      run -> SKIP
    OR
      run; RECEIVE stop FROM master -> run := false
    END
  END
END
```

Example 8: Deadlock

It is easy to write a program that deadlocks the processors of a CSP system. The following example is based on the program given by Kieburtz and Silberschatz [1979].

```
STRUCT
  mess1, mess2, mess3 : ;

START proca
  BEGIN
    SEND mess1 TO procb;
    SEND mess2 TO procc
  END

AND procb
  BEGIN
    RECEIVE mess3 FROM procc;
    RECEIVE mess1 FROM proa
  END

AND procc
  BEGIN
    RECEIVE mess2 FROM proca;
    SEND mess3 TO procb
  END
END
```

4.3 An Implementation of CSP

It is evident that CSP is not merely another abstraction of a von Neumann machine, although a CSP program could be simulated by a von Neumann machine. This section describes an appropriate architecture for a CSP system, following the principle that the language should determine the nature of the processor (top-down design) rather than the processor determining the nature of the language (bottom-up design).

The distinguishing features of CSP are: concurrent processes, and communication between processes. A single process has many of the characteristics of a von Neumann machine. There is therefore no need for a revolutionary architecture: the nature of CSP suggests that it can be implemented by a system containing several conventional processors.

4.3.1 CSP System Architecture

The processes of a CSP program have local data and exchange messages: there is no global data. This is a sensible policy to adopt at the hardware level also, because it eliminates problems of memory interlocks.

The only assumption made in a CSP program about the timing of events is that corresponding <u>send</u> and <u>receive</u> commands are executed simultaneously. Therefore, there is no need for a common clock in the support system, provided that output from one process can be synchronized to input to another process when necessary. There are a number of points in favour of the 'no common clock' rule (cf. section 3.1.2):

1. Lamport [1978] has described the difficulties of clocking arbitrarily large distributed systems.

2. One reason for having a common clock is shared memory, which CSP does not require.

3. In order to be incorporated into the system, a processor must be able to communicate with the other processors in the system, but need not have any other features in common with them. This means that a CSP

system can incorporate specialized processors with very different performance characteristics; the advantages of this were discussed in section 3.2.

The term 'common clock' is used here to denote a timing signal that synchronizes events throughout the system. The concept of 'system time' which enables events to be time-stamped and therefore ordered (or at least partially ordered) is not precluded, and is in fact an important part of the communication protocol proposed for CSP.

Although it must be possible for any process to communicate with any other process, it is not feasible to provide the N(N-1) connections needed to fully connect N processors. (See section 3.2.3 above for a more detailed discussion of this point.) The alternative chosen for the proposed implementation is to connect every processor to a common bus. The common bus has the following characteristics:

1. A processor is either connected to the bus or it is disconnected.

2. The bus has a clock to control the timing of data transfers. When a processor is disconnected, the bus clock does not influence its behaviour. Thus the bus clock is not a common clock in the sense defined above.

3. A processor may request use of the bus. Such a request will eventually be granted. Once a processor is connected to the bus, it may send a message to another processor.

The single system bus is a potential bottleneck. This problem is discussed in more detail in section 4.4.2 below, in which the proposed implementation is evaluated.

The bus is a shared resource which may be used by only one processor to communicate with another processor at any one time. The bus must therefore be administered by a <u>bus controller</u>. The bus controller notices a request for the bus, waits (if necessary) for the bus to become free, and then grants the request. When it tells the requesting process that it has the bus, the controller also issues the system time.

Since there is no common clock, a single request line joining all processors cannot be used. Each processor is independently connected to the controller, and the controller samples each request line in turn. If two processors request the bus simultaneously, the bus will be given to the processor whose request line is sampled first. If <u>t</u> seconds are required to sample a request line, then a processor in an <u>N</u> processor system waits an average of <u>Nt</u>/2 seconds to obtain the bus, unless another processor is already using it.

There is a choice of methods for providing the system time. A fast clock, perhaps giving the time in microseconds, could be used, but it is not necessary. A slower clock, perhaps counting milliseconds, would need fewer bits to encode the time. The only disadvantage of a slow clock to a CSP program is that a <u>choose</u> command may select the <u>receive</u> command acknowledging the more recent <u>send</u> command, because the <u>send</u> commands apparently occurred at the 'same' time. Another kind of clock could simply count the number of bus requests, and use the value of this counter as the system time. This clock would provide a total ordering of communication events, and would 'run' more slowly than the fast (microsecond) clock mentioned above.

Although the bus controller has a special status in the system, it has a CSP flavour and can in fact be 'programmed' quite easily in CSP:

```
STRUCT
  request, finished : ;

START bus
  VAR
    active, time : INTEGER;
  BEGIN
    active := 0; time := 0;
    REPEAT
      active = 0 ->
        CHOOSE
          [p:1..100]
          RECEIVE request FROM process[p] ->
            SEND time TO process[p];
            active := 1;
            time := time + 1
        END
    OR
      active = 1 ->
        CHOOSE
          [p:1..100]
          RECEIVE finished FROM process[p] ->
            active := 0
        END
    END
END
```

A process that wants the bus executes the command

```
SEND request TO bus
```

It can use the bus as soon as the command

```
RECEIVE time FROM bus
```

succeeds, and when it has finished using the bus it must execute

```
SEND finished TO bus
```

The process <u>bus</u> in this example functions as a monitor [Hoare, 1974].

When the sending process has obtained the control of the bus, it must attract the attention of the receiving processor. It does this by placing the identification of the receiving processor on the data lines of the bus, and then raising the interrupt line. Every processor is interrupted, and looks at the code on the data lines; one processor identifies the code as its own, and raises the acknowledge line on the bus, and waits for the message.

4.3.2 CSP Processor Architecture

Each process in a CSP program is executed by one processor in the CSP machine. In this rather conservative design, each processor is assumed to be a von Neumann machine. It is easy to see how all of the commands of CSP can be executed by a von Neumann machine, except for the commands send and receive. This section therefore describes the implementation of the send and receive commands.

Suppose that the sending process s contains the command

```
(1)   SEND e TO r
```

and the receiving process r contains the command

```
(2)   RECEIVE v FROM s
```

These two commands are equivalent to the assignment statement v := e

and they must be executed simultaneously. In practice, of course, process s will not usually reach command (1) at exactly the same time that process r reaches command (2). Consequently, one of the processes must wait for the other. Suppose that command (1) is reached first; then the processor executing process s (which we may call processor s) will wait. When processor r reaches command (2) it must not wait, because the processors would then be deadlocked, and so it must have some way of discovering that s is waiting.

Suppose that the system has a public bulletin-board. Then processor s could pin a bulletin to the bulletin-board saying 'I have a message for process r'. Later on, process r would reach command (2), and would look on the bulletin-board for a message from s. Once r has seen s's bulletin, the command can be executed. Alternatively, process r might reach command (2) before process s reached command (1). In this case, process r would pin a message to the bulletin-board saying: 'I am waiting for a message from process s'.

Unfortunately, the public bulletin-board is a form of shared data, and so we cannot use it. However, there is no reason why each processor should not have its own private bulletin-board, and this is the basis of the communication protocol described here.

Two forms of communication take place between processors. A <u>bulletin</u> is a short communication which can be intercepted and placed on the <u>bulletin-board</u> of the receiving processor. This is done by interrupting the receiving processor. A <u>message</u> is a longer communication containing data. (In command (1) above, <u>e</u> is data.) A message can only be passed between two processors when both are prepared for it: in other words, messages neither use nor require the interrupt routine.

A bulletin must contain the identification of the sender and a flag specifying the direction of the proposed message. The sending processor must also provide information that enables the receiving processor to decide whether it can accept the message or not. This information consists of the time that the bulletin was issued, and a <u>structure identifier</u> defining the format of the message.

When a process fails or terminates it sends a termination message bulletin to all processes that might be expecting to exchange data with it. A <u>send</u> command issued by a process <u>s</u> to a receiving process that has terminated aborts <u>s</u>. A <u>receive</u> command issued by a process <u>r</u> to a terminated process fails, but does not necessarily abort <u>r</u>. An unacceptable message does abort <u>r</u>.

These considerations suffice for an informal description of the implementation of <u>send</u> and <u>receive</u>; the remaining details will be discussed after these descriptions.

Description of SEND

The sending process s first looks at its bulletin-board; if there is a bulletin saying that the receiving process r has already failed, then s is aborted. If there is no such bulletin, then s constructs a bulletin, sends it to r, and waits for a bulletin from r. S uses the common bus to send this bulletin, but releases it while waiting. The bulletin contains the number (or other identification) of s's processor, the time obtained from the bus administrator, and the structure identifier of the data.

The bulletin that s eventually receives may say that r has terminated, in which case s is aborted. Otherwise, it says that r is ready, and s immediately transmits the data (the bus is open for this transmission because it was obtained by r). The send command is then complete.

SEND Protocol

1. If receiver has terminated, then abort this process.

2. Get common bus.

3. Send a bulletin containing identification of sender, time, and structure identifier, to the receiver.

4. Return common bus.

5. Wait for a reply bulletin from the receiver.

6. If receiver has terminated, then abort this process.

7. If receiver is ready, then send data.

8. Remove bulletin from bulletin-board.

Description of RECEIVE

The receiving process r waits for a bulletin from s to appear on the bulletin-board. (This wait is unnecessary if the bulletin is there already.) If the bulletin says that s has terminated, then the receive command fails. Otherwise, the bulletin describes a message that s has for r; if the message is not acceptable, then r is aborted. (Note that abortion implies termination, and so a bulletin announcing the termination of r will be send to s, and so s will also abort: see step 6 of send protocol.) If the bulletin is acceptable, r obtains the common bus, sends a bulletin to s announcing its readiness, and receives the data.

RECEIVE Protocol

1. Wait for bulletin.

2. If sender has terminated, this receive command fails.

3. If the message described by this bulletin is unacceptable, abort this process.

4. Remove bulletin from bulletin-board.

5. Get common bus.

6. Send 'ready' bulletin to sender.

7. Receive data from sender.

8. Return common bus.

One drawback of this organization is the need to assign memory for a bulletin-board in each processor. The bulletin-boards need not be very large, however. When a process has sent a bulletin, it either waits or terminates. Therefore, no process can send a second bulletin until the first is acknowledged. In the worst case, a bulletin-board in an N-processor system would contain N-1 bulletins. In practice, the compiler could allocate less space to the bulletin-board of a processor that only communicated with a few other processors.

The receive protocol is suitable for a single receive command. A CSP choose command may, however, contain several receive commands, exactly one of which is to be exexuted. The non-determinism of such a command is resolved in this way:

1. If, on entry to the choose command, there are no relevant bulletins on the bulletin-board; then the processor waits for a bulletin specifying an acceptable

message, and selects it.

2. If there is exactly one acceptable bulletin on the bulletin-board, the coresponding message is selected.

3. If there are several acceptable bulletins on the bulletin-board, the oldest bulletin (that with the earliest time stamp) is selected. If two acceptable bulletins were issued at the same time, one is chosen arbitrarily. (This can only happen if the clock is slow running in the sense of section 4.3.1 above.)

When the two processes have agreed to transfer data, the receiving process _r_ requests the bus, and sends a 'ready' bulletin to the sending process _s_. The amount of data to be sent has been agreed upon (it is specified by the structure identifier in the first bulletin sent by _s_) and so no special protocol is required for the message transfer.

The processors of a CSP system could be conventional minicomputers or microcomputers programmed with the necessary CSP primitives. A processor designed for a CSP system could have several special features, either provided by hardware, or microprogrammed. In particular, a special interface to the common bus is desirable, so that a processor can reject interrupts intended for another processor without a break in execution.

4.3.3 Compiling CSP

Many features of a CSP compiler follow conventional practice. In this section we discuss only those aspects of the compiler that are related specifically to CSP.

The compiler must allocate processes to processors. Since we are assuming a one-to-one relationship between processes and processors, this can be done quite easily: process names are mapped onto consecutive integers which are the addresses of processors. The problem is harder if some processors have special characteristics, in this case the programmer might have to specify that a particular processor contained, say, fast floating-point calculation, or access to a special device.

The compiler calculates the length of each message from the global structure declarations. These codes and lengths are used, when the <u>send</u> and <u>receive</u> commands are compiled, to construct the appropriate bulletins.

The non-deterministic nature of the <u>choose</u> command implies that the code for the command cannot be fully compiled, but must be partly interpreted. For a <u>choose</u> command, the compiler must generate code that:

1. evaluates boolean guards;

2. rejects guards containing false boolean expressions;

3. constructs a list of eligible messages (each entry in

the list contains the number of the sending process, the type of message expected, its length, and the address of the first instruction to be executed if the message is received);

4. sends the list to a run-time procedure which takes the appropriate action.

Each process may terminate or abort; when either happens it must send a termination bulletin to all processes that might be expecting to receive a message from it or send a message to it. If it aborts, it must also send a message to the output process explaining what happened.

The compiler can perform type checking, and can emit code for range checking, within a process, using standard techniques. It cannot easily check the compatibility of messages passed between processes, but this is automatically checked by the run-time system anyway.

There is no reason why the compiler should not compile processes independently of one another, provided that it has a table of process names and structure definitions for the whole program.

4.3.4 Loading CSP

The task of loading a CSP program is most easily carried out sequentially, although it is possible to imagine several processes loading concurrently from a random access file. A simple loader would be a sequential process running in a single processor, sending code in the form of structured messages to other processors. Each processor would contain a bootstrap program capable of receiving a message and storing the appropriate code in memory.

4.4 Evaluation of the CSP System

This section evaluates first the language CSP and then the proposed implementation.

4.4.1 Evaluation of the CSP Language

CSP is not intended to be an adequate language for serious programming. This evaluation considers two questions: does CSP provide a foundation on which a useful programming language could be built, and would such a programming language be useful?

CSP has adequate control structures, but lacks facilities for abstraction. One solution to this problem would be to add conventional procedures and functions, defined locally to a process. A better solution would be to incorporate procedures and functions as processes, but allow a shortened form of call. For example, the command sequence

```
SEND parameter TO proc;
RECEIVE results FROM proc
```

could be abbreviated to

```
proc(parameters,results)
```

This format also informs the compiler that the calling process is performing no actions between invoking the procedure and using the results that it returns. The compiler could therefore allocate the same processor to both processes.

This solution does not adequately accomodate 'utilities' -- procedures and functions that are used by many processes. Short utilities, such as the standard mathematical functions, should probably be incorporated in the conventional way in the memory space of the calling process. Longer utilities should be autonomous processes; if they are frequently used, multiple copies could be loaded.

A more serious limitation of CSP is its essentially static structure. The number of processes, their names, and the structure of the messages that they can exchange, are all determined by the program text. These are fundamental properties of the language and attempts to change them lead to considerable increases in the complexity of the design.

The ability to create and destroy processes may not be as serious as it seems. It is possible to construct very elegant algorithms that employ recursive process creation and non-determinism in the sense of Floyd [1967a]. It is also possible to design algorithms that exploit parallelism with a fixed number of processes. Practical experience with CSP is required before the importance of this restriction can be fully assessed.

The requirement that the length of each message be known during compilation is a more serious restriction. A useful implementation of CSP would have to support variable length message without relaxing the security provided by the present implementation.

It is not difficult to modify the syntax in such a way that the <u>send</u> and <u>receive</u> commands are symmetric, thereby allowing a <u>send</u> command to be used as a guard. The communication protocol must also be made symmetric, because a process may have to choose a <u>send</u> command that is acceptable to it. This can be done within the framework of the implementation described, but only at the expense of a more complex exchange of bulletins. The language gains in expressive power; for example, the <u>repeat</u> command of the process <u>buffer</u> of example program 3 can be written in a more symmetric way:

```
REPEAT
  in - out < 100;
  RECEIVE buf[in\100] FROM producer ->
    in := in + 1
OR
  in - out > 0;
  SEND buf[out\100] TO consumer ->
    out := out + 1
END
```

The requirement that corresponding send and receive commands are executed simultaneously is controversial. Hoare [1978] argues that message buffering is not a primitive operation and should not be an implicit language feature. Kieburtz and Silberschatz [1979] find this argument unsatisfactory for the following reasons:

1. Software buffering by the destination process is not easy to accomplish.

2. The assumption underlying the concept that buffered communications are not primitive is that memoryless communication channels are the norm; this assumption may become invalid when VLSI technology permits store-and-forward transmission to be implemented as easily as common bus connections.

3. CSP further assumes that no communication may be lost. This is not always so; for example, in a system refreshing a graphics display, it is better to receive up to date information, even at the expense of missing some information altogether.

These objections are not very convincing. It is true that software buffering is not always easy to accomplish, but that does not necessarily make it easier to accomplish by hardware. Thus, neither of the first two objections carry much weight. The third objection is even weaker. The fact that a special situation can be constructed in which it is preferable to lose messages surely cannot be used as a design criterion for a language in which communication is a primitve concept.

Kieburtz and Silberschatz [1979] also object to CSP on the grounds that processes can deadlock (cf. Example 8 of section 4.2). This is scarcely a valid criticism of a programming language, although it might be a valid objection to an implementation. Indeed, all concurrent programming constructions, such as semaphores, conditional critical regions, and monitors, permit deadlock. The more important question is whether the compiler or run-time system can detect deadlock. It is true that it is not always easy to detect the possibility of deadlock from a CSP source program. Further research is required to determine compile-time or run-time deadlock avoidance strategies.

This thesis does not discuss in depth the verification of CSP programs, for the reasons described in section 1.3.1. An individual process of a CSP program has semantics similar to those of other procedural languages, and can be verified by standard techniques. The proof assumes that receive commands receive valid data in a certain sequence, and it proves, amongst other things, that send commands transmit data in a certain sequence. The remainder of the verification procedure consists of demonstrating that the assumptions made about received data, and the assertions made about transmitted data, are consistent.

4.4.2 Evaluation of the CSP Implementation

The common bus of the proposed system is a potential bottleneck. The extent to which this will lead to a loss of efficiency cannot be determined without a dynamic analysis of non-trivial programs, because bus loading varies inversely with the amount of processing performed between send and receive commands. The example programs of section 4.2 cannot be considered realistic because they perform a negligible amount of computation. A more realistic example might be a compiler written as five concurrent processes:

scanner
symbol-table
parser
code-generator
optimizer

The common bus can be replaced by a digitally controlled switching network. The links required for a particular program can be determined by the compiler, and created by the loader. The number of links required for a typical CSP program is much less than the number of links required to connect every processor to every other processor. Networks which provide arbitrary interconnections between processors, but only a limited number of distinct paths, are called <u>delta networks</u>. Patel [1979] describes the construction and analyses the performance of delta networks for microprocessors.

The one-to-one relationship between processes and processors is probably unduly restrictive. In a more sophisticated system, several small processes could be supported by one processor. Ideally, processes sharing a processor would be message-bound, so they would not compete for processing time. This extension requires a slight modification to the communication protocol and an algorithm for deciding how processes should be allocated to processors. The converse, dividing a process between several processors, is better done at the source language level.

The local memory requirements of processes will vary considerably, and will probably not match the memory available to the processors. There are several ways in which this problem can be mitigated:

1. Allocate processes to a processor until its memory is exhausted.

2. Provide each processor with a different amount of memory, and attempt to match required memory to available memory during loading.

3. Provide memory in the form of blocks which can be switched from one processor to another.

4. Consider memory to be a cheap resource.

4.4.3 Directions for Future Research

There are a number of possible directions for future research into the potential of CSP-like programming languages. Some of these are enumerated below.

1. Elaboration of the language

The version of CSP presented here is inadequate for serious use. Research is required to extend CSP in such a way that it becomes useful without sacrificing its attractive features. Desirable extensions include: variable length messages, dynamic creation of processes, and more general input and output capabilities.

2. Simulation of a CSP System

Potential problems of CSP, such as overloading of the common bus, could be evaluated most easily by a conventional sequential program simulating a CSP system. Implementing a

CSP program has three advantages over constructing a CSP system: first, the cost is less; second, more measurements can be obtained; and, third, it is easier to modify the simulation program to model different implementations than it is to alter hardware.

3. Construction of a CSP System

If the results of the simulation are encouraging, a small CSP system could be constructed using microprocessors.

4. Other Applications of CSP

CSP is a notation for concurrent processes as well as a programming language for concurrent processing. Such a notation may have applications outside the area of multiprocessor architecture: for example, CSP might provide a sound foundation for a simulation language.

4.5 Conclusion

CSP provides a procedural notation for programming communicating sequential processes without shared data. It can be used to construct parallel versions of traditional sequential algorithms. It also enables many of the low-level activities traditionally associated with operating system software, such as peripheral drivers and resource schedulers, to be coded in a high-level language. Moreover, a feasible implementation has been presented, demonstrating that CSP is a practical language.

CSP does not, of course, provide an immediate panacea for all of the problems of programming language design discussed in the earlier sections of this thesis. It does, however, provide partial solutions to some of these problems.

1. The von Neumann bottleneck -- the single path between a fast processor and a random access memory -- has been replaced by many paths between processors and their memories. The performance of a CSP system implemented in the way described here might be limited by the common bus, however. A more general implementation of CSP could exploit the fact that there are now many data paths, and with appropriate programming techniques it should be possible to adjust the flow of data so that all paths are efficiently used. A CSP system is a network in which the nodes are processing units and storage units, and in which edges carry data; the topology of the network and hence the loading of the nodes and edges can be adjusted by the compilers and loaders of a CSP system. In this way, CSP overcomes some of the limitations of the von Neumann architecture discussed in section 1.1.2.

2. Section 2 of this thesis introduced the term 'semantic gap' to describe the divergence between hardware and

software design. The description in section 4.3 of the implementation of CSP shows that the semantic gap can be narrowed in some respects by designing hardware with an architecture that corresponds to the architecture of the language. This thesis provides only an introduction to this design methodology; CSP requires more elaborate constructs, such as control and data abstraction facilities, and the implementation of these could also be continued down to the hardware level. The objective of this design methodology is to ensure that optimization strategies adopted at a low level actually improve the efficiency of operations performed at higher levels, instead of merely increasing their complexity.

3. CSP contains keys to both the solution of certain problems of contemporary programming and their unification. The problem of accessing shared data is eliminated. Access to shared resources can be controlled by a single CSP process functioning as a Hoare monitor. An abstract data type can be implemented as a CSP process, although the communication protocols of CSP would have to be elaborated to make this feasible.

4. Traditional multiprocessor systems require an elaborate operating system to schedule resources dynamically [Enslow, 1977]. In CSP, much of the resource

allocation is performed by the program and the compiler. This follows the trend established by Pascal [Wirth, 1971], which allows the compiler to check the legality of many operations and assignments, and Concurrent Pascal [Brinch-Hansen, 1975], which allows the compiler to check the legality of operations on shared data structures. CSP continues this trend towards more homogeneous systems in which there are fewer distinctions between the operating system and the users' programs.

5. CSP exploits the potential of modern technology. CSP processors could consist of a microprocessor, communication software in ROM, and a comparatively small amount of RAM. These processors would be identical, and therefore cheap to produce. A CSP system would be simple to maintain, because a defective processor can simply be switched off until it is replaced.

CSP is not a revolutionary architecture, because the building block, the processor, can be a small von Neumann machine. Further study is needed to determine whether the CSP processor ought to be a von Neumann machine, or whether there is another kind of architecture that is more appropriate.

Appendix 1: A Notation for the Productions of a Grammar

The notation used in this thesis for the productions of CSP grammar is similar to the notation described by Wirth [1977]. It is an extension of Backus-Naur Form (BNF), in which the symbol '::=' is written '=', the meta-brackets '<' and '>' are not required for non-terminal symbols, and terminal symbols are enclosed in quotes ("). Additional structuring facilities are provided by brackets ([...]) and braces ({...}).

```
   grammar = { productions } .
```

Braces '{' and '}' mean 'zero or more repetitions'; this production therefore allows a grammar to be empty.

```
   production = identifier "=" expression "." .
```

The '=' corresponds to the '::=' of BNF, and the '.' terminates the production. In contrast to BNF, non-terminals are not delimited, and terminals are enclosed in quotes (").

```
   expression = term { "|" term } .
```

The bar "|" denotes alternation, as in BNF.

```
   term = "[" factor "]" | "{" factor "}" .
```

Brackets '[' and ']' denote an option - 'zero or one occurrence'.

```
   factor = terminal | "(" expression ")" .
```

Parentheses '(' And ')' are used for grouping.

```
   terminal = """" string """" .
```

Quote (") is repeated when it is employed as a terminal symbol. The string can contain any characters.

```
   identifier = letter { letter | "-" } .

   letter = "a" | "b" | ... | "z" .
```

Appendix 2: The Grammar of CSP

```
program = global-declaration parallel-command .

parallel-command = "START" process { "AND" process } "END" .

process = process-identifier range local-declaration
     "BEGIN" command-list "END" .

process-identifier = identifier .

range = "[" ( integer-constant | identifier ":" subrange )
     "]" | empty .

subrange = integer-constant ".." integer-constant .

global-declaration = "STRUCT" { type-clause } | empty .

type-clause = identifier { "," identifier }
     ":" type-descriptor { "," type-descriptor } ";" .

type-descriptor = [ "[" subrange "]" ] simple-type | empty .

simple-type = "INTEGER" | subrange | "CHARACTER" .

local-declaration = "VAR" { var-clause } | empty .

var-clause = identifier { "," identifier }
     ":" ( simple-type | array-type ) ";" .

array-type = "[" subrange "]" simple-type .

command-list = command { ";" command } .
```

```
command = null-command | assignment-command
     | input-command | output-command | choose-command .

null-command = "SKIP" | "WAIT" .

assignment-command = target ":=" expression .

target = simple-target | structured-target .

simple-target = variable .

variable = identifier | component .

component = identifier "[" simple-expression "]" .

structured-target = constructor
     "(" variable { "," variable } ")" .

expression = simple-expression | structured-expression .

structured-expression = constructor
     "(" simple-expression { "," simple-expression } ")" .

constructor = identifier .

input-command = "RECEIVE" target "FROM" process-desriptor .

output-command = "SEND" expression "TO" process-descriptor .

process-descriptor = process-identifier
     [ "[" simple-expression "]" ] .

choose-command = ( "CHOOSE" | "REPEAT" )
```

```
    guarded-command { "OR" guarded-command } "END" .

guarded-command = range guard "->" command-list .

guard = boolean-expression | input-command
     | boolean-expression ";" input-command .

boolean-expression = comparison { "&" comparison } .

comparison = simple-expression comp-op simple-expression .

simple-expression = [ add-op ] term { add-op term }
     | boolean-constant | character-constant .

term = factor { mult-op factor } .

factor = variable | integer-constant | "(" simple-expression
     ")" | function-identifier "(" simple-expression ")" .

boolean-constant = "FALSE" | "TRUE" .

character-constant = "'" character "'" .

integer-constant = digit { digit } .

comp-op = "<" | "≤" | "=" | "≠" | "≥" | ">" .

add-op = "+" | "-" .

mult-op = "*" | "/" | "\" .

function-identifier = identifier .

identifier = letter { ( letter | digit | "_" ) } .
```

References

The following is a list of the references consulted while this thesis was being prepared. Only those references marked with an asterisk (*) are actually cited in the text.

Ackerman, W.B. [1979*]
Dataflow Languages
Proceedings National Computer Conference, 1979, p1087-1095

Ashcroft, E.A., Wadge W.W. [1975*]
Clauses, Scope Structure and Defined Functions in LUCID
Fifth Principles of Programming Languages Conference, 1977, p17-22

Ashcroft, E.A., Wadge, W.W. [1977*]
LUCID, A Non-procedural Language with Iteration
Communications of the ACM, 20, #7, July 1977, p519-526

Ashcroft, E.A., Wadge, W.W. [1978*]
Lucid: Scope Structure and Defined Functions
Research Report CS-78-01, University of Waterloo

Atwood, J.W., Clark, B.L., Grushcow, M.S., Holt, R.C., Horning, J.J., Sevcik, K.C., Tsichritzis, D. [1972*]
Project SUE Status Report
Technical Report CSRG-11, University of Toronto, 1972

Backus, J. [1978*]
Can programming be liberated from the von Neumann style? A functional style and its algebra of programs
Communications of the ACM, 21, #8, August 1978, p613-641

Baker, H.G. [1978]
Shallow Binding in LISP 1.5
Communications of the ACM, 21, #7, July 1978, p565-569

Balzer, R.M. [1973*]
An Overview of the ISPL Computer
Communications of the ACM, 16, #2, February 1973, p117-122

Banâtre, J.P., Routeau, J.P., Trilling, L. [1979]
An Event Driven Compiling Technique
Communications of the ACM, 22, #1, January 1979, p34-42

Bezivin, J., Nebut, J-L, Rannou, R. [1978]
Another View of Coroutines
ACM SIGPLAN Notices, 13, #5, May 1978, p23-35

Birtwistle, G.M., Dahl, O-J, Myhrhaug, B., Nygaard, K. [1973]
SIMULA Begin
Auerbach, 1973

Bloom, H.M. [1975*]
Conceptual Design of a High-level Language Processor in High Level Language Computer Architecture, ed. Chu
Academic Press, 1975

Brinch Hansen, P. [1975*]
The Programming Language Concurrent Pascal
IEEE Transactions on Software Engineering, SE-1, #2, June 1975, p199-207

Brinch Hansen, P. [1978]
Distributed Processes: A Concurrent Programming Concept
Communications of the ACM, 21, #11, November 1978, p934-940

Chand, D.R., Yadav, S.B. [1978]
On the Application of Data Abstraction Facilities
ACM Conference, 1978, p639-645

Chesley, G.D., Smith, W.R. [1971*]
The Hardware-Implemented High-Level Machine Language for SYMBOL
Spring Joint Computer Conference, AFIPS 1971

Chevance, R.J. [1977]
Design of High Level Language Oriented Processors
ACM SIGPLAN Notices, 12, #10, October 1977, p10-32

Claybrook, B.G. [1977]
A Facility for Defining and Manipulating Generalized Data Structures
ACM Transactions on Databases, 2, #4, 1977, p370-406

Cowan, D.D., Lucena, C.J.P. [1978]
A Data-directed Approach to Program Construction
Technical Report CS-78-02, University of Waterloo

Cowart, B.E., Rice, R., Lundstrom, S.F. [1971]
The Physical Attributes and Testing Aspects of the SYMBOL System
Spring Joint Computer Conference, AFIPS 1971

Davis, A.L. [1979*]
A Data Flow Evaluation System Based on the Concept of Recursive Locality
National Computer Conference, 1979, p1079-1086

Demers, A. et al [1978]
Data Types as Values: Polymorphism, Type-checking, and Encapsulation
Fifth Principles of Programming Languages Conference, 1977, p23-30

DeMillo, R.A., Lipton, R.J., Perlis, A.J. [1977*]
Social Processes and Proofs of Theorems and Programs
Fourth Principles of Programming Languages Conference, 1977, p206-214

Dijkstra, E.W. [1968*]
Co-operating Sequential Processes
in Programming Languages, ed. Genuys
Academic Press, 1968

Dijkstra, E.W. [1975*]
Guarded Commands, Nondeterminism, and Formal Derivation of Programs
Communications of the ACM, 18, #8, August 1975, p453-7

Dijkstra, E.W. [1976*]
A Discipline of Programming
Prentice-Hall, 1976

Ellis, T.M.R. [1979]
Parallel Processing in an Adaptable Application Oriented Language Processor
SOFTWARE: Practice and Experience, 9, 1979, p183-190

Enslow, P.H. [1977*]
Multiprocessor Organization -- A Survey
ACM Computing Surveys, 9, #1, March 1977, p103-129

Feldman, J.A. [1979*]
High Level Programming for Distributed Processing
Communications of the ACM, 22, #6, June 1979, p353-367

Floyd, R.W. [1967a*]
Non-deterministic Algorithms
Journal of the ACM, 12, #4, October 1967, p636-644

Floyd, R.W. [1967b*]
Assigning Meanings to Programs
in Mathematical Aspects of Computer Science, ed. Schwartz pp19-32, 1967

Floyd, R.W. [1979*]
The Paradigms of Programming
Communications of the ACM, 22, #8, August 1979, p455-460

Friedman, D.P., Wise, D.S. [1978]
Unbounded Computational Structures
SOFTWARE: Practice and Experience, 8, 1978, p407-416

Gagliardi, U.O. [1973*]
Report of Workshop 4: Software Related Advances in Computer Hardware
Proceedings of a Symposium on the High Cost of Software
S.R.I., 1973

Geschke, C.M. et al [1977]
Early Experience with MESA
Communications of the ACM, 20, #8, August 1977, p540-552

Gostelow, K.P., Thomas, R.E. [1979]
A View of Dataflow
National Computer Conference, 1979, p629-636

Grune, D. [1977]
A View of Coroutines
ACM SIGPLAN Notices, 12, #7, July 1977, p75-81

de la Guardia, M.F., Field, J.A. [1976*]
A High Level Language Oriented Multiprocessor
Proceedings of 1976 Conference on Parallel Processing
IEEE, 1976

Guttag, J. [1977]
Abstract Data Types and the Development of Data Structures
Communications of the ACM, 20, #6, June 1977, p396-404

Hanson, D.R., Griswold, R.E. [1978]
The SL5 Procedure Mechanism
Communications of the ACM, 21, #5, May 1978, p392-400

Haynes, L.S. [1977*]
The Architecture of an Algol 60 Computer Implemented with Distributed Processing
Proceedings of the Fourth International Symposium on Computer Architecture
IEEE, 1977

Hewitt, C., Atkinson, R. [1977*]
Synchronization in Actor Systems
Fourth Principles of Programming Languages Conference, 1977, p267-280

Hoare, C.A.R. [1969*]
An Axiomatic Basis for Computer Programming
Communications of the ACM, 12, 1969, p576-580

Hoare, C.A.R. [1972]
Towards a Theory of Parallel Programming
in Operating Systems Techniques
Academic Press, 1972

Hoare, C.A.R. [1974*]
Monitors: An Operating System Concept
Communications of the ACM, 17, #10, October 1974,
p549-557

Hoare, C.A.R. [1978*]
Communicating Sequential Processes
Communications of the ACM, 21, #8, August 1978,
p666-677

Hunt, J.G. [1979]
Messages in Typed Languages
ACM SIGPLAN Notices, 14, #1, January 1979, p27-45

Ingalls, D.H.H. [1978]
The Smalltalk-76 Programming System: Design and
Implementation
Fifth Principles of Programming Languages Conference,
1977, p9-16

Jacobsen, T. [1978]
Another View of Coroutines [1978]
ACM SIGPLAN Notices, 13, #4, April 1978, p68-75

Johnson, S.C. [1978*]
A Portable Compiler: Theory and Practice
Fifth Principles of Programming Languages Conference,
1977, p97-104

Kahn, G. [1974*]
The Semantics of a Simple Language for Parallel
Programming
International Federation of Information Processing
North Holland, 1974

Kieburtz, R.B., Silberschatz, A. [1979*]
Comments on "Communicating Sequential Processes"
Transactions on Programming Languages and Systems, 1,
#2, October 1979, p218-225

Keller, R.M. et al [1979]
A Loosely-Coupled Applicative Multiprogramming System
National Computing Conference, 1979, p613-622

Kessels, J.L.W. [1977]
A Conceptual Framework for a Nonprocedural Programming Language
Communications of the ACM, 20, #12, December 1977, p906-913

Knuth, D.E. [1971*]
An Empirical Study of FORTRAN Programs
SOFTWARE: Practice and Experience, 1, 1971, p105-133

Kuhn, T.S. [1970*]
The Structure of Scientific Revolutions
University of Chicago Press, 1970

Lakatos, I. [1976*]
Proofs and Refutations: The Logic of Mathematical Discovery
Cambridge Univerity Press, 1976

Lamport, L. [1978*]
Time, Clocks, and the Ordering of Events in a Distributed System
Communications of the ACM, 21, #7, July 1978, p558-565

Ledgard, H.F., Taylor, R.W. [1977]
Two Views of Data Abstraction
Communications of the ACM, 20, #6, June 1977, p382-384

Liskov, B., Snyder, A., Atkinson, R., Schaffert, C. [1977*]
Abtraction Mechanisms in CLU
Communications of the ACM, 20, #8, August 1977, p564-576

Lewis, B. [1978]
Further Comments on "A View of Coroutines"
ACM SIGPLAN Notices, 13, #7, July 1978, p31-33

Maurer, W.D. [1978]
Register Type Bits - A Proposal for Efficient Instruction Repertoire Extension
ACM SIGPLAN Notices, 13, #9, September 1978, p34-35

McKeeman, W.M. [1967*]
Language Directed Computer Design
Fall Joint Computer Conference, 1967

Mizell, D. [1978]
Verification and Design Aspects of "True Concurrency"
Fifth Principles of Programming Languages Conference, 1977, p171-175

Myers, G.J. [1978*]
Advances in Computer Architecture
Wiley, 1978

von Neumann, J. et al [1946*]
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument
in Collected Works of John von Neumann, ed. Taub
MacMillan, 1963

Ogden, W.F. et al [1978]
Complexity of Expressions Allowing Concurrency
Fifth Principles of Programming Languages Conference, 1977, p185-194

Patel, J.H. [1979*]
Processor-Memory Interconnections for Multiprocessors
Sixth Symposium in Computer Architecture, p168-177

Prabhala, B., Sethi, R. [1978]
Efficient Computation of Expressions with Common Subexpressions
Fifth Principles of Programming Languages Conference, 1977, p222-230

Pratt, V.R. [1977*]
The Competence/Performance Dichotomy in Programming
Fourth Principles of Programming Languages Conference, 1977, p194-200

Price, R.J. [1979]
A Language for Distributed Processing
National Computer Conference, 1979, p957-967

Reynolds, J.C. [1979]
Reasoning about Arrays
Communications of the ACM, 22, #5, May 1979, p290-298

Rice, P., Smith, W.R. [1971*]
SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems
Spring Joint Computer Conference
AFIPS, 1971

Robinet, B.J. [1975*]
Architectural Design of an APL Processor
in High Level Language Computer Design, ed. Chu
Academic Press, 1975

Ruggiero, W. et al [1979]
Analysis of Data Flow Models using the SARA Graph Model of Behavior
National Computer Conference, 1979, p975-988

Schmidt, J.W. [1977]
Some High Level Language Constructs for Data Type Relations
ACM Transactions on Databases, 2, #3, 1977, p247-261

Shave, M.J.R. [1978]
The Programming of Structural Relationships in Dynamic Environments
SOFTWARE: Practice and Experience, 8, 1978, p199-211

Shaw, M., Wulf, W.A., London, R.L. [1977*]
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators
Communications of the ACM, 20, #8, August 1977, p553-564

Tanenbaum, A.S. [1978*]
Implications of Structured Programming for Machine Architecture
Communications of the ACM, 21, #3, March 1978, p237-246

Tennent, R.D. [1976]
The Denotational Semantics of Programming Languages
Communications of the ACM, 19, #8, August 1976, p437-453

Thurber, K.J., Myna, J.W. [1970*]
System Design of a Cellular APL Computer
IEEE TC, C-19, #4, 1970, p291-303

Turner, D.A. [1975*]
SASL Language Manual
Technical Report CS/75/1, University of Glasgow, 1975

Turner, D.A. [1979]
A New Implementation Technique for Applicative Languages
SOFTWARE: Practice and Experience, 9, 1979, p31-49

Watson, I., Gurd, J. [1979]
A Prototype Data Flow Computer with Token Labeling
National Computer Conference, 1979, p623-628

Wilkes, M.V., Stringer, J.B. [1953*]
Microprogramming and the Design of Control Circuits in an Electronic Computer
Proceedings Cambridge Philosophical Society, Part 2, 49, p230-238, April 1953

Williams, R. [1978*]
A Multiprocessing System for the Direct Execution of LISP
Fourth Workshop on Computer Architecture for Non-numeric Processing, p35-41

Winograd, T. [1979]
Beyond Programming Languages
Communications of the ACM, 22, #7, July 1979, p391-401

Wirth, N. [1971*]
The Programming Language Pascal
Acta Informatica, 1, #1, 1971, p35-63

Wirth, N. [1977*]
What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?
Communications of the ACM, 20, #11, November 1977, p822-823

Yau, S.S., Fung, H.S. [1977*]
Associative Processor Architecture -- A Survey
ACM Computing Surveys, 9, #1, March 1977, p3-27