Programming in Parallel:

A study in the psychology of software

A

Master Thesis

by:
Geert de Haan

supervised by:
dr. P.T.W. Hudson

University of Leiden
Unit of Experimental Psychology
Hooigracht 15
2312 KM Leiden
The Netherlands
Contents :

Title page: i
Contents: ii
Acknowledgments: iii
Abstract: iv

Chapter 1: Programming Considered as a Fallible Activity 1
  1.1 Introduction 1
  1.2 The design of computer-languages 1
  1.3 Why computer programming is so difficult 3
  1.4 Overview of what follows 7

Chapter 2: Parallelism in hard and software 9
  2.1 Why the future is parallel 9
  2.2 von Neumann goes parallel 10
  2.3 All sorts and conditions of parallel computers 12
  2.4 How to cook in parallel 18
  2.5 The hard case of parallel software 21

Chapter 3: How to Investigate Parallelism: time and knowledge 30
  3.1 Introduction 30
  3.2 Talking about time: psycho-linguistics 31
  3.3 From psycho-linguistics to programming 34
  3.4 Timing in natural and programming language 40
  3.5 Thinking about programming: cognitive psychology 47

Chapter 4: An Experiment about Programming Knowledge 56
  4.1 Introduction 56
  4.2 Introduction to the experiment 59
  4.3 Method
    4.3.1 Method: Stimulus materials 63
    4.3.2 Method: Questionnaires 64
    4.3.3 Method: Subjects 65
    4.3.4 Method: Procedure 66
  4.4 Results
    4.4.1 Results: cloze tests and text questions 66
    4.4.2 Results: experience and opinions 69
  4.5 Discussion 72

Chapter 5: Parallel Programming, hindsight and directions 77
  5.1 Summary 77
  5.2 Conclusions and Recommendations 82

References 93

Appendix A: Written instructions - group and individual testing 100
Appendix B: Stimulus texts - recipes and programs 103
Appendix C: Questionnaires - recipes, programs and background 110
Acknowledgments:

I owe a great deal of thanks to the following people:

**Patrick Hudson**, PhD., who kicked me under the ass at the right time, supplied the idea, supervised it, and inspired me all the way through. Without him, you would not have so much to read, and I would not have an honours degree.

**Karin van Opzeeland**, MSc., who kept me from starvation, by providing me with all the necessary physical and emotional food to continue.

**Thomas Green**, PhD., who taught me the basics of Human-Computer Interaction and lots more. Without him, I still might have known a lot about HCI, without, however, a vision on the field.

I would also like to thank all the people; friends, family, teachers and fellow-students, who indirectly provided the opportunities to arrive at this point. Of these, I would especially like to mention Leo Maris, Lex van der Heijden, and Martin Enkeling.
Abstract:

Programming a sequential computer is difficult, and this difficulty will increase with the arrival of parallel computers. At present, not very much is known about the psychological processes involved in programming, and there is a need for more adequate models of this behaviour. Nevertheless, something can be said about the relative difficulty of the various approaches to programming in parallel; by the use of general psychological insights, the existing knowledge about programming and other higher mental processes, and by the use of the psycho-linguistic knowledge about use and understanding of temporal relations and time-words.

In the following, the difficulty of programming is discussed, followed by a review of the main approaches to parallel computation and their associated difficulties. This is followed by a discussion of programming from a psycho-linguistic view of time, and a discussion of psychological modelling based on the present knowledge about programming. After a less successful empirical study about the factual and procedural knowledge involved in programming, the complete study is summarized and several conclusions and recommendations are drawn from it.
Chapter 1: Programming Considered as a Fallible Activity

1.1 Introduction

For many years computer programming was the exclusive domain of computer scientists. It is only in the recent past that experts from other domains have become involved with computer science, and only still more recently that computer scientists began to listen to these foreign sounds. One of the main reasons behind the involvement of, for instance, linguistics and cognitive scientists is that they themselves had began to use the computer. They noticed that many problems could be solved more fruitfully using knowledge from their own domains rather than by adhering to the engineering approach. For example, at the end of the sixties, extensive discussions around the use of GOTO statements were going on in the area of computer science. When, a few years later, the discussion had been more or less resolved into the doctrines of structured programming, psychological studies on programming were published, showing structured programming to be in need of, at least modernization (see: eg. Green, 1986).

Another reason for the involvement of foreign scientists with computer science is that the computer has gradually moved out of the engineering laboratory over the years. When programming had evolved from the binary and machine code levels to the higher language level, formal tools and grammars like BNF (Backus-Naur Form) could be brought into use to replace designers' intuitions. In this way, Niclaus Wirth's Pascal was the first language to be formally defined (Jensen and Wirth, 1978). More recently psychologists have began using formal grammars like BNF as a tool to study the human factors of interactive software (see eg. Reisner, 1983; Payne and Green, 1986).

1.2 The design of computer-languages

In the past, the design of computer languages heavily depended on the intuitive insights of the designer, resulting in, for example, languages that excel in functionality, but place high demands on human cognition. On the one hand, the artificial intelligence language Lisp can be taken as a typical instance of a language which is said to be very well suited for its purposes. On the other hand, due to the sheer number of functions, lack of an overall structure and "Lots of Incredibly Silly Parentheses" (Siklossy, 1976), it should not come as a surprise that it is also a very difficult language, especially for the novice programmer. In this context, Anderson and Jeffries (1985) report that many novice Lisp errors are caused by short term memory overload, in particular when the errors do not force detection, because the results seem to look all right. Something quite easily overlooked in design is the fact that cognitive performance often involves trade-offs.

In language design, there is often a trade-off between two or more performance features. The Lisp example exemplifies the trade-off between the functionality of the language and amount of information the user has to keep in mind. In this respect, declarative languages are generally easier to work with, though of course lacking the flexibility of functional languages. The language Cobol provides another example of such a trade-off. The language was originally developed as the standard computer language for business data processing. One of the design targets was to make Cobol programs self-documenting, so as to facilitate program reading and comprehension (Philippakis and Kazmier, 1981, p.18). Consequently, Cobol programs are quite lengthy and require much typing. It is precisely this amount of text that causes a lot of misspelling errors which are probably
among the most frequent kind of programming error. Whereas it is relatively easy for a compiler to detect and signal misspellings, it is much harder to overcome another consequence of length of Cobol programs, namely that it quite difficult to acquire an overview of a program, especially when it is large or complex. Another well known shortcoming of the Cobol is its overcompleteness. In order to enable real general purpose programming, Cobol contains so many special functions that hardly anyone can grasp the complete language, and only part of it is effectively used by any one programmer.

Almost all the main languages at the present day are languages from the past, as tradition and conversion costs seem able to enforce the use of these psychologically cumbersome languages; this is what Shneiderman (1980, p.6) calls the historical imperative. All these professionally important languages can be characterized by the fact that they are relatively difficult to use and are generally compiled. This may explain, at least partially, why so many amateur programmers, to the dismay of the computer experts, rely on the interpreted language Basic. Basic can not survive being compared with a modern language like Pascal or C in terms of neatness or speed. Nevertheless, in Basic the user may specify a problem simply from start to end in one sequence and even line by line, without having to be bothered with function and subroutine calls. In combination with interactively checking and debugging the program on the spot, Basic frees the user from having to know all the ins and outs of the language before getting down to actual work. This is quite similar to the popularity of graphical user interfaces among managers and domestic computer hobbyists. Although these interfaces are rather clumsy in that they require constant user interference, they allow for progress to be made in little steps instead of demanding a lot of knowledge from the user, and much planning in advance.

1.3 Why computer programming is so difficult

Having discussed some of the difficulties experienced with the present programming languages, the question can be asked what it is that makes them as difficult as they are. Although there are several points that may cause a language or programs written in a particular notation to be more and less easy to understand: "There is no central source of difficulty in programming, no single core problem that can be miraculously be dissolved" (Green, et.al., 1980, p.894). The reason why there is no central problem depends on the limitations of the human cognitive system, the characteristics of the problem in the outside world, and the way in which information is exchanged between the two. These limitations cause trade-offs to occur.

Consider, for example, the language or notation used to convey the information. A particular notation is more and less suitable for expressing certain problems: a mathematical one works well in stating mathematical problems but it is quite useless for describing history, which is a lot easier in English. Similarly, Lisp is well suited for A.I. purposes, but not for heavy number crunching. Of course, it is not for psychologists to say if a particular notation is useful for number crunching or not. However, they can say something about the suitability of notations for humans to work with, and explain why that is so. Roman notation, for example may be used to represent numbers, and everyone knows that one can not expect people to be fast and error free in adding or multiplying such numbers. However, explaining why this is so is something quite different, especially when it comes to using the insight in different situations. The argument about Roman numerals may be used as a serious warning against the unrestrained use of the intuitive "everyone knows that". After mentioning the difficulty of doing arithmetic using Roman numerals, Green (1980) asks his readers: "How come all those Roman and early Europeans, right up to the Middle Ages, failed to think of the Arabic system? More urgently, how can we be sure
that our programming notations are not just as backward?” (p.274). In the past decades, machine, assembly and ever higher level languages were developed to hide the lower level details of the computer from the programmer. Notations were designed to make effective use of the hardware, or in order to facilitate the expression of problems, mainly in the area of artificial intelligence. Apart from a number of ‘toy’ languages (eg. Grade, see: Sengler, 1983; Logo), programming language design has never started with the capabilities of the programmer.

The control structures are a well known source of difficulty, especially for novice programmers. Here, they will be used to show that even modern programming languages contain parts that, although intuitively looking perfect, they are not necessarily and allow for improvement. The most important control structure in machine, and assembly languages is the branch instruction in combination with some kind of test. The combination was used in the 'early' high level languages as statements like IF condition GOTO statement or GOTO statements ON condition. If it was not already, then after a number of articles against the use of GOTO's (eg. Dijkstra, 1968; Wulf, 1972), the If-statement was added to most languages in its present form: IF condition THEN statements ELSE statements. In this way, the If-statement is generally favorite, at least, because in comparison with the original, it makes the structure of the program much clearer. However, Sime, Arblaster and Green (1977) showed that using such a structure may be nice, but using a more redundant structure is even nicer. In their experiment, novice programmers using the form: IF condition DO statements; NOT condition DO statements; END condition were ten times faster in correcting their mistakes then the ones working with the usual form. These results can be taken as a simple example showing the likelihood of a grossly underestimated importance of perceptual factors in programming.

Another control structure is the repeating loop, in assembly language represented by a number of statements enclosed between some test and branch instruction on an index variable and a label at the start or end of the block. Almost without exception, programming languages provide for some kind of FOR loop, a loop with a preassigned number of iterations, such as: FOR index = start, stop, increment DO. Other looping constructs from the equivalents of Pascal's WHILE condition DO statements and REPEAT statements UNTIL condition. These loops differ with respect to where the test for the next iteration is placed: before or after the block. Depending on whether the number of iterations is known beforehand, and if not, where the test has to take place, each of the looping constructs is particularly suited to solve specific problems. This is exactly what Soloway and co-workers employed in a study into the ability of novice and intermediate level programmers to choose the appropriate looping construct (Soloway, Ehrlich, Bonar and Greenspan, 1982).

Soloway and colleagues asked programmers to design programs to calculate the average of a number of digits for respectively, a given number, when their sum exceeded a certain value, and after a special value was read in, but excluding that value. These problems can be most easily solved using a FOR, a REPEAT and a WHILE loop respective. It is also possible to use the other constructs, but only at the cost of increased program size and, more importantly, complexity. Soloway et.al. found that only 42% of the novices and 47% of the intermediate level programmers used the appropriate looping construct for each problem. They conclude: "Given the above data, one wonders if teaching all three is a good strategy." (p.43), but equivalently, one may wonder as well if keeping all three, or keeping all three in their present way is a good strategy. This example can be used to indicate that there can be, and here, there are, discrepancies between the ways of thinking, implicitly demanded by a programming language and the way in which humans actually think.
Given these two examples, one could now indeed ask if it is wise to continue designing new languages, starting from the computer architecture instead of from human cognition; especially when considering the advent of concurrent processing. Concurrent processing is used as the denominator of all attempts to construct computers with more than one central processor, which consequently should be able to execute more that one program or program statement at the time (for a layman's introduction, see: Veen, 1986, p.V-VIII). It is generally agreed that concurrent processing will lead to increases in the complexity of software. Because programs will no longer necessarily be executed in the statement order as they are written, the programmer will have to take account of this, or alternatively, she will have to state explicitly the order of execution. Referring to one of the first major (because 64) multiprocessor computers, Nickerson (1986, p.63) writes: "the ILLIAC-IV did not prove widely useful in solving complex problems, to the surprise and disappointment of its developers. The problem is not that there exist no tasks that require very large amounts of computation; the problem appears to be that we are not yet very proficient at designing algorithms for these tasks that effectively exploit a multiprocessor capability."

Considerable work has been done on the problems of programming conventional computers, and though some of these results may be applied to the programming of parallel computers, little is known about the specific problems and difficulties regarding the latter. However, even if psychology can not provide all the answers, and certainly no simple and clear cut answers, computer usage is becoming too important to be left entirely to the intuition of the technicians. This will be especially true for concurrent programming; as a new field of application without well established languages and tools, it provides the opportunity to do things right from the start, instead of having cognitive ergonomists pick up the pieces and do the patchwork.

A this point it is worth to mention the importance of human performance as an economical factor in computer programming, especially in comparison to the hardware costs. For instance, Nickerson (1986, p.269) writes: "...whereas in the early days of computers the hardware for a system often accounted for the larger portion of the costs, today the reverse is true", after which he quotes Birnbaum (1982) stating that between 1955 and 1985 the performance to cost ratio of computer hardware rose by a factor of 1 million, while programmer productivity increased only by a factor of about 3.6 during the same period. It might be added that the complexity of software has also increased, but on the other hand, de Kerf (1977) estimates that in 1980, 90 percent of the financial budget of computer centers would be spend on personal. Therefore, it seems worthwhile to spend at least more attention to the human factors in programming. And as long as human cognition and performance remain the key factors in programming, psychological research will be a prerequisite for optimum use of the available hardware.

1.4 Overview of what follows

In the following chapters, the problems introduced above will be treated in further detail. In Chapter 2 the computational techniques will be discussed. The chapter opens with the questions why nowadays serial computers are no longer fully adequate, how they work, and which attempts are made to increase their performance. Next, some characteristics of parallel computers are mentioned, and a classification of the major kinds of parallel computers is presented, followed by a comparison between how each of them works and a parallel kitchen as a more or less real life setting. The final paragraph deals with how to program a parallel computer, focussing on having to specify exactly what or how to process in parallel, do so implicitly or not at all.
For each of these approaches, examples of code are presented, using a newly designed language and an existing one with adjustments for parallel programming. For each approach, some remarks are made about the difficulties the programmer will meet.

Chapter 3 deals with the linguistic and psychological ways to investigate parallelism. The first paragraph consists of a linguistic analysis of time related words in natural language, based on the assumption that language is the mirror of the mind. It is briefly described how words describing the occurrence of events in time develop during childhood and how they are used in everyday speech, and how their use might explain some of the difficulties involved in understanding computer programs. The paragraph ends with a comparison between the meaning event related words in natural and computer languages. In the following chapter, the adequacy of the main psychological models of human cognition for research into programming is discussed, which focusses on the difference between the representations of common and programming knowledge, of which the latter can be characterized as shallow and highly individual.

In chapter 4 an empirical study of the difference between procedural and factual knowledge in programming and cooking is presented. Starting with a discussion of related work on programming, the reason to study these kinds of knowledge, why and how it will be investigated is explained. This is followed by a description of the experiment itself are a discussion of the results and possible implications.

The final chapter 5 starts with an extensive summary of the paper, followed by a discussion of the conclusions, which follow from the investigation. These are twofold, first there are conclusions to be drawn regarding the investigation of programming knowledge, and secondly, the recommendations following from the more and less suitability of the approaches to parallel programming.
Chapter 2: Parallelism in hard and software

2.1 Why the future is parallel

It depends on one's point of view who can be said to be responsible for the design of the first device worthy of the name computer. But, ever since, the development of still smaller, cheaper and faster computers has gone on with increased effort. At present, one may witness a shift from the kind of computers that process their instructions in a strictly sequential way to computers that can do several things at the same time. Although the development of the sequential computer is not yet completely finished, and parallel computers are not readily available, it is clear that the latter type will mean a big leap forward for computer science.

There are several reasons why parallel computers will become increasingly important. The first reason is an economic one. In the first digital computers memory and the central processing unit were made of different materials, and as memory was much cheaper, it was self-evident that there should be a lot of memory and only one processor. Nowadays, memory and processor are both made of silicon. According to Hillis: "In a typical computer more than 90 percent of the silicon is devoted to memory. While the central processor is kept wonderfully busy, this vast majority largely sits idle. At about $1 million per square meter, processed packaged silicon is an expensive resource to waste" (Hillis, 1987, p.86).

The second reason underlying the trend towards parallelism is that for a growing number of applications sequential computers are becoming too slow. The very first machines were used for a lot of simple numeric calculations, exemplified by names like "Automatic Sequence Controlled Calculator", "Card Programmed Calculator" and so on (de Kerf, 1977). Thereafter, the range of applications spread to text processing, electronic mail and data analysis, which still can be done quite satisfactorily using a reasonably fast machine; this has become especially true since the invention of the micro-processor meant that computers could become small and affordable (Whitney, 1980, p.91). In certain areas of research like astronomy, nuclear physics and aerospace technology, there has always been a demand for ever faster machines. More generally, in a number of applications like database management and transaction processing, problems have become too large to be adequately handled by sequentially operating computers. As sequential machines can not be made very much faster than they are at the moment, the only way to increase computational speed is to switch to parallelism.

The growth in popularity of parallel computers is also due to a change in the nature of the applications for which computers are presently being used. More specific, applications are not only getting too intensive, but the range of applications has also come to include those fields where the dominant problems are inherently parallel. Among others, these areas include computer art and animation; engineering with Computer Aided Design and modelling; and artificial intelligence and cognitive science, including research into knowledge representation, vision, neural networks and machine learning, and pattern matching.

For example, in recognizing objects the human visual system does not analyze the visual field point by point, as this would be far too time consuming and inflexible. Rather, the visual system can be described as a hierarchy of levels, with numerous elements at each level working in parallel on the whole of the picture, ending with some high level description (Marr, 1980).
Similarly, in computer animation, calculating the values of the pixels that constitute a moving object on the screen is so computationally intensive, that even hobby computers are being supplied with special purpose chips, working in parallel with the central processor.

Reviewing the history of computer applications, one might say that at first computers were mainly used for those things people are not very good at. Indeed, it is said that this was the main motivation behind the design of the predecessors of the computer, Babbage's difference and analytical engines (Graham, 1986, p.39). Now that the computer can not become much better using conventional techniques, it seems that computers are to become more and more like the human cognition system with its massively parallel powers.

2.2 von Neumann goes parallel

The majority of current computers are called von Neumann machines after John von Neumann. Early computers were hard wired, meaning that a program was 'written' using electrical wires, much the same as telephone operators connecting lines with a wire. Von Neumann brought forward the idea of storing the program's instructions in memory, where at the time the data were already stored; hence the name "stored program" computer (Stone, 1980, p.8). This made it possible to change from program to program in a much easier way than before, besides making it possible to change a program on the run.

Because of the separation between processor and memory, von Neumann machines work strictly serially. This means that first an instruction is fetched from memory, then it is decoded and specific operations (eg. addition, etc.), data and addresses are determined. Next, the instruction pointer is updated to point to the following instruction, and finally, the instruction is executed and its results stored at the appropriate place, after which a new cycle starts all over again.

In theory, logic signals can travel with the speed of light, or approximately thirty centimeters per nanosecond. Even though this seems quite fast, it may be clear that the speed of sequential computers is limited. This is all the more because, in practice, this speed is never attained. Propagation delays, switching time and the time necessary for electronic signals to stabilize all conspire to slow the absolute maximum speed down (Stone, 1980, p.363).

In computer architecture the term 'Von Neumann bottleneck' is used of a single processor device, to denote the limiting factor of having a separate processor and memory. Because of this, within the framework of the Von Neumann architecture, only relatively small increases in speed are attainable.

Among these are attempts to increase the speed of the hardware, by using 'high speed' materials like Gallium Arsenide and ever more larger scale integrated circuitry to reduce travelling distance. The speed of execution of the instructions by the arithmetic-logic unit is enhanced by an increased word width and processing databits in chunks, instead of bit by bit.

At the level of the central processor, scalable, writable and otherwise reduced instruction sets seek to speed up instruction encoding and execution, utilizing a few fast and frequently needed instructions, and the use of pipelining allows for overlap between all four stages of the processor cycle, somewhat similar to an assembly belt.

At the levels above the processor, speed is increased by dual memory access allowing for storage into memory to take place during instruction fetches. Further, special purpose peripheral processors are used to free the central processor from anything except for the real computational tasks, and finally, fast cache memory is used to collect data from slower memory and peripheral devices to insure that they are available when the processor needs them.
Using the methods just mentioned, it is estimated that computer speed may increase by a factor between two and eight. However, it should be realized, that within the von Neumann framework, each additional factor speed increase costs by several factors. Therefore, parallelism is not only a more promising approach, but in the long run it will inevitably become the principal one.

Parallelism is not something completely new. Zimmerman and Sips (1978) mention that already in 1958, the National Bureau of Standards had begun using a Pilot system. Featuring three processors, this system can be classified as the first multiprocessor system. Ever since, most of the so-called supercomputers have always featured some kind of parallelism. Parallelism is neither something completely new, as it is in various ways responsible for many of main advances in computation power of the sequential computers. Dedicated processors for floating point and input-output operations, pipelining and utilization of parallelism at the binary level have been responsible for exponential decreases in execution time. However, these approaches differ from the present developments in computer architecture in that they still adhere to the von Neumann framework in principle. That is, some pieces of hardware are hired to carve the bosses' food, do the clerical work or collect the pieces, but it is still one boss who takes the decisions. The parallelism at stake concerns all attempts to spread the work among a number of processors, consisting of anything between two and several tens of thousands of processors, without an almighty central executive in charge with everything.

2.3 All sorts and conditions of parallel computers

One of the problems, associated with finding general categories of parallel computers, is that one easily ends up with enumerating a number of more or less distinctive features. Parallel machines differ from sequential ones in their ability to do a number of things simultaneously, and if possible, rapidly and cheaply. However, while serial computers tend to share a common general structure, no two designs for organizing parallel computers have been even remotely similar. This is because, at the architectural level, the choices to be made concern sliding scales, while answers to one question may often depend on the answers to other questions. As a first introduction to present day parallel computers, and the problems associated with designing them, consider the following classification.

According to Durham (1987), the parallel computers which are commercially available at present, can be classified within two groups of 'farms' and 'cubes'. The word 'farm' is used for machines with up to 30 general purpose processors, each essentially comprising a powerful computer on its own. Each of these processors may optionally contain additional special purpose processors, local memory, etc., and usually they share a global memory. Instead of having a single queue of programs waiting to be run, similar to the conventional computer, a farm computer also possesses a processor pool, listing all idle processors. A processor may then be assigned to a process, fetch its code and data from global memory, execute it, finish, and return to the processor pool. The computers from the farm group are typically used as high performance general purpose computers, and for (ie. fault tolerant) on-line transaction processing.

The computers called 'cubes' consist of a great number of relatively dumb processors, each having similar limited capabilities and some local memory. The processors, or rather intelligent memory cells, are arranged within a network, usually according to a hypercube topology (hence, the name 'cube'). Plainly stated, hypercubes are the family of all cubes featuring more than 6 planes which generally form a regular pattern, but are not necessarily
situated at the outside. In such a cube, the processors are placed at the vertices and communicate with one another by sending messages along the line segments (Hillis, 1985).

This classification reflects two different strategies for attaining high speed at a low price. One is to keep the communication between computations to a minimum, by dividing the task into large processes that operate on their own private data. In this strategy, costs can be kept to acceptable levels by the use of standard commercial processors, with some added hardware. The second strategy tries to minimize costs by using a large number of identical VLSI (very large scale integration) processor chips, provide for simple and cheap communication and coding the program in a special format. Here, programs are segmented into very small, easily communicated processes, which are analyzed by special hardware to determine which processes can execute in parallel.

One of the things to be inferred from the explanation exemplifies the fact that many design considerations are related. Choosing for a coarse grained computer will place little demand for communication facilities, except maybe during program loading, but a programmer has to be aware of, and care for the partitioning of the program in separate processes. On the other hand, choosing a fine grained computer requires the design of special hardware and extended communication facilities, but generally releases the programmer from segmentation problems. To this might be added that extension of a 'farm' computer requires programs to be rewritten, whereas, on a fine grained machine only recompilation, if anything at all, will be required.

Another classification, proposed by M.J. Flynn (discussed by Stone, 1980, p.364-367), is based on more theoretical grounds. In this classic, but somewhat coarse grained view, four classes of computers are distinguished, depending on the parallelism within the instruction and data streams. The single-instruction single-data stream (SISD) computer is the conventional one thing at a time serial computer. In a single-instruction multiple-data stream (SIMD) computer, each instruction operates on a number of data sets at the time. A SIMD computer typically consists of a single central control processor, and a number of processing elements with little, if any intelligence. By replicating the instruction stream in a serial computer, a multiple-instruction single-data stream (MISD) computer is created, in which each operant is operated upon simultaneously by several instructions. No such a computer has ever been created yet, but stretching the definition a little, an example may be found at the binary level, where bits can be read and added to other bits at the same time (see: eg. Gosling, 1980). The fourth kind of computers are of the multiple-instruction multiple-data stream (MIMD) type. Here, the computer is composed of a number of processors, each of which is itself a complete computer. It may be noted, that Flynn's classification only distinguishes between SIMD and MIMD kinds of parallel computers, and even fails to make a clear distinction between the computer of the farm and cube types.

In order to refine the level of granularity of types, the classification of Veen (1986) may be useful. Instead of using 'SIMD' and 'MIMD', the closely corresponding distinction between synchronous and asynchronous processing is used. The distinction between the two is that: "If the parallel operations are synchronized at the machine language level, scheduling and allocation needs to be done by the programmer. In asynchronous machines the processes that run in parallel need to be synchronized whenever they communicate with each other." (Veen, 1986, p.10). It should be noted that Veen makes the distinction between SIMD and MIMD type of computers at the machine language level, which means that both parallelism at the binary level (ie. clock, parallel arithmetic on the bits a single data element), as well as networks of conventional machines are excluded.

Within the SIMD type of computer, a further distinction can be made between vector processors, processor arrays and associative processors. In associative processors many primitive processing elements are directly
connected to their data; those elements that are active in a given cycle all execute the same instruction. An example of an associative processor are Thinking Machine's Connection Machine (Hillis, 1985), what Durham (1987) called a 'cube' computer.

A vector or array processor (not to be confused with processor arrays) is a computer designed to operate on vectors as basic units of data, instead of individual words as the SISD computer. The most popular of today's supercomputers, such as the CRAY and Control-Data machines attain their speed by special high-speed hardware, and pipelines on instruction and data streams (Levine, 1982). Programming on a vector computer can be made more comfortable by the use a standard scientific language, such as Fortran, and a vectorizing compiler, which attempts to find the parallelism 'hidden' in a sequential program, and to translate it into efficient code. However, the programmer still has to be careful, as Perrott and Zarea-Aliabadi (1986, p.9) write: "The art of programming a vector processor in standard FORTRAN involves assisting the compiler to vectorize as much of the program as possible."

Processor arrays are in essence vector computers implemented as an array of processors with a limited communication among the processing elements. A processor array, such as the much famed, but ill fated Illiac IV differs from an associative processor by the fixed connection network between the processing elements. In the Connection Machine the network can be reconfigured to match a particular problem. Which is particularly important in problems with a highly irregular (data) structure (Hillis, 1985, p.27). Here, the programmer will carefully have to mold the structure of program and data after the hardware, on penalty of high inefficiency.

On the side of the MIMD or asynchronous types of computers, Veen (p.10) distinguishes between fine grained and coarse grained, or large process computers, based on the size of the instruction unit executed by each of the processors. This was discussed before, but it must be added that the feature of granularity of not restricted to asynchronous computers, as, for example, the Connection Machine is also a fine grained computer.

Asynchronous fine grained computers are divided between dataflow and reduction machines, a distinction which is equivalent to the one between data-driven and demand-driven execution. Here, both types of machines are essentially similar, except that in data-driven execution operations are performed whenever the operands are available, whereas in demand-driven execution the opposite holds. As the name indicates, reduction machines are based on reducing complex expressions to the values of their constituents, which may be computed independently. As such, they are used for functional and logical notations like Lisp and Prolog. Note that dataflow and reduction are not opposites. The opposite of dataflow is flow of control, used for the conventional computer, where the order of execution is important, as is reflected in the associated imperative languages such as Fortran. Dataflow computers are mainly used in combination with single assignment languages, such as Id, and Sisal (see: Arvind and Nikhil, 1987; Gurd et.al. 1986). Here, data-driven execution is made possible by having values assigned to variables, once and for all; hence, the name of the language type. Although fine grained asynchronous computers are being heavily investigated, so far no such computer has left the laboratory stage.

Computers in the category of coarse grained asynchronous machines are readily available, ranging from add-on accelerator boards for personal computers to the newest minisuper and supercomputers. In the computer types of Durham (1987), the farm computer is an exemplary example for this group. Coarse grained asynchronous computers complete the classification of Veen (1986), which is presented in table 1, and to which examples of computers in each class are added.
Synchronous

or SIMD: Associative Processors: Connection Machine
Processor Arrays: Illiac IV, ICL DAP
Vector Processors: Cray-1, CDC-205

Asynchronous

or MIMD: Fine Grain: Reduction: Parallel Reduction Machine
Dataflow: Manchester Dataflow Machine
Coarse Grain: Fixed Architecture: Doom, Sequent Balance
Flexible Architecture: Inmos Transputer

Table 1: A classification of the main types of parallel computers

Although Veen does not, Treleaven (1986, p.44-45) distinguishes two subcategories within the class of coarse grained asynchronous computers, comparable to the difference between their SIMD equivalents: processor arrays and associative processors: parallel Unix systems and VLSI-based systems. In general, the former type, has a more or less fixed architecture and is mainly used for transaction processing systems and high performance general purpose computing. The name Unix systems, referring to the most popular operating system for these machines is somewhat misleading, because this category consists of all systems based on powerful commercially available processors with a fixed but adjustable architecture (eg. Test et.al., 1987).
The Philips Decentralized Object Oriented Machine (Doom for short) is a Esprit research project, which also belongs to this category. It is based on a style of programming in which program data and code is hidden in a number of units or objects which can be processed in parallel, and exchange messages to request information or operations from each other (Bronnenberg et.al., 1986; Odijk, 1987).
The main representant of the VLSI group is the Inmos transputer, which is itself a very fast processor, but in addition it can be flexibly linked to other transputers in a still more network (May, Shepherd and Keane, 1986). According to Anning and Hebditch (1986) such a network could provide the computational power of a supercomputer at a fraction of the price. However, besides mastering the dedicated language Occam, programming such a system demands an extreme care in segmenting and assigning processes to transputers, because: "There is a danger with the language that it makes one or two transputers do all the work while the others try to catch up." (p.45).

It should be noted that the classification just discussed is not the only one. First, the SISD, etc. classification is a general and theoretical one, whereas the practical questions may have a different focus, besides which most of these machines are not build for general purposes. For instance, in speed, SIMD machines differ from MIMD computers by no more than a multiplicative constant, and both can simulate each other (Hillis, 1985, p.24), therefore, it may be more important to focus on other features like communication and granularity. To this can be added that existing machines may be classified differently, depending on which point is stressed. For instance, a supercomputer, such as the Cray-1 can be characterized as a pipeline processor, but also as a processor array featuring different functional units. It may even be regarded as a MIMD system because, although to a limited extent, these units can process data independently (Levine, 1982).
Secondly, already indicated by the previous example, many of the feature by which machines may be classified actually involve matters of degree rather than of fact. For instance, there is a trade-off between the number and the size (both, physical as in terms of computational power) of the processor. Where then, one may ask, is the line to be drawn between the farms and the cubes?

Thirdly, a number of architectures which form an important direction in research are not described here, because they do not appear to fit in any category. These are neural networks, Boltzman machines, and probably systolic arrays, which are more and less hard wired, self-adjusting networks capable of learning. As such, they are situated somewhere between data and instruction streams, as well as between hard and software, and comparable to nothing but the brain (Hopfield and Tank, 1986; Hinton, 1987).

2.4 How to cook in parallel

To explain things in a less technical sense, consider the following comparison between the workings of a parallel computer and a real-life task, such as cooking. Here, the question is how a number of meals would be prepared supposing that the kitchen was organized according to the architecture of each of the kinds of parallel computers.

To start with the most simplistic case, a SISD-cook or serial 'food-processor' would cook the meals one by one, until finally everyone had been served. This approach is suitable when each guest wants to have totally unique dinner served and served at a different time for all the other guests. This approach is most inefficient if all the guests want to eat the same meal at the same time.

The MISD cook would also prepare the meals one by one, but notwithstanding the fact that a cook has a limited number of hands, etc., and that cooking potatoes generally takes longer than cooking spinach, all the dishes would be prepared simultaneously, take equally long or short, and be finished at the same time.

In all the following parallel kitchens, the chief cook would be provided with a number of assisting cooks. In the pipelined SIMD kitchens, all these cooks, just as many as there were ingredients, would be lined up behind an assembly belt on which plates moved by. All the cooks would, if there were any ingredients, take a particular ingredient from the plate, prepare it and return it to the next cook. For some reason or another, with any change of menu, or just a single ingredient, the whole kitchen was to be cleaned. This approach, with great resemblance to a canned food factory, is very suitable when an enormous number of guests would like to have exactly the same meal. However, the opposite holds, if only one in every tenth guest was to request something else.

In the processor array kitchen, a fixed number of cooks would be working side by side on an equal or smaller number of meals. Here, each individual cook would either be busy, preparing the same meal as all of the others did, or do nothing. As such, if all the guests would ask for the same meal, all the cooks would be constantly busy. However, if each guest would ask for a different meal, then only one cook at the time would be busy, while all of the others would be taking a break.

The associative processor kitchen would contain a large number of little cooks, and each of the cooks being handed a plate with raw as well as cooked ingredients, and if one particular raw ingredient was present, he or she would cook it. Then, after all the cooks had done their part, the plate would be passed to another little cook, somewhere in the direction of the dinner room, and a new plate would be handed. When, after an instance of time, the plate reached the dinner room, everything on it would be done. Here, if not all the guests would have the same meal, a number of cooks would be off. However, because little cooks generally receive low wages, it
would not be as bad as in the processor-array and pipeline kitchen. Note, that it is not very efficient if all of the cooks should wait for each other, and that is what is abolished in the asynchronous kitchens.

In the fine-grain asynchronous kitchen, the chief cook would be busy, putting all the necessities for the meals of the guests on the kitchen table. Then, many little cooks would each take his or her special ingredients, prepare them, and put them back on the table. In a dataflow kitchen, none of the cooks would be doing anything, unless all the ingredients he or she needed were on the table. That is, if there were not both tea and boiling water on the table, a tea-cook would wait for them to arrive. The difference with a reduction kitchen is that there, when a reduction cook would need anything, it would ask for it. On the other hand, if all of the ingredients would be readily available, no reduction cook would do anything, unless being asked by someone. Both of these kitchens can be very efficient, not in the least, because of their cheap labor. However, for some strange reason, the little cooks would be completely exhausted after doing something, and would have to be send off. Therefore, before really being useful, either the exhaustion of the little cooks should be cured, alternatively, each should be enable to prepare any, instead of only one particular ingredient.

The final kind of kitchen is the coarse grained asynchronous kitchen, and much like a human kitchen, it may be somewhat small and badly furnished, but the cooks are generally of the highly qualified SISD type. In some of these kitchens, each of the cooks will be preparing the meals for his or her own guests in a separate corner while sharing the most expensive equipment. Here, hiring more cooks would save on equipment costs, but if the more of them would need to use the same tools at the same time, their respective guests would have to wait. In such kitchens, the efficiency of equipment use increases with every additional cook, but diminishes relative savings in equipment.

Kitchens may become extremely efficient if the cooks would work together as a team, although this requires some flexibility on behalf of the cooks and quite some deliberation on task assignment. Here, cook-one would then chop a sufficient number of tomatoes for the meals of all guests, while cook-two could do the same with the potatoes, and so on. Besides, being smart cooks, they would be able to avoid having to use the same tools. In principle, the more cooks, the more efficient work could have been done. However, for a kitchen manager, this would take a lot of careful planning and decision making to have the cooks fully cooperate with each other. Here, the kitchen manager would have to avoid that, for instance, one or few of the cooks do all the work while the others hang around, or to keep some cook from accidentally cook all of the tomatoes like the potatoes.

2.5 The hard case of parallel software

Having reviewed the technical side, it is time to put forward the question how the human programmer is to operate a parallel machine, and maybe more important, determining which of the available programming approach is best suited for the human cognitive system. The human brain is the best existing example of a (eg. massive) parallel computer, and consequently, human beings are often characterized as high bandwidth processors, being able to process a lot of information on different levels at the same time. Therefore, one would expect it to be relatively easy for a human to program his silicon counterpart.

However, there are a number of important practical things, which severely limit the bandwidth of the information exchange between man and machine. First, the physics of the input and output apparatus put constraints on the information. Here, the increasing use of interfaces featuring graphics, speech and pointing devices might be taken
as sign of the growing recognition of the human capability to deal with higher bandwidth information. Secondly, to be of use, the information must be clear, complete and nonambiguous. Nowadays computers are relatively expensive and slow, so it should not be expected that, at short notice, they will be supplied with the immense amount of knowledge which enables humans to understand each others telegram style of utterances.

In the course of years, computers have become easier to use without the need to completely specify exactly what there is to be done and how to do that in binary. Rather, it is made possible to ‘talk’ to the computer in much more human way, and thereby, the complexity decreased. The price paid for this was quite fair, if hardly noticeable at all, because of decreasing costs and increasing speed of the hardware.

Switching to parallel computers will mean a dramatic increase in computational speed, but probably in complexity as well. First, the same difficulties of using a Von Neumann machine will remain, not in the least because often, serial computers are to be used as a front end machines. Second, additional difficulty will arise, because of having to deal with more than one thing at a time. The main questions arising from the preceding are the following. First, which are the main approaches to expressing parallelism in programming languages. Secondly, can anything preliminary be said about the programming difficulty of these approaches. Regarding the first question, it is a general finding that each approach is strongly related to the possibilities, the different types of machines provide, although there are no one to one relations.

Before discussing the main classifications, consider the following related alternatives. Hillis and Steele (1986) distinguish between data parallel and control parallel programming languages, a distinction which parallels the difference between synchronous and asynchronous multiprocessor systems. In the former, a set of data is processed in parallel, and the associated programming languages will have special parallel datatypes, matrix and For-All-Elements operations. In asynchronous multiprocessors, several processes may be active simultaneously, which will be reflected by operations for the creation or deletion a processes, constructs for synchronization and for message and data exchange. Here, one may note that there is a remarkable similarity between programming an asynchronous multiprocessor and serial multiprogramming system.

Another interesting classification is presented by Treleaven (1986), although for practical reasons, he preferred to use another. It is based on programming style, in which parallel computers as well as the associated languages are classified on how precise one has to be to specify how something is to be done versus having to specify merely what is to be done. Starting with procedural languages and asynchronous multiprocessors, the classification continues with object oriented processing, followed by single assignment and functional programming on dataflow and reduction machines to logic and production system languages, and the associated rule-based and connectionist machines. Note that programming on the latter machines is not necessarily easier, that is, choosing rules for use in a production system like Ops-5 should be done with at least the same care as choosing a process structure in a procedural language.

Three main approaches can be distinguished where it concerns expressing parallelism in programming languages; explicit parallelism, implicit parallelism, and no specification of parallelism at all. Here, the order of the approaches more or less reflects the amount of attention the programmer has to give to actual execution of the parallelism in a program. Note that if the programmer does not has to pay such attention, it may be sensible to do, comparable to the fact that low-level knowledge of a serial computer may help creating clever programs. Implicit parallelism is used to denote the languages where the programmer either can not or does not have to
indicate which parts of a program can be processes in parallel, whereas in the case of explicit parallelism, the programmer just has to.

**No parallelism** has to be expressed in three situations. First, there are some languages, like Prolog and production system languages which focus on attempting to prove or disprove statements which express relations between still lower level statements, and ultimately what may be called 'facts'. As such, a large part of running such a language consists of searching memory for matching clauses, a process which is particularly suited for parallel execution, and unsuited for serial machines. A major part of Japan's fifth generation computer project is to build a Prolog-based parallel inference engine (Feigenbaum and McCorduck, 1983). It is however a question if Prolog-like languages will ever become popular outside the world of academic artificial intelligence research, because these languages are not easy to use for general purpose problem solving, such as numeric and procedural problems.

In the second place, parallelism does not have to be specified on some coarse grained MIMD machines, on which the presence of a parallel architecture or the actual amount of parallelism may be hidden under several layers of the operating system from the view of the application programmer (eg. de Bruijn, 1987). However, this approach will not get the utmost from parallelism due to large overheads. In the long run it may be better to regard it as a attempt to speed up the serial architecture.

In the third place, parallelism does not have to be specified, when a compiler is used to detect the parts of a program that can be done simultaneously. Parallelizing or vectorizing compilers are used mainly on vector processors and processor arrays used for scientific calculations. Such calculations often consist of repeating-loop calculations on arrays values, and loops with arrays are particularly suited to vectorization.

Consider for instance, the following Fortran loops featuring only one repeated statement:

```fortran
DO 1 I = 10, 30
   A(I) = A(I) + 5
1 CONTINUE
```

```fortran
DO 1 I = 10, 30
   A(I) = A(I-2) + 5
1 CONTINUE
```

**Figure 1**: Vectorizable and non-vectorizable Fortran fragments

Here, the low-level operations consists of fetching the contents of an address in memory, adding 5 and storing the result in memory again. If this statement is repeatedly executed, then the three operations could be executed at least partially in parallel. Here, a smart compiler should be able to convert the loop on the left into parallel code. The loop on the right however, will at best be only partially parallelizable, because most times, the values of the calculation depend on the results of earlier iterations. This may be taken as an example of a situation in which a smart programmer might know and be able to use some 'dirty tricks' to patch the problem.

The main reasons behind this approach are independence of the target machine hardware, software and programmer portability and the amount of existing software. A major disadvantage is the fact that such compilers are often not able to detect the parallelism inherent in the problem, whereas programmers could, but may not be able to express it, or influenced by the underlying serial nature of the language, may not come to think of it.
When the decision not to use a parallelizing compiler is made, then the main options are the use of either implicit or explicit expression of parallelism. The difference between explicit and implicit parallelism should not be thought of as an absolute one, but rather as a scale, ranging from having to specify where to do what on which data to something like do this to those data.

For both of these there is an additional choice of adapting an existing language with extensions to enable evaluation of expressions in parallel, or to use a newly designed language. The use of an existing language with extensions is mainly motivated by the same reasons as using vectorizing compilers, but also depends on the ease of implementation.

Examples of the use of an existing language in which explicit parallelism is expressed are given by Perrott and Zarea-Aliabadi (1986), who discuss the use of Fortran on, among others, the Illiac IV array processor. Because this machine can work on arrays with 64 data elements at once, it is particularly suited for the use of explicit parallelism. In the Fortran adapted to this machine, one may declare an integer array with the default number of Processing Elements, by the statement: "PE INTEGER V(*)."

Examples of Fortran code are given of parallel arithmetic operations using arrays as operands. On the right, the serial equivalent of the last statement on the left is presented as a means of comparison.

```
V(* - 1)
DO 99 I = 1,64
IV(*, 5, 10) / V(*)
IF (M(I,5) .GT. 0) V(I) = 0
IF (M(*,5) .GT. 0) V(*) = 0
99 CONTINUE
```

**Figure 2 : Illiac-IV and standard serial Fortran fragments**

The processors of the Illiac IV have a limited capability to communicate, which is used in the code at the top-left, by which an array is rotated by one (i.e. V(1) = V(2) ... V(64) = V(1)). It may be clear by comparing the last line on the left and the code fragment on the right that the Illiac IV is fast; here at least 64 times faster than a serial machine. However, note that this speed increase is gained at the cost of pushing the problem into the particular array structure. The task of the programmer will then not only consist of translating a problem into an algorithm, but also of fitting the algorithm onto the arbitrary array length.

Here, reading and understanding a program will be much more difficult, because of the added difficulty of having to distil the algorithm from the machine dependencies. This means that here, speed increase is gained at the cost of getting further from what is generally accepted as the goal of parallelism from a strictly cognitive-ergonomic point of view. That is, to be able to express the parallelism inherent in a problem, about which it is hypothesized that it facilitates human productivity and understanding.

Another example of explicit parallelism, presented below, is a fragment of code in the notation Occam. Because Occam was specially designed to use the Inmos transputer chip as a building block for flexible MIMD systems, Occam is both a flexible, as well as a small and low level language (Inmos, 1984). As such, a designer may be able to build a mini-supercomputer out of transputers, but the programmer has to write quite a lot of code to merely produce one of the standard programming examples.

Further, Occam is quite different from other languages for MIMD machines, like concurrent-c and ADA, because those languages were principally designed to support high level multiprocessing, featuring facilities like control
of remote tasks and complex message exchange. The example shows the code for a simulation of a serial data line, connecting two parallel lines by (de-) multiplexing, freely adapted from Welch (1986, p.368), with added comments, and without the demultiplexing subroutine.

CHAN OF INT hard.link: --this program
PAR --multiplexes and
plex.2.to.1 (in.0, in.1, hard.link) --demultiplexes data
plex.1.to.2 (hard.link, out.0, out.1) --in parallel

PROC plex.2.to.1 (CHAN OF INT in.0, in.1, out)
WHILE TRUE
INT x.0, x.1: --values in this subroutine are taken
SEQ --from and put into channels in sequence
PAR
in.0? x.0 --but, the values are taken from the
in.1? x.1 --channels in.0 and in.1 in parallel
out! x.0; x.1

Figure 3: Explicit control parallelism in Occam

Notice that indentation is a meaningful part of the language and not only used for the sake of clarity. In order to execute this program the programmer will have to add much extra code, because Occam is such a low level language. Nevertheless, the example indicates both a strong as well as a weak point in explicit MIMD programming.

A strong point of languages like this, is their suitability for (eg. real-time) simulation and modelling, and not only at the lowest levels. The weak point is that the programmer may loose control of the system as a whole, because only the single elements can be controlled. Occam, as well as the other higher level languages with explicit parallelism are relatively new and therefore well designed, and enable neat programming. But the biggest problem with MIMD programming is not neatness, but not loosing track. As such, it may be that a completely different approach like object oriented programming will prove more useful, especially to the novice programmer.

As an example of an existing language which expresses implicit parallelism, the following Lisp program was adapted from Hillis (1985, p.44). This program takes two points A and B in an arbitrary network of points G, and calculates the shortest path between A and B. In the program, each point is represented by a data structure which contains a label and a number of links to its neighbors. In fact, each node is a processor in the connection machine, but the programmer does not have to know that, and neither that the '@' (ie. a greek alpha) means to perform for all in parallel. Because of this, the language CmLisp can be said to feature implicit parallelism. Here, the '@' is to be read as 'do-for-all', the '* (ie. dot) as 'exclude-from-all', while the '#' means to reduce the following to one value.
(DEFUN PATH-LENGTH (A B G)  
  @(SETF (LABEL *G) +INF)  
  (SETF (LABEL A) 0)  
  (LOOP UNTIL (< (LABEL B) +INF)  
    DO @(SETF (LABEL *(REMOVE A G))  
      (1+ (#MIN @(LABEL *(NEIGHBORS *G)))))  
      (LABEL B))  
  )

Figure 4: Implicit data parallelism in CmLisp

The algorithm can be read as follows: set all the labels in the network, not the network itself, to +infinity, but reset point A by zero. Then relabel all the points, except A to one plus the minimum of it's neighbors labels, and repeat this step until point B's label is finite, and equal to the length of the shortest path(s) from A to B. In other words, what happens is that all points receive some 'odd' value +INF. Then, starting with a zero at point A, all the points at distance one (eg. neighbors) from A get A's value plus one is one. In the next loop all points with distance two get the previous value one plus one makes two; and so on, until point B is reached via the shortest path.

The reason to include the CmLisp example was to show the flexibility of a SIMD machine. As opposed to the Illiac IV, on the Connection Machine this is quite possible, because the design focussed on the connections, rather than merely a lot of processors.

Programming a connection machine is much easier than programming an array processor, because the programmer is not bound to a certain array-width. Also, because the programmer can use programming language constructs which seem to better reflect the way human being think: concepts like 'for-all' and 'except-for' frequently occur in human speech, where as 'for index is 1 to 100 do' rather not.

As a final example, Sisal was chosen as a newly designed parallel programming language with implicit parallelism, in which the programmer can not express parallelism. Although Sisal may run on a conventional machines like the Dec Vax, it is specially designed to be run on dataflow computers by the use of single assignment, which means that any 'variable' used, must receive a value, once and for all.

Single assignment languages like Sisal, Val and Id are free of side-effects, which means that the values of the variables in an expression do not depend on where and when the program is executing, nor on how it came there. In procedural languages however, the value of a variable may be changed anytime and in different parts of a program, therefore the values of expressions can only be established with certainty when it executes. The following program, adapted from Veen (1980, p.47), calculates so-called Fibonacci numbers, which results from adding succeeding natural numbers, starting with one (eg. 1, 1, 2, 3, 5..), until one such number exceeds the value of max.
fibnumbers := FOR INITIAL
  fib1 := 1 ; fib2 := 1
REPEAT
  fib1, fib2 := OLD fib2, OLD fib1 + OLD fib2
  WHILE fib2 < max
  RETURNS ARRAY OF fib2
END FOR

Figure 5: Implicit non-expressible parallelism in Sisal

This example, which shows the resemblance between Sisal and Pascal contains two notable things. First, in Sisal the values of local variables in loops are not retained between iterations, except when the word 'old' is used to indicate the need for a value from the immediately preceding loop. It may be noted that the use of the keyword old creates data dependencies between the iterations of the loop, and comparably to the example of the vectorizing compiler, only partial parallelization will be possible.

Secondly, the program defines the name 'fibnumbers' as the resulting array of fibonacci numbers without declaring its dimensions. Because any array assignment demands the creation of a new array, there is, except where the programmer needs it, no need for explicit indexation.

In the previous paragraph, various examples of the implementation of parallelism at the programming level were shown. Most but not all machines also allow other programming approaches to be used, than the particular examples might suggest. In general however, the hardware and the costs of it will usually place restrictions on how parallelism to be expressed in programs. Supercomputers for instance, are too expensive to be used as easily operated general purpose machines, rather than for number crunching by experts.

Some preliminary remarks were made on the difficulty for the programmer when working with a particular type of language. Preliminary, because at present little is known about the language features which determine the difficulty for the human programmer in expressing parallelism. However, because even for supercomputers the human costs are rapidly succeeding hardware costs as the limiting factor in computer use, further increases in productivity may be expected to come from cognitive engineering, rather than from increases in the MIPS per dollar ratio.
Chapter 3: How to Investigate Parallelism: time and knowledge

3.1 Introduction

The science of linguistics studies the way human beings represent certain aspects of reality by means of a symbolic language, and the rules that exist within such a system. The relations between reality and language are not pre-existing, but "Each community must tacitly work out its own segmentation of reality according to whatever heuristic principles seem applicable." (Pearson, 1977, p.228). In this light, language is the mirror of mind, reflecting the way in which reality is perceived. The following chapter is about two aspects of language, how event-timing is dealt with in natural and programming languages, and how knowledge of programming languages at the mental level is to be investigated and modelled.

Looking at computer parallelism from a psycho-linguistic point of view, the concern is with the perception and description of tasks and events occurring in the course of time. Therefore, the first paragraphs below will be devoted to the similarities and differences between how the timing of events is specified by man and machine, and how this is related to the difficulty of programming.

From a cognitive-ergonomic stance, the concern is with the mental representations of parallel programming knowledge and the factors which determine the ease of forming and using these. As such, the nature of programming knowledge will be the subject of discussion in the last paragraphs, focussing on the question of the adequacy of present day models to account for this kind of knowledge.

In the following psycho-linguistically oriented section, a number of general observations, and a brief inspection of individual event-related words in the Dutch language will be used to point at the linguistic importance of the concept of time. Thereafter a more extensive look at the description of tasks and events in English, and how this ability develops in children, will be used to show the difficulty of dealing with time. These findings will be applied to understanding software and yield some conclusions about the general difficulty of programming parallel computers. This section is closed by a theoretical and empirical analysis of the nature of time-constructs, and their use in parallel computer languages.

3.2 Talking about time: psycho-linguistics

It may be noted that, from Pearson's remark above, it follows that there may be cultural differences between languages, and here, between how languages treat time. Reviewing evidence regarding temporal reference systems, Kuczaj and Boston (1982) subscribe to this point of view, but they add that these differences all seem to be restricted to conventional time systems. Regarding personal (and consequently logical) time, they argue that there are cross-cultural universals: "These hypothesized universals are: (1) the notion of temporal sequence and nonsequence (priority versus simultaneity); and (2) the notion of temporal containment (period A within period B)." (p.390). As such, a culture might have chosen to use the sun and the seasons as a time-scale, or Greenwich Mean Time, but all languages provide the means to describe events happening before, after, and at the same time as others.

Looking at individual event-related words, a short inspection of a dictionary of the Dutch language yielded a list of about 150 of these words. Two observations seem worth mentioning. First, the temporal relations as expressed by the words in the list range from very weak (eg. 'close', 'continue') to very strong (eg. 'while', 'before');
as such, the strictness of the criterion used determines how many time-related words will be found. The second and closely related observation is that many of the words do not express relations between events or between events and time directly, doing so only in terms of considerations of general knowledge of the world. For instance, if one considers that situations or events can both be each others causes and effects, then the concept of causality does express a temporal relation, a relation which will differ between efficient and final causation. In a sense, most or even all verbs could be included in counting, because often they describe actions changing states, apart from serving as the means to express temporal relations by tense. Thus, measuring by the number of time related words, it seems that the concept of time is indeed quite important.

On the other hand, time does also appear to be a fairly difficult concept to grasp. This is exemplified by the well known difficulty school children experience when learning to cope with the various tenses, especially the more exotic ones, like the progressive future and unfinished past tense. The difficulty with time is also indicated by its relatively late development in speech, like most abstract concepts. Foss and Hakes (1978, p.276) write: "... distinguishing between 'then' and 'now' requires being able to focus on the relationship between what happened earlier and the present. The ability to engage in this kind of decentering develops only slowly, over a period of years. An even more difficult task for children is dealing with relationship between two events when neither is in the present."

Several observational studies have shown that children first learn to use sentences about events in the present tense; after that, sentences relating present and past events, and finally sentences about events that lay completely outside the present (eg. Wells, 1985; Clark, 1970; Friedman, 1978, also for some critical remarks). Wells (1985) for instance, observed a significant tendency for reference to Neutral time ('now') to precede reference to either Past or Future, and a strong tendency within each of the three time divisions for reference relative to speech time to precede reference to a point in time ('at 3 o'clock'), and for the latter to precede references relative to a point in time (p.154-156). This order of development clearly mirrors the linguistic distinction between speech time (the time of the utterance), reference time (the time to which the utterance refers), and event time (the time at which the event occurred). The three times may all coincide when, for instance, describing one's own immediate experiences, but two, or even all of them may also differ from each other, like in: "John had already talked to Mary last Friday."

In English, tense is the minimum requirement for describing the occurrence of an event in time. Using tense, it is possible to specify the occurrence of an event relative to the time of speaking by means of just a single clause. Historically, there were only two tenses in English, past and present (Pearson, 1977, p.215). However, the latter should rather be called non-past or unmarked tense because it may also be used to specify future tense, like "My plane leaves on Monday" or without any temporal implications at all, like "Goats eat grass". Thus, even though the use of tense is necessary to specify the occurrence of events in time, the opposite does not hold. Using single clauses in combination with tense restricts the complexity of the temporal relations that can be expressed. Clark (1973) writes: "if the speaker uses only single clauses, the only way that he can recount a succession of events is by keeping to the chronological order of their occurrence... The same applies if the speaker uses coordinate clauses to describe a succession of events." (p.586).

To be more specific about the occurrence of events than relative to speech time, for instance, in order to specify it in conventional time or relative to other or events, tense must be used in combination with aspect, mood, and adverbials and prepositional phrases with temporal implications. Of these, aspectual modifications
(ie. continuous, perfective, etc.) are concerned with the temporal status of an event or state, rather than with its location in time (Wells, 1985, p.157). Adverbs and prepositional phrases usually form the main source of information for understanding the temporal situation of or relations between events, apart from tense.

Several findings from studies about the development of children's understanding and use of temporal descriptions yield useful results for the study of programming languages. The reason why developmental data on children is applicable to such an adult occupation like programming, is its use as a measure of cognitive difficulty. It is generally agreed that children not only use cognitively less complex utterances, but also that the order in which children learn to use language closely reflects their growing ability to cope with more and more difficult problems. As such, when a programming language employs many concepts which arise relatively late in cognitive development, the language will be more difficult to learn and to use without committing errors.

As an example of using developmental data consider, as was mentioned above, that children's understanding of time starts with the immediate experience of the present, and later on spreads to future and past, until finally temporal relations can be handled completely independent of the present. For instance, reviewing research on the development of the concept of time, Friedman (1978) writes that, 'as he calls it- experiential time appears well developed in young children, while logical and conventional time, are still poorly developed before the age of about 8, and only fully mastered and integrated during adolescence (p.294-295). From this, in combination with the reasoning about cognitive difficulty, it would follow that human performance would decrease from temporal judgement tasks which require sequential information, and use the present as point of reference, to those which involve more complex judgments about relations in the past. In this context, one might look at some purely psychological research about picture recognition.

Here, it is known that humans are rather good at judging whether pictures had been presented earlier or not (see eg. Baddeley, 1980, p.211-213), a task which involves a judgement relative to the present. Using a more or less similar task and requiring more complex temporal judgments about respectively, the order of stimulus items, their position and the lags within the presentation of stimulus items, one might expect that performance would decrease in the same order. This is exactly the result which Jackson (1986) reports; the performance on the lag and position judgement tasks is considerably lower than the performance on the temporal order task, which merely demands sequential information to be retained.

3.3 From psycho-linguistics to programming

Several important implications for programming language design follow from a classical observational study by Eve Clark (1970, reported more fully in 1973), in which she presents several detailed findings about children's description of temporal relations between events. It can be summarized as follows. When a child first begins to talk about events related in the past, they will use single clauses and describe the events in the order in which they occurred. Later on and, or and then are used to connect the clauses into more complex sentences, like: "I ate lunch and then I went to play". After this, children begin to describe simultaneous events using when. Still later, the child will use before and after to indicate the sequence of events in utterances, as in: "I ate lunch before I went to play". Finally, only after all these forms have appeared in their utterances, will children
begin to describe temporal relations, in which the events are mentioned, in an order that does not correspond to their order of occurrence, like: "Before I went to play, I ate lunch".

This order of development can be explained by considering two opposing demands on children's use of language: functionality and simplicity. To start with, there are functional reasons for choosing particular words and syntactic forms. First, words like since, before and after allow the speaker to be more specific about the relation between the events compared to and then, besides allowing the events to be mentioned in a non-chronological order. Mentioning events in a non-chronological order may be necessary because, at least in English, it is common usage to place the topic or the theme of a sentence in the front and reserve the rest of the sentence for comments on the topic or the rheme.

However, functional choices may often conflict with the limited capabilities of children. According to Clark (1973), there are two independent principles of simplicity, a child will use to choose particular words and a syntactic form to describe the temporal order of two events. "The first principle is that order of mention, the order in which two events are described by a speaker, is 'simpler' when it coincides with chronological order, that is, the order in which the events are perceived to have happened." (p.586).

The second principle is derivational simplicity, which says that single clauses, and clauses connected by a coordinate are most simple. Further, "... the simpler form of sentence with a subordinate clause is the one with the subordinate clause second, that is, to the right of the main verb." (p.589). As an example, the sentence "He came home when he was ready" has a more difficult order of mention, but is derivationally simpler then "When he was ready he came home", for which the opposite holds. In English, it is only for the subordinating conjunctions before and until that the chronological order of mention and the derivational simpler form correspond; for the majority of temporal conjunctions they differ (Clark, p.589).

Using the considerations of functionality and simplicity, Clark (1973) was able to give a theoretical account for the particular order of development of the children she observed, but here, the interest is focussed on the principles of simplicity. Reviewing several studies on Clark's principles, Flores d'Arcairs (1978b) writes that the first one, about the order of mention, has acquired an almost unequivocal support. The evidence concerning the second principle, stating that sentences with a main / subordinate clause structure would be easier to understand than sentences with a subordinate / main structure is less clear, as it is not always investigated, and if so, not always found (eg. Flores d'Arcairs, 1978c, experiment II). Nevertheless, Flores d'Arcairs (1978b, p.361) concludes that sentences with a main / subordinate structure are simpler, and that children rely most on the information of the main clause.

According to a review of word meaning acquisition by Blewitt (1982), several studies have supported Clark's finding about words which indicate simultaneity to precede words indicating succession, whereas others failed to do so, or found the opposite. In trying to explain these differences, Blewitt draws to attention to what may well serve to be the third conclusion of the previous: context. She writes that the fact that many studies do not provide a clear picture, points at the importance of the linguistic and nonlinguistic context for children's performance in word meaning studies (p.155).

Regarding the effects of the nonlinguistic context, French and Nelson (1985) criticize the high error rate and late development of the use of temporal descriptions, as is found in many comprehension studies. When they asked children to describe what happened during a familiar event, such a getting dressed or grocery shopping, they
found that the children were quite competent in producing the words. They conclude among others that the 'common' result is largely due to nature of the comprehension tasks used. Instead, they propose a theory in which the child learns to use relational terms in some well-known context, from which understanding spreads to other contexts, until finally, the meanings of the terms lose their boundness to specific contexts.

A series of experiments reported in Flores d'Arcais (1978a) shows the importance of linguistic contextual information. Here, the question was asked if, and to what extent, the understanding of the meaning of connective would depend on the information provided by the context of the sentence. Experiments in which children of various ages were required to make some kind of judgement on connectives within sentences were compared with a sorting experiment, in which the connectives occurred as isolated words. Ability to sort connectives lagged behind the ability to deal with them in the context of a sentence, that is, when many children show they are able to distinguish connectives embedded in sentences, they may not yet be able to classify them, on semantic grounds, in isolation.

The following will discuss how several of these conclusions can be used in the study of the design and understanding of programming languages, especially those dealing with multiple events. To start with, there is ample evidence that the principle of the order of mention applies to the comprehension of information in general, rather then solely to sentence comprehension.

When information is presented in an order in which is to be used, it will be comprehended most easily. If, on the other hand, the information needs to be restructured before it can be used, comprehension will generally be impaired. Consequently, it will be easier to tell if John is taller than Pete, from the sentences: "John is taller than Bill", and "Bill is taller than Pete", and harder from the sentences: "Bill is taller than Pete" and "John is taller than Bill".

Still more difficulty will be met when the relation words differ between the sentences, and for instance 'shorter' is used in one of them. In this way, sentence ordering can explain many of the difficulties in sentence comprehension in general (see eg. Foss and Hakes, 1978), and various reasoning problems (Johnson Laird, 1983).

Similarly, the order of presentation of information can be applied to understanding some of the difficulties in software comprehension. A program can be easily understood if, among other things, one can see how it executes by merely looking at the order of the statements on a listing. In a very simple computer program, this order of the statements on paper will exactly mirror the order of their execution. If one statement precedes another, it will execute before the other, and vice versa. Put it differently, there is a transitive relation between the order of any two statements on paper and in reality.

In a slightly more difficult program, there will only be a partial relation between order of presentation and order of execution. In a loop, one statement will be executed before or after the other, but only when considering a single iteration, not when taking a wider point of view. The program can be complicated still further, when the difference between order of presentation and execution is progressively enlarged by the introduction of irregular jumps, subroutine calls, and ultimately interrupts and asynchronous execution.

In the latter situation, different parts of the program, and several instances of the same part can be active at the same time. Therefore the only way to determine which statement executes before or after another is by carefully observing the program as it runs. Here, the listing can only serve as a very abstract and simplified representation of the program when it runs.
Thus, the relation between the spatial ordering of statements in a listing and their temporal ordering during execution is an important indicator of the difficulty involved in program understanding. This conclusion puts forward a problem where it concerns parallel programming. On the one hand there are urgent reasons for the parallel and possibly asynchronous execution of computer programs. On the other hand, however, the programmer will benefit most from working according to a strictly sequential programming style.

There are two approaches which seem to offer acceptable solutions to this; first, using virtual parallelism, in which the actual parallelism is hidden from the programmers view underneath a sequential language. The second approach is comparable to breaking up a large program into a number of smaller subprograms in order to facilitate reviewing what is done, rather then how it is done. Less stringent forms of this approach are programming with abstract data-types and modular programming, and a more strict form is taken in object oriented programming languages. Here, the code within each object is sequentially specified, but may execute in parallel with other objects, while the interactions between objects provide process synchronization.

Next, derivational simplicity and context effects will be discussed in turn, and it will be shown how the two are related. According to Clark (1973), subordinating conjunctions like before and after are used to avoid the contingency in the relation between the events, which is implied by the use of the coordinate conjunction and. However, which particular conjunction will be used does depend on what the speaker has chosen as the topic or the theme of the sentence. The topic comprises what there is to talk about, and as a convention, sentences start with it and continue with the comment or rheme, which is whatever there is to say about the topic. The probable aim of this convention on sentence order is to facilitate the process of communication.

Clark (1977) uses the term Given-New Contract for the implicit agreement, first to mention the Given information, which the listener is supposed to have already, so he can prepare to receive the subsequent New information. In this sense, the theme / rheme order is a convention to facilitate information exchange by providing a particular syntactic context. Although the theme / rheme order is primarily meant to be used within the context of single sentences, it may be used in a much broader sense. For example, the title, preface and content page of a book, the first paragraph of a chapter and lines of a paragraph may all be regarded as a theme, setting a semantic context by briefly summarizing what will be discussed thereafter.

Context effects exist in both natural as well as programming languages, in which a certain amount of redundancy allows information to be inferred from the context, which is otherwise not explicitly present. For example, syntactic rules and regularities in the ordering of the elements in both languages make it possible to predict which character, word or sentence should be at a certain position. At the lower levels of character and word order, the amount of context-information will probably not differ very much between programming and natural languages. With the exception of the various names (eg. Christian names, foreign places, variables, etc.), both languages have very strict, although largely implicit rules on character ordering.

Regarding higher level context, an important difference between natural and programming languages is that the latter especially deal with how things are to be done, whereas natural language deals with what is the case or to be done. The how is clearly expressed by procedural programming languages, but in declarative languages order of execution is also important, either because otherwise expressions can not be evaluated at all, or not efficiently. For example, in a language like Prolog, it is not possible to prove 'X is-mother-in-law-of Y', from 'Y is-spouse-of Z' and 'Z is-offspring-of Y', if instead of the latter rule 'X is-mother-of Z' is given. In languages like
Prolog, the *how* depends on *what* is explicitly given, because it lacks the general knowledge to see the equivalence between offspring-of and mother-of.

Concerning computer programs in general, the main and often only sources as well as destinations of semantic content are human programmers. For a programmer it makes a difference if a piece of code computing an average is named accordingly. For a computer language, there may be a distinction between integers and reals, but as long as the syntax is correct, it will just as happily eat spaghetti-styled uncommented programs, saturated with goto's and single character names, as any other. Similarly, it is up to the programmer to choose a meaningful name to serve as "topic" for a datastructure or a subprogram in order to summarize *what* it is or does.

Having established that programming languages lack a great deal of semantic content, and making the reasonable assumption -if not stating a fact- that meaning greatly facilitates understanding it follows that a suitable approach to parallel programming, and programming in general, would stress the provision of information about *what* is done to *what*. When discussing the effects of the order of the presentation of information, the conclusion reads that to take either a virtual or an object oriented approach to parallel programming would be the best thing to do. Here, the argument is in favor of the choice for an object oriented approach, as this clearly provides the best opportunities for expressing *what* is done to *what*.

To end this section, there is a small reservation to make about the conclusion. It may be noted that an object oriented approach only provides those opportunities, it does not always enforce their use. Here again, it is up the programmer to use them or not.

### 3.4 Timing in natural and programming language

In various approaches to parallel computation, the order of program execution is expressed by special token words and statements. In this situation a linguistic analysis might be useful, to find similarities and differences about the handling of event-timing between natural and programming languages. In the following, the description of events and tasks in time is discussed; first, in the logical sense and in natural language, and thereafter, in programming languages.

In the most simple situation, an event can be described as taking place *at* a certain point in time. Here, the word *at* is used when the occurrence of an event exactly coincides with a temporal point of reference. Using *at*, there is no need to find out when the event will happen, relative to the point of reference. The opposites *before* and *after* describe on which side of a reference point the event is to happen, and therefore, in addition to establishing the reference point, it is necessary to determine on which side of it the event lies. Also, *before* and *after* are less specific about the timing of something happening. When an event is to happen *after* today, it may happen tomorrow, but also the day after or next week.

In the logical sense, an event taking place *at* a certain point in time could be understood from a combination of *before* and *after*. This is because Wells (1985) reports that, in children, reference to point-time always precedes reference *relative* to point-time, both in the present, the future and in the past. One may, however describe events as happening *within* a certain interval, which may indeed be understood from a combination of *before* and *after*. In this context, Wells (1985) reports that Time Up To, Time From and especially During are the last temporal categories to emerge. To this may be added that the word *at* restricts events to specific points in time, leaving no freedom to move them,
whereas the word *within* allows the event to occur anywhere during the interval, be it tea-time, tomorrow, or next year.

Thus, in a strictly logical sense, *before* and *after* are the basic constructs to describe events in time, and in combination they form the derived constructs *at* and *within*, which differ with regard to how precisely they describe the event or task in time. In a psychological sense, *at* is the most primitive concept, when the point of time at which the event is to occur is identical to the point of reference in the description. In more complex descriptions, the identity is lost and the point of reference is used as a temporal boundary; either on one side, like in *before* and *after*, or on two sides, like in *within*.

In human speech, temporal constructs are used more loosely than the above may suggest. For instance, *after* in the sentence "*cook the tomatoes, after cooking the potatoes*", is used as a direction or suggestion. As such, there are circumstances in which it is better to interpret temporal descriptions in relation to the particular context, rather than in the strictly logical sense. For example, having forgotten the potatoes, while busy cooking the tomatoes, hardly anybody would start all over again, or completely abandon cooking the potatoes.

A more complex level of description is reached when events and tasks are described relative to each other's occurrence, rather than on a time-scale. Here, part of the complexity is due to the uncertainty of the occurrence of the other events, which may determine if and when something is to happen, or to be done, or not at all. In this context, one may not have to cook potatoes at all, after being instructed with the sentence "*cook the potatoes after I have told you to do so*". In the same sense, a task may never have to be done, if not all of the prerequisite events or tasks have been fulfilled.

Another part of the increased complexity stems from the enduring nature of the events and tasks, which may make it necessary to distinguish between beginning and end points. Consider, for instance, the sentence "*cook the potatoes, and then cook the tomatoes*". In this example, it is a matter of the interpretation exactly *when* to start cooking the tomatoes; either after having finished the potatoes, or alternatively, anywhere after having begun with them.

Both of these points are important to the parallel execution of computer programs. Consider, for instance, the difference between the simultaneous execution of heterogeneous tasks and composed tasks executing in lock-step. In the former case, the main task has started when only one of the subtasks has done so, and the point at which the main task can be regarded as finished will depend on whether all of the subtasks have finished. In the case of lock-step parallelism, the beginnings and endings of the subtasks all coincide, and the execution of the main task can be completely determined by the execution of any one subtask. This difference is quite similar to the difference between *at* and *within*, except that here, the closing boundary of *within* is determined by what actually happens.

The distinction between lock-step and boundary-based parallelism is a logical one, which does not necessarily have to exist psychologically, or in natural language. Regarding natural language, there are indeed no words to be found in a standard dictionary which clearly represent the distinction. This leaves the question, whether there are simply no words for it, or that the distinction is not natural to human thought. In case of the former, it will not be too difficult to make the distinction, but one will have to use a multiple of words in order to do so. However, if the distinction is not a natural or psychologically real one, the need to make it will be a probable and likely cause of error.
A relatively straightforward way to determine if the distinction between the parallelism of lock-step and boundary-based tasks is psychologically real, is to ask people about the meaning of words related to doing things at the same time. Suppose that people were asked to indicate if using such a word meant that the beginnings and endings of the tasks involved, were allowed to coincide (i.e. "may"), or either should coincide (i.e. "must"). If the distinction is real, then one might expect that these words are understood, either in a very strict way, in which beginnings and endings should coincide, or in the opposite category, in which beginnings and endings of two tasks may, but do not necessarily have to coincide. On the other hand, if it is not a real distinction, then one would rather expect something like a sliding scale. As a part of a study, more completely reported in chapter four, 19 people were asked about the meaning of five such words, signifying parallel execution of two tasks. The tasks were described as taking place: at the same time, simultaneously, in parallel, one while the other takes place, and one during the other. For each of the words, the subjects were asked if the begin (end) of the one task must or may coincide with the begin (end) of the other task. The results of these 10 questions are presented in table 2.

<table>
<thead>
<tr>
<th></th>
<th>beginnings:</th>
<th></th>
<th>endings:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>must</td>
<td>may</td>
<td>must</td>
<td>may</td>
</tr>
<tr>
<td>at the same time</td>
<td>19</td>
<td>0</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>simultaneous</td>
<td>17</td>
<td>2</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>parallel</td>
<td>13</td>
<td>6</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>while</td>
<td>0</td>
<td>19</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>during</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td>17</td>
</tr>
</tbody>
</table>

*Table 2: Number of subjects stating that beginnings and endings of two variously related tasks may or must coincide*

The results show two things about the words the subjects were asked about. First, none of the words is unanimously understood as a description of either the lock-step type of parallelism or the boundary-based type of parallel execution; even when there is a general agreement about the must or may of the beginning of the subtasks, the opinions about the ending points are more diverse. That is, although there is a strong statistical relation between the opinions about the beginning and ending points for each word, none of the words are unanimously understood as enforcing a must, or allowing a may on both beginning and ending points. In the second place, it is possible to detect a tendency in the direction of a sliding slope between the two extreme points between lock-step and boundary-based parallelism, with tasks taking place in parallel falling somewhere in the middle between the two extremes. This may either indicate that the meaning of doing things in parallel has not yet been firmly established in natural language, or that it has an ambiguous by itself.

To summarize the above, in a logical sense it is possible to be completely specific about the meaning of temporal constructs, but when they are applied to the real world, the interpretation is some free. In natural language, one may still be fairly specific in describing how tasks and events occur in time, although it is not always necessary nor wise to use time-words in a very strict sense. To a certain extent, there is agreement about how to describe the occurrence of events and tasks in relation to other tasks and events, but the difference between lock-step and boundary-based parallelism does not seem to be very realistic.
In programming languages for parallel processing, there are several ways to express the simultaneous execution of statements or tasks. It may be noted that it is not necessary for programming languages to use temporal words for this purpose. Furthermore, it is neither necessary to use temporal relations explicitly to express parallel program execution. About the language Sisal, one could say for example, that no parallelism is expressed at all, or alternatively that it is, but rather in terms of something like 'do for all elements' than in temporal terms. This is similar to what Hillis and Steele (1986) call "data-parallelism", in which the programmer indicates which data to process in parallel, rather than which processes to execute in parallel.

In this situation, the actual execution of a program may be guarded by specific temporal constraints, but knowledge of this is not readily available, or it is not very useful, or it is more obvious to use other then temporal models to handle parallelism, such as spatial models, or models from set-theory. Consider, for example, an implementation of a binary summation of an integer-array on a connection machine. Under usual circumstances, it is fairly easy to determine how much time is required for such a process, which may be useful for comparison with serial algorithms, but hardly for anything else.

That parallelism is not expressed by explicit timing instructions is not to say that it is no good. Seen from a psychological point of view, the ability to do something to all data seems very natural. In this respect, the 'invention' of string variables was a good first step, and if the programmer could use 'strings' of various data types, such as Sisal's streams, there are only gains to be made, especially when these can be processed in parallel. However, when the do-for-all does not free the programmer from low-level details, such as array indexes, nothing will be gained in the psychological sense.

In several approaches to parallel programming, the programmer has to be concerned with the timing of events, although it is not necessary to use time constructs explicitly. Explicit timing of processes would be exemplified by a statement like "After( process1, process2 )", but it is equally valid to implement such a temporal relation by "Wait( process1, StopSignal ) ; Run ( process2 )". The specification of temporal relations between tasks in the last way is most common, and can be characterized by the implied use of before and after. Implied timing can be found in all the notations in which synchronization is attained by the manipulation of tasks, or by using messages, signals, or shared data.

Task manipulation is a relatively clear way of expressing synchronization in programs. If one task starts or stops another, the latter will only run before or after the point of deletion or creation. At a lower level, a more efficient way of synchronizing processes is to have them wait for each other's messages, signals, etc. Here, a program, or a part of it will only execute after a particular message or signal, or access to shared data is received. The use of these implied timing methods is quite common in languages for MIMD or multiprogramming systems (eg. Ghezzi, 1985; Frederickson et.al., 1985).

Finally, the most explicit form of temporal word usage is found in the language approaches which feature a Do In Parallel construct, such as Occam (Inmos, 1984), Myrias Fortran and concurrent Pascal (Frederickson et.al., 1985, p.263). Here, a weak form of within is used in which statements may execute somewhere between the begin and end points of the block, and only after all the statements have executed may control leave the block. In Occam, instead of leaving it, control begins with a new iteration of the block, as the Do In Parallel is implemented as a do-forever loop. Because in all these cases the ending point of the block is determined by the
execution of the statements in it, they may be regarded as any ordinary statement block with an implicit \texttt{wait} at the end if it, waiting for all the signals set by the execution of each of the statements in it.

In the first part of this section, it was shown that in the human sense, not all of the logically possible temporal constructs exist, and if they do, these words are used more freely. When parallelism is not expressed in temporal terms, it may be expressed in other more humane ways, such as in terms of sets, using words like \textit{all} and \textit{except-for}. However, when program execution is specified in either an implicit or an explicit temporal manner, a difference between natural and programming languages shows up. In the latter, concern is especially with the direct synchronization of the execution of processes at certain points, whereas in natural language, concern is less with the \textit{synchronization}, as with the \textit{coordination} of tasks and events. People as intelligent beings are able to deal with such a higher notion of process coordination, but at the same time, they are probably less well able to deal with process synchronization.

It is not natural for people to deal with tasks of a more or less equal duration, as the \textit{Do In Parallel} construct presupposes. Instead, people use sentences like "While cooking the potatoes, cut the tomatoes, and keep the onions from burning". Sentences like this allow two remarks to be made about the way people handle parallelism, and express it in language. First, in putting the sentence into effect, the fact that cooking potatoes will probably take much longer than cutting tomatoes does not lead to any problems. As such, this may be taken to indicate that \textit{while}, and similarly \textit{at the same time}, \textit{simultaneously}, etc. are not understood as spending equal \textit{amounts} of time to each of the tasks, but as paying attention to the other task(s) \textbf{during} the time that attention is paid to the one task.

As a second remark, in the sentence no mention is made of either signals or events, that is, people do not seem to think in such low-level terms of 'burning union smell' and 'potatoes cooked' signals, and nor of 'three o'clock' events. These remarks may be partly understood because natural language is meant for intelligent agents, with a lot of general knowledge of the world, and not for computers. However, even to intelligent agents, instructing a computer at such a low level is difficult; amongst other reasons because it means dealing with a large amount of information in a very precise manner, for which their general knowledge of the world is of little or no use. If this was not recognized as a general fact long ago, then humans would still be programming in binary or hex. 

What is needed then, is the ability to specify process parallelism or temporal relations within programs in a more human-like manner. Here, one may think of instructions like "\texttt{After( process1, While( process2, process3 ) )}" and "\texttt{NotWhile( process1, process2 )}", which allow the programmer to specify the temporal relations at a more meaningful level, without having to bother with the implementation details, such as events and signals. As an alternative, or to bridge the gap between the present and more ideal circumstances, there is a need for at least several and better tools to assist the programmer in analyzing the time dependencies in, and general structure of programs, and furthermore, tools to measure or simulate durations and occurrences of the program's modules on the run.

3.5 Thinking about programming: cognitive psychology

Software design is the process of translating a set of task requirements or functional specifications into a concrete computer program that will perform the task. Understanding software is the opposite process to building a mental representation of the set of functions a program will perform, from its listing. Both understanding, as well as producing software involves a number of peripheral or perceptual, and central cognitive processes.
Questions concerning perceptual processes are often studied, probably because it is relatively easy, but they play only a supportive role in programming. Planning and problem solving are the central processes in human cognition, but psychology has not been able to say very much about them. Consequently, several questions may be asked about the subject and method of investigation, about which approach is most suitable, and that will be the subject matter of this chapter.

There are several reasons to pay attention to which method should be used to investigate computer programming. First, because computer programming involves many different skills at various levels of complexity, it makes sense to consider in advance which of those are the most important and worthwhile investigating. In the past, a major portion of research has been directed towards the evaluation of perceptual language features. Perceptual features are important to programming because they determine the ease by which meaningful information is expressed in, or extracted from program listings. As an example, Kesler and colleagues (1984) showed that understanding programs in which the indentation was used to highlight their functioning was better than the use of excessive indentation, or none at all.

On the other hand, not all questions are equally important and or useful. To find out if it is better to use a FOR statement or alternatively a DO statement to implement a simple loop may answer a particular question of a programming language designer, but it may not be more widely applicable. Therefore, it may be better to restrict attention to questions whose answers tell something about the underlying psychological processes and conceptual knowledge. Doing so will not necessarily render all the seemingly superficial questions obsolete, as several of these do indeed contribute to a deeper understanding of human cognition. In general, though, difficult to see how the study of perceptual features will bring much insight into the essential processes involved in dealing with software.

The second reason to consider the method of investigation in advance has to do with the idea that processes such as planning and problem solving belong to what Dennett (1984, p.1453) described as "something ... warm and cuddly". In his article, Dennett discussed the views of two opposing schools of thought in cognitive theory, which center around the question whether or not thinking is to be regarded as formal, rule based, and symbolic information processing, or alternatively, as the mysterious and elusive activity taking place somewhere in the head. Because computer programming involves both the 'tough' manipulation of formal symbols, as well as 'cuddly' planning, problem solving and probably some esthetics, the claim that knowledge of all aspects of the psychology of programming is possible, may be disputed itself. Here, it is considered to be of little use to ask if any scientific knowledge of the mental is possible or not. It does however, make sense to ask to what extent cognitive processing should be considered as symbol-manipulation, governed by formal rules, disposed of any meaning, or alternatively, if it is rather not a computational process, guided by its own activity and giving rise to meaning. Therefore, when asking the question how to attain knowledge of mental processing, it seems better to weaken one of the extreme positions described by Dennett to create a more workable contrast. In this view, the extreme positions can be described as the logic-based models of cognitive processes versus computational models, based on neural networks (Hinton, 1987).

The question under discussion is important to the study of programming because the two approaches form competing hypotheses, and programming, as an activity which involves a broad range of cognitive processes, may serve as a test-bed to decide between the two options. It is not impossible to distinguish between the true and the false, because of their paradigmatic nature (Schneider, 1987), it will eventually become necessary to
establish which is more fruitful one, maybe even by sheer voting. Consider, the following occurrence as a little anecdotal example of the contrast in everyday work. In order to see how they felt about it, P.J. Barnard presented a cognitive model of short-term memory to his colleagues, which consisted of a collection of interacting modules, forming a system whose behaviour was entirely driven by the available data (Barnard, 1985). After presenting the main lines of the model, several of his colleagues hastened to ask where he had left the control structures.

Thirdly, and finally, one of the key issues in computer programming is its development in the course of time. Whereas a reader may eventually become an excellent reader, an expert in programming will continue developing his or her expertise. This is because in computer programming, and in other disciplines dealing with abstract knowledge, such as mathematics, what may be learned next depends on what is already known, at least, to a higher degree then it does in say, reading. To stress the importance of memory in programming, one might use Schank's words (1982, p.226): "... we will really find out about memory when we begin to ask the right questions. Work focussing on static conceptions of memory asks the wrong questions". Consequently, any approach to the study of programming should consider the dynamic nature of the knowledge involved, and preferably, take it into account.

In the following, a layered model of programming knowledge will be presented, and it will be shown how it fits together with various research findings. A factor of key importance in programming is the ability of people to perceive organizing principles within the information they have to deal with, apart from the question, if any such structure seems far from obvious. Regarding programming knowledge, it is possible to distinguish three levels of abstraction at which one can organize the material.

The keywords and basic constructs of a language form the lowest level of organization. At this level, there is often considerable overlap between different notations, which will employ similar or slightly different words for the same construct, like the use of a Do-loop in one language and For-loop in another. In this context, Soloway and colleagues (1982) use the concept of implementation plans to name the language-dependent mental frames with variable slots, which represent how to use, for instance, a For-loop in Pascal.

The second level consists of what Soloway et.al. call strategic plans, which have a similar structure of frame and slots as the implementation plans, but are used to solve language-independent low level problems, such as calculating the summation of the elements of an array. Because these plans are independent of a particular language, they need implementation plans to be translated into programming code. In this setting, Wiedenbeck (1986, p.698), uses R. Brooks' term 'beacons' for the standard pieces of code, such as a swap of values, which may make it easier to get at the workings of a program, by removing the need to understand a program on a line by line basis. With the growth of programming knowledge, the elements at this level may become quite numerous and more complex, but they will keep their characteristic of being well-defined.

Well-definedness is probably not a main characteristic of the plans at the third and highest level. In comparison with the lower level programming plans, agreement among researchers about the nature of the plans at this level is somewhat lower. For example, Soloway et.al. (1982) mainly focus on novice programming, and present fairly simple and straightforward 'strategic plans', whereas Guindon and Curtis (1988) include design heuristics and time planning. There is, however, agreement at the point that high-level plans may be best regarded as possible approaches to find solutions for a particular set of problems. As such, one may find plans ranging, from the general scheme of a binary-sort to the manageable modules into which a system designer decomposes an otherwise too complex problem (Jeffries et.al., 1981).
This three level model can be used to explain several findings about programming, as well as say something about what is a good method for investigating programming. To start with, it should be noted that, unlike the popular statement saying that programming is a semantically rich area (e.g. Bhaskar and Simon, 1977, Simon, 1979, cited in Abelson, 1981, p.423), it is definitely not. It is beyond doubt that Do-loops are more abstract than real or even imaginary apples, and binary-sort has a much more shallow meaning than sorting real apples. By implication it follows, that a programmer will have bring his or her own organization into the material, starting at the lowest, syntactic level. At the level of the syntax, there is a more or less well defined language with some likeness to natural language, but with very little meaning. At the higher levels, the information structures consist of more meaningful generalizations of the lower levels. However, it is only when there is a sufficient amount of lower level knowledge present, that the higher level knowledge may develop, and the development of the former will always precede the latter.

There is indeed evidence from several studies to showing that novice programmers work principally syntax-based, and expert programmer function-based, which is generally thought to be responsible for higher performance scores. Among others, these studies include program statements recall (Abelson, 1981), program sorting and segmenting (Yu and Robertson, 1988), program debugging (Vessey, 1985), and system design (Jeffries et.al., 1981). In this context, it may also be expected that complete recall of a program is an easier task then cued recall of specific statements immediately following a statement, serving as a cue. This is because in the latter task, a great deal of the syntactic and functional context is not present, or can not be used. Wiedenbeck (1986) reports exactly the expected result for more experienced subjects. Furthermore, she mentions (p.705) that in pilot tests, novice programmers were hardly able to recall anything, unless explicitly instructed to memorize the program.

In a number of studies, results are reported which may be understood most easily from the view that programmers have to create their own organization within programming knowledge. First, there is a wide variety of higher level structures both in system design (Jeffries et.al., 1981; Guindon and Curtis, 1988) and program design and recall (Vessey, 1987). Secondly, if programming knowledge differs between individuals, then similarly, and except for the most common blunders, what made a programmer commit a bug would also differ. In this respect, Soloway and colleagues (1982) were better able to explain various errors when they used the strategic design plans they inferred from the individual programs, than when they used purely syntactic considerations. Thirdly, in the study by Vessey (1987), she was unable to show the effects of manipulating the higher level knowledge structures in the results, which is to be expected if these are highly individual. That is, if what is known at the higher levels depends on the knowledge at the levels below, then increases in the higher level knowledge will go with still greater increases in individual differences.

Returning to the contrast between controlled processing and network models, it appears that there is a principal difficulty for both approaches. The logic-based models are able to explain mental phenomena, and make predictions of the effects of the presence of specific features, but only when a structure within the stimulus material is given in advance, such as in language and various reasoning problems. Here, they may explain some of the problems the programmer faces in general, like the overall size and complexity of the rules of the language, the clarity between a construct and its meaning, and the syntactic marking of the scope of constructs. However, when at the higher and more interesting levels the general structure of programming language disappears into individually different planning structures, this type of model is no longer useful.
The network models are generally quite robust against noise and local disruptions, such as individual differences, might be conceived. However, this kind of model works best when there is a large amount of data, such as visual perception and the semantic representation of meaning, where it does not matter if for one person the word apple is only weakly related to the word pear, as long as they are for most people. Consequently, in programming, network models can explain the effects of for instance perceptual cues like indentation, ease to distinguish variables from constructs, and having more information then only syntax to discriminate among pieces of code.

At the higher levels of programming knowledge, there are only few structures which are common among programmers, besides, the relations between different functional constructs and plans is much less clear than those between the typical semantic network nodes, like apples and pears. Because of this, it will be very difficult, to make specific predictions from network models which are applicable to expert programmers in general. See for example, Schneider (1987) for a critical but positive discussion of the power of connectionist models in making predictions and explanations.

As a possible solution to overcome the difficulties in modelling programmer knowledge and behaviour, one might simplify the situation by looking only at individual, skilled and error-free behaviour. For example, Card, Moran and Newell (1983) build a general model of text-editing, and fitted it to each of their subjects by setting appropriate parameter values. In this respect however, one might ask, that, if it already necessary to adapt a model of a relatively low-level and straightforward skill like text-editing to the individual, to what extent it is possible to model a high-level and intuitive skill like programming in an adequate way, and what the usefulness of such a models will be.

To sum up, it was shown that low-level and specific language features are most easy to investigate, also the least important. Investigating the higher level characteristics is important, but difficult to deal with for either of the approaches, due to shallow nature of programming knowledge and the individual differences caused by it. Two directions may be taken if investigations into programming are to be successful. First, one might look at those factors which supposedly do not differ too much between programmers. In the following chapter, one such an attempt to gain insight using the latter approach will be described. A second, and at least on the longer term more promising direction, is the development of tools for knowledge representation. If cognitive science succeeds in its attempts to model the knowledge of the individual by itself, rather than in terms of the model based on the population, then one might ring the bells of glory.

Of course, for parallel computation there remains a popular alternative to psychological research, which can be described as 'learning the hard way'. It might however be noticed, that learning by doing was responsible for things like Roman numbers and the von Neumann architecture itself. If someone would have done a little thinking in advance, people would not have had to struggle for ages with Roman numbers. Likewise, if someone would have done a few difficult psychological experimentation, instead of designing computers and programming languages like it used to, then parallelism might already be effectively employed, much like homo-sapiens herself does.

In the preceding chapter, the scientific investigation of programming, and in particular parallel programming was discussed, taking as starting points, the development of the concept of time, the meaning of temporal words, and the difficulties of the present psychological paradigms in taking account of programmer
behaviour. The second paragraph dealt with the possibility to use psycho-linguistic evidence about the concept of time, in order to draw several useful conclusions about parallel programming. Considering the possibilities to express temporal relations between events, time should be a very important concept in natural language. However, it is also difficult to deal with, judged by its rather late development in children. It was argued that the development of understanding temporal relations in children can be used as a measure of the cognitive difficulty, adults will meet in computer programming.

The third paragraph contained a more extensive treatment of the subject, starting with a discussion of a classical study in children's development by Clark (1970; 1973). In this study, Clark showed how the development of children's description of events can be explained by the principles of the order of mention and the derivational simplicity. Other researcher have generally supported these findings, and in addition, pointed at the importance of contextual information. Using the order of mention to explain some of the difficulty involved in programming, lead to the conclusion that programmers would benefit from a sequential approach, with hidden parallelism. Also, an object oriented approach may be used to express parallel execution at the higher, more meaningful levels, where it is easier to understand what is done.

Derivational simplicity and the difference between main and subordinate clauses were shown to derive from the choice for a theme / rheme order of clauses, or the given-new contract, which provide a particular information context to facilitate message understanding. Such redundancy at the (higher) level of meaning is, unlike lower level context effects lacking in programming languages. Therefore, it should be facilitated to express meaning in computer programs, and this may be done, again, within an object oriented programming approach.

In the fourth paragraph, the meaning of individual time words was discussed. In human language, the word at is the basic concept which pinpoints events down to a particular time-slice. The words before and after provide one sided bounds, and in combination form within. When considering the description of events and tasks relative to each other, distinguishing begin from endpoints, a distinction shows up between multiple tasks in lock-step, and tasks taking place between boundaries. Subjects do not appear to make the distinction, which may be taken to indicate that it is not psychologically real one, and consequently may cause problems if programmers are to use it. Rather then adhering to the logical meaning of time words, humans use time-words less strictly and possibly more in accordance to the real world. Time-words are not always used to express parallelism in programming languages, which may either hide it, or use other means to express it, such as, by the use of parallel data structures. Languages with concern for event-timing do not have to make explicit use of temporal relations, and may instead use expressions, which are easier to understand. However, if they do, it should be done using time words, which are meaningful to the programmer, such as before and while. Presupposing tasks of equal duration and low level detail should be avoided, or alternatively, made easier to handle by appropriate tools.

In the final paragraph, the difficulties in psychological research were discussed, which are met when the focus is directly on programmer knowledge. It is important to reflect on the methodology to investigate programming, not only because there is a wide choice of subjects to investigate, which are not as valuable, but also because it is necessary to account for the dynamic nature of programming knowledge. A final reason is, that this issue is concerned with the two main paradigmatic approaches to cognitive psychology of the present, which view information processing either as self-regulating network activity, or as logic-based symbol manipulation without meaning.
The most important and interesting of the programming skills are the hardest to investigate, because the 'shallow' and dynamic nature of the knowledge involved leads to gross individual differences, which neither of the two approaches is able to handle. At short notice, it may be best to research problems with little individual variability between subjects, whereas on the longer term, the development of tools which enable the knowledge of the individual to be modelled seems most promising.
Chapter 4: An Experiment about Programming Knowledge

4.1 Introduction

This chapter contains the empirical part of the question about the difference between knowing-how and knowing-that. Knowing how to do things forms the basis of all behaviour, but almost all the psychological models of human memory developed during the past decade deal with representing semantic knowing-that information (Wolters, 1983).

Interest in the question of knowing-how was renewed with the field of human-computer interaction, where several important questions arose about the development of skill and expertise, and the difference between novice and expert problem solving. An important difference between real-life and computer tasks is that the former usually provide clues and constraints on which action to take, whereas the latter usually does not (Norman, 1986). Because of this, knowing how to do things, such as how to delete a file becomes relatively more important than knowing that, like for instance, knowing that the command 'delete' is to be used to delete things. In the following, the concept of expertise will be employed as a means of gaining insight into the difference between procedural 'knowing-how' and factual 'knowing-that'.

Experts are generally considered to have more knowledge than beginning users, whose knowledge contains blind spots, and may contain erroneous information as well. The often cited study of de Groot (1965, discussed in Baddeley, 1976, pp. 271-273) was the first of a series of experiments showing that the knowledge of experts is organized in many and larger structures, or 'chunks', than the knowledge of novices. In this study, chess masters were far more successful in reconstructing a position on a chess board after a five second exposure than were weak players. However, when the pieces did not constitute a position from a chess game, but were placed in a random order, the difference between the two classes disappeared.

Similar results have been found about memory for program code. For example, Barfield (1986) reports that expert programmers' performance did not differ from novices' when programs were presented as random lines of code, whereas presenting programs in working order or as random chunks lead to a remarkably better performance on behalf of the expert programmers. There is also a difference between experts and novices where it concerns speed. For example, Schmidt (1986) reports that when presenting a program at a line at a time, experts take less time to read than do novice programmers. And according to Wiedenbeck (1985), experts are both faster and more accurate in making simple evaluations of Fortran code fragments.

Regarding these chunking studies, the question could be asked to what extent, expertise consists of factual knowledge or knowing-that, and to what extent it consists of procedural knowledge or knowing-how. According to Vessey (1987), the representations employed in these studies, "are all basically symbolic and possess the overriding characteristic that the principal type of information that dominates them is sequencing information" (p.66).

Sequencing information, or knowing 'what to do when' is a kind of procedural knowledge which also characterizes many real-worldly kinds of expertise, like driving a car or a bicycle. Regarding expertise, Allport (1980), presents several examples of skilled subjects being able to perform two tasks at once, like playing novel and technically difficult piano pieces while shadowing English prose. It is also a common observation, however, that most beginners have difficulty with knowing 'what to do when', and often seem to take bizarre actions, using
established knowledge from outside the specific field (e.g. Rumelhart and Norman, 1981; Bonar and Soloway, 1985).

In programming, it is not clear whether the supremacy of expert performance stems from a larger and more rapidly accessible knowledge base only, or alternatively also from a qualitative difference in knowledge. Several studies seem, at least at first sight, to support the latter position. Jeffries et.al. (1981) analyzed protocol data during the production of software by programmers with various levels of skill. They report that although all programmers used similar problem decomposition methods, the less experienced tend to a more rigid top-down approach to subprogram solutions, whereas the expert does it iteratively, choosing from several alternatives. Protocols were also analyzed in a study by Vessey (1985) about expertise in debugging, and in a more extensive one by Rist (1985) in which she investigated the structure of plans in programming at large. These studies yielded a similar overall result: novice programmers take a more focal view of the programs they work on than do experts, and stick closely to the lines of code. Experts on the other hand take a more global stance, easily switching between levels of abstraction. They treat the code merely as a means to generate and test hypothesis to see if the operation of the program keeps within constraints.

However, protocol data does not provide conclusive evidence for the notion that expertise is more than a matter of the amount of knowledge, because a better know-how may also be explained by a larger know-that. The difference Jeffries et.al.(1981) mention regarding the rigid approach of novices writing computer programs while the expert considers alternatives may well be explained because the former lacks the large base of known solutions and design schemata of the latter. The same reasoning applies to program understanding by the typical novice and expert in Rist's study (1985).

Such criticism does not apply to an observational study of Kahney (1983), though it is rather limited. He required subjects to copy programs using a different pencil after each additional view of the original was taken. In general, the typical novice extracted information from the programs a line at the time, whereas more expert programmers extracted more of the structure, especially on the first viewings.

There is some experimental evidence available about similarities and differences between know-how and know-that. First, Vessey (1987) tested the hypothesis that expert programmer performance would be more affected by the lack of a match between the information structures of a program and those in long term memory than would novice performance. Subjects were required to reproduce a program from memory, using four more and less desirable structures to validate a field in a record; where 'desirable' was taken to mean suitable as a validation routine in a business environment.

The conclusion read: "Although expert programmers recalled more than novice programmers, there were no qualitative difference in the types of structures the two groups recalled" (p.65). Also, when programmers were requested to construct, rather than reproduce a validation routine, they yielded very diverse program structures, supporting the notion that there are no qualitative differences between the types of chunks experts and novices possess.

Gilmore and Green (1984) did show that programs written in a procedural or in a declarative language are not stored in a uniform 'mental language', as it was proposed by Shneidermann and Mayer (1979). Instead, they argue "that the mental representation of a program preserves some features of the original notation" (p.31), by which they mean the accessibility of the how or the that information of a program. Subjects were required to answer
sequential 'what if' and circumstantial 'what is' questions from a printed or a recalled computer program, which was written in a procedural or in a declarative language. Performance was best on matched pairs of questions and notation, and when information in the program was highlighted by cues, performance on recall of unmatched pairs of questions and notation improved.

Returning to the notion of expertise, during program comprehension, there are two points where programmers might benefit from a special knowledge of how to do things. The first is during program comprehension, when information is extracted from a program. One way to do so is to understand each line of code in sequence and continue until the complete program is understood. A more advanced way would be to adopt a strategy or heuristic like looking at the information provided by indentation, variable names etc..

In the second place, during program reconstruction, know-how might aid in fitting the recalled pieces together in a meaningful way. In this context, the protocol data by Jeffries et.al. (1981) and Rist (1985) might be taken to indicate that program production is probably guided by programming strategies, and rules on 'good programming'. At this stage, it is also possible to apply erroneous know-how, like Anderson et.al. (1985) report about novice programmers, who rely on the well-formedness of results to decide whether a program is correct.

4.2 Introduction to the experiment

For many well-studied skills, particularly those in everyday life like using a typewriter or driving a car, there is vast evidence for the presence of a know-how factor. Many of these skills either preclude a know-that explanation, or either paying attention to the factual side of knowledge may disrupt skilled behaviour. In this context, one might consider driving a car or using a typewriter as typical examples of the classical kind of skill. Two main features distinguish these skills from the ones involved in cognitive task like programming. First, although these skills may be as complex as a skill like programming, to a certain extent, they lack a cognitive component. That is, skills like these require no more than a direct mapping between the input and output systems, whereas cognitive skills for the greater part take place inside the head, and have rather weak or indirect relations with the outside. As such, for instance type-writing can be learned in a foremost procedural way; by doing and guiding the action by the results of it, the skill may become automated. The major part of programming can not be learned in this way, as Wiedenbeck (1985) states: "Although the expert programmer may achieve a degree of automation in doing simple tasks, programming in the full sense of problem solving is not something which becomes automated. Only the most stereotypical and routine subtasks are likely to be automated" (p.388).

In the second place, a cognitive skill like programming is concerned with highly symbolic information, whereas everyday skills deal with real objects which provide more as well as more redundant information. That is, objects in reality provide more information about what they are (for), how they are to be used and how they related to each other than do program fragments. Norman (1986) remarked that many tasks in everyday life require only a minimum of learning, problem solving, and planning because natural and contrived properties of the environment combine to constrain the set of possible actions. It would be of interest to see if subjects perform better at the same task using either realistic or highly symbolic objects.

In this chapter, two task domains will be studied: cooking and programming. These tasks are both relatively complex, requiring factual as well as procedural knowledge but cannot be learned in a strictly procedural manner. That is, knowledge of results is not instantaneously available, nor can be directly used to guide further action as applies to many everyday skills. Rather, such knowledge is only available after an interval
comparable for both tasks of at least several minutes, and it does require a certain amount of evaluation. However, the tasks do differ with respect to the nature of their 'ingredients'. Whereas cooking uses meaningful objects from real life, programming is based on symbolic languages and fairly abstract ideas.

Using cooking and programming as task domains, the next step is to choose a task that does not cause any domain bias. For example, the use of (e.g. verbal) observational data would not be very sensible to compare performance between these domains, because cooking and programming differ with respect to how well they can be acted out or verbalized. Therefore, memory for textual information will be used. In accordance with the earlier mentioned criticism of the chunking studies, that the use of single lines or fragments of code is not a good way to test memory, these texts will be presented in a psychologically realistic form; as a recipe for a complete meal and a runnable computer program.

Among others, the language used in cooking differs from a programming language in the amount of text, and the amount of syntactic and semantic redundancy. This could easily lead to a different understanding of recipes and programs, because of a different way of reading, and not due to the different nature of the materials. In order to prevent reading strategies or know-how to occur on the perceptual side, and to make it equally difficult to understand each of the versions of the recipe and program, the texts were presented with an equal proportion of words and tokens exchanged by dots. Presenting a text as a cloze test will also yield the number of filled-in words as a measure of text understanding.

In order to make it more and less easy to extract the procedural or know-how and the factual or know-that information from the text, the presentation of the text is varied, comparable to the Gilmore and Green (1984) study. Here, the interest is in the type of information rather than the type of language and therefore the same language will be used with different orderings of sentences.

A standard recipe or a neat computer program convey the reader both procedural and factual information, and from these the factual information probably somewhat better that the procedural information. Procedural information can be made more easily accessible at the cost of access to factual information, if the instructions are presented in an order such that they would be executed in the most efficient way. In this way, a version of the recipe reflected the order a good cook would adopt and one version of the program was written, trading the clarity of separate modules for rapid execution.

Procedural and factual information can both be made less accessible if the instructions are presented in such a sequence that if effectuated would result in the least efficient execution. In this way, a most disorderly recipe was made to serve as a base-line condition; a most disorderly version of the program was dropped in the due course because it would be very similar to the 'rapid' program, or contain errors.

Comparable with the study by Gilmore and Green (1984), different types of questions will be used to test the comprehension of factual 'that' and procedural 'how' information. In order to test for differences in factual knowledge, subjects were requested to indicate if a sentence preceded or followed another sentence in the text. It should be noted that answering this type of question only requires knowledge of the text; not of the task described in the text.

To answer such a question, three operations are necessary. First, each sentence in the question has to be located in the mental representation of the text. Then the relative locations have to be compared on ordering, and the appropriate response generated. It is hypothesized that better access to the factual information will facilitate performance, while worse performance is expected in the opposite case.
Expectations about the effect of the accessibility of procedural information are mixed. Following the studies of Shneiderman and Mayer (1979), and Gilmore and Green (1984), if people build up a model of the task instead of only a mental picture of the text, accessibility to procedural information will have an inverse effect on performance. That is, if procedural information parallels the ordering of sentences in the text performance will be facilitated, and if the information is differently ordered it will cause interference and worse performance. If, on the other hand people only create a mental picture of the text and no model of the task, than the accessibility of procedural information will have no effect at all.

To test for procedural knowledge, or the insight into how the task described in the text is actually carried out, questions were made with the same sentences as in the 'factual' questions, requiring some amount of mental problem solving in the form of 'what if'. For the recipes, subjects were asked to indicate if the ordering of the sentences in the question was efficient or not, in terms of the necessary time to cook the meal. For the programs, the question was, if for some reason or other the two sentences within the program were exchanged, the code would need few or gross changes to reestablish its proper function.

In order to answer these questions, a mental model of the structure of the task is a preliminary. Next, the consequences of exchanging the subtasks referred to by the two sentences will have to be established, using the given criterion of cooking efficiency or gross change of program code. Only then can an appropriate response be generated. It is hypothesized that better access to the procedural information will facilitate performance, while worse access will degrade performance. Better access to 'what-is' information will also yield better performance, because factual knowledge can be expected to facilitate building up a model of the task, but only so far as the missing words method is not perfect.

4.3.1 Method: Stimulus materials

The stimulus material consisted of 6 or 7 pages of instructions, text and questions. The first page served to introduce the experiment and the procedures to the subjects. It briefly explained the goal of the experiment, and gave some directions about how to fill in the words and answer the questions, how much time would be allowed, and not to consult the texts when answering questions about them. As the procedure changed in the course of the experiment from testing in groups to individual testing, timing directions were changed accordingly.

The text containing the recipe for a meal was adopted from a vegetarian cookery book (van Essen, 1976, pg.78-79), and described how to cook Moussakas, Greek Salad and Danish Custard. The computer program, which described how to merge two files with sorted and unsorted financial data into a third file, was designed by the experimenter.

There were three versions of the recipe, one, referred to as apart, presented the instructions for cooking each of the three courses in a separate paragraph. In each paragraph, the instructions were ordered in a most time-effective way for preparing the single course. In this version, the recipe was presented in the way usually found in cookery books and, apart from minor variation, similar to the original source text. The second and the third texts listed the recipe as a whole, without separate paragraphs for each course. The instructions in the second text, referred to as realistic, were ordered in a most effective way, in terms of the necessary time, for the preparation of the recipe as a whole. The third version of the text, referred to as disorder, described the most time-consuming way of preparing the recipe, though within the constraints of the temporal logic of cooking. Excluding numbers and title, the recipes consisted of 509 and 510 words and 44, 46, and 47 lines of text.
Two versions of the computer program were used, mimicking the *apart* and *realistic* versions of the recipe. It was not possible to produce a *disorder* program to do the transaction processing without committing logical errors. The code of the *apart* program text was grouped in code blocks, each performing a subprocess (e.g. sorting input, comparing old- and mutation records). In the *realistic* program, these code blocks were merged so far as possible in order to save source code, and theoretically speed. Excluding numbers and single character tokens, such as `='; the programs consisted of 279 and 247 words and 72 and 55 lines of code and comments.

Each of the texts was presented on a separate page, except for the *apart* computer program which took a little more than one page. The texts were presented as cloze tasks in order to obtain a measure of the subjects' ability to deal with the material, and secondly to prevent large individual differences caused by differences in manner and level of processing. With the exception of the titles, every fourth word was turned into as many dots as there were letters in it. Numbers and single character tokens were skipped. For the cooking texts, this resulted in 127 words to be filled in. In each of the program texts, five lines meant for commentary were restored to their undotted form. For the *realistic* program, this resulted in 45 words to be filled in, and in 55 for the *apart* program. In order to prevent the loss of information caused by the dotting to influence results, two versions were made for each of the texts using a different first word to start the dotting. Due to the small number of subjects it was not possible to take this factor into account, and it was subsequently dropped from further analysis.

**4.3.2 Method: Questionnaires**

Three questionnaires were used, two about the contents of the texts and a third to determine a number of background variables. In the cooking questionnaire, two briefly described cooking instructions were related with each other. First, it was asked if one instruction (e.g. "boil 2 eggs hard") was presented "Before / After" another instruction in the text. Next, it was asked if the order of instructions as they were presented in the question was "Indeed / Not" an efficient order so as to "spend least time possible cooking".

In the program questionnaire, the question posed was whether one line of code preceded or followed another in the program, and if the order of lines in the program was essential to the workings of the program. Essential meant that with a reversed order, the structure of the program would have to be subjected to many changes. The questionnaires started with an example and an incentive to work though answering all the questions; even in case of doubt. In every other question, the response categories "Before / After" and "Indeed / Not" were reversed.

The third questionnaire started with 5 questions about the meaning of temporal concepts relating to the parallel execution of tasks. The question was, if one task takes place: *at the same time as, simultaneously with, in parallel with, while, and during* another task, whether the beginnings and endings of these tasks *must or may* coincide. Next, 5 open questions asked for age, years of experience in cooking and information science, and which computer languages were mastered excellently, reasonably and somewhat. The following 7 questions asked for a rating on a 5 point scale, of the difficulty of the texts and questions about them, the regularity of cooking, and the neatness and efficiency of the computer program. A final question asked if the subject would be available for a future experiment about the evaluation of parallel programming texts.

All the texts and questions were produced on a laser printer. The written instructions are included as appendix A, the stimulus texts as appendix B and the questionnaires as appendix C.
4.3.3 Method: Subjects

Twenty-one subjects participated in the experiment on a voluntary basis. Of these, 12 subjects were paid fl.7.50 for their co-operation; 4 students were recruited from the Psychological Institute and tested as a group, and another 6 students were recruited at the Mathematical Institute and tested group-wise as well, and 2 students were individually approached and tested. A further 9 subjects, completed the questionnaires on a voluntary basis, and without other supervision than the directions on the first page. Of these, 1 unemployed and 8 professionally employed at the university automation department.

Subjects' ages were evenly distributed between 19 to 37 years of age (mean: 27.1). Experience in cooking on a regularly basis ranged from 0 to 15 years (mean: 5.7), and two subjects cooking seldom. Experience in information science ranged from less than 1 to 18 years of study and/or employment (mean: 4.0), and from 1 to 6 mastered programming languages (mean: 3.5). Including disputable entries, like: SPSS, ADS and DBaseIII, 18 different languages were mentioned. Programming data for one subject was excluded from analysis, because of failing to meet the criterion of having mastered at least one computer language. Two subjects failed to produce answers to the program questions, and were partially excluded.

4.3.4 Method: Procedure

For the subjects tested individually or in groups, a session started with a brief explanation of the procedure, after which they were requested to read the first page of the questionnaire, containing essentially the same information. It was explained that the purpose of the experiment was to compare different texts, rather than the subjects' ability, and that the main interest was in the answers to the questions about the texts, with the number of completed words being of lesser relevance. Questions about the procedure were answered, but questions about the purpose of the experiment were postponed until after the session.

Next, the subjects were asked to turn to the next page and start filling in the missing words, and to start answering the questions about the text as soon they had all the words completed, or when, after some 13 minutes they were told to start answering the questions. If necessary, then after another 7 minutes the subjects were told to hurry-up with the questions. This procedure was repeated for the second text, which was followed by 5 or more minutes for answering the background questions and providing comments. When the subjects had finished, after about 45 minutes, they were debriefed, thanked for their participation and paid off. The procedure and timings worked out quite satisfactory.

For the subjects completing the questionnaires on their own, additional timing instructions were included on the first page, asking them to work through, to spend maximally a quarter of an hour completing the words, about 7 minutes answering the text questions, and as much time as they would want on commentary and background questions.

4.4.1 Results: cloze tests and text questions

This section presents the results of the cloze tests, and the answers to the questions about the texts. Table 3 presents the results of the cloze tests. For each of the texts, the average proportions are given of missing words,
correct and incorrect words. A word was considered correct if it matched the text, and if it was consistent with a proper outcome of following the instructions; either an edible meal or a correct set of transactions.

<table>
<thead>
<tr>
<th></th>
<th>Missing</th>
<th>Correct</th>
<th>Incorrect</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipe:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.170</td>
<td>.826</td>
<td>.004</td>
<td>(7 x 127)</td>
</tr>
<tr>
<td>realistic</td>
<td>.147</td>
<td>.843</td>
<td>.010</td>
<td>(7 x 127)</td>
</tr>
<tr>
<td>disorder</td>
<td>.152</td>
<td>.838</td>
<td>.010</td>
<td>(7 x 127)</td>
</tr>
<tr>
<td>Program:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.191</td>
<td>.752</td>
<td>.057</td>
<td>(10 x 55)</td>
</tr>
<tr>
<td>realistic</td>
<td>.126</td>
<td>.794</td>
<td>.080</td>
<td>(10 x 45)</td>
</tr>
</tbody>
</table>

**Table 3**: average proportions of filled in words in brackets: the number of observations

Because of the low proportion of errors, further analysis on these results was carried out only on the number of correct words, without the need to analyze incorrect words. Analyses of variance were carried out on the numbers of correct words. Where needed, the numbers of correct words were proportionally adjusted per subject between texts with unequal numbers of words to fill in. This is essentially similar to running analyses of variance on proportions while avoiding gross rounding errors. The difference of the number of correct words between the recipes was not significant ($F(2,18) = .020$, ns.), neither was it between the programs ($F(1,18) = .161$, ns.), nor between recipes and programs ($F(1,20) = 1.813$, ns.).

<table>
<thead>
<tr>
<th></th>
<th>Correct</th>
<th>Incorrect</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipe:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.690</td>
<td>.310</td>
<td>(7 x 18)</td>
</tr>
<tr>
<td>realistic</td>
<td>.690</td>
<td>.310</td>
<td>(7 x 18)</td>
</tr>
<tr>
<td>disorder</td>
<td>.794</td>
<td>.206</td>
<td>(7 x 18)</td>
</tr>
<tr>
<td>Program:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.643</td>
<td>.357</td>
<td>(9 x 14)</td>
</tr>
<tr>
<td>realistic</td>
<td>.714</td>
<td>.286</td>
<td>(9 x 14)</td>
</tr>
</tbody>
</table>

**Table 4**: average proportions of answers to factual questions in brackets: the number of observations

Table 4 presents the average proportion of correctly and incorrectly answered factual questions for each of the texts. Four missing answers to questions about the disorderly recipe were estimated by taking the majority of the treatment group's answer to the question.

Analysis of variance did not show a significant difference in the number of correctly answered factual questions between the recipes ($F(2,18) = 1.179$, ns.), neither between the programs ($F(1,16) = 1.286$, ns.), nor
between recipes and programs (F(1,20) = 1.384, ns.). Whereas determining the correctness of the procedural questions about the computer programs is a straightforward and relatively easy task, the determination of the correctness of the procedural questions about the recipes created certain difficulties. Two judges, employing either a time-requirements analysis, or using their extensive cooking experience, reached only a moderate level of inter-judge agreement. Therefore, it was decided to use the answer of the overall majority of subjects as the preferable one, except for those questions for which there existed strong reasons for one of the answer categories; either as physical or as efficiency requirements. For 8 out of 18 questions, the answer of the majority was used as a criterion. Table 5 presents the average proportion of correct and incorrect answers to the procedural questions for each of the texts. Three answers to questions about the disorderly recipe were missing and estimated using the majority of the treatment group's answers.

<table>
<thead>
<tr>
<th></th>
<th>Correct</th>
<th>Incorrect</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recipe:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.540</td>
<td>.460</td>
<td>(7 x 18)</td>
</tr>
<tr>
<td>realistic</td>
<td>.619</td>
<td>.381</td>
<td>(7 x 18)</td>
</tr>
<tr>
<td>disorder</td>
<td>.603</td>
<td>.397</td>
<td>(7 x 18)</td>
</tr>
<tr>
<td><strong>Program:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.611</td>
<td>.389</td>
<td>(9 x 14)</td>
</tr>
<tr>
<td>realistic</td>
<td>.635</td>
<td>.365</td>
<td>(9 x 14)</td>
</tr>
</tbody>
</table>

Table 5: average proportions of answers to procedural questions
in brackets: the number of observations

Analyses of variance showed that the number of correct answers to procedural questions did not differ between the recipes (F(2,18) = .660, ns.), neither did it differ between the program texts (F(1,18) = .242, ns.), nor between recipes and programs (F(1,20) = .924, ns.).

Analyses of variance carried out on both factual and procedural questions showed that the number of correct factual answers (mean proportion = .702) was significantly larger than the number of correct procedural answers (mean proportion = .605, F(1,20) = 11.941, p < .01). This must be attributed to the difference between procedural and factual answers for the recipes (mean proportion: factual = .725, procedural = .587, F(1,18) = 9.881, p < .01), because for the programs the difference failed to reach significance (mean proportion: factual = .679, procedural = .623, F(1,16) = 2.761, ns.). It should be noted, however, that a somewhat different procedure was used to establish the correctness of the procedural questions about the cooking texts.

One of the features varying from one question to another was the physical distance between the two sentences of the question, within the text. For each text and question type, correlation coefficients were calculated between the number of correct answers and the distance between the two sentences of the question. These results are presented in table 6.
### Table 6: Correlations between the number of correct answers and the physical distance between the sentences in the questions in brackets: the number of observations

<table>
<thead>
<tr>
<th></th>
<th>Factual</th>
<th>Procedural</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipe:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>.615</td>
<td>-.061</td>
<td>(18)</td>
</tr>
<tr>
<td>reality</td>
<td>.343</td>
<td>.265</td>
<td>(18)</td>
</tr>
<tr>
<td>disorder</td>
<td>.393</td>
<td>-.435</td>
<td>(18)</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apart</td>
<td>-.065</td>
<td>-.175</td>
<td>(14)</td>
</tr>
<tr>
<td>reality</td>
<td>.103</td>
<td>.338</td>
<td>(14)</td>
</tr>
</tbody>
</table>

### 4.4.2 Results: experience and opinions

This section presents the results of the questions in the third questionnaire, about experience in the respective domains, opinions about the texts and questions, and the meanings of 5 words signifying temporal relations between events. In order to determine the importance of experience in cooking and programming on performance, the third questionnaire contained 4 questions about this experience.

Experience in cooking was measured as the number of years, ranging from 0 to 15 years (M = 5.71, S = 4.41), and as the regularity of cooking, rated on a 5 point scale (eg. daily, almost daily, ... seldom) (M = 2.67, S = 1.25). Experience in programming was measured as the number of years of study and/or employment in information science (M = 3.95, S = 4.21), and the number of languages known; excellently, reasonably or somewhat (M = 3.45, S = 1.50). The number of known languages as a measure of experience because the application of weights according to the extent of this knowledge did not result in better fitting data.

Correlation coefficients were calculated between these measures of experience and performance on the texts. The languages included: ADS, ALGOL, APL, Assembler, BASIC, C, COBOL, DBASE, EASYTRIEVE, FORTRAN, LISP, MODULA, PASCAL, PL/1, SIMULA, SMALLTALK, SPSS and SQL. Table 7 presents the correlation coefficients between cooking and programming experience and the numbers of correctly filled in words, and correctly answered factual and procedural questions within the same domain.

<table>
<thead>
<tr>
<th></th>
<th>Filled-in Factual</th>
<th>Procedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooking:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>years</td>
<td>.381 (21)</td>
<td>.089 (21)</td>
</tr>
<tr>
<td>regularity</td>
<td>-.398 (21)</td>
<td>-.396 (21)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Filled-in Factual</th>
<th>Procedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>years</td>
<td>.536 (19)</td>
<td>-.021 (18)</td>
</tr>
<tr>
<td>languages</td>
<td>.310 (20)</td>
<td>.266 (18)</td>
</tr>
</tbody>
</table>

### Table 7: Correlations between experience and correctly filled in words, and correct factual and procedural answers in brackets: the number of observations
Six questions were used to collect the subjects' opinions about the stimulus materials, each to be rated on a 5 point rating scale (e.g. very difficult, difficult, ... very easy; and scored as 1, 2, ... 5). Opinions were asked about the difficulty of each of the texts, and the associated questions. For the programming texts, there were two additional questions about the efficiency and neatness of the program. Table 8 presents these results.

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>difficulty of the cooking text: questions:</td>
<td>2.810</td>
<td>1.006</td>
<td>(21)</td>
</tr>
<tr>
<td>difficulty of the program text: questions:</td>
<td>2.600</td>
<td>0.735</td>
<td>(20)</td>
</tr>
<tr>
<td>efficiency of the program:</td>
<td>2.778</td>
<td>0.916</td>
<td>(18)</td>
</tr>
<tr>
<td>neatness of the program:</td>
<td>2.895</td>
<td>0.912</td>
<td>(19)</td>
</tr>
</tbody>
</table>

**Table 8**: Mean (M), Standard Deviation (S) and the number of subjective opinions (N) about the stimulus materials

Lower scores indicate more affirmative opinions

All of the correlation coefficients, calculated in each of the domains, between the opinions and the numbers of correctly filled in words, and correctly answered factual and procedural questions, failed to reach a statistically significant level and are not mentioned.

Five questions asked about the meaning of the temporal constructs relating two tasks, were stated (in Dutch) as follows:

if given: tasks a and B must take place **simultaneously**

than the begin of task A **Must** / **May** coincide with the begin of task B

than the end of task A **Must** / **May** coincide with the end of task B

In 4 similar questions, **simultaneously** was exchanged by **in parallel, at the same time, while, and during**.

<table>
<thead>
<tr>
<th>Beginnings: must may</th>
<th>Endings: must may</th>
</tr>
</thead>
<tbody>
<tr>
<td>at same time 19 0</td>
<td>15 4</td>
</tr>
<tr>
<td>simultaneous 17 2</td>
<td>13 6</td>
</tr>
<tr>
<td>parallel 13 6</td>
<td>9 10</td>
</tr>
<tr>
<td>while 0 19</td>
<td>3 16</td>
</tr>
<tr>
<td>during 0 19</td>
<td>2 17</td>
</tr>
</tbody>
</table>

**Table 9**: number of subjects stating that beginnings and endings of two variously related tasks may or must coincide.
In order to be complete, the results of these questions which have been presented before in paragraph 3.4, are presented again in Table 9, as the number of subjects having chosen either may or must, for the beginnings and endings of the tasks, and for each of the concepts. The correlation of the number of must (may) responses between the beginnings and endings of tasks is very strong (Rxy = .988; t(3) = 10.96, p < .001). This allows for the responses of beginnings and endings to be added together, which makes it easier to apply a Chi-square to test for different response patterns between concepts.

The difference between at the same time and simultaneous is not significant (Chi(1) = 1.583, ns.). Although the Chi-square may not be used to test for a difference between within and during (Siegel, 1956, pp. 110), inspection of the data shows that the difference will inevitably not be significant. The response patterns between parallel and simultaneous (Chi(1) = 3.897, p < .05), and between parallel and while (Chi(1) = 21.518, p < .001) are significantly different, and so will be all the non-adjacent concepts. Finally, there is a strong correlation between the order of presentation of the concepts in table 9 (eg. at the same time = 1, simultaneous = 2, etc.) and the numbers of must (may) responses for both the beginnings and endings (Rxy = .930; t(8) = 7.147, p < .001).

4.5 Discussion

In this study, subjects answered factual and procedural questions about a recipe and a computer program, which were presented as cloze tests. Apart from the respective task-domains, each of these texts had a similar instructional or functional content, but differed in the way the information was presented; in particular with respect to the ease with which the factual and procedural information could be obtained. It was hypothesized that the difference in relative accessibility would lead to certain differences in the performance of the respective groups of subjects. These differences concerned the number of words filled in, the number of correct answers to specific procedural and factual questions, and to these types of questions in general, and finally the total number of correct answers. However, no such differences were found and very few effects were found at all, which drastically limits the points left for discussion.

First, in the absence of clear effects, much data does not have any immediate interesting practical or theoretical implications. As such, all the questions of the third questionnaire about experience and subjective opinions are left in the results-section for inspection. Secondly, the results about the meaning words, which signify temporal relations between tasks were already discussed in paragraph 3.4. In the third place, even in the absence of overall effects, it might have been interesting to look at performance differences on specific questions and words to be filled in, if these could be interpreted in some meaningful way. A number of such analyses were carried out, but none of these yielded anything interesting.

In the following, the discussion will be restricted to the next points; first, several methodological factors which may have been partially responsible for a flattening out of the effects, and secondly, the difference between the number of correct procedural and factual questions, which may be taken as a genuine difference between the recipes, but not between the computer programs. As such, it has implications regarding the nature of the knowledge used in programming and the real-world knowledge of cooking.

In the stage of planning this experiment, a small pilot experiment was run, which yielded the expectation that a number of about 48 subjects would suffice in order to obtain clear results. However, only 21 subjects could be found, from rather diverse sources, which is the unlucky consequence of seeking students for volunteers during
a term holiday. Both a small sample size, as well as the fact that the subjects did not come from a homogeneous population make it less likely to find genuine differences between means (Hays, 1973, p.360-361), while the latter may also invalidate the assumption of equal variances between groups, which underlies the use of analysis of variance.

A second factor which may explain the established pattern of results is the application of *know-how* by the subjects. This seems a bit paradoxical, because the use of cloze tests in combination with time constraints was meant to prevent the occurrence of perceptual strategies, and to avoid large individual differences in the amount of information extracted from the stimulus texts. However, within these constraints it appeared that the subjects had enough freedom to choose their own strategy in fulfilling the task. In terms of a model like the GOMS analysis (Card, et.al., 1983), such a task strategy is a kind of a higher level strategy, because it does not merely concern *knowing-how* to choose a method to establish a pre-given goal, like a perceptual strategy on how to get as much relevant information from a text. Instead, a task strategy consists of *how* to choose which of the optional subgoals is easiest to solve. Here, the fact that the proportion of incorrect words was considerably lower than the number of words not filled-in makes it reasonable to assume that the subjects found it more important to produce correct words, rather than as many words as possible.

In order to produce correct results, it is easiest to pick the words whose correctness can be locally established, without the need to join information from different parts of the text, resulting in a more superficial understanding of the stimulus texts. A similar effect may have evolved from merely being told that questions would follow to indicate how well certain parts of the texts would be remembered, whereas the subjects in other studies were explicitly instructed to study the stimulus material for recall (eg. Abelson, 1981; Wiedenbeck, 1986, experiment I).

At this point, it may be noted that although there is not much to conclude about the results of the experiment, there are several indications of principal differences between programming and natural languages. First, in looking over the filled-in words, it was decided to consider a word correct if it was consistent with the rest of the text, and the overall goal of the instructions. For the recipes, this has slightly increased the number of correct, and decreased the number of incorrect words. Alternatively, it might have been decided to consider a word correct if it meant about the same as the original word. However, the latter strategy would negatively bias the recipe, because due to the nature of natural language, for each word there are almost always several suitable replacements words. The fact that this was hardly ever possible in a programming text does not necessarily indicate that programming languages are more strict on the allowable tokens, but only that they are when the instructions are part of a meaningful whole; a program. In natural language, sets of instructions like a recipe, allow a larger amount of freedom in choosing the actual wording.

A similar reasoning applies to the determination of the correctness of the answers to the procedural question. It may be recalled that for the recipes, this correctness was determined by the majority of answers, except when a particular answer was self-evident. This procedure was followed, because it was not always clear which answer to consider correct, and at least for some questions, more than one answer appeared reasonable. A similar procedure could have been followed in the analysis of the procedural questions about the programs, which would have resulted in a higher proportion of correct answers, although all these answers were evidently erroneous. Both the structure of the transaction processing task and the need to provide explicit instructions on
'what to do when' largely limit the freedom to choose a solution schema. A real worldly task like cooking does not seem to impose such a very strict solution schema, also due to the intelligent nature of the human agent.

With regard to the difference between the numbers of correctly answered procedural and factual questions, it was hypothesized that the apart versions of the texts would facilitate answering factual questions, because in comparison to the other text versions, the factual information was easier to access. Similarly, the realistic text-versions presented the procedural information most clearly, and were hypothesized to facilitate answering the procedural questions. The proportion of correctly answered factual and procedural questions did not differ for the program texts, but it did for the recipes. It was expected that the procedural questions would yield a lower proportion of correct, because in comparison to answering factual questions, answering these requires additional mental operations and memory capacity.

The fact that the disadvantage of the procedural questions was only found for the recipes and not for the programs is not simply caused by the different way of establishing the correctness of the questions, but it indicates a difference between real-world and symbolic knowledge. This is because, when only those answers are considered about which the judges did agree, than the correct procedural questions (mean proportion = .571) is still quite lower than the equivalent factual questions (mean proportion = .743). Also, this difference is three times as big as the difference between the correct procedural and factual questions about the programs (mean proportions, respectively .623 and .679).

When one considers that the same amount of information is presented by the comparison lines in both the factual and in the procedural questions, than this difference can be interpreted as the measure of performance deterioration caused by the additional memory load and mental operations. It is not possible to account for the observed difference as the result of a different usefulness of the comparison lines in the question as cues to the information in memory, because if this had been the case, then the difference between the procedural and factual answers would still have been similar between programs and recipes.

The conclusion should read that it is easier to reason about programs than it is to reason about recipes. However, why this is so requires some speculative thought. It may be that answering questions about programs is a relatively flat task which may be done at the level of the mentally represented text. That is, regardless of whether it concerns factual or procedural ones, answering questions about a program may only involve looking and manipulating a mental listing, without the need to create an additional semantic representation. As was noted earlier, the relations between the words of a program is rather strict, which will aid building and maintaining a mental listing.

On the other hand, answering questions about a cooking recipe could involve mental representations of both the text and the semantics, with different processes to answering procedural and factual questions. Building a semantic representation of a recipe may be facilitated by the meaningful nature of the material, and it will be needed more, because the text has both less strictly related words, as well as more words. In this case, the presence of the higher level semantic representation will support and facilitate answering factual questions, but it will be the only representation used in answering procedural questions.

To conclude this section, the best thing to do is to investigate further the question of knowledge representations for real-world tasks like cooking, and symbolic tasks like programming, before accepting speculation for truth.
5.1 Summary

This study dealt with programming parallel computers, a subject which is closely related to computers architecture and to human cognition. The architecture is important because it provides the playroom of computerized problem solving, and human cognition is important because it determines the ease, with which the available playroom can be used for problem solving. Up to the present day, computerized problem solving has been governed by the von Neumann bottleneck, which demanded that problems should be molded into serial processes, making them more and less easy to deal with for humans.

Now that various attempts are made to overcome the bottleneck problems by the introduction of parallel computers, the problems will have to be fit into different mouldings. Consequently, different demands will be placed on human cognition, which may be a good thing, but equally possibly it may be a bad one. Because the success of computerized problem solving depends for the greater part on human beings, it is important to look at the consequences of the technical changes for human cognition. This should be done in advance, if one is willing to maintain the view that humans are able to learn from their previous mistakes. A small attempt in that direction was taken in the previous chapters, and attention was paid to various aspects of human cognition, computer architecture, and the place where they meet: the programming language.

In chapter 1, the difficulties the human as a programmer meets were discussed. Many of these problems arise from the use of languages which were designed largely on intuitive grounds, to reach a certain functionality, or to make effective use of the available hardware. If more attention was paid to the architecture of human cognition, instead of the computer's, at least some of these problems could have been avoided. This is especially important as these languages keep on being used, because of the historical imperative. The development and growing use of parallel computers again provides the opportunity either to repeat the mistakes made, or do things right from the start. Therefore, it was argued that psychological research into programming is necessary as it may provide the insights to enable optimum use of the available hardware and in the long term save sums of money, which would be wasted otherwise.

Chapter 2 dealt with parallel computation. Several reasons were mentioned, supporting the notion that sequential computer architecture has reached its height and is about to begin the way down, while it is beyond doubt that the future lies in parallelism. The reason connected to the economics of computation is the cost, or rather the waste of the hardware by the sequential computer. Existing applications are also becoming too intensive to be satisfactorily performed on a one-thing-at-a-time machine, while many of the more recently evolved problems are inherently parallel and consequently most effectively solved on parallel hardware. A description was given of von Neumann architecture, the factors which limit its performance and the attempts followed to increase its performance, most of which constitute primitive forms of parallelism.

The following paragraph discussed several ways of classifying parallel computers, including the distinction between data parallelism and control parallelism. One such classification, based on the parallelism within the instruction and data streams, and the granularity of parallel processes was more broadly treated in order to provide a general overview of the hardware yet for sale or under investigation. This was followed by an attempt to explain the workings of each category of machines, using parallel kitchens as their real world equivalents.
The final paragraph of the chapter dealt with the task of the human programmer. It was argued that, although interacting with the computer has become much more in accordance with the requirements of the human programmer, it is still a complex task. Switching to parallel computers will probably increase the complexity of the programmer’s task, but the exact extent will depend on the programming approach adopted, which is also dependent of the hardware. In order to acquire an overview of the programming approaches, a classification was presented, based on the degree to which parallelism would have to be specified. A distinction was made between having to specify parallelism implicitly, explicitly, and not at all.

No parallelism has to be specified in a number of artificial intelligence languages, mainly used for research, where the details are hidden by the implementation of the language. Parallelism hidden by the operating system is the presently popular approach taken in general purpose computing where it, at least in the short term, provides higher performance. No parallelism has to be specified, at least not in principle, when a vectorizing compiler is to detect the parts of a program which can execute in parallel. This approach which is mainly used for supercomputers, however, does require the programmer to assist the compiler in its work, and therefore, to be indirectly aware of the workings of the machine. When parallelism has to be specified, either implicitly or explicitly, a further subdivision can be made between data and control parallelism.

Explicitly specified data parallelism refers to the situation in which arrays of values are all processed at once, and although these SIMD-computers are among the most powerful devices, they are hard to program, because they force the programmer to adapt a problem to the architecture of the machine. In the approach with explicitly specified control parallelism, the programmer has to specify which process to do when, and sometimes also to arrange the hardware configuration. Using this approach very fast MIMD-machines can be build for little money, though it provides the programmer with the tedious job of having to control all processes.

The approaches based on implicit specification of data and control parallelism do not differ so much from each other as their explicit equivalents. When using implicit data-parallelism, the programmer specifies that a particular operation is to be done on a whole set of data, without having to worry how the hardware works it out. In implicitly specified control parallelism, the programmer might in principle have to indicate that all the iterations of a loop can be done in parallel, and because loops generally process arrays, the situation would be similar to data parallelism. However, implicit control parallelism is generally applied to the situation in which the programmer does not have to do anything special at all, except for using a particular language (eg. dataflow).

At present, only a few computers using implicitly specified parallelism have left the laboratory, but in the longer term this seems to be the most promising approach to general computing. While the MIMD-computers featuring explicit parallelism seem to keep promises for applications which require a particular flexibility and for high-speed number crunching applications. Because not much work has been done on the most important processes involved in programming, only preliminary remarks, based upon general psychological insights could be made about the consequences of adopting a particular approach to programming a parallel computer.

Chapter 3 discussed two psychological approaches to investigating parallelism, and programming in general; the linguistic analysis of time, and the nature of programming knowledge. The linguistic analysis is based upon the assumption that there are strong relations between how people think and how they use natural language. The analysis showed that, although natural language offers all the means to describe the occurrence of events in time, such descriptions are relatively hard to learn to use, and this hardship can be used to explain some of the difficulty experienced in programming.
Based on the work of Clark (1970; 1973), it was stated that the main factors affecting the comprehension of temporal relations consist of the order of mention, context effects, and derivational simplicity, of which the latter is a special kind of semantic context effect. The argument of the order of mention would favour a strictly sequential order of program statements, and an object oriented approach, if the order would have to be broken to enable reviewing the program. An object oriented approach also follows from the context argument, because this approach allows the programmer to work on a meaningful level, and concentrate on what is done, instead of how it is done.

Furthermore, in natural language, temporal words are not used as precisely as they are supposed to be used in a logical sense. When subjects are asked for their opinion about words related to parallelism, it appears that none of these words are understood according to a very strict meaning, and secondly, that the distinction between lock-step and boundary based parallelism does not seem to be psychologically real. As such, having to deal with the distinction may be a potential cause of programming errors.

Regarding the use of time-related words to express parallelism in computer languages, they are not always used, which need not be a problem if there are appropriate alternative models for the temporal expression of parallel execution, such as provided by data-parallelism. When parallelism in computer languages is expressed by temporal words, these are used differently from natural language. In the latter, the words serve to describe the position of tasks and events in time and relative to each other. In programming languages the words are used to synchronize process execution at certain points. When parallelism in programming languages is to be expressed in temporal terms, there is a need for much more humane ways of expression than the present ones. In the meantime, tools could be used to allow the programmer to work at more meaningful levels.

The psychological approach to investigating parallelism was the subject matter of the last section. Psychological investigations are focussed around two opposing schools of thought, respectively, the logic-based theories of human cognition and the network models. Here, it was argued that the both of the theories were quite able to account for certain aspects of programming, but that these aspects are merely supportive, rather than grasping what seems to be essential in computer programming, namely problem solving and the planning of actions. Neither of the two models is able to account for these more or less warm and cuddly activities, which depend heavily on individually different representations of knowledge. Programming knowledge, which may then be characterized as 'shallow', can only be investigated in a difficult way, considering knowledge of the individual, or the knowledge shared by everyone. The alternative to such a difficult approach to investigation, is to learn about parallel programming by just doing, but this has not proved to be very successful.

Chapter 4 presented an empirical study of the difference between procedural and factual knowledge. These types of knowledge which belong to the knowledge of every human may underlie the differences in the performance of experts and novices. Previous work in the area of programming has proven that expert programmers possess at least a larger and more developed body of factual knowledge than do novice programmers, both in terms of low level, as well as higher level knowledge structures. However, the evidence for a similar distinction with regard to procedural knowledge is much weaker, because it is gathered using protocol analysis techniques which are foremost descriptive tools. It may therefore still be possible that experts derive their expertise simply from more knowledge, and if so, then programming is a purely knowledge-based activity without any strategic planning component.
Next, the specific considerations behind the experiment were described. Programming knowledge is fairly abstract and supposedly semantically poorer in comparison to the typical skills from real life, therefore, programming listings would be compared to cooking recipes. In these texts, procedural and factual information were made more and less accessible by manipulating the organization with which the information was presented. The subject's task was to fill in the missing words in two texts, and answer from memory several questions about the actual order and the sensibility of that order. It was assumed that the questions about the actual ordering of sentences requires recall of factual information, and that the questions about the most sensible ordering would require factual as well as procedural information to be employed. The hypothesis under investigation did predict an interaction between question type and type of information, made more and less accessible in the text. The results might also show or fail to show differences between the two task domains.

Because the results differed very little between conditions, which is partially due to the difficulty of the task, there was not much to say conclusively. There was a small but significant difference between the number of correctly answered factual and procedural questions for the recipes, whereas no such difference existed for the program listings. It was hypothesized that this difference was caused by the presence of a mental representation of the cooking task, in addition to a mental representation of only the text, which might be taken to prove the semantic poverty of programming knowledge in comparison to real world knowledge used in cooking.

5.2 Conclusions and Recommendations

One of the lessons to be learned from the preceding chapters is that at present, the psychological models of human cognitive processing are not as useful in the area of computer programming as they were in the more conventional areas. The main problem for the present psychological models is that programming knowledge is shallow and highly individual; consequently the development of this knowledge depends more on what a programmer already knows than the knowledge in other areas. Partial evidence for this statement was obtained from the experiment conducted as a part of this study, and more conclusive evidence was found in the literature. Because it is very difficult, if at all possible, to say anything interesting about programming behaviour using the present general models of cognitive processing, other models which instead attempt to represent knowledge may prove to be more useful.

There is already some work done on modelling the knowledge involved in working with computer user interfaces. The motivation for this work was to become able to determine the ease of use of interactive software without having to perform costly and time consuming behavioral research (Reisner, 1983). In Reisner's approach, the interactive commands the user has to learn in order to operate particular systems are represented in a number of formal rules in BNF-grammar. Then the number and the complexity of these rules can be used as measures for the ease of use of a system in comparison to others. This research has been extended by a number of formal models of the behaviour of the users of interactive systems, which either stress user performance (eg. Anderson, 1983; Card et.al., 1983) or competence (Payne and Green, 1986). Although the knowledge involved in using an interface is much more limited than programming language knowledge, there does not seem to be a principle difference between the two fields. Both kinds of knowledge can be described by sets of conditional rules, within the limits of the complete interface language definitions. In addition, one might speak of the most efficient organization of this knowledge, as the organization of the hypothetical 'expert user'. More important in this respect is that these models, at least in
principle, provide the means to describe the extent and organization of this knowledge on an individual basis, and that they allow for predictions about future behaviour. As such, they are able to capture, first, how different choices and operations can be used to reach otherwise similar goals, and secondly, with the exception of Card et.al., where and why misconceptions might arise. Anderson (1983) and Payne and Green (1986) explain individual differences in knowledge by proposing a different set of conditional rules for every person, and Card et.al. (1983) take the view that there is one set of rules, but that each person has individual weights attached.

A similar approach comes from the work done on programming plans (eg. Soloway et.al. 1982; Rist, 1985; Yu and Robertson, 1988), in which programmer behaviour is described and explained by the intentions and knowledge underneath. Within the universe of discourse, which is given by the language definition, the different ways to reach similar high-level goals, and differences in programming knowledge are understood by the presence of different plans. These, in turn, are inferred from a combination of the behavior of a programmer and the problem worked on. Either way, being able to model individual user knowledge will not only increase the knowledge of human-computer interaction by a considerable amount, but eventually, this kind of model may also become a directly usable tool for software design.

The second conclusion to be drawn from this study is that the various approaches to parallel programming, and programming in general, impose different amounts of difficulty on the programmer, which may be understood by two characteristics of the human cognitive system. In the first place, the human cognitive system is limited in its information processing capacity. Especially, and this is a second characteristic, if the information cannot be organized or 'chunked' in meaningful units, for people. The twenty-sixth right parenthesis on the last line of a Lisp program may have a high functional meaning, but to the novice it means just as little as the twenty-five other ones. The two characteristics are closely related, because chunking information meaningfully does partially determine how much processing capacity is needed. They are, however, not identical, because even if it is possible to structure information meaningfully, it may be too extensive to handle. In general, one may say that the first point refers to the amount of detail the programmer has to handle, and the second point refers to not having to be concerned with the details.

As an example, consider assigning a word to a string array. In the usual case, the array will have to be declared with the correct length, the individual characters of the word assigned to the correct array-cells, and the remainder of the array filled with spaces, for example using a loop. Here, compared with other languages, the programmer has to work at a relatively low level, and is forced to consider details like array bounds, cell numbering and the correctness of the looping index, in all of which mistakes can be, and are made. In this context, Kernighan and Plauger (1974) mention 62 elementary rules on good programming style, among which to watch out for what they call 'off-by-one' errors, occurring when loops are executed once too often or too short, and probably with good reason. One might ask if it is not better to trade some of these potential sources of error against probably minor decreases of speed and available memory. For example, the simple convention in the language C to add a finishing null to a string variable would make the looping part superfluous, although the programmer still has to be concerned with some low-level detail. A still better solution can be found in among other, the often criticized language Basic, in which the programmer does not have to mind about string length and cell numbering at all, because this is dealt with by the interpreter.
or compiler. In this respect, the diminished possibility of error, and the ability to work at a relatively high level make Basic more advanced then the so-called 'professional' languages. Those who do not agree may ask themselves what was the last time they asked for a piece of paper with space for 63 characters or less.

Of course, certain features are necessary to allow specific operations, or for the expression of efficient algorithms. For example, the ability to address specific elements in an array is a precondition for the expression a binary search algorithm, and it is hardly possible to write an efficient operating system without such things as signals and interrupts, or assembly code. Here, the point is not to leave out all features which might cause difficulty, but instead to make their use optional, depending on the programmer and the problem. Or, to put it differently, problems should not be forced into a mould depending upon the features, present or absent, of a specific machine or language, especially if these are a known or suspected cause of difficulty and are not strictly necessary to solve the problems.

Returning to the characteristics of the human information processing system, there are two main recommendations to be made, following from this study and the literature on human-computer interaction in general. The first recommendation is, whenever possible, to hide specific low-level details of machines, languages and their implementation, operating systems, etc. in the same sense as higher level languages have come to replace assembler programming, which itself made programming in binary redundant.

The second recommendation is a plea to make computer system design less a matter of hardware driven engineering, by giving cognitive or mental engineering the place it deserves based on the factual importance of human factors in computer efficiency and utilization. In practice, this means making computer programming less dependent on the workings of the cpu, and more on the 'natural' workings of the human mind, in so far as these are known.

Regarding the first recommendation, it may be that the sharply increased popularity of computer usage is not due to their price and availability. Rather, this popularity seems to be due to the fact that in order to operate them, it is no longer necessary to be a hardware and assembly language expert, able to write ones' own operating system, device drivers and especially, application software. Just like the majority of the present computer users is merely to be able to find the switches and read the manuals, making the job of the programmer easier could lead to increased quality and productivity. This is partially reached with the introduction of database query languages and fourth generation languages, but these have the disadvantage that in order to use them optimally, knowledge of a third generation language and various implementation details is still necessary.

Just as there is a waste of effort in first learning Basic or Logo to get acquainted with computers, and later on having to learn a more sophisticated language like Modula, or Ada to use them efficiently, the thought might occur of a language which may serve to learn the general and introductory, as well as the specific and advanced features of computer use. Being able to walk or ride a bicycle is still not a prerequisite for driving a car. Apart from facilitating the process of learning to program, and fewer opportunities to commit errors, hiding details may be expected to lead to more reviewable program code, as well as increases in programmer productivity by not having to reinvent the wheel over and over.

Of course, there are a number of practical limitations to adopting such a proposal. First, languages and their implementations may become somewhat too big and slow to suit the present machines. To this, it may be remarked that 'once', the machines were also too small and slow to suit higher level languages, and not so very
long ago IBM thought that 640 Kb would do for its pc, which clearly was not. The same argument holds for processing speed, and for instance, the use of graphic interfaces. In the second place, not all of the languages and machines are fit or meant for introductory purposes. For example, Lisp is a difficult language, but it is so small and simple that there are hardly any details left to hide, except maybe, for the parentheses argument, mentioned earlier. Similarly, languages and machines featuring explicit parallelism using vectors or arrays are used exclusively for their raw speed, and here, for some time to come, nobody will consider trading speed against programmer friendliness.

Leaving these exceptions for what they are, there still remain lots of situations in which computer usage is not principally centered around speed and it would pay to base design, to a greater or lesser extent, on the proposed guideline. In general, the advantage may be expected to be largest in the area of general purpose usage, software engineering and educational services. In the latter case, too much detail too early, such as may be said about the present situation, is the most probable cause of confusion and low learning rates. In software engineering, making computer languages and associated libraries more transparent may be expected to lead to higher production and lower error levels, which is still getting more important than mere processing speed. In programming languages, the features which force the programmer to be concerned with low level details involve, among many other, the indices and boundaries of arrays, multiple signals and synchronization points.

There is a trade-off between the ease of hiding detail and generality of such an attempt, depending on where and how to make the changes. It is relatively easy to provide different levels of access to libraries, especially to system dependent system and language libraries. Here, each level of access provides different trade-off points between ease of use, speed and flexibility, like the difference between formatted and format-free input and output in Fortran. String and array handling in Basic is an example of different levels of access to a programming language itself, but this is generally still relatively new and less common. In most object oriented languages, the mechanism of inheritance provides a partial means for levels of access.

Two further points may be worth of mentioning. First, having different levels of constructs and libraries is generally restricted to those operations which are most commonly used at present, such as memory management and input-output. With the increased use of multiprogramming and multiprocessing systems, there will be a growing need to do the same for the less familiar management of signals, events and processes. The need for this will be much higher, because this field is presently less well understood, as it may also be inherently more difficult to understand.

The second point to make is that the gains will not be as large as they could have been, if the programmer is allowed to work at a high level in some context, but not in a closely related one. As was mentioned before, it may be expected that making the use of string and array indices redundant, will directly lead to fewer opportunities to err, as well as to indirect gains, following from working at a higher more meaningful level, such as increased productivity and a more surveyable program code. However, if indices are not necessary for reading strings but are needed for string copy and assignment, then the programmer will often still have to be bothered about indices, and there will only be the partial gain of the decreased opportunity to err. To this may be added that the presence of such an ambiguity or incoherence will itself be liable to induce errors.

To conclude the first recommendation, it could be said that programming languages and libraries will be easier to learn and use depending upon the degree, in which they allow low level detail to be hidden; and used, only if they are needed to solve a problem, or the programmer wants to.
The next recommendation is closely related to the second characteristic of human cognitive processing, which is at its best, if it is possible to 'chunk' information in a particularly human way, at a high and meaningful level. In order to overcome the first characteristic, the fact that the information processing system is limited in its capacity to handle information, the information must be chunked in larger pieces which put a smaller demand on the available resources. However, not every way of chunking information is equally useful, because the operation itself requires a smaller or larger amount of information processing, and the resulting pieces of higher level information may not be equally easy to store and retrieve in memory.

Consider trying to remember a relatively long series of digits. A mathematically elegant way, is to regard the digits as a number, and remember it as the product of two or more smaller numbers. In this way, the resulting numbers may be easy to remember, but the method is hardly useful, because calculating these numbers is usually too difficult. For people a more useful way to remember a large number is to try to find some syntactic pattern in it, although the result is not as easy to store and retrieve.

What then seems to be the best way of chunking in one respect does not have to be the best way for people, and if a particular way of processing information is presupposed, like for example, by using a certain programming language, the human may turn out to be a hampering factor, as is often the case. Consequently, if one asks what the most suitable language for the human programmer looks like, the answer is to look for the one which is, among other things, the easiest to organize in chunks. However, at present there are no complete insight in how humans actually do this.

There are several theories and models, discussed above, which describe a number of important characteristics, such as the presence of inconsistencies, and the number of features needed to distinguish rules and probable 'chunks', and thereby the ease of use of a system (ie. Reisner, Payne and Green, and Anderson). Models like these require that the system is, at least partially, specified in advance, and it is difficult to apply them to open questions, like: "what is the best way to synchronize programs, using time words". A similar reason applies to experimentation, for which there are additional drawbacks, in terms of time, effort and money. However, if the alternatives are known, then both models and experimental work can and should be used, because in such cases they offer rather cheap tools to avoid making otherwise costly mistakes.

Because it is not yet possible to use the theoretical models and tools to guide in finding the answer to open design questions, another choice was made in this study. This started with the idea that it the way in which humans deal with information in one context may provide useful design principles for handling information in another context. To put it in different words, if people 'chunk' temporal information in general without referring to exact durations, and begin and ending points, then this may a more efficient way for the human cognitive system of handling temporal information in the context of programming, than with reference to exact timing.

It may be objected that doing so does not guarantee designing the best and most efficient systems, because it is still possible to design systems, which are based on a bad habit like Roman numbering, if such a habit is 'good practice' in some context. However, when such reasoning by analogy is applied in combination with psychological principles and insights, using a strict methodology, such inefficiencies will easily show up. In a sense, "reasoning by analogy" uses the wrong wording, because here it does not merely concern the way people act in reality, but the way they have to act because of their particular processing system, which can be experimentally proven.
Further, using psychological insights, even without a firm theory, is certainly more promising than adhering to purely technological considerations, because by not taking human factors into account, systems may become not only equally difficult to deal with as other systems, but they may become even harder to use. In this context, one may think of a simple comparison between arithmetic with decimal and binary numbers.

Regarding programming in general, in this study it was argued that the language designer should consider the principle of order of mention and context effects, which, as was concluded, would favor object oriented programming approaches. Approaches like these allow for a reasonable compromise between the human way of saying that certain things have to be done, and the requirement of the machinery to state exactly how things have to be done.

Where it concerns parallel programming, reasons in favor of object oriented programming are the possibility of hiding the parallelism under a 'virtual' layer, and the possibility of specifying it at a meaningful level. It goes without saying that further research at this point is both promising, as well as necessary, in order to make the programmers' life easier and increase productivity. There are also indications that various data parallel approaches may be suitable to the human programmer, because these make it possible to view programs in similar terms as are already often used in real life (e.g. all, except, any, etc.). It seems worthwhile to investigate the reality of such spatial or set-theory metaphors, as they were called, and determine if they are applicable to facilitate parallel programming.

There remains a lot to be learned about the use of explicit time words in parallel programming. For instance, to see if the lock-step type of parallel execution does indeed cause difficulty for programmers. It might accidentally not exist in natural language, in which case it is a simple matter of inventing the word to close an unsystematic linguistic gap. On the other hand, if it can be shown that having to deal with exact timings does pose problems, like for instance in using Occam, then there is evidence, instead of intuitions to do something about it. In the latter case, the alternatives are to develop supporting tools, or to try to develop a set of synchronization constructs, corresponding better to human information processing. Also without such evidence, and more or less following from the first recommendation, there is reason to think about better timing constructs. A semaphore, for example, is a very abstract low-level solution of the synchronization problem, lacking any recognizable counterparts in the real world. This does not only make it relatively hard to master, but, also in terms of development time and errors, it would pay to implement a more humane 'one-thing-at-a-time' construct in place of it. The same holds for many other fairly low level synchronization constructs, until more attention is paid to enable the programmer to focus on the problem, instead of having to consider the know-how and know-when because of the machinery.

To conclude this section, for each of the approaches to parallel processing, an attempt is made to answer the question after the future prospects. Starting with the explicit parallelism in SIMD machines, it seems better to leave pipelines and processor arrays to the experts, because in the short term, it is not possible to make these approaches more programmer friendly without significantly affecting the speed to costs ratio. On the other hand, some progress can be made by the use of new and specially designed languages, in order to avoid some of the ad-hoccer of the present versions of Fortran and to hide some of the hardware dependent characteristics (see: Perrott and Zaria-Aliabadi, 1986; Perrott et.al., 1987).
From the viewpoint of software ergonomics, the ultimate goal is formed by the approaches to programming in which the underlying machine architecture is completely hidden from view, but this is from hiding parallelism under a sequential layer. On the short term, the present approaches are quite useful for dusty decks, that is, for not having to throw away the efforts and investments associated with the piles or decks of old fashioned punchcards (eg. Test et.al., 1987). However, the choice for hiding the parallelism underneath the operating system requires a certain amount of compilation or execution overhead, and there is no possibility of expressing the parallelism inherent in problems. Therefore, this approach does not seem very promising for the future further away, with the possible exception of the logic machines for specific knowledge based applications. However, programming in logic has not yet shown to be very suitable for general purpose computer applications, nor do languages like Prolog appear to be very easy to use.

With regard to implicit data parallelism on both SIMD as MIMD machines the future seems bright, especially if a combination of high speed computing is provided together with a way of expressing parallelism, which is either programmer friendly, or problem oriented, or both. Although some discussion about the different concepts of each proposed language is possible, the general provision of concepts such as all, any and except, and the speed and theoretical algorithmic potential of this kind of parallelism (Hillis and Steele, 1986) keep open the promise of a happy marriage. As some languages already do, it would be very sensible to allow these concepts to be used without the need to use any indices. That is, 'do for all indices' is nice as well as necessary for certain applications, but often a 'do for all' is all that is needed.

MIMD computers featuring coarse grained offer great advances in term of speed, but yet, it is difficult for the human programmer to get at this speed without going through several hardships, which are probably too hard for the general application programmer. One possibility is using an object oriented programming style, which facilitates or at least enables understanding parallelism at the semantic level. In such languages objects or abstract data types serve both as real-world operations and entities, and as specifications of parallelism, whereas the first role is left out of other approaches (eg. Ghezzi, 1985; Goldberg, 1984). As such, the difference is between calling a procedure 'delete' to delete a 'file-of-ascii', sending an object 'document' to another object 'eraser'. An alternative which is focussed more on raw speed is using a language with constructs such as signal, semaphore and interrupt to synchronize processes. In this approach, there seem to be few if any constructs for process control which are both programmer friendly, as well as efficient. What seems to be needed here then, is the development of such higher level constructs as one-at-a-time or do-in-lock-step. Apart from expressing process control, course grain MIMD programmers will have to deal with the politics of load balancing (Pountain, 1986); assigning tasks to processors in a way that every one gets a fair share of the work and none too much (Anning and Hebditch, 1986). However, because compilers are not very good at this, and it is a tedious job for humans, there is a clear need for tools to analyze the timing of program execution.


Appendix A: Written instructions, in order of appearance -
- instructions for testing in groups
- instructions for individual testing

Appendix B: Stimulus texts, in order of appearance -
- Recipe: realistic version
  disorder version
  apart version
- Program: disorder version
  apart version

Appendix C: Questionnaires, in order of appearance -
- questions about recipe knowledge
- questions about program knowledge
- questions about experience and opinions