CONTROL STRUCTURES FOR PROGRAMMING LANGUAGES
FISHER, DAVID ALLEN
ProQuest Dissertations and Theses; 1970; ProQuest Dissertations & Theses Global

70-21,590

FISHER, David Allen, 1942-
CONTROL STRUCTURES FOR PROGRAMMING LANGUAGES.

Carnegie-Mellon University, Ph.D., 1970
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CONTROL STRUCTURES

FOR PROGRAMMING LANGUAGES

by

David A. Fisher

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania
May 1970

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Carnegie-Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Control Structures for Programming Languages

PRESENTED BY David Allen Fisher

ACCEPTED BY THE DEPARTMENT OF Computer Science

Alan J Perlis — MAJOR PROFESSOR — 4 May 1970 — DATE

Alan J Perlis — DEPARTMENT HEAD — 4 May 1970 — DATE

APPROVED BY THE COMMITTEE ON GRADUATE DEGREES

[signature] — CHAIRMAN — 5/5/70 — DATE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PLEASE NOTE:

Some pages have indistinct print. Filmed as received.

UNIVERSITY MICROFILMS.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

The research reported here is the result of an investigation of several aspects of the control structure of programming languages. By control structures is meant programming environments or operations which specify the sequencing and interpretation rules for programs or parts of programs. This dissertation attempts to demonstrate the thesis: complexity diminishes and clarity increases to a marked degree if algorithms are described in a language in which appropriate control structures are primitive or easily expressible.

A number of control structures extant in programming languages and systems are catalogued and then used as a guide to develop a programming language which has a control extension facility. This language has not only the mechanical necessities for control extensions, but also has primitive control operations for sequential processing, parallel processing, alternative selection, monitoring, synchronization, and relative continuity. These operations are the source of the clarity of control descriptions because they span our conceptual notion of control and because they can be easily composed to form other more specialized control structures.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The thesis is demonstrated by using the control description language to give formal descriptions of itself, the simulation language Sol, and a variety of specific control structures. Some nonstandard control structures (including a nondeterministic control called sidetracking, the continuously evaluating expression, and a multiple sequential control) are invented to further illustrate the thesis. It is also shown that the use of nonstandard controls in some cases leads naturally to efficient implementations.

The most extensive example given, uses the sidetrack control for parsing context-free languages. Beginning with a simple breadth first search recognition process, it is refined by the incorporation of appropriate control structures. Minor additions are then made to permit parsing, translation, and error correction. It is also shown that hybrid recognizers which merge several well known syntax analysis techniques can be easily built and an implementation of the sidetrack control which achieves the time bounds of Earley's parsing algorithm is also given.

This investigation has only scratched the surface, but it is hoped that it has contributed to the understanding of control, illustrated the simplicity and clarity that can result from a conscious examination of control, and demonstrated the large, but unrealized, potential for variability in control.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Chapter I. Introduction 1

Chapter II. Review of Control Structures in Programming Languages 16

A. The Sequential Machine 17

B. The go to Statement. 20

C. Programming Languages — General Characterization 22

D. Recursion. 23

E. Coroutines 25

F. Parallel Processing. 28

G. Synchronization and Monitoring 32

H. Pseudo-parallel Processing 36

I. Nondeterministic Programs. 37

J. Other Control Structures in Programming Languages 40

K. Conversation and Other Nonprogramming Control 44

L. Summary of Control Structures. 48

M. Control Definition Facilities. 52

Chapter III. A Formal Device for Describing Control . . .55

A. Data Structures and Operations 62

B. Primitive Control Operations 66

C. Syntax 72

D. Environmental Data Structure 76

E. Environmental Control Structure. 77

F. Extension Facilities 78

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

G. Some Examples. 83

H. The Interpreter. 87

I. Implementation Strategy for Sequential Machine 93

J. Experience with an Implementation of CDL . . 103

K. The Control Structure of SOL 106

Chapter IV. Multiple Sequential Control 121

A. Sequential Processing. 121

B. Interface Operations 125

C. Stack and Queue Storage. 130

D. Semaphores 135

E. Operating Systems. 138

Chapter V. Parsing as a Problem in Control. 142

A. Terminology. 143

B. A Nondeterministic Control 143

C. Sidetracking 144

D. Multiple Parallel Return 147

E. Production Language Syntax 148

F. Interpreter for Sidetrack Control. 149

G. An Efficient Implementation. 153

H. Computation of Time Bounds 160

I. Hybrid Recognizers 163

J. An Extension Using a Delayed Execution Control. 167

K. Error Correction 171

Chapter VI. Continuously Evaluating Expressions 176

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A. Some Uses as an Expression 176
B. Some Implementations 177
C. A Tree Representation. 179
D. Compilation of Tree Form Expressions 184
E. Continuously Evaluating Systems. 186
Chapter VII. Further Study and Conclusions. 187
References 200

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

I am deeply indebted to Professor T. A. Standish who, as my advisor, guided my research and to Professor Alan J. Perlis for his helpful suggestions. I want also to thank Professors Standish, Perlis, Habermann, and Reddy, Jim Leathrum, Harvey Bingham, and Bob Chen for their efforts in reading this paper.

I am indebted to Mr. John T. Lynch of the Burroughs Corporation who provided machine time and facilities for the CDL implementation, and to Dr. J. F. Leathrum who helped with that implementation.

As a recipient of an IBM fellowship during the past year I gratefully acknowledge my indebtedness to the International Business Machines Corporation for financial support.

I would like to thank Mrs. Agnes R. Toole for patience and outstanding effort in typing the final draft. I also thank Marjorie Fisher and Barbara Toole for their help.

Finally I want to express my sincere appreciation to my wife for her understanding, patience, and support throughout this effort.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I. INTRODUCTION

> "There is one important difference between ordinary logic and the automata which represent it. Time never occurs in logic, but every network or nervous system has a definite time lag between the input signal and the output response. A definite temporal sequence is always inherent in the operation of such a real system."
>
> — John von Neumann [vN 56]

This dissertation is an investigation of several aspects of the control structure of programming languages. The various control mechanisms used within current programming languages are examined, several alternative views of control are discussed and a formal mechanism for describing the control structure of programming languages is developed. A number of nonstandard control structures are introduced to illustrate the insight and simplification a conscious examination of control can provide in a variety of problem areas. The major intent of this effort is to acquire a better understanding of control. In addition, in a few cases implementation techniques are examined.

Despite the many studies concerned with the formal description of various aspects of programming languages, control structure (with the exception of isolated cases dealing with specific controls, usually parallel processing) has been largely ignored. There have been several proposals for describing data structures [Mc 64] [St 67], operators [Le 66] [GP 67] and syntax. Compiler-compilers [BM 62] [Fe 66] [Ch 65] and formal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

languages for describing languages [NN 66] [Bö 66] [Str66] have also been proposed (see also [FG 68]). Gorn [Go 61], in fact, identifies thirteen different kinds of specification languages. More recently there has been considerable interest in extensible languages [Jo 69] [Ir 69] [Be 69] [Ma 69], that is, languages within which the user can modify the language to tailor it to his own needs.

Yet in all the above investigations there is no explicit concern with the control structure of languages. In each case either sequential processing and/or subroutines are taken as the given control structures and all programs are composed from these. Indeed, much more can be said and tools can be provided for describing a broad class of interesting regimes of control. This dissertation hopes to demonstrate that the study of control can simplify and illuminate some areas, and that the use of appropriate control structures can produce clarity of expression without loss of efficiency.

By control structures we mean programming environments or operations which specify the sequencing and interpretation rules for programs and parts of programs. Thus, the control structure of a language might include a general rule for the evaluation of expressions (e.g. all arguments are to be evaluated and then the operation applied to the resulting values of some of its arguments to determine which of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

others are to be evaluated). The general rule for statements might perscribe that they are to be executed one at a time in the lexicographic order of the program description, but control operations such as the unconditional transfer and iteration statements violate this general rule.

We intend our definition of control structure to include not only sequencing, but the lack of sequencing rules or even the indeterminancy of the desired sequencing. Consequently, parallel processing and nondeterministism are also control structures. It may be necessary to process two parts of a program sequentially, their processing order may not matter, or it may be that they can be processed in either order but not concurrently. It may be necessary to process only one component, which one being indeterminant until the other has been processed. These are all examples of control structures. A definition of control, however, does not give insight into the implications and applications of control.

We begin our investigation of control structures by examining the control structures of programming languages. For the majority of available languages there are just four control mechanisms: sequential processing, iteration, subroutine control, and the conditional. Within these basic control structures there are minor variations which arise from the restrictions on and interaction of the given control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

capabilities. It may or may not be possible to enter into the middle of an iterative statement from outside, subroutines may or may not be recursive. Variations also arise in the way control interacts with the environment. For example, the interaction of lexicographic and dynamic scope, and the use of control to evaluate parameters is accompanied frequently by shifts in environment.

A few languages have control structures in addition to those mentioned. Algol-60, for example, has call by name procedure parameters which have several useful interpretations each generalizing to a different control structure. Pure Lisp (i.e. Lisp without the program feature), on the other hand, has only recursion and a conditional operation, and illustrates how little in the way of control is necessary for a complete system.

Other control mechanisms have appeared in the literature, not as the control structure of programming languages, but as implementation techniques for compilers and operating systems. Coroutines first appeared as a technique for implementing Cobol compilers [Co63a]. Most operating systems use some form of pseudo-parallel processing. We will be examining a few of these.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One must, however, be careful not to assume that the control structures which are found in current programming languages or software systems are the only useful controls or even constitute the general form of useful control structures. The control structures above reflect the sequential control of the machines on which they are implemented. Even the discrete simulation languages, which were designed to simulate parallel activities, have no parallel processing control structure. Instead they have a pseudo-parallel control in which the programmer can specify the interleaving of several sequential processes.

Any attempt to identify useful control structures should not be restricted to the examination of programming systems. It is quite possible that control mechanisms which would be useful and desirable in programming languages have not been provided in current systems because of limitations in hardware, software or both. We will look for the control structure of various active components of the real world. Control structures which have proven useful in the real world may also be useful in programming systems, and any attempt to model accurately the real world requires the facility to simulate these control structures.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Once a number of control structures have been identified, we can begin to develop a facility for describing control. A look at the means for describing other language structures will give us some perspective on the problem. One approach would be to design an eclectic language which draws together all the control structures which we are able to catalogue, in effect a union of the control structures of other languages. There are a number of problems with this approach. Such a language might be a supermarket housing all the available known forms of control, but lack the descriptive power to describe any new forms of control.

Regardless of the control structures provided within a language, applications will arise in which other control structures are desired. At the time a language is designed one cannot forsee all applications to which it will be put. Consequently, it is impossible to build into a language in advance a fixed set of control structures suitable to all applications. A similar problem occurs with data structures [St 67], but the problem is more severe for control. Data are objects within the environment provided by a language, and operators are provided to modify, interrogate and compose them. Within a given language the primitive data structures can be used to simulate the desired structures (although not necessarily efficiently). Lists can be used to represent vectors, vectors can be used to represent arrays. Control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

structures on the other hand are an integral part of the programming environment. Rather than having objects which can be manipulated within the environment they provide a regime which governs the way in which the program will be evaluated. For most languages the only way of combining control structures is by embedding (i.e. subroutines and nested loops). Control structures which are not primitive to the given language can be simulated only by simulating a new programming environment, that is, by writing an interpreter.

Because control is strongly entwined with the programming environment of a language, the control description facility must be concerned with all aspects of the programming environment. The other major component of a programming environment might be called the environmental data structure, the block structure of Algol [Na 63], for example. For other components, such as procedure parameters and the rules for passing them, it is not clear whether they should be considered as control or data. It does not really matter. The point is that if there is a formal means for describing the external syntax of a language and a corresponding internal representation for every program of that language, a formal means for describing the data structures and operators of the language, and a formal means for describing the control structure of the language; then the combination of these should provide a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

complete description of the language. In this view, the control structure encompasses all aspects of a language not embraced by the data or syntax.

In addition to their limitations on generality, there are two other problems which result from eclectic languages: inefficiency and inflexible notation. Languages such as PL/1 [PL 66] and Formula Algol [PIS66] which have incorporated several data structures into one system have been costly in both time and space. Although any program uses only a small subset of the facilities of the language, it must pay the overhead for the unused generality. This situation could be somewhat remedied if the user were able to specify that subset which he is using, and if, thereafter, the compiler could restrict itself accordingly.

As the number of primitive components of a language increases, notations must be provided to differentiate between them. The language APL [Iv 62] has a large number of primitive operations and many infix operator symbols to represent them, but this represents a change in quantity rather than form. Experience with APL has shown that the large number or operators is an inconvenience only in the initial learning of the language and that these operators provide conciseness of expression once learned.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The situation is different with control structures. The environmental control structure is the overall processing scheme for programs. Generally, only one such scheme is required for a given programming task. Consequently, there are few requirements for explicit reference to the control structure. Most references are implicit, and those which are not, usually use only functional notation. Functional notation has been used to specify subroutine calls, coroutine calls, and calls on nondeterministic procedures; but since only one type of call is provided in a given environment there is no conflict in distinguishing which is intended. When several control structures are available in the same environment, specifications must become explicit and the functional notation alone is no longer sufficient. Thus conciseness and clarity of expression, some of the advantages from using control structures well suited to a task, can be lost.

The above problems point out two different goals for describing a programming language or more particularly its control structure. One goal would be to provide the user of the language with a description from which he could determine: what can or cannot be done easily within the language, what the various notations of the language mean, what types of objects can be manipulated and how, the scopes of names and values, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the relationships between the processes described. The Algol-60 report [Na 63] is an example of a description of this type. This kind of description is useful to the user because it does not hide important features of the language in the mechanism of an implementation and, if unambiguous, is useful to the implementor because it does not confuse the properties of the language with the properties of a particular implementation scheme.

The other goal is to give a description which indicates how programs of the language can be processed, and provides techniques for the efficient mechanization of the various components of the language. Efficiency is of critical importance in the use of digital computers. Most descriptions of real programming systems emphasize this goal. The translator for a language is a formal unambiguous procedural description of that language. Unfortunately, in too many cases it is the only complete description. With little difficulty one can find language features which were determined by the implementation and not by the needs of the applications for which the language was intended.

These are both important goals and they are not entirely independent. The approach here, however, will be to provide a descriptive facility which permits language and control structure descriptions which are independent of their

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mechanization. For control structures this means that the description facility, a programming language itself, will have an environment in which the desired control structures can be formed by composition of the primitive controls of the description language. To do this the control structure must be reduced to objects which can be manipulated within the description language. Thus, expressing control as objects in the environment of the description language has no effect on where the control resides in the language being described.

The primitive control operations are simpler than most of the control structures to be described. There will be six types of control operations, one reflecting each of the functions in control structure: sequential processing, parallel processing, testing, monitoring, synchronization, and relative continuity.

The kind of description which we would like to give for any language is exemplified by Lisp [Mc 60]. Perlis expressed this view and its importance in his Turing lecture [Pe 67].

> "...its description consciously reveals the proper components of language definition with more clarity than any language I know of. The description of Lisp includes not only its syntax, but the representation of its syntax as a data structure of the language, and the representation of the environment data structure also as a data structure of the language. Actually the description hedges somewhat on the latter description, but not in any fundamental way. From the foregoing descriptions it becomes possible to give a description of the evaluation process as a Lisp program using a few primitive functions. While this completeness of description is possible with other languages, it is not generally thought of as part of their defining description."

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

> "Why is it so important to give such a description? Is it merely to attach to the language the elegant property of 'closure' so that bootstrapping can be organized? Hardly. It is the key to the systematic construction of programming systems capable of conversational computing."

If we are to get a thorough understanding of any programming language, we must have a clear representation of its evaluation process. What is the source of the clarity and elegance of the Lisp description: There are several and we will point out three we think important: the programs and environmental data structure can be represented as data within the language, the interpreter was written so that its control structure is the same as the control structure of the program being interpreted, and the language has very few primitive components whether syntax, data or control. In our language for describing control we have attempted to emulate these. List structures are provided for representing the abstract (internal) syntax and a set notation similar to that used in the Vienna report [LLS68] is used to represent the data structures (including those of the environment). Clarity in the description of a control structure is achieved by a programming convention. The number of language components which must be described is entirely a function of the language being described and should affect only the extent of the description and not the complexity.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus far we have emphasized our desire to provide language descriptions which are not clouded by particular implementation techniques. In those cases where we wish to describe efficiency techniques or implementation strategies we will replace all or part of the above descriptions with descriptions which use only those control and data structures available in some special environment.

Also important is what is added to our understanding of control and how this influences our ability to cope with other problem areas. Can the use of control structures better suited to a programming task simplify that task and expose the significant problems in that problem? We think so, and to support this view will examine several task areas with respect to control.

The view of control found in most programming languages, the view that processes are essentially sequential, is characterized by the multiple sequential control environment which contains most of the common control structures of high level languages. This environment provides a chance to look at the problems which arise with several control structures in one environment and, because the control environment is similar to that of a sequential machine, to compare how control structure affects storage allocation policies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Two control structures of our own invention illustrate some of the advantages resulting from conscious examination of the control structure for a problem. A control structure called sidetracking is applicable to problem areas involving nondeterministic procedures, and will be looked at in the context of parsers for context-free languages. Within this framework a common nonprocedural notation (BNF) is used to describe the syntax, an interpretation of that description as sidetrack control is used to provide a recongizer for the language, and a sequential implementation of the control is used to achieve the time bounds of Earley's algorithm in a different form, a form dictated by the control structure which was itself chosen to correspond to a convenient external notation. Yet this same difference enables a simple argument for the time bounds required by the method.

Another control structure, the continuously evaluating expression, is as its name implies, an expression which is continuously evaluated. Whenever its value is required the output is accessed directly without additional computation. One advantage in looking at this control structure is that it does not conform to the usual sequential view of control. A look at common programming tasks with this control in mind reveals a number of places where it seems quite natural. The result is that one gets a different perspective on the task. Because the continuously evaluating expression is so unlike most controls

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we encounter, some effort will be spent to develop an efficient method for implementing it with sequential control.

Finally, a number of other nonstandard control structures will be suggested. One of particular interest is a form of delayed execution which can be used to transform the sidetrack representation of a recognizer into a translator. Because the notation used for this translator is the same as that for describing the syntax (internal and external) of the control description language, this control structure together with the initial self description of the language forms a closed bootstrap system not only for the interpretation but also for the translation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

II. REVIEW OF CONTROL STRUCTURES IN PROGRAMMING LANGUAGES

> "The study of forms of control and scheduling in languages and computer systems has not received the attention it deserves" — Peter Wegner [We 68]

There is very little literature which deals directly with control. The design and implementation of programming languages has been a major concern to computer science, but the problems of control structure have been largely ignored. Each language has an environmental control structure and a few control operations, but these are usually an integral part of the language and are unalterable either within or without the language.

The reasons for this may include the fact that the primitive control structure of a sequential single processor digital computer is sufficient to simulate any other control structure. Consequently, there is limited incentive to develop other control structures. The rewards from using more complex or other control structures are largely in clarity and conciseness of expression and in programming ease, measures which are difficult to quantify. As Perlis [Pe 67] puts it "Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily." The control structures of programming

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

languages have not kept pace with the increased flexibility in syntax and data.

A. THE SEQUENTIAL MACHINE We begin by examining the control structure of a single processor machine. The only environmental control structure is sequential control, a general processing rule in which the instructions of a given program are arranged in sequence and processed in the order of that sequence. Also, at least on the earliest machines, there are just three control operations. The control operations provide a means to violate the sequential processing rule. There is some form of conditional whose successor is determined by the value of a variable. Often the conditional causes instructions to be skipped depending on the variable value. The control transfer or branch operation permits one to specify the next instruction explicitly. Because this successor may occur earlier in the given instruction sequence, the transfer can be used to program loops and in conjunction with the conditional can be used to limit the number of iterations the loop will be executed.

The third control operation is primarily a data operation. It allows the content of storage cells to be dynamically altered. Because we are dealing with von Neumann machines, machines in which the data and instructions are kept in the same storage, the data operations can be used to modify the instruction sequence itself. This has been acclaimed as one of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the major sources of power of the digital computer, yet it is a technique seldom used in modern machines except for the initial loading and translation or programs. In the earlier machines it was widely used but today the facility is unavailable in most high level languages and its use is generally considered bad coding practice in machine and assembly languages. We will look for the causes of this change.

As frequently used patterns of instructions were identified, single instructions were provided to replace those patterns and thereby increase the conciseness, clarity and efficiency of programs. One goal of this thesis is to identify other useful control structures by repeating this act of specilization on more complex patterns of instructions.

Why was instruction modification used? In many instances with iterative (loop) control it is necessary not only to execute a sequence of instructions repetively, but also to make systematic changes to the data addresses at each iteration. In later machines this function was provided by index registers which allowed the addresses in instructions to specify only the base address of a vector of storage and each time the instruction is executed the address is augmented by the content of the index register.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Instruction modification was also useful when a sequence of instructions appeared several places in a program. Economy of storage was gained by storing these subsequences only once and transferring control to them each time they were required. To return control to the point of call it was necessary, before each transfer to the sequence, to alter the last instruction of the sequence to be a transfer to the point of call. Later machines had subroutine call instructions which transfer control and also retain what would have been the next instruction address, and subroutine return instructions which cause a transfer to the retained address.

It can be argued that the need for instruction modification has been eliminated by the incorporation of an execute instruction into the instruction set of some machines (the execute instruction causes its operand to be executed as if it were an instruction). A closer examination of the use of the execute instruction, however, reveals that its use in practice is not to cause the execution of a computed instruction but to select among alternative instructions (i.e. it is the indirect addressing and not the instruction modification ability which is used). The execute instruction, for example, is sometimes used to implement Algol call by name parameters. For this purpose each actual call by name parameter is viewed as a (or several) subroutine which is evaluated whenever the corresponding formal parameter name is encountered

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the procedure body. The execute instruction is used to select among the various actual parameter expressions (often by executing a variable whose value is a call on the appropriate subroutine).

The ability to modify instructions dynamically was used for only a few purposes (i.e. indexing, subroutines) and as special instructions for these purposes were introduced the need for dynamic modification of instructions was eliminated. The value of dynamic instruction modification was not from a direct requirement, but from its simplicity and generality which made it a natural building block for other frequently used patterns of instructions. The usefulness of the *go to* statement is quite similar and as will be seen in the next section, specilization of the uses of *go to*'s have, if not eliminated, at least diminished the need for a *go to*.

B. THE *GO TO* STATEMENT The *go to* can also be used to build a number of control structures. We have seen how it (or the control transfer) is used to form loops and for calling and exiting subroutines. In most programming languages some type of intrative control structure, *DO* loops in Fortran [Ba 57] and *for* statements in Algol [Na 63], is available. Subroutines are called using functional notation and the return is implicitly specified as the successor of the last instruction of each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

routine. Since special control structures are provided for iterative and subroutine control, the *go to* is no longer required for these functions..

It has been argued that the *go to* should be eliminated from programming languages. Dijkstra [Di65a] [Di68a] argues on aesthetic grounds, primarily elegance, clarity of expression and ease of conveying information; Perlis [Pe 68] on grounds of enabling and simplifying program proofs. Because control structures such as iterative and subroutine control are less general that the *go to* their special properties can be used to advantage (e.g. a push down stack implementation of subroutines). Thus we would add efficient implementation as an additional advantage to eliminating *go to*'s.

Some of the more common uses of the *go to* can be eliminated as follows. Any *go to* which leads backward without forming a loop can be eliminated by reording the program. Any one which leads backword and forms a loop (with a conditional) can be replaced by a statement of the form *do x while y* where *y* is the condition. When two *go to*'s lead forward to the same point then each of them terminates a sequence of instructions. These *go to*'s can be eliminated by making these sequences be alternatives to a conditional statement. In any case Böhm and Jacopini [BJ 66] have shown that only composition, a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conditional, and an iterative control are necessary to describe any computation (i.e. they are equivalent to a Turing machine).

If requirements of ease, clarity, or simplicity are important then Böhm and Jacopini's constructions are useful only when sequential, iterative, and subroutine control are natural for the task at hand. This may not always be the case. If coroutines were required (but not given as primitives) then the elimination of *go to*'s would make the simulation of the coroutines less elegant, proofs would not be simplified, and the possible imposition of a stack (i.e. subroutine) environment on coroutines would most likely make them less efficient. We believe that control transfer statements should be eliminated from programming languages, but only when their elimination does not make the desired control structures less efficient or more difficult to use.

C. PROGRAMMING LANGUAGES -- GENERAL CHARACTERIZATION Most programming languages reflect the machines on which they are implemented. Their primary control is sequential. In addition there is usually some form of iterative, subroutine, and conditional control and a *go to* operation. The iterative controls vary in the form of the termination condition (e.g. exhaustion of a list, or the value of a Boolean expression becoming true and when the termination condition is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tested. Algol [Na 63] tests before each iteration while Fortran [Ba 57] tests after each iteration. A significant property of subroutines is whether they may be recursively defined and/or called. Recursion will be discussed in section D of this chapter.

Sequential control means that programs consist of sequences of statements which are evaluated in the lexicographic order of their sequence in the program text. Statements represent actions and modifications to the environment which are evaluated for their side effects. There must also be value producing structures called expressions. These two calsses of evaluation are reflected in other control structure. In Algol there are both function designators (value producing subroutine expressions) and procedure statements. There are also conditional expressions and conditional statements.

D. RECURSION Recursion is not just an extention of the subroutine concept but an important control structure in its own right. Recursive control was first introduced to programming languages in IPL [Ne 61] to facilitate the description of processes which involved many similar levels of control. In IPL-V the recursive control is superimposed on a sequential (statement structured) machine. Pure Lisp [Mc 60], on the other hand, has recursion (as expressions)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and a conditional (expression) as its only control.

There are several reasons why some languages permit recursion while others do not: it may relfect a feeling that recursion is (or is not) a natural way to describe certain processes, it may be a reflection of the storage policy, or possibly some combination of these. In list processing languages the data (i.e. lists) have recursive structures and consequently it is convenient to describe processes recursively. The availability of recursion in list and string languages (e.g. Lisp, IPL-V, Snobol) may be a reflection of their storage policies. In these languages the programs are represented in the same form as the data (i.e. as lists or strings). In Fortran the relative storage addresses for data are fixed at compile time and recursion is prohibited. Because the storage need be allocated only once, Fortran saves processing time by excluding recursion. Did the lack of a need for recursion permit an efficient storage policy, or did the desire to use a particular storage policy exclude a useful mechanism (i.e. recursion)?

Recursion does not necessarily mean less efficient storage policies. A dynamic storage allocation scheme commonly used for Algol will in some cases require less space for a computation than would the corresponding Fortran program. Algol permits recursion and each procedure (and possibly block)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is allocated space as it is entered. Thus, more processing time is required than with a fixed allocation scheme, but because a procedure is allocated space only if it is called and then only for the duration of that call, the same storage can be used by several procedures at different times. Again it may have been the availability of such a mechanism that led to the inclusion of recursion in Algol, or it may be that a desire to include recursion dictated a dynamic allocation policy. One case in which a storage allocation policy makes recursion available without additional cost is in multiprogramming systems which use reentrant code (i.e. one copy of a procedure is shared by several programs each with their own copy of the data). The reentrant property requires that the data be allocated independent of the program, and that data addresses cannot be fixed at compile time. Consequently, run time allocation is necessary and the incremental cost, if any, to allow recursion is small.

E. COROUTINES Coroutines are a specialized control for situations in which the natural division of a process into subtasks is not hierarchical. There are at least three views of coroutines which yield the same structure: mutual subroutines, procedures with *own* storage, and symmetrical control. The mutual subroutine view is convenient when using coroutines but gives little insight into the mechanisms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

involved. In this view coroutines allow each of several procedures to be written so that each procedure calls all the others as if the others were subroutines. For example, a parser and a lexical analyzer might act as coroutines. The parser would be written as if the lexical analyzer were a subroutine which returns the next lexical quantity whenever called, where the calls can occur anywhere within the parser. The analyzer would be written as if the parser were a subroutine which disposes of a lexical quantity, with calls on the parser at any convenient points within the analyzer. This is essentially the view of coroutines given by Conway [Co 63a] although his specific intent was to give an implementation technique for Cobol compilers.

Procedures with *own* variables, variables which retain their values between calls, provide the same facility as coroutines. The *own* variables are used not only to retain the values of local data for the procedure, but also to retain the state of processing within the procedure so that processing can continue from that point at the next call. This is very inconvenient in a language like Algol because the only mechanism for returning is to label each return point and provide a *switch* which selects among those labels. This means that the return points cannot be embedded in expressions, procedure parameters, or even within *for* statements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Programs are simplified by separating them into logically disjoint parts and describing each part separately, usually in the form of subroutines. Some problems, however, do not have such a hierarchical structure. They (e.g. translators) may instead require several interdependent stages of processing. Knuth [Kn 68] gives two control mechanisms which may be applicable to these tasks, the multipass algorithm in which the stages of processing are done to completion one at a time and the coroutine which allows the execution of the various stages to be interleaved. This view of coroutines emphasizes the symmetric relation it establishes between the processing stages.

Regardless of the point of view, any implementation of coroutines is such that at each point of call, the calling routine is suspended and the called routine is resumed at its last point of call. The advantage of the coroutine as a control structure is that each of several processes can be described as a principal (vs. subroutine) routine with minimal concern for the interface with other processes. Contrast this with subroutines which achieve the same result using *own* variables.

Coroutines as described above are sometimes called explicit coroutines to distinguish them from the pseudo-parallel or implicit coroutine found in many simulation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

languages (see section H of this chapter). The property shared by the explicit and implicit coroutines is interleaved execution with only one process active at a time.

F. PARALLEL PROCESSING A parallel processing control structure is one in which several paths of control can be active at the same time. There have been a number of proposals for specifying parallel processing, most in the form of operators. It should be made clear at this point that when we speak of parallel processing (or any other control structure), we are referring to its virtual structure and not how it might be implemented. This is consistent with the view of parallel processing proposed by Gill [Gi 58]. In fact, he points out the conceptual similarity between time sharing (a users view) and parallel processing. Yet he defines time sharing as interleaved sequencing (the implementer's view) of independent programs to achieve better utilization of resources. Thus the control structure is dependent on our point of view; time sharing may be either a parallel or an interleaved control.

Conway [Co63b] introduced the *fork* operation which permitted parallel paths of control to be formed and the *join* operation which allowed a single control path to proceed only when all forked paths reach a common point. Operations which would establish parallel paths and later terminate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

them without rejoining were used in CL-II [CL 63]. Similar operations in the form of *fork* and *quit* were proposed by Dennis and Van Horn [DV 66] and in the form *start* and *halt* by Wirth [Wi 69]. Gosden [Go 66] distinguishes between these two kinds of forking, suggesting *and* when the paths rejoin and *also* when they do not. Most of these proposals have been for use in assembly and operating systems, and it has even been suggested that they be used in Algol [An 65].

These proposals have attempted to add parallel operators to a serial environment rather than forming a parallel environment. It has been shown [Fi 67] that the ability of processes to be processed in parallel is a nontransitive relation. That is, if W and X can be processed in parallel and so can X and Y, it does not follow that W and Y can be processed in parallel. When we assume that all processes (or statements) are sequential in the order given unless explicitly specified as parallel, then processes which can be executed in parallel can be given only by specifying every parallel pair of processes. More convenient notations such as the *do together* [Op 65] are inadequate to describe all parallelism. The *do together* allows a set of statements to be executed in parallel and then rejoined into a single path of control.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

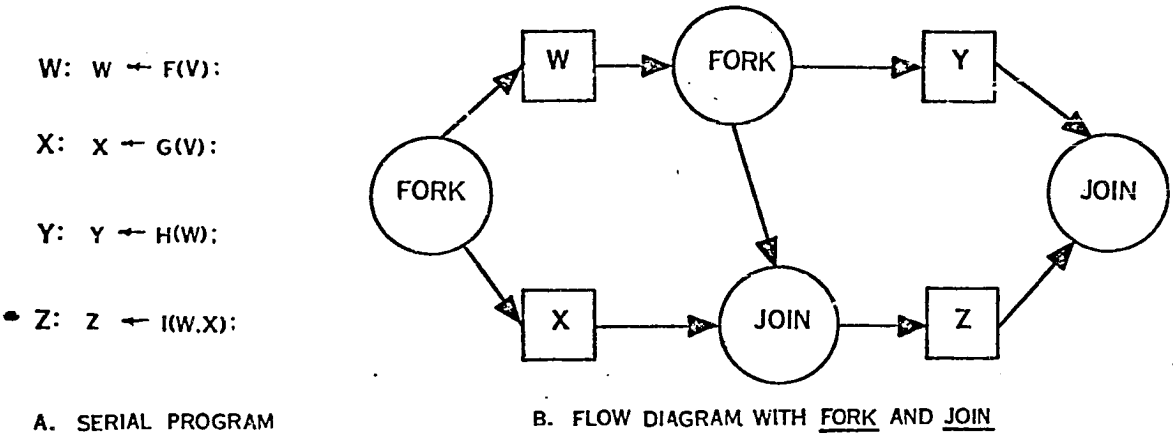

Figure II-A, nontransitivity of parallelism

Consider the program in figure II-Aa where v, w, x, y, and z and variables and f, g, h, and i are functions. The potential parallelism in this program can be given with forks and joins as shown in figure II-Ab, but it cannot be described with the *do together*. If W and X are done together then X and Y can be done together only by doing X together with the sequence (W,Y), but then Y and Z can be done together only by doing Z together with the sequence (W,Y), and this is impossible since W must precede Z. If it is assumed that all processes can be processed in parallel unless explicitly specified as sequential, then only a covering relation need be given because the requirement that processes be sequentially processed is a transitive relation.

The lack of success in the above efforts may be in part due to their attempting to provide a general facility for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

describing parallel processing rather than a parallel control structure appropriate to some task area. We know of only one language, Sol, with a parallel control structure, but there are a number of languages which achieve a potential for parallel processing through distributive operators which apply to multiple component data structures. Most notable of these is APL [Iv 62] which includes both vector and matrix operations.

Sol [KM64a] [KM64b] is a simulation language with parallel processes resembling Algol procedures. The processes are described independently and coordinated by *wait until* statements which suspend a process until given conditions (on global variables) occur and by *wait* statements which delay processes for given (simulated) time intervals.

A discussion of parallel processing would not be complete without mentioning some of the more unusual parallel machines. The most common parallel system configuration is the multiprocessor, i.e. several central processors with a shared storage. We will look at two other arrangements, the Solomon [SBM62] and the Holland [Ho 59] machines. The Solomon computer has a single sequential control unit, but the operations are distributed over an array of processing units. These processing units have their own local memory and can communicate with their four near neighbors. Operations are broadcast from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the control unit to the processing units so that the machine operates in a lock-step fashion with each processor either executing or ignoring the broadcast operation at each step. A Solomon machine with 64 processing units per control unit and some additional indexing capability within each processing unit is being built as Illiac-IV [Il 66]. The Holland machine also has processors arranged in an array, but instead of a central control each processor has its own control. Each processor in the Holland machine is very limited in both operations and storage so that more complex operations are built using several processors some which serve only as data transfer paths.

G. SYNCHRONIZATION AND MONITORING It is sometimes necessary for various control paths to communicate. Any mechanism which is used to guarantee harmony in a communication will be called synchronization. In well structured situations where the paths to be synchronized are known when the program is described the *join* can be used. In other cases the requirement is for mutual exclusion from data or critical sections of code which can be processed by only one control path at a time. One solution to this problem [DV 66] uses two operations, *lock* and *unlock*. The *lock* has one argument, a datum, and allows the processing to continue only when it successfully changes a lock bit associated with the datum from the reset to the set state. The operation is indivisible,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that is, if two *lock* operations are executed simultaneously then they will act as if one and then the other were executed. The *unlock* resets the bit.

Mutual exclusion occurs whenever only one process at a time can execute a given sequence of instructions. The associated sequence of instructions which can be executed by only one process at a time is called a critical section. We have already seen how mutual exclusion can be guaranteed using the *lock* and *unlock* operations. A mutual exclusion facility has been proposed for Algol [Wi 66] in the form of shared procedures. Only one instance of a shared procedure can exist at a time and any attempt to call a shared procedure when another call is in progress will be delayed until the other call is completed. Mutual exclusion is achieved in Sol [KM64a] [KM64b] by the *seize* and *release* operations which apply to facilities (i.e. SOL data structures) and guarantee that only one process has control of the facility between the *seize* and the corresponding *release*.

Mutual exclusion involves two mechanisms: indivisibility and nonbusy waiting. Indivisibility is necessary so that the availability of a critical section can be determined and marked "unavailable" before another process can even determine its availability. Dijkstra [Di65b] and Knuth [Kn 66] have given a programmatic solution to this problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

without using indivisible operations. Their algorithms require that each process cyclically test the availability of the critical section. These algorithms, however, lack the second mechanism required for mutual exclusion, the nonbusy wait. By a nonbusy wait is meant any mechanism which apparently cyclically tests a condition, but does so without drawing on the normal processing capabilities. Nonbusy waits are mechanized in two ways: by special dedicated processors (e.g. hardware fault interrupts) or by having the process which makes the condition true trigger resumption of the waiting process (this method is detailed in chapter III). The indivisability guarantees that only one process can simultaneously enter a critical section while the nonbusy wait guarantees that other processes waiting to enter the critical section will not waste processing capability.

The nonbusy wait is an example of monitoring, in this case for the availability of a critical section. Most machines provide a hardware monitoring facility in the form of interrupts. The conditions include abnormal arithmetic indications, erroneous machine conditions, and requests from peripheral and real time devices.

Operating systems often provide both monitoring and testing operations for completion of tasks and availability of resources. A difference between hardware interrupts

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and monitoring is their effect on other processes. When a hardware interrupt condition occurs, nonmonitoring processes are interrupted (i.e. suspended) while the interrupt process is executed. Monitoring on the other hand need have no direct effect on the process being monitored. Alternatively, hardware interrupts can be viewed as monitors with the additional property that their associated processes operate in a different time frame than other processes. In particular the process associated with an interrupt can be carried to completion between any two consecutive steps of other processes (this view is developed further in section II-L).

The semaphore [Ha 67] [Di68b] [Wi 69] offers a more elaborate mechanism for synchronization. Semaphores are variables which take on integer values. Two operations, *P* and *V* are the only operations applicable to semaphores. The operation *P(s)* decrements the value of semaphore s by one and permits processing to be continued only when the resulting value is nonnegative, otherwise the process waits until a *V(s)* is performed. *V(s)* increments the value of the semaphore by one and if the resulting value is non-positive resumes one of the processes halted by previous *P(s)* operations. Two processes can communicate by initilizing a semaphore to 0. The sending process executes a *P* operation, and the other responds with a *V*. For

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mutual exclusion a semaphore is associated with each critical section. At any given time, outside *P* or *V*, the semaphore will contain the difference between the number of processes which can simultaneously enter the critical section and the number currently in or attempting to enter the critical section. Each process performs a *P* operation before entering a critical section and a *V* upon exit.

H. PSEUDO-PARALLEL PROCESSING Discrete simulation languages [TL 66] provide an almost parallel control environment. The world to be simulated is viewed as either a collection of activities, logically distinct time consuming processes, or as a collection of events, instantaneous happenings corresponding to changes in the state of the system. The activities to be simulated are often concurrent, but the languages (except Sol) force the user to specify their interleaving explictly in a serial environment. Simula [DN 66], for example, has activities which take a form similar to Algol procedures. Activities can be created independently of their execution so that several can exist at the same (subroutine) level. Partially processed activities can be suspended and later resumed, but they cannot be run simultaneously. Instead they must be scheduled and managed in a gueue. No explicit operations are required for synchronization or mutual exclusion in this pseudo-parallel environment because these functions can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

achieved through appropriate queue management. In practice this means it is difficult to determine from a program where synchronization and mutual exclusion were required. Other discrete simulation languages have similar sequential job schedulers for either activities or events.

Like coroutines, a set of pseudo-parallel processes are executed in an interleaved manner, and thus have been called implicit coroutines. Recall that (explicit) co-routines were viewed as symmetric subroutines. Reynolds [Re 69] proposed a pseudo-parallel control in the form of label variables. A label variable holds a value which is a static copy of the state of a process. If the state of a process is assigned to a label variable then neither further execution of the process nor resumption of the copy which is the value of the label, will effect the content of the label. This can be contrasted with element variables in Simula which cannot be copied and hold the dynamic state of a process. Reynolds also shows that label variables can be used in an implementation of coroutines and finite path nondeterministic algorithms.

I. NONDETERMINISTIC PROGRAMS The solution of many problems can be represented as a search tree in which each terminal node is a potential solution. The task is to traverse this tree beginning at the root and to find a solution node.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ideally, a nondeterministic control structure would be a sequencing mechanism which would at each choice point (i.e. nonterminal node) always select the correct branch (or possibly all branches) which lead to solution nodes.

Floyd [Fl67a] uses three operations for nondeterministic algorithms. The *choice* function selects a value (ideally the correct value) from a given set. Algorithms are written which permit paths to be traversed leading to a failure or no solution. When such a path is identified, *failure* is executed; when a solution is found *success* is executed. The operations are mechanized using an exhaustive "depth first" search procedure called backtracking. The algorithm is executed making some one choice at each *choice* point. When a *failure* is encountered, the algorithm backs up, restoring all variables, to the last *choice* point for which all choices have not yet been exhausted and makes another choice. Note that this method does not guarantee a solution when the search tree is infinite.

Fikes [Fi 68] proposed a nondeterministic language which had a *select* function similar to *choice* and a *condition* statement which is used to give Boolean constraint expressions whose value must be true. This provides a convenient way of stating nonprocedural solutions to problems with complex constraints. A program to find two distinct digits, x and y,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

whose sum is fifteen might be given as:

condition (x←select(0,9))<(y←select(0,9)) & x+y=15;

This statement says, in effect, select an integer x in the range 0 to 9 and an integer y in the range 0 to 9 such that x is less than y and the sum of x and y is 15.

In Fikes mechanization, a variable is defined by each call of the *select* function. For each of these variables there is a list of the values over which the variable can range. The value returned by a call on *select* will be the name of the variable thereby defined. In the above example, the first call on *select* would create a variable S_1 whose range is the integers 0 through 9, and would return the name S_1 which would then be assigned to x. The other call on *select* would create a similar variable S_2 and return the name S_2 which would be assigned to y.

When a *condition* statement is encountered, the Boolean expression is evaluated. If the resulting value is true then no further action is taken. If the resulting value is false then no solution exists. If the value is an expression containing the names of variables, then that expression

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

becomes a constraint on those variables. In the example above, the value of the *condition* statement would be the expression:

$$S_1<S_2 \ \& \ S_1+S_2=15$$

Thus, the interpretation of a program produces a set of variables with their associated ranges and a set of Boolean constraints on those variables. Various problem solving methods can then be applied to deduce the values of the variables. For example, the constraint $S_1<S_2$ eliminates 9 from the range of S_1 and 0 from the range of S_2. The constraint $S_1+S_2=15$ restricts the range of both variables to the interval 6 through 9. A case analysis using the resulting ranges, range $(S_1)=6,7,8$ and range $(S_2)=6,7,8,9$, and the combined constraints yields the two solution pairs (6,9) and (7,8).

J. OTHER CONTROL STRUCTURES IN PROGRAMMING LANGUAGES Whenever the available languages are illsuited to a given task, new languages are invented. This usually means that additional data or control structures are required. If only a new type of data is required the new data could be embedded in an existent language. In general, however, new data are introduced because their structure is significantly different

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from what is available. Consequently, the sequencing mechanisms satisfactory for the old data will not be satisfactory for the new data. In algebraic languages iterative control is established by counting because the data (i.e. vectors and arrays) are of fixed size and addressed by position. Strings could be introduced into these languages, but the positional control which is easy to implement would be clumsy in use. With strings the interest is in manipulating patterns rather than integer counters. Languages like Snobol [FGP64] and Comit [Co 62] have strings as their primitive data structures. The control structure is such that one is always sequencing strings and making replacements dependent on and determined by the patterns satisfied by the substrings encountered. In list languages [Mc 60] the data are seen as having two components, the first list element and the rest of the list which is itself a list. Consequently, list languages tend to have a recursive control structure.

Landin [La 66] attempts to eliminate explicit sequencing (or at least unnecessary sequencing) in Iswim by using more structured statements. Primary among these is the *where* which has a statement as its left-hand argument and a definition local to that statement as its right-hand argument. The idea is to use expressions in preference to statements, to use *where* instead of local assignments, and to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

use multiple valued functions and parallel assignment statements in preference to multiple use of single valued functions and simple assignment statements. Judicious application of these rules yields programs without explicit sequencing or assignments. By eliminating sequencing which is not crucial to a computation Landin hopes to obtain more transparent programs. Although the goal was not the same, the CPL language [BBH63] has many of the Iswim constructs. There is a *where* clause which permits local variable definitions, a *result of* expression which permits local function definitions, and a simultaneous assignment. The simultaneous assignment has a list of variables as its left-hand arguments and a list of expressions as its right-hand arguments. The expression values are assigned to the variables on a one for one basis such that the assignments occur simultaneously. That is, all variables and expression values are determined before any assignments are made. The exchange of two variables a and b can be written as $a,b \leftarrow b,a$ (Interestingly, a procedure to exchange the values of two arbitrary variables cannot be written in Algol).

The first control structure examined above (section II-A) included a means to modity and compute instructions. In Formula Algol [PI 64] [PIS66] a symbolic representation of expressions can be manipulated so that expressions can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be manipulated so that expressions can not only be evaluated but be the object of a computation. This is useful for problems where mixed symbolic and numeric computations arise. The ability to mix the computation of expressions and their evaluation does not require a capability for modifying programs within the language. In fact, a capability for program modification is available in some list processing (e.g. Lisp) and string (e.g. Snobol) languages but not in Formula Algol. In Formula Algol programs cannot be modified or computed, but instead expressions (i.e. data structures) are computed as the values of *form* variables. These expressions can then be evaluated by the *eval* operator.

The generator as used in IPL-V [Ne 61] is another form of repetitive control structure. A generator is a process which produces a sequence of outputs and applies to each a specified process. The Algol *for* statement can be viewed as a generator in which the for list specifies the sequence of values to be generated and the body of the *for* statement is the specified process. The feature of the generator which distinguishes it from "equivalent" process composed from a conditional and a *go to* (e.g. the Algol descriptions of for list elements in section 4.6.4 of the Algol report [Na 63], is that the environments of the generator and the specified process are independent and not hierarchially defined. Both the generator and the specified process are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

inaccessible by the other. The sequences of Algol statements (in section 4.6.4 of the Algol report), which are purportedly equivalent to *for* statements, include labels which should be inaccessible within the body of the *for* statement. If one of the more liberal interpretations [Kn 67] is given to *for* statement (i.e. some of the arguments to a *for* statement are call by value), then local variables must be used to generate the successive values of the control variable, but these variables are also inaccessible from the body of the *for* statement. Generators can usually be terminated at any iteration: In IPL-V each iteration produces a Boolean value which if false terminates the generator, and in Algol a *for* statement can be terminated by a *go to* which exits the *for* body.

K. CONVERSATION AND OTHER NONPROGRAMMING CONTROL An attempt to identify and classify control structures should not be restricted to programming systems. It is quite possible that control relations which might be useful and desirable in programming systems have not been provided in current computing systems through limitations in hardware, software or both. We will look for control relations among the active components of the real world. Relations which have proven useful in the real world may also be useful in programming systems, and any attempt to model accurately the real world requires the facility to simulate these controls.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider the relation between two individuals in a conversation. With a cursory examination we might conclude that first one and then the other participant becomes active in an interleaved fashion where only one is active at a time. This is similar to the coroutine.

A closer look at a conversation reveals that although the speaking may occur in an interleaved fashion, the non-speaking participant monitors and processes the message concurrent with the other's speaking and indeed may interrupt it. Thus, we might view each participant as three process: a message generator, a message processor, and a monitoring process.

Let G, P and M comprise one participant in a conversation and g, p and m the other where G and g are message generators, P and p are message processors, and M and m are monitors. When G is active we expect that p will be active to interpret the message generated by G, that g will not be active, and since g is inactive that P will be inactive. Both M and m should be active monitoring G.

When G wishes to pass the conversation to g, G may resume P and suspend itself. Process m will then resume g upon discovering the abnormal condition on the message generated by G. The condition might be the termination of a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

question or the pause after G went inactive. Alternatively, we could argue that G only suspends itself and does not resume P, that P is resumed by M upon discovering output from g.

If M discovers an abnormal condition (such as an improper construction) when monitoring G, then M will interrupt G so that G can make another attempt. That is, M will suspend G and then resume G at a different point. On the other hand, if m discovers an abnormal condition when monitoring G, m cannot interrupt G directly, but will resume g.

We noted earlier that G and g are active in an interleaved fashion. Now we see that G and p, g and p, and P and p generally act in an interleaved fashion. G and p (or g and P) are active at the same time. This relation of simultaniety is a form of parallel routine. Since M and m are always active they are parallel routines relative to one another and relative to any other process whenever the other process is active.

The above description of a conversation points out the inadequacy of English for this purpose. Some sort of graphical representation is suggested. Several candidates have appeared in the literature (e.g. Petri's nets [Pe 66] or Dennis's program graphs [De 68]) but these devices were

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

designed to describe data flow rather than control flow. A graphical device for describing control flow will be introduced in chapter VI.

Another relation which does not occur in the conversational example, becomes apparent if we look at a manufacturing process. The relation between a machine and its operator, or between the operator and his foreman is one of subordination. One process is initiated by, performs a task for, and is terminated before a second process. If the subordinate process is performed within an inactive period of the process which created it (i.e. the operator is inactive after initiating a machine operation and until that operation is complete) then we have the concept of the subroutine. When this condition does not hold we have another example of parallel routines but with an additional subordination property. The above is a dynamic interpretation of subroutines. Alternatively one could take a static view and argue that the action of the machine operator is not a subroutine because it has a potential for parallel processing whether used or not.

Several of the control structures we have examined can be viewed as relations. By a relation we mean a predicate designating the properties which obtain only among two or more processes. We have identified a subordinate relation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in which a process exists for and during the course of another process, an interleaved relation in which one and then another of a group of processes becomes active but no two are active simultaneously, a concurrent relation in which processes are simultaneously active. We have seen these control structures in programming languages in the form of subroutines, coroutines, and parallel routines, which have properties of subordination, interleaving, and concurrency, but not in their most general form, probably because they were never isolated as processes.

L. SUMMARY OF CONTROL STRUCTURES We have seen a variety of control structures, some of which were viewed as time dependent relations between processes. A first step in better understanding these relations is to observe that they are in fact relations and not, as their names imply, catagories of routines or processes. The conventional view that the control structure can be described by monadic predicates such as subroutine, coroutine and parallel routine is not adequate. Our view proposes that the prefixes sub-, co- and parallel- describe relations among processes rather than a property of a single process.

Many control structures can be described by the relations which obtain among processes (dyadic or polyadic predicates), but the complexity and more importantly the (possibly data dependent) variations in these relations with respect to time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

makes this approach unattractive. This difficulty can be eliminated by describing the actions necessary to create the relations rather than the relations themselves. This approach is used to develop the multiple sequential control in chapter IV.

The multiple sequential environment is composed of processes each of which has a sequential control structure of its own and may have a variety of controls relative to the other processes. Although this environment is sufficient for describing most of the common control structures of programming languages and an unlimited variety of related control structures, there are a number of controls which do not fit this model. Monitoring, for example, requires a continuous operation which is not provided by sequential control.

A control which is closely related to monitoring is that a combinatorial (hardware circuit) logic. Here there are a number of simultaneous asynchronous processes which intuitively are continuously active and can be thought of as instantaneous relative to the steps of the sequential machine control. Two control structures are involved. One, the continuous processing control resembling that of an analogue computer, contrasts with the discrete step sequential control of the digital computer. The second

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coontrol is the instantaneous effect achieved by performing continuous operations between the steps of the sequential control. The relation between the analogue (i.e. parallel continuous) and the digital (i.e. discrete sequential) machine control has been discussed in the literature and techniques for automatically generating sequential representations of continuous parallel processes [Pa 67] and parallel representations of sequentially described processes [Be 66] [Fi 67] have been developed.

Petri [Pe 66] shows that if both the existance of an upper bound on the speed of signals and the existance of an upper bound on the density with which information can be stored are taken as postulates, then the theory of automata is incapable of representing the actual physical flow of information in the solution of a recursive problem. Instead he proposes a theory of discrete objects which can be combined only by finite techniques, and no metrics are assumed for either time or space but rather time is introduced as a strictly local relation between states. In particular, he introduces switching networks called nets in which the nodes are nonmetric switching elements and the connections between them, called locations, represent the state. Time is then only the temporal succession of changes in the (discrete) values of locations as imposed by the definitions of the switching elements. He goes on to show that communication

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(or synchronization) can be temporally established between arbitrary asynchronous automata.

It is convenient to view time as a continuum which is divided into discrete intervals corresponding to the steps of a computation. State changes within the course of a process occur only at the points of time which mark the end of these intervals. Other processes which appear to be instantaneous relative to a given process can then be introduced by choosing its state change points in such a way that they all lie on a single interval (i.e. between two consecutive state change points) of the given process. In light of Petri's work, however, the assumption of a time continuum is erroneous for real automata, and the process of choosing smaller and smaller intervals can not be repeated indefinitely. Thus we will take the alternative view that time is defined only by the sequence of state changes. A process A will be called continuous relative to another process B if and only if communication is established between A and B in such a way that state changes in B are temporarily delayed while the entire action of A is carried to completion.

We do not pretend to have covered all control structures or even all those of programming languages, but have attempted to illustrate the many varieties of control and the significance of variations in these for particular tasks. These

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

controls provide a representative sample which illustrates the limited facilities for altering and combining control in most current programming languages, the importance of control structures in revealing the transparency of programs, and the influence of data on the appropriate control for a task.

M. CONTROL DEFINITION FACILITIES Other than the work reported here we know of only one other investigation of control structures in programming languages. Leavenworth [Le 68] has proposed a method for programmers to define their own control structures. This is done by selecting a machine with a small set of state components as a base language. The user can then extend the control structure of this language through operations which allow him to modify the state directly. That is, the base language is described by means of the abstract machine which interprets the language and new control structures are introduced by writing functions which transform the state of this abstract machine.

The base language is similar to Landin's Iswim [La 66] and has only a conditional, recursion, the *where* clause (in the form: *let* $x=e$), and state accessing and installation functions as its primitive controls. Consequently, any new

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

control structure must be described in terms of recursive and conditional controls acting on data structures of stored machine states. This makes the language a convenient vehicle for describing scheduling algorithms and other implementation strategies for various control structures. However, a limited ability exists to give control descriptions which convey the intended interpretation of a control structure (e.g. a scheduling algorithm which has the same effect as a parallel control structure can be described but it will not convey a concept of concurrency).

Leavenworth's system is illustrated by a nondeterministic control which uses Floyd's backtrack primitives [Fl67a]. In contrast to Floyd's implementation, Leavenworth views the choice function as a generator of multiply defined machines. The description of the backtrack primitives then becomes a problem in simulating parallel computations on a sequential (or recursive) machine. The control structure thus obtained is a pseudo-parallel implementation of nondeterministic control.

In a more recent paper [Le 69], Leavenworth has extended these ideas and given several additional examples. The *go to* operation is implemented as an assignment to the state of the interpretative machine. Two loop (i.e. iterative) control structures, explicit and implicit coroutines, and nondeter-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ministic control are also described.

The one other language which deals directly with control is PPL [St 68] [St 69]. Several control mechanisms are proposed including parallel processing, continuously evaluating expressions (see chapter IV), control structure macros, interrupts, and traps. The specific means of incorporating these tools into the language are not entirely resolved. The general idea (excepting continuously evaluating expressions) is that the programming environment is composed of many sequential processes with a capability for controlling interfaces among them.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

III. A FORMAL DEVICE FOR DESCRIBING CONTROL

> "Our experience with the definition of functions should have told us what to do: not to concentrate on a complete set of defined functions at the level of general use, but to provide within the language the structures and control from which efficient definition and use of functions within programs would follow. — A.J.Perlis [Pe 67]

The control structure of a programming language is often an integral part of the environment provided by that language. A facility for describing control must include an ability to deal with the environmental structure of languages. The abstract machine which interprets programs of a language, provides a description of the environment (both data and control) for that language. Control operators could be described within a language in a manner similar to functions, but this does not enable one to get at the enclosing control regime imposed by the language. Instead there must be a way to modify the interpreter for the language, and this in turn requires a formal specification of the internal machine representation for programs of the language and of the machine which interprets those representations.

Any compiler of course provides a precise description of a language. Their purpose however is to specify an efficient implementation for executing programs of the language on some

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

given object machine. An implementation is the totality of things necessary to execute a program of a language or to process some control structure. The understanding of a control structure that one derives from an implementation is dependent on the control structure of the language in which the implementation is described. If an implementation is given in the assembly language of some machine, then it conveys a method for mechanizing the control on that and similar machines. If the implementation is given in a higher level language but in a language which has only sequential, subroutine, and conditional control, then the implementation may convey a strategy for mechanizing the control on any machine having those controls. There are however many useful and interesting properties of control structures which cannot be described in terms of the control structures available in most programming languages. If parallel processing is described through its inexact simulation in terms of sequential processing, subroutine, and conditional controls by means of some scheduling algorithm, then we might come to understand parallel processing as a particular interleaved execution of sequential processes and might learn how to implement parallel processing on a sequential machine, but the idea of concurrent execution would not be conveyed. Concurrency is a concept which cannot be composed from sequential primitives.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Our desire here is to provide a set of primitive control structures which include not only sequential processing but also concurrency, nonbusy waiting, and other control concepts which cannot be described in terms of sequential, subroutine, and conditional control. In fact, in order to describe these control operations we will have to use a language (i.e. English) in which these concepts are meaningful. We will however give another description of these control structures in terms of sequential control to show how they might be implemented on a sequential machine. By chosing primitive control structures which encompass ideas of concurrency, nonbusy waiting, and indivisibility, we hope to be able to give formal descriptions of other control structures which are clearer, more concise, and more closely resemble their descriptions in English than would their description in other formal languages which have only sequential, conditional and recursive control. We would like to give formal definitions of control structures which correspond to our intutive view of the control and without necessarily providing an implementation for the control on a sequential machine.

Control descriptions should provide only the information necessary to understand the control and should restrict the implementation on any real machine as little as possible. The efficiency of any implementation derived from the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

description should be limited only by the generality of the control structure which was described and the properties of the object machine. That is, the description of a control should be only as general or restrictive as the control being described and should not impose other constraints by virtue of the means of description. Implementations for particular classes of machines can of course be given within the same formal language by restricting the set of control primitives used for the description.

The desire to provide control descriptions which correspond to our intutive view of the control rather than descriptions which provides mechanism which achieve the same effects in other ways, does not rule out description by composition of more primitive concepts, but it does require that the set of primitives used in the descriptions not only span the space of control structures commonly found in programming languages but also include primitives which might be used in a natural language description.

All control structures must be available with in the description language either directly or by composition. In fact, because control descriptions will often take the form of an interpreter, the interpreters will be written in such a way that the dynamic control structure of the interpreter is always the same as the control structure of the program

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

being interpreted. If parallel paths are called for in a program then parallel paths will be called in its interpreter to execute the respective program paths. If a coroutine is called in a program, it will be interpreted by a coroutine within its interpreter. In short, when any control action is called for in a program, a similar action will be carried out as part of the interpretation process for that program.

In many cases the control structure is implicit in the syntax of the language and is unalterable by the user. Few control structures are provided in most languages and there is little capability for combining them to create other controls. This situation is desirable in problem oriented languages because it provides an environmental structure suitable to a particular task area and makes the constant parts of the control implicit in the notation (and in the implementation). Although the control structures available in a given language are few, the number of useful control structures is large. The task area before us, that of describing control, requires a means to combine controls and a set of primitive control structures which can be used in combination to describe any other control structure.

An interpreter will have the same control structure as the program it interprets, but will not use the same primi-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tives to construct those controls. In the program the control structure will appear as control operators or implicitly as part of the environmental structure, while in the interpreter they will be formed by composition of the control primitives.

The choice for the control primitives was indicated in chapter II where we saw only a few functions underlying control. 1) There must be means to specify a necessary chronological ordering among processes and 2) a means to specify that processes can be processed concurrently. There must be 3) a conditional for selecting alternatives, 4) a means to monitor (i.e. nonbusy waiting) for given conditions, 5) a means for making a process indivisible relative to other processes, and 6) a means for making the execution of a process continuous relative to other processes. A primitive control operation will be provided for each of the above six functions.

Is this set of control operations necessary or sufficient? This depends on what we mean by necessary and sufficient. If we take the conventional view that the semantics of a programming language is an implementation of that language, then we must conclude that not all of these operations are necessary. In fact, as will be shown in section III-I, all these controls can be implemented in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

terms of sequential, recursive, and conditional control. This view of semantics holds that each control structure is a black box having some effect on the outside world. To derive the semantics of a control from its description one first looks inside the black box to determine what its effect will be on the outside world and then takes those effects but not the manner in which they are achieved as the semantics.

Here we take an alternative view that both the effects and the manner in which they are achieved are important. This view requires that the primitives with which we describe a control structure not be restricted to some small set (e.g. sequential machine controls). Many aspects of the primitive control structures proposed here can be given in terms of sequential primitives, but the totality cannot. We see no obvious way to describe one of these primitive control operations in terms of the others without loss of some essential ingredient of their intended interpretation (e.g. primitives which are not described by programs). For example a language for numeric computation might take the sine function as a primitive. Certainly, it is understood that the operation will yield the sine of its argument to only some finite precision, but to define the sine by a power series approximation would only detract from the clarity of the language definition.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The system proposed here for describing control is a programming language and as such must not only have operations for composing control structures, but must also have data structures and operations, an enclosing environmental data and control structure, and a syntax. In each of these areas we have attempted to keep the language simple. The design follows.

A. DATA STRUCTURES AND OPERATIONS The data with which an interpreter must deal are representions of the programs it interprets. In addition it must be able to cope with the data structures (including the environmental data structure) of any language it interprets. The latter requirement imposes a need for a data definition facility within the language. Ideally this facility would include a capability for structure definition, for naming the structures and their component parts, the rules of combination, copy and erasure for both the structure and content. These goals were proposed in the first Turing lecture [Pe 67]. Probably the most extensive proposal along these lines is a thesis [St 67] in which descriptor formulae are used to describe data structures. These descriptors contain enough information to determine the predicates, selectors, constructors, and storage allocation policies for the data.

Because our primary concern is control and not data we

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

will take a simpler approach of providing a single but general composite data type which can be used to represent other data structures. These composite data will be called constructs. In addition we introduce a countably infinite class of elementary objects called atoms. These two classes of objects are similar to the composite and elementary objects respectively, as used in the Vienna report [LLS68]. Constructs are composed of parts which are themselves either constructs or atoms. The components of a construct are uniquely named by objects (in fact atoms) called selectors. Familiarity with the objects ([LLS68] chapter 2) used in the Vienna report is assumed and only the following two points of difference between those objects and the constructs used here are pointed out.

Constructs have an identify independent of their structure and consequently the concept of copy is meaningful in our system. Using the notation of the Vienna report, instead of the condition $(Vs)(s(A_1)=s(A_2)) \equiv A_1=A_2$ where A_1 and A_2 are constructs and the s's are selectors, we have the weaker condition $A_1=A_2 \supset (Vs)(s(A_1)=s(A_2))$. Thus objects correspond to structural equivalence classes of constructs. This distinction is made because in programming systems it is convenient to both copy objects and retain the identity of objects after their structure or content has been modified.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A direct consequent of distinguishing between a construct and its copy is that there is then no need to restrict constructs to finite nesting. The total number of components which comprise the substructure of a construct must be finite, but because constructs have an identity independent of their structure, a construct may appear as one of its own components. Successive application of selectors to a construct will not necessarily lead to an atom after some finite number of applications.

Several functions are defined on these data. A selection function, *select(s,c)* yields that part of construct c named by selector s or if none exists then the atom *SELECTERROR*. The constructor function $\mu_o(s_1,v_1, \ldots s_n,v_n)$ where $n \geq 0$, forms a new construct which has for each $0<i \leq n$, the value of v_i as a part and the value of s_i as the corresponding selector. The operation $\mu(c,s_1,v_1, \ldots s_n,v_n)$ where $n \geq 0$ allows incremental changes to be made to a construct. For $0<i \leq n$ the value of v_i is substituted for that part of construct c named by selector s_i, or if c had no part named s_i then part v_i is added to c with s_i as the selector. The value of the operation μ will be the construct c. Note that for all selectors s, constructs c, and objects v:

$$\text{select}(s,\mu(c,s,v)) \equiv v$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Atoms will be used as tokens to represent objects whose substructure is of no concern. There is a unique atom associated with each identifier of the concrete (external) syntax. Selectors must be atoms. The atoms for the identifiers *TRUE* and *FALSE* are taken as the Boolean values. The predicate *atom(x)* yields *TRUE* if the value of x is atomic and *FALSE* otherwise. Depending on whether x and y are the same objects or different the operation $\equiv(x,y)$ written $x \equiv y$ yields *TRUE* or *FALSE* respectively. The equivalence operation, $\equiv$, applies to constructs as well as atoms. Since it is an identity test $copy(x) \equiv x$ is *FALSE* for any construct x. The symbol $\not\equiv$ is used to designate the nonequivalence operation. Real numbers are also atoms. Other relational and arithmetic operations will be introduced as needed.

Programs can be represented (internally) as lists. Lists are data structures with two parts, a head which is any arbitrary object and a tail which is either itself a list or the terminator *NIL*. Lists will be represented as constructs with two parts named *H* and *T*. A list of two elements x and y can be formed by $\mu_o(H,x,T,\mu_o(H,y,T,NIL))$. The operations $select(H,x)$, $select(T,x)$, and $\mu_o(H,x,T,y)$ correspond respectively to the Lisp [Mc 60] operations *car(x)*, *cdr(x)*, and *cons(x,y)*. Programs will usually be represented as lists with an operation as the head and the list of arguments as the tail where arguments have the same form.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B. PRIMITIVE CONTROL OPERATIONS Throughout the control description language (hereafter called CDL) no distinction is made between statements and expressions, that is between parts of programs which are executed for the changes they make in the environment and those which are executed for the values they produce. There will be only one class called expressions which will always have a value (which may be ignored) and can have side effects (i.e. can cause change in the environmental context of the computation). Initially we will represent all programs in the functional form $f(x_1, x_2, \ldots x_n)$ where f is an operator and the x_i's are expressions for the arguments. Each program will be an expression which is either an operator followed by a list of arguments which are themselves expressions or an atom which references the environment.

Sequential processing. The sequential processing control operation, *seq*, causes its arguments to be evaluated in the order given from left to right in such a way that the evaluation of an argument will not begin until the evaluation of all arguments to its left is complete. The value of the right most argument will be taken as the value of the entire expression. The sequential processing operation will be used to indicate that the arguments must be evaluated in order of their left to right listing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parallel processing. The parallel processing control operation, *par*, causes its arguments to be evaluated independently (i.e. in parallel) as if each had its own processor. No assumptions are made about the relative speeds of these processors, so there are no guarantees concerning the chronological order of their side effects. The effects can however be controlled by synchronization of the parallel control paths. The value of the right most argument will be taken as the value of the entire expression and will be emitted after all arguments have been evaluated. The parallel processing operation will be used to indicate that the arguments can be processed concurrently; that the processing order is of no concern. If within any argument control is passed to another routine and not returned, the parallel operation will not be completed. The establishment of parallel paths has no effect on the environmental data structure of a program. In particular, copies of the environment are not made for each path, so that they must share the same global variables and define their own local variables.

As a general rule we will favor the use of the parallel processing operation over the sequential processing operation to avoid specifying an unnecessary order which can restrict the choice of implementation. On the other hand, we must be careful not to require a (pseudo) parallel implementation of parallel described controls when it is not indicated. If a control is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

described using the parallel operation but no synchronization arises among the various paths then the interpretation should be that the arguments can be evaluated concurrently, sequentially in any order, or in any interleaved fashion. Any order convenient to an efficient implementation is satisfactory.

The conditional. The conditional operation, *cond*, has an even number of arguments. Beginning with the left most argument, every other argument will be evaluated in order from left to right until one having value *TRUE* is encountered. The argument immediately to the right of the one with value *TRUE* will then be evaluated and its value taken as the value of the entire expression. If none of the odd numbered arguments has value *TRUE* then the value of the conditional will be *UNDEF* and none of the even arguments will be evaluated. With the exception of the default case this operation is the same as the Lisp conditional [Mc 60].

Monitoring. The monitoring operation, *monitor(s,c,r,v,exp)*, provides for nonbusy waiting. The left most four arguments will be evaluated in parallel and will be evaluated only once. The value of *s* should be a selector, the value of *c* a construct, the value of *r* a binary relational operator, and the value of *v* can be anything. The monitor operation has the effect of cyclically

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

testing the condition *r(select(s,c),v)*. When that condition becomes *TRUE* then the testing will be terminated and the expression *exp* will be evaluated. No assumptions are made about the relative speeds of the processor which evaluates the expression *exp* and the processor which evaluates the program which caused the condition to become *TRUE*. Any time the value of the condition becomes *TRUE* it will be detected by the monitor but the monitor may take arbitrary long time to respond. The value returned by a *monitor* operation is the process which is apparently cyclically testing the condition, and is returned as soon as the testing cycle (actually a nonbusy wait) is initiated.

There are two ways in which a monitoring process can be terminated: the value of the condition becomes *TRUE* or the operation *unmonitor(p)* is executed where *p* is the monitoring process. The expression which is the right-most argument to *monitor* will be evaluated if and only if the condition becomes *TRUE* before an *unmonitor* is executed.

Note that the *monitor* operation computes a value and then tests for the condition that that value is related to the value of a specified variable. The monitor operation will prove useful for describing interrupts and wait until conditions. If, for example, we wanted to monitor for the value of an arbitrary expression becoming *TRUE* the monitor operation could be used to detect all changes in the variables of the expression and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expression reevaluated only when the values of those variables change. One interesting generalization of monitors is the continuously evaluating expression discussed in chapter VI.

Synchronization. The operation *synch* provides a number of equivalent functions: synchronization of parallel or interleaved processes, mutual exclusion, and indivisibility of operations. The *synch* operation takes three arguments. The left most argument is evaluated first and its value must be a construct. The second argument is an expression which will be evaluated only when no other *synch* operation is evaluated its second argument for the same construct (i.e. the value of the first argument). The third argument is an expression which will be evaluated in all other cases. The value of the entire expression is the value of whichever argument is evaluated. If several *synch* operations are executed simultaneously then exactly one will have its second argument evaluated. In no case does the *synch* operation cause waiting.

The *synch* operation can be used for mutual exclusion to guarantee that only one process at a time executes a critical section of a program or simultaneously modifies a datum. *Synch* also provides for the relative indivisibility of operations because for all *synch* operations with a given value of their first argument, execution of the second argument cannot be simultaneous or interleaved, i.e. they are indivisible relative to the construct which is the value of the first argument.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Continuous processing. A process is continuous (see definition in section II-L) relative to a given process if and only if all actions of the process occur between two consecutive steps (i.e. state changes) of the given process. We will extend this definition in light of our parallel processing environment. The continuous processing control operation *cont(exp)* causes the evaluation of the expression *exp* to be continuous with respect to all control paths which are parallel to the control path which executes the *cont* operation (providing they themselves are not in the midst of a *cont* operation). Because continuous processing operations can be embedded, many levels of relative continuity can occur. These roughly approximate the priority levels found in some operating systems. An interrupt control can be formed by embedding a *monitor* within the argument for a *cont* operation. Continuously monitoring will be continuous (this is unnecessary because the monitor will detect any change in the condition regardless), but rather means that when the condition becomes *TRUE* the triggering of the evaluation of the monitor expression and the evaluation of that expression will be continuous.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C. SYNTAX It is at times convenient to refer to the various representations of programs and the states in the interpretation of programs. For this purpose the following definitions are made. A program is a data structure which describes the actions necessary to accomplish a given task. Programs may be represented in several forms. The two forms of concern here are the concrete syntax, the external physical embodiment as strings of symbols, and the abstract syntax, the representation as lists, atoms and constructs for processing by the interpreter.

A context-free grammar will be used to describe both the concrete syntax and the abstract syntax. This notation is formalized in chapter V. Here it will be presented by example. The notation for the concrete syntax differs from the Backus-Naur Form (BNF) used in the Algol-60 report [Na 63] only in that here strings of upper case letters represent terminals, and any character preceded by a $ represents that character as a terminal symbol. The productions:

```
sexp = id $( apl $)
     | pexp
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

say that a simple expression (sexp) is either the external representation of an identifier (id) followed by the external representation of an actual parameter list (apl) enclosed in parantheses, or is the external representation of a primitive expression (pexp). Within each alternative of a nonterminal, concatenation of the component parts is indicated by juxtaposition of their symbols.

The abstract syntax takes a similar form. The productions:

```
sexp = id apl
     | pexp
```

say that a simple expression is either a list whose head is the internal representation of an identifier and whose tail is the internal representation of an actual parameter list, or is the internal representation of a primitive expression. Within each alternative of a nonterminal, list formation with the component parts is indicated by juxtaposition of their symbols. Strictly, the right most symbol of an alternative is taken as the tail of the list formed from the other symbols in the order given. If an alternative contains only one symbol then no list is formed.

By combining the alternatives in the two grammars on a one for one basis, their correspondence can be specified. The first alternative of:

```
sexp = id $( apl $)   # id apl
     | pexp           # pexp
```

says that the internal representation corresponding to any

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

simple expression of the form identifier followed by an actual parameter list enclosed in parantheses, is the list having the internal representation of that identifier as its head and having the internal representation of that actual parameter list as its tail. In the concrete syntax the nonterminals id, capid, and symbol are assumed to represent lower case identifiers, upper case identifiers, and special characters respectively. In the abstract syntax they represent the corresponding atoms. The nonterminal empty represents the null string in the concrete syntax and has no meaning in the abstract syntax. In the abstract syntax no distinction is made between id and capid.

The initial syntax for CDL programs is given in figure III-A. The same notation will be used when introducing syntactic extensions and when describing the syntax of other languages.* Programs have a recursive form with each level either an atom or an operator followed by a list of arguments where the operator and arguments are themselves expressions. An atomic expression indicates a reference to the part of the environment with that name. The inclusion of a colon (:) after an actual parameter indicates that the value of that parameter is a list and that each element of that list is to be treated as an actual parameter in the order given. That is, if the value of u is a list of three elements a, b, c and the values of x, y, and z are respectively a, b, and c then for most operations f :

---

* When the concrete and abstract syntax specifications for an alternative are identical, the # is omitted and only one of the specifications is given.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

f(u:) ≡ f(x,y,z). Exceptions arise because some operations (including most control operations) act on the values of those expressions. Binary infix operators have Iversonian [Iv 62] precedence (i.e. they have no hierarchy and are right associative).

```
program = exp
exp = exp2
   | exp2 biop exp          # biop exp2 exp NIL
exp2 = sexp
sexp = pexp
   | pexp $( apl $)         # pexp apl
   | pexp $[ apl $]         # pexp apl
pexp = id
   | $( exp $)              # exp
apl = exp                   # exp NIL
   | exp $:                 # $: exp NIL
   | exp separator apl      # exp apl
   | exp $: separator apl   # $: exp apl
separator = $, | $;
biop = relop | $+ | $- | $× | $/ | ...
relop = $≡ | $≢ | $< | $> |  ...
```

Figure III-A, initial syntax.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

D. ENVIRONMENTAL DATA STRUCTURE Convenience and clarity in writing programs is the main concern in picking the environmental data structure. Here an Algol-like structure in which local definitions can be made to override any global definition for the same selector was chosen. The scopes of identifiers (selectors) are as in Algol-60 [Na 63] determined by the lexical structure of programs and not by their dynamic nesting (i.e. routine call structure). Each local environment will be a construct in which each identifier there defined names a part. One additional part, the immediately global environment, will be included in each local environment. The immediately global environment is the local environment at the time the now local environment was created, and may be selected with the name *GLOBAL*. The operation *ref(s,env)* selects the most local occurrence of selector *s* in the environment *env*.

```
exp = exp2 WHERE sdef                  # WHERE exp2 sdef NIL
    | exp2 WHERE $[ deflist $]         # WHERE exp2 deflist
deflist = sdef                         # sdef NIL
    | sdef $; deflist                  # sdef deflist
sdef = id $= exp                       # VAR id exp NIL
```

Figure III-B, syntax for scope.

The operation *where(e,d)* is used to specify the scope of local environments. The *where* operation establishes a new local environment containing the parts defined in *d* and then evaluates the expression *e* in the context of that environment. The *d* is a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

list of definitions. One form of definition is *var(s,e)* which defines the name *s* (a variable) to have an initial value which is the value of expression *e*. Notationally we borrow from Iswim [La 66] and use the form *e* <u>*where*</u> *d* to mean *where(e,d)*. Syntax extensions for *where* and variable definitions are shown in figure III-B.

E. <u>ENVIRONMENTAL CONTROL STRUCTURE</u> The environmental control structure is in the form of processes. A <u>process</u> is a dynamic instance of a routine. A static picture of the progress of a control path within a process is recorded in its <u>activation record</u>. The activation record has two parts, the environment in which the process is operating and the state of progress of a control path within the process.

The evaluation of each operation constitutes a new process. Consequently, as each operation is encountered within a process, further processing of a control path within the process is suspended while the operation is performed. If that operation is defined by a program then the process which is the dynamic instance of that program to perform this operation will have in its global environment a part named *CALLER* which is an activation record describing the suspended state of the control path in the calling process. When the operation is completed, control is returned (i.e. transfered) to the calling process by the operation *return(v,ar)* where the value of *v* becomes the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

value of the called operation and *ar* is the activation record for the calling process. References to the parts of an activation record yield the current value of the parts of the local environment of the associated process.

F. EXTENSION FACILITIES The first extension facility permits the definition of new functional operations in the form of routines. A routine, like a procedure or subroutine in most programming languages, represents a class of processes all having the same programmatic description. When a routine is called, a process is created to carry out the actions of that routine in the context of a given set of values, the actual parameters. A routine definition has the form routine *routine(f,x, e)* where *f* is an atom which names the operation, *x* is a list of formal parameters (i.e. selectors used in the local environment of any process for the routine to name the values of the actual parameters), and *e* is an expression called the body of the routine. The actual parameters of a call on a routine are evaluated in parallel and then associated with the formal parameter names to form the local environment in which the body of the routine will be evaluated. The pairing of the formal parameter names with the values of the actual parameters is as follows. Beginning with the left most formal parameter name and the left most actual parameter value and moving to the right, they are paired on a one for one basis. To accommodate routines with variable numbers of parameters, the left most formal parameter name may be followed by a colon (:) to indicate that the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

remaining (after the above pairing actual parameters are to be formed into a list and that list paired with the right most formal parameter name.

Routines are called like subroutines, that is the calling process is suspended, a process which is an instance of the routine is created, the values of the actual parameters of the call are placed in the local environment of the process to be referenced by the corresponding formal parameter name, and the execution of the new process is begun. The termination of the process however can be controlled by the *return* operation. For example, the process for a routine may act as a subroutine by returning to its caller, as a coroutine by calling a dummy routine which then returns to the processes caller, or as a parallel routine by establishing parallel paths, one which returns and one which continues processing. When the evaluation of the body of a routine is completed without encountering a *return* operation, the associated control path is terminated and no return is made. Because the most common form of return will be to the caller with the value of the expression for the body as value (i.e. a subroutine return) this form is given an abbreviated representation in the concrete syntax. The syntax for routine definitions is given in figure III-C.

The second extention facility permits changes to the interpretation rules for programs. It permits changes to the environmental control structure by defining changes in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

machine which interprets the language. When an expression is processed by the interpreter, the interpreter must evaluate and examine the operator to determine the interpretation of the arguments. For functional operations the arguments will be evaluated and the appropriate operation applied to the resulting values. For control and other environmental operations unusual interpretations may be given to some of the arguments, they may not be evaluated (e.g. some of the arguments for the *cond* and *synch* operations) or they may be evaluated according to special rules (e.g. the first four arguments to the *monitor* operation and all arguments to the *where* operation).

To specify the interpretation of an operation as if the specification were part of the underlying interpreter, three things are necessary: we must be able to reference the expressions for the actual parameters of the call, to reference the local environment of the call, and to call upon the underlying CDL interpreter as if it were a routine. Formal parameter names written in uppercase letters will be used to indicate that it is to be paired with the expression for the actual parameter rather than the value of the actual parameter and that the actual parameter is not to be evaluated. The local environment of the process which called any routine can be obtained by referencing the variable *CALLER*. The operation *eval(exp,e)* is defined and has the same effect as executing the expression *exp* in the context of environment *e* (a routine for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the *eval* function is given in section III-H). We can then contract (i.e. write down the terms and conditions) for the interpretation or evaluation rules which are to be applicable for some given scope, and then continue programming within the context of those rules. Thus, a capability for contract control structures which can be defined for a given context and then used as the environmental control structure throughout that scope is provided. Control extensions also allow language extensions for particular task areas to have not only a concrete syntax which makes the constant control for that task implicit, but to have an interpreter unique to the task and to take advantage of the constant features. This can be contrasted with macro extensions which allow implicit external specifications but do not permit the economies of constancy to be realized in the interpretation.

There are a few system variables which are useful in describing control structures and can be referenced as global variables by any expression. The selector *PATHS* names the number of nonterminated control paths currently in the system, *ACTIVE* names the number of active (i.e. nonmonitoring) control paths, and *SELF* names the current local environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
sdef = id $( fpl $) body        # ROUTINE id fpl body NIL
     | id $( $) body            # ROUTINE id NIL body NIL
fpl = fpname                    # fpname NIL
     | fpname $:                # $: fpname NIL
     | fpname $, fpl            # fpname fpl
body = $= $= exp                # exp
     | $= exp                   # RETURN exp CALLER NIL
fpname = id
     | capid                    # capid NIL
pexp = $[ condlist $]           # COND condlist
     | capid                    # QUOTE capid NIL
     | $$ symbol                # QUOTE symbol NIL
condlist = exp2 $→ exp          # exp2 exp NIL
     | exp2 $→ exp $; condlist  # exp2 exp condlist
```

Figure III-C, additional syntax.

Additional primitive operations which have not been defined are *quote(x)* which returns its (unevaluated) argument as its value and *copy(x)* which makes a copy of the value of *x*. If the value of *x* is a construct then the value of *copy(x)* will be a new construct having the same named parts as does the value of *x*. If the value of *x* is an atom the value of *copy(x)* will be that same atom. Syntax specifications for *quote* and a convenient notation for conditional expressions (borrowed from Lisp) are also given in figure III-C.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The syntax specifications are not a part of the CDL, but are made in a metalanguage and are used either to define the internal representation of programs so that an interpreter can be described, or to extend the concrete syntax of the CDL for clarity or convenience in writing control descriptions. However, there is in chapter V, an interpreter for the syntax specifications which could be used to make the syntax specifications into an extension facility within the language.

G. SOME EXAMPLES A few examples of routines which will be useful later are given to illustrate the language and particularly the control extension facilities. The operation *ref(s,env)* which is used to reference the most local occurrence of selector *s* in the environment *env* has already been defined. A related operation *asgn(s,env,v)* which replaces that value is also useful. A definition of *asgn* as a routine follows:

```
asgn(s,env,v) = [select(s,env)≠SELECTERROR → par[μ(env,s,v); v];
    select(GLOBAL,env)≠SELECTERROR →
                    asgn(s,select(GLOBAL,env),v)];
```

The following notation will be used for *ref* and *asgn*:

```
exp = sexp $• exp2 $← exp # ASGN sexp exp2 exp NIL
    | sexp $← exp         # ASGN sexp SELF exp NIL
exp2 = sexp $• exp2       # REF sexp exp2 NIL
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The *ref* operator (•) has higher precedence than the other binary operators, but less than functional application. The "•" will be read as "of" and can be used to form a compound selector. The expression $H \bullet T \bullet T \bullet x$, for example, is read H of T of T of x and references the third element of list x.

A list of an arbitraty number of elements can be built using the routine *list* with the elements as arguments.

```
list(x:) = x;
```

The routine *cons* forms a list of the values of the given arguments, but with the value of the last argument as the tail of the list. This is a generalization of the *cons* operation of Lisp [Mc 60] such that $cons(x) \equiv x$ and $cons(x_1, x_2, \ldots x_n) \equiv cons(x_1, cons(x_2, \ldots x_n)\ldots))$ where $n \geq 2$. The definition follows.

```
cons(x,y:) = [y≡NIL → x; TRUE → μ₀(H,x,T,cons(y:))];
```

The routine *not* produces the complement of a Boolean value.

```
not(x) = [x → FALSE: TRUE → TRUE];
```

We now describe some nonprimitive control operations. The parallel distribution control operation $pdist(exp, id, x)$ creates a list from the successive values of the expression *exp* applied to each element of the list which is the value of x. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

identifier *id* is local to each evaluation of *exp* and has as its value the associated element of list *x*.

```
pdist(EXP,ID,x) = evpd(x) where [ e=caller;
     evpd(x)=[x≡NIL → NIL; TRUE →
        cons(eval(exp,μo(GLOBAL,e,id,H•x)),evpd(T•x))]];
```

The parallel application control operation *papp(exp,id,x)* is similar but does not form a list. The value of *papp* is the value of *exp* applied to the right most element of the list which is the value of *x*.

```
papp(EXP,ID,x) = evpa(x) where [ e=caller;
     evpa(x)=[x≡NIL → UNDEF;
        T•x≡NIL → eval(exp,μo(GLOBAL,e,id,H•x)); TRUE →
            par[eval(exp,μo(GLOBAL,e,id,H•x)); evpa(T•x)]]];
```

An operation for sequential application *sapp(exp,id,x)* is defined below. Note that the only difference between the *papp* and *sapp* operations is the single occurrences of *par* and *sar*. The *sapp* operation is similar to the IPL-V generator (see section II-J) except that *sapp* lacks the capability for termination at each iteration.

```
sapp(EXP,ID,x) = evsa(x) where [ e=caller;
     evsa(x)=[x≡NIL → UNDEF;
        T•x≡NIL → eval(exp,μo(GLOBAL,e,id,H•x)); TRUE →
            seq[eval(exp,μo(GLOBAL,e,id,H•x)); evsa(T•x)]]];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An iterative loop expression of the form *until c do exp* will cause the expression *exp* to be executed repetitively as long as the value of *c* is *TRUE*. The expression *c* will be evaluated before each iteration and will not be evaluated until the previous iteration (if there is a previous) is completed. Because *untildo* tests before each iteration it is similar to the *for while* of Algol. A monitoring expression of the form *when* $s \cdot c \; \Phi \; e_1$ *then* e_2 where Φ is a relational operator will yield the value of e_2 but only after the value of the condition $s \cdot c \Phi e_1$ becomes *TRUE*. The expressions s, c, e_1, and e_2 will each be evaluated only once.

```
untildo(C,EXP) = evud() where [e=caller;
     evud()=[eval(c,e)→UNDEF; TRUE→seq[eval(exp,e); evud()]]];
when(s,c,RELOP,v,EXP) ==
     monitor(s,c,relop,v,return(eval(exp,caller),caller));
```

The syntax for *untildo* and *when* is given below:

```
exp = UNTIL exp1 DO exp2 # UNTILDO exp1 exp2 NIL
      WHEN sexp $• exp2 relop exp1 THEN exp2
                  # WHEN sexp exp2 relop exp1 exp2 NIL
```

Two other list operations which will be useful are *member(x,y)* which looks for an occurrence of x on list y and *push(x,y)* which pushes x onto the head of y. The latter

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operation is similar to *cons* except that after a *push* operation *y* will reference the new list while after a *cons*, *y* referenced the old list. *Member* and *push* can be described as routines.

```
member(x,y) = [y≡NIL→FALSE; x≡H•y→TRUE; TRUE→member(x,T•y)];
push(x,y) = μ(y,H,x,T,cons(z,T•y)) where z=H•y;
```

H. THE INTERPRETER The routine *eval* is an interpreter for any program of the CDL; *eval(exp,env)* evaluates the expression in the context of the environment *env* according to the current extension of the CDL interpretation rules. Thus for any expression the value of *eval(e,self)* will be the same as the value of *e* regardless of the current evaluation rules. *Eval* is given as a routine in figure III-D.

If the expression *exp* is atomic then it is used as a selector to reference the environment *env*. Otherwise it is split into two parts, its head which is an operation and its tail which is a list of expressions for the arguments. The operation is evaluated and called *f*, the arguments are unraveled (i.e. each argument which represents a sequence of arguments is elevated so that each of its elements becomes an element of the argument list) and called *x*. The unraveling produces a list of the expressions for the actual parameters, having each actual parameter which was not followed by a colon

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
eval(exp,e) = [atom(exp) → exp•e;
TRUE → [atom(f) → [f≡SEQ → sapp[eval(y,e);y;x];
         f≡COND → evcon(x)
             where evcon(x)=[x≡NIL→UNDEF; eval(H•x,e)→eval(H•T•x,e); TRUE→evcon(T•T•x)];
         f≡MONITOR → monitor(eval(H•x,e); eval(H•T•x,e);
                  [rel≡UNDEF→H•T•T•x; TRUE→rel] where rel = eval(H•T•T•x,e);
                  eval(H•T•T•T•x,e); eval(H•T•T•T•T•x,e));
         f≡SYNCH → synch(eval(H•x,e),eval(H•T•x,e),eval(H•T•T•x,e));
         f≡CONT → cont(eval(H•x,e));
         f≡RETURN → return(eval(H•x,e),STATE•eval(H•T•x,e));
         f≡WHERE → eval(H•x,papp[evdef(d,lenv);d;T•x] where lenv=μ₀(GLOBAL,e));
         f≡QUOTE → H•x;
         TRUE → f(pdist[eval(y,e);y;x]:)];
     H•f≡ROUTINE → interp(H•T•T•T•f,e,pp(H•T•T•f,x,e,μ₀(GLOBAL,H•T•f)))]
     where [ f = [y≡UNDEF→H•exp; TRUE→y] where y=eval(H•exp,e);
             x = ravel(T•exp,e)]]
```

Figure III-D, the interpreter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as an element. Arguments followed by colon stand for several arguments and are evaluated, each element of the resulting list is quoted, and the quoted values are taken as elements of the list of actual parameter expressions. The value of *f* should be either an atom naming a primitive operation or a list which represents a routine.

If *f* is an atom it is compared to each of the selectors for the primitive operations which have special interpretations for their arguments. These include only *seq, cond, monitor, synch, cont, return, where* and *quote*. If one of these is found then the arguments are treated accordingly. For the other primitive operations the arguments are evaluated in the context of the calling process and used as actual parameters to call the operation within the context of the interpreter.

If *f* is nonatomic then it must be a routine. Routines are lists of four elements: *ROUTINE, env, fpl,* and *exp* where *env* is the local environment in which the routine was defined, *fpl* is the list of formal parameter names, and *exp* is the expression for the body of the routine. A new local environment is formed and the expression for the body of the routine is evaluated in the context of that environment. The local environment contains an activation record for the caller, and a global environment which is the environment of the routine definition. The remainder of the local environment consists of the actual parameters named by the formal parameter names.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
where [ ravel(x,e) = [x≡NIL → NIL;
        H•x≠$: → cons(H•x,ravel(T•x,e));
        TRUE → cons(pdist[list(QUOTE,y);y;eval(H•T•x,e)]:,ravel(T•T•x,e))];
    evdef(d,e) = μ(e,H•T•d,[H•d≡VAR → eval(H•T•T•d,GLOBAL•e));
                            H•d≡ROUTINE → cons(ROUTINE,e,T•T•d)]);
    pp(fpl,apl,e,lenv) = [fpl≡NIL → lenv;
        H•fpl≡$: → [atom(H•T•fpl) → μ(lenv,H•T•fpl,pdist[eval(y,e);y;apl]);
                    TRUE → μ(lenv,H•H•T•fpl,apl)];
        TRUE → par[pp(T•fpl,T•apl,lenv);
                   [atom(H•fpl) → μ(lenv,H•fpl,eval(H•apl,e));
                    TRUE → μ(lenv,H•H•fpl,H•apl)]]];
    interp(exp,e,lenv) == eval(exp,μ(lenv,CALLER,μ₀(GLOBAL,e,STATE,caller)))];
```

Figure III-E, auxiliary routines.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Four auxiliary routines: *ravel*, *evdef*, *pp*, and *interp*, are used in defining the interpreter. These are given if figure III-E. The routine *ravel(x,e)* evaluates, unravels, and quotes any expression on list *x* which represents several actual parameters. The routine *evdef(d,e)* makes a single definition in the local environment *e* where *d* may be either of the definition types, *VAR* for variables or *ROUTINE* for routines. The expression for the initial value of a variable is evaluated at the time of definition in the context of the global environment of the definition. The body of a routine is saved together with the local environment *e*, so that at each call on the routine, the body can be evaluated with *e* as the global environment. The syntax for definitions is given in figure III-B and III-C. The parameter passing routine *pp* pairs the values of the actual parameters *apl* with the formal parameter names *fpl* in the given local environment *lenv* of a process and evaluates the actual parameter when indicated. The *interp* routine is used to form the activation record for the caller whenever a routine is called. The activation record has two parts, the local environment of the caller and the state (i.e. activation record) of the interpreter for the caller. Note that *interp* does not return to its caller, this is because the return will be made by *eval* when a *return* operation is encountered (see figure III-D).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The description of the interpreter as a routine is intended to clarify the details of the interpretation of programs in the control description language and to illustrate how interpreters are described within the language. The definition of *eval* also shows the relation between control structure definitions and the interpreter. For example, a parallel conditional operation which evaluates the first of each pair of arguments in parallel and takes as the value of the expression the value of the second argument of the pair whose first argument was *TRUE*, could be added to the *eval* routine by adding another case when *f* is atomic (see figure III-D). The code to be inserted might appear as:

```
f≡PCOND → seq[papp[[eval(H•y,CALLER)
      V←eval(H•T•y,CALLER)];y;x]; v] where v=UNDEF;
```

On the other hand, this same operation can be made available within the language, for any given scope, and without modifying the interpreter by the following definition:

```
pcond(X:) = seq[papp[[eval(H•y,e)
      V←eval(H•T•y,e)];y;x]; v] where v=UNDEF;
```

Thus it is seen that the effect of a control definition is to temporarily add a new primitive operation in the underlying interpreter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I. IMPLEMENTATION STRATEGY FOR SEQUENTIAL MACHINE The control description language can be of practical value for experimenting with control structures only if it is implemented on some real machine, that is, it has a description in the language of some sequential machine. We have given a description of the interpreter (section III-H) which could be transliterated into any given language providing the given language has the data and control structures used in the CDL description of the interpreter. What is needed then is a description of the CDL data and control structures in terms of the data and control of the given sequential language.

The CDL data structures can be readily implemented with lists. Using the terminology of Lisp, a CDL construct can be represented as either an association list (i.e. a list of pairs) or as a property list (i.e. a list on which every other element is an attribute (or selector) and the other elements are the associated values). Many languages have lists as primitive data structures and in any case methods for their implementation on sequential machines are well known (see [Mc 60] or [Ne 61]).

Each of the primitive control operations of the CDL must be described in terms of the control structure of a sequential language. This will be done by assuming a sequential language which is syntactically identical to CDL but has only sequential control structure. This language will then be extended to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

include routines which have the same effect as the primitive CDL control structures. To be more explicit we assume that the given language has the sequential processing control operation *seq*, a conditional, recursive subroutines, and a *return* operation. It is assumed that there can be only one path of control at a time, but the explicit use of the *return* means that is not necessary for control to pass among routines in a strictly hierarchical manner. Consequently local storage for all routines cannot be allocated in a single stack. Finally, we assume that the processes (activation records for routines) can be treated as data (otherwise the *return* operation would be meaningless). In particular the selector *CALLER* will be used to reference the process which called the process in which *CALLER* is used. The sequential implementation follows.

Two global variables *queue* and *n* will be defined. *Queue* is a list which is used to hold those processes which would be currently running were it not that there can be only one path of control at a time. Each element of the *queue* is a list of the form *(m,p)* where *p* is a routine which has made a call but not yet been returned to, and *m* is the number of levels of the continuous processing operation, *cont*, in which the call was embedded. The *queue* will be arranged so that entries at the greatest level of *cont* embedding will be at the top. The variable *n* holds the number of levels of continuous operations in which the currently running process is embedded. The system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

variables *PATHS* and *ACTIVE* will also be kept as global variables of the interpreter.

A couple of routines for managing the queue will be useful. The routine *q(m,p)* will be used to queue the process *p* at *m* levels of *cont* operation embedding. Among processes with the same value for *m* the queuing order will be first-in-first-out. The *q* routine is given below.

```
q(m,p) = seq[ ACTIVE←active+1; PATHS←paths+1;
              QUEUE←q2(list(m,p),queue)]
    where q2(x,q) = [q≡NIL → list(x);
        H•H•q<H•x → cons(x,q);
        TRUE → cons(H•q,q2(x,T•q))];
```

The *unq* routine removes the first entry from the head of the queue and returns control to its process. If the queue is empty then all processing is complete.

```
unq() == [queue≠NIL → seq[ACTIVE←active-1; PATHS←paths-1;
         N←H•H•queue; QUEUE←T•queue; return(UNDEF,p)]
    where p=H•T•H•queue ];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parallel processing. A sequential implementation can now be given for each of the CDL control primitives. The first operation is the parallel processing operation *par*. The arguments to *par* may be evaluated in any order, but if any one of them is temporarily suspended (e.g. by execution of a *monitor* operation) that suspension should not prevent the continued evaluation of the other arguments. Thus, the routine will sequentially queue up processes to evaluate each of its arguments. These processes will share access to four variables *wl*, *vl*, *rtns* and *p*. A list of the evaluators for the arguments which have been queued but not yet returned a value is kept in *wl*. A list of the values returned by the right most argument is kept in *vl*. The number of returns to be made from *par* is kept in *rtns*; this will normally be one unless some arguments return more than one value. The variable *p* contains the process which called *par*.

The routine *evpar* (see definition below) acts as the evaluator for each *par* argument. Each call on *evpar* sequentially: 1) adds itself to the list of evaluators *wl*, 2) for the right most argument the argument is evaluated and added to the value list *vl* or for any other argument the evaluation of the argument is delayed (i.e. queued for later processing) while *evpar* is applied to the remaining arguments, 3) when the evaluation of an argument returns a value then its evaluator is removed from *wl* or if it is not on *wl* (i.e. this was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not the first value returned by the argument) then the number of *par* returns *rtns* is incremented, 4) if all arguments have returned at least one value then *rtns* returns are made to the process *p* in such a way that every value returned by the right most argument is returned to *p* at least once. The *fork* routine is used to create pseudo-parallel control paths in a sequential environment; it sequentially: 1) queues its caller for later processing, 2) evaluates its argument in the context of its caller, and 3) continues the processing of a queued process.

```
par(X:)==evpar(x) where
     [ wl=NIL; vl=NIL; rtns=1; p=caller;
     evpar(x)==seq[ WL←cons(self,wl);
          [T•x≡NIL → VL←cons(eval(H•x,p),vl);
           TRUE → seq[fork[evpar(T•x)]; eval(H•x,p)]];
          WL←par2(wl) where par2(y)=
               [y≡NIL → seq[RTNS←rtns+1; NIL]; H•y≡self → T•y;
                TRUE → cons(H•y,par2(T•y))];
          [wl≡NIL → until rtns≡0 do
               seq[RTNS←rtns-1; [T•vl≠NIL → VL←T•vl];
                   fork[return(val,p)]] where val=H•vl;
          unq( )];
     fork(EXP)==seq[q(n,caller); eval(exp,caller); unq( )]];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Monitoring. The monitoring operation *monitor(s,c,r,v,exp)* provides a capability for nonbusy waiting. The evaluation of an expression *exp* is initiated when the value of a given variable *s•c* is related to a value *v* by the relation *r*. The *monitor* operation can be implemented by associating a construct of monitor processes with each construct which is not local to the interpreter (i.e. which can be monitored by the user). The monitor list will be a part of the monitored construct and can be referenced by the selector *MONPART*. For each part *s* of a construct *c* which is being monitored the value of *s•MONPART•c* will be a list of the processes monitoring the value of *s•c*. The *monitor* routine, providing the condition *r(s•c,v)* is not already *TRUE*, will add a new monitor process to the list *s•MONPART•c*. All monitor processes on a list *s•MONPART•c* will be executed whenever the value of *s•c* changes.

The routine *monitor(s,c,r,v,exp)* evaluates the arguments *s,c,r,* and *v* only once. If the condition *r(s•c,v)* is *TRUE*, then the process which called *monitor* is queued for later processing, the expression *exp* is evaluated in the context of the process which called *monitor*, and then control is passed to another queued process. If the condition *r(s•c,v)* is not *TRUE* then a monitor process is created and added to the list at *s•MONPART•c* and a list of *s,c,* and the monitor process is returned as the value of the *monitor* routine. The monitor process is represented by a list of the form *(r,v,exp,e,m)* where *e* is the context for the evaluation of *exp* and *m* is the number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

levels of *cont* operations in which the call on *monitor* was embedded.

```
monitor(s,c,R,v,EXP) = [relop(s•c,v) → mon2()
    where mon2()==seq[q(n,caller);eval(exp,caller);unq()];
    TRUE → seq[[s,MONPART•c≡UNDEF → μ(MONPART c,s,NIL)];
        s•MONPART•c←cons(p,s•MONPART•c);
        PATHS←paths+1; list(s,c,p)]
    where p=list(relop,v,exp,caller,n)]
    where relop=[y≡UNDEF→r;TRUE→y] where y=eval(r,caller);
```

The routine *unmonitor(x)* terminates the monitoring process *x*, where *x* was returned by a call on *monitor*, by removing the process from the appropriate monitor list. The process *x* is of the form *(s,c,p)* where *p* is or was an entry in the list *s•MONPART•c*.

```
unmonitor(x) = ((H•x)•MONPART•H•T•x←unm2(H•T•T•x,
                                (H•x)•MONPART•H•T•x))
    where unm2(x,y)=[y≡NIL → NIL;
                H•y≡x → seq[PATHS←paths-1; T•y];
                TRUE → cons(H•y,unm2(x,T•y)];
```

All changes to the values of variables must ultimately be made by the operation μ. Thus it is only necessary to test monitor conditions when a μ operation is performed for the user

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(i.e. all operations which do not act on variables local to the interpreter). The routine μ will make the assignments to the specified variables and will also evaluate the monitor conditions associated with each of those variables. If any monitor condition is *TRUE*, the monitor process will be removed from the monitor list, and the monitor expression will be queued for later evaluation. Because some of the expressions which are queued during a μ routine may be embedded in more *cont* operations than was the μ call, it is some times necessary to requeue the μ routine.

```
μ(c,x:) = [x≡NIL → seq[req(); c];
           TRUE → seq[mu2(H•x,H•T•x); μ(c,T•T•x:)]]
     where mu2(s,val)=seq[μ(c,s,val);
         [s•MONPART•c≠UNDEF → sapp[mu3(y:);y;s•MONPART•c]]]
     where mu3(relop,v,exp,e,m)=
         [relop(val,v) → seq[mu4();eval(exp,e); unq()]]
     where mu4()==seq[PATHS←paths-1; q(m,caller);
                      return(UNDEF,CALLER•caller)];
req()=[queue≠NIL → [n<H•H•queue → seq[q(n,caller); unq()]]];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Synchronization. The operation *synch(c,x,y)* quarantees that for any construct *c* only one process can be be executing the critical section *x*. If a *synch* operation is executed on a construct for which a *synch* operation is in progress then *y* and not *x* will be executed. This operation can be implemented by associating a Boolean variable with each construct which is not local to the interpreter. The variable will be called *SYNCHPART* and will be *FALSE* only when a critical section is being executed and *TRUE* otherwise. Because the expression *y* may include a call on *synch(c,x,y)*, the execution of *y* will be delayed until other queued prccesses (hopefully including the critical section associated with *c*) have been executed.

```
synch(c,X,Y) = [SYNCHPART•c → seq[SYNCHPART•c←FALSE;
                    VAL←eval(x,caller);
                    SYNCHPART•c←TRUE] where val=UNDEF;
               TRUE → seq[delay(); eval(y,caller)]
                    where delay()==seq[q(n,caller); unq()]];
```

Continuous processing. The continuous processing operation *cont(x)* guarnatees that evaluation of the expression *x* will occur between two consecutive steps of any process which is embedded in fewer *cont* operations. We have associated with each process a count of the number of *cont* levels in which it is embedded and then have executed the processes in strict accordance with this count (i.e. higher count processes are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

executed first). All that is then required by the *cont* routine is to increment the count for the current process by one, to evaluate the expression *x*, and to then decrement the count to its former value. When the count is decremented it may be necessary to queue the process which called *cont* because other queued processes will then be at a higher level of *cont* embedding.

cont(EXP) = seq[N←n+1; VAL←eval(exp,caller); N←n-1; req(); val]
where val=UNDEF;

One further routine is needed. We assumed that those constructs which are not local to the interpreter have two special parts *MONPART* and *SYNCHPART*. Consequently, a $\mu_o()$ routine is needed for their initilization.

μ_o(x:) = $\mu(\mu_o$(MONPART,μ_o(),SYNCHPART,TRUE),x:);

The above descriptions of the control operations in terms of sequential control are intended to show only how they might be implemented on a sequential machine and should not be taken as definitions of the control operations. Neither do these descriptions provide the most efficient implementation (e.g. if the monitor processes on a monitor list were ordered by the value of *v* then the number of conditions tested when a μ operation is executed could be reduced), but rather to give the general form of a sequential implementation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

J. EXPERIENCE WITH AN IMPLEMENTATION OF CDL A programming system [Le 70] based of the Control Description Language was implemented of the B5500 during the summer of 1969. This system is written in Algol, follows the strategy suggested in section J of this chapter, and is interpretive.

The primitive data structures (including constructs and lists) are built from simulated 23 bit words having a 4 bit data type field and a 19 bit relative address field. Successive list elements are kept in successive simulated words with exceptions indicated by a special data type *link* which points to the continuation of the list. A *link* to the maximum address value indicates the end of a list. Simulated words are kept two per Algol real variable in dynamic *own* arrays. Garbage collection involves a double copy of the storage which is in use. This operation compacts the in use portion of storage into a contiguous area and attempts to eliminate links by making lists contiguous.

The result of this implementation were both discouraging and encouraging. On the negative side the implementation consists of approximately 1500 Algol statements and is difficult to follow. The lack of clarity in the implementation is a direct result of restricting ourselves to the Algol control structure. It is necessary within the CDL interpreter to set aside the state of the interpretation when monitor processes become *TRUE*, when *cont* operations are encountered, and when

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parallel paths are established, and these can occur when the interpreter is embedded in several levels of routine calls. Algol has no provision for setting aside its stack environment and therefore Algol procedures cannot be used as CDL routines. That is the interpreter requires a pseudo-parallel control structure which Algol does not have. Thus, the system was implemented with all local storage for the interpreter kept in the simulated words under the control of the garbage collector instead of in Algol variables which would be managed by the stack hardware. For the same reason, the interpreter is written using only one Algol procedure level. When routines are needed they are given Algol statement labels and called by first saving a return label and then executing a *go to* to the desired routine.

Because we were unable to take advantage of the Algol stack and instead had to explicitly manage the storage in *own* arrays, there is also a significant degradation in the execution times of CDL programs as compared to similar programs written in Algol. a (decimal) order of magnitude might be expected because the system is interpretive but the first experiments showed closer to four orders of magnitude. A large portion of this time was attributable to our inexperience with list processing in a paged environment. With some changes in the storage management and garbage collection the processing time was reduced by a factor of 50. Experiments with the resulting system have shown an additional factor of 20 to be attributable to the explicit

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

storage management.

Three consequences of implementing the system directly in Algol were an opaque program, excessive processing times, and difficulty in making changes to the system at the Algol level. As it turns out the B5500 is not in fact an Algol machine with respect to control structure. The B5500 is a multiprogramming Algol machine and as such has provision for a form of pseudo-parallel control (i.e. the stack can be set aside and later recovered). The B5500 version should be reimplemented by first extending Algol to include operations which set aside Algol stacks and later recover them. The CDL interpreter can then be implemented directly using procedures as routines.

Despite the speed, experience with the use of the system has been encouraging. The system was implemented as an interactive system with the interpreter executing expressions directly from consoles as well as from stored programs. A sidetrack parser was written in the system and required about 50 minutes of connect time for loading and initial debugging. This parser then provided a syntax extension facility which was used in conjunction with the control extension facility to implement a subset of APL. This latter effort produced a usable (vector only) subset of APL and required a total of 20 man-hours (not mine) for the implementation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

K. THE CONTROL STRUCTURE OF SOL The Sol language will be used to show how existing programming languages are described using the control description facility. Sol is particularly useful for this purpose because it contains a large number (twenty) of control operations ranging from a sequential statement structure resembling that of Algol to parallel processing and interrupt controls. In Knuth's and McNeley's paper "A Formal Definition of Sol" [KM64b] the external syntax for Sol is given in BNF notation and the semantics in English. Here we have translated those English descriptions into CDL. The syntax is given in figure III-F at the end of this section.

Each Sol program or model consists of a set of processes which are executed in parallel. Initially each process has one transaction (i.e. control path). Whenever all transactions are waiting for some condition or for a specified amount of simulated time then the simulated time clock *time* for the model is increased to the minimum time *minwaittime* for which any transaction is waiting. When any transaction executes a stop statement the simulated time clock is stoped and the simulation is complete.

```
sol(GDL,PL) = seq[ papp[evprcs(1,p:);p;pl];
     until stop do when active=0 then TIME•genv←minwaittime];
stop = FALSE;
minwaittime = 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There is a single global environment shared by all processes. It contains any global variables defined in the model and the variable *time* which is the simulated time.

genv = evdecl(gdl,μ_o(TIME,0));

An important characteristic of Sol is that a number of statistics are gathered during a simulation. Thus, for each process the number of transactions *notrans*, the name of the process *name*, a list of the local declarations *pdl*, and the list of statements for the process *sl* are kept. The initial transaction for each process is executed beginning at the first statement of the process and such that initially the transaction is not waiting, is not interrupted, has a local environment consisting of the local definitions of the process and the variable *priority* which contains the priority of the transaction for resolving conflicts when several transactions simultaneously requests the same resource. The table of statement labels is that of the process.

evprcs(notrans,name,pdl,sl) == evtrans(T•sl,0,0,0,
evdecl(pdl,μ_o(GLOBAL,genv,PRIORITY,0)),H•sl)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each transaction has a list of statements *next* to be executed, the next simulated time *waittime* for which it is waiting, the time *inttime* when the transaction was last interrupted, the number of times *noints* it is currently interrupted, a local environment *env*, and a table of statement labels *labtab*. Statements are executed sequentially until the end of the statement list (i.e. *next≡NIL*) is encountered at the highest procedure level, but only when the transaction is neither interrupted nor waiting for a simulated time. Each statement is executed by making its successor be the next statement and then evaluating the current statement in the context of the transaction interpreter *evtrans*.

```
where evtrans(next,waittime,inttime,noints,env,labtab)==
     evprocedure(next,env,labtab)  where
evprocedure(next,env,labtab)=
     (until next≡NIL do when NOINTS•self≡0 then
      seq[NEXT←T•next; eval(s,self)] where s=H•next)
where [
```

Note that the block structure of the Sol interpreter conforms to the dynamic structure of Sol models with statements local to procedures, procedures local to transactions, transactions local to processes, and processes local to models.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Sol control operations take the form of statements and are defined below. Statement labels are ignored in the evaluation.

```
label(ID)=UNDEF;
```

The start statement *new transaction to label* causes a new transaction to be formed. The new transaction has the same process and a local environment which is identical to the transaction which executed the *newtrans*. The new transaction is neither interrupted nor waiting. Execution within the new transaction begins at the given label *label*. The *newtrans* routine also increments the number of transactions *notrans* for the associated process. This latter operation requires mutual exclusion because several *newtrans* processes could be in progress simultaneously.

```
newtrans(LABEL)==par[ return(UNDEF,caller);
     exclusive(findenv(NOTRANS),NOTRANS←notrans+1);
     evtrans(evgo(label,labtab),0,0,0,e,labtab)]
     where e=copy(env);
exclusive(x,EXP)=excl( ) where [p=caller;
     excl( )=synch(x,eval(exp,p),excl( ))];
findenv(name)=fe(caller) where fe(x)=
        [select(name,x)≡SELECTERROR→x; TRUE→fe(GLOBAL•x)];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The cancel statement *cancel* causes transactions to "die". There is also an implied cancel statement at the end of each process.

```
cancel( )=NEXT←NIL;
```

The replacement statement *var ← exp* replaces the value of the variable *var* by the value of the expression *exp*. *Var* may be the reserved Sol identifier *priority*.

```
replace(VAR,EXP)=var•env←eval(exp,env);
```

The wait statement *wait exp* causes the transaction to be delayed for the value of *exp* rounded to an integer units of simulated time.

```
wait(EXP)=seq[WAITTIME←TIME•genv+max(0,round(eval(exp,env)));
     until waittime≤TIME•genv do when NOINTS•self≡0 do
          [waittime>TIME•genv →
               seq[exclusive(findenv(MINWAITTIME),MINWAITTIME←
                         mingreaterthan(waittime,minwaittime,TIME•genv));
                    when TIME•genv≠TIME•genv then UNDEF]]];
mingreaterthan(x,y,z)=[x>z → [y>z → min(x,y); TRUE → x];
                       TRUE →[y>z → y; TRUE→z]];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The wait-until statement *wait until exp* causes the transaction to be delayed until the value of *exp* becomes *TRUE*. The condition is tested once for each increment of the simulated time.

```
waituntil(EXP)=until eval(exp,env) do
    when TIME•genv≠TIME•genv then UNDEF;
```

The enter statement *enter store, exp* is a request for the value of *exp* rounded to an integer units of storage space from *store*. The transaction remains at this statement until the requested storage is available and all other transactions of greater or equal priority which have been waiting for storage space have been serviced. Because of the strict servicing order a list of requests in the order to be serviced is kept with each store as *req*. Whenever space is allocated or released statistics are gathered. The *TOTALOCCUPANCY* of a store is total number of space-time units of the store which have been used. The *MAXUSE* of a store is the maximum storage units inuse during any time interval. Each element of the request queue for a store contains the requesting transaction *trans* and the amount *amt* of space requested.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
enter(STORE,EXP)==seq[exclusive(store•genv,
        REQ•store•genv←q(μ₀(TRANS,caller,AMT,round(eval(exp,env))),
                        REQ•store•genv))
        where q(x,y)=[y≡NIL → list(x);
            PRIORITY•env>PRIORITY•ENV•TRANS•H•y → cons(x,y);
            TRUE → cons(H•y,q(x,T•y))];
    allocate(store•genv)];
allocate(store)==[exclusive(store,
        [REQ•store≠NIL → [CAPACITY•store≥INUSE•store+amt →
         seq[[et>0 → par[TIME•store←TIME•genv;
                TOTALOCCUPANCY•store←TOTALOCCUPANCY•store+
                                    INUSE•store×et;
                MAXUSE•store←max(MAXUSE•store,INUSE•store)]]
                where et=TIME•genv-TIME•store;
            INUSE•store←INUSE•store+amt;
            REQ•store←T•REQ•store; TRUE]]
        where amt=AMT•H•REQ•store]) →
    par[return(UNDEF,transaction); allocate(store)]]
    where transaction=TRANS•H•REQ•store;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The leave statement *leave store, exp* returns the rounded value of *exp* units of space in *store*.

```
leave(STORE,EXP)==seq[exclusive(store•genv,
        REQ•store•genv←cons(μ₀(TRANS,caller,AMT,
                AMT,-round(eval(exp,env))),REQ•store•genv));
    allocate(store•genv)];
```

The seize statement *seize facility, exp* is a request for the use of the facility *facility*. The transaction remains at this statement until the requested facility is available and all other transactions of greater or equal priority or control strength which have been waiting for the facility have been serviced. Because of the strict service order a list of requests is kept with each facility as *req*. Each element of the request queue for a facility contains the requesting transaction and the control strength *cs*. The control strength of a request is given as the rounded value of *exp*. Whenever the control strength of a request is greater than the control strength of the request which current controls a facility, an interrupt will occur so that the request of higher strength can continue ahead of the transaction which has control of the facility. A record is kept of the time at which a facility changes from nonbusy to busy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
seize(FACILITY,EXP)=exclusive(facility•genv,seq[
     [REQ•facility•genv≡NIL→CHANGETIME•facility•genv←TIME•genv;
      TRUE → interrupt([cs≤CS•H•REQ•facility•genv → global;
                        TRUE → TRANS•H•REQ•facility•genv])
        where interrupt(trans)=exclusive(trans,
            seq[[NOINTS•trans≡0 → INTTIME•trans←TIME•genv];
                NOINTS•trans←NOINTS•trans+1])];
     REQ•facility•genv→q(μ₀(TRANS,global,CS,cs),
                                        REQ•facility•genv)
        where q(x,y)=[y≡NIL → list(x);
            cs>CS•H•y → cons(x,y);
            (cs≡CS•H•y)∧(y≠REQ•facility•genv)∧
                (PRIORITY•env>PRIORITY•ENV•TRANS•H•y)
                    → cons(x,y);
            TRUE → cons(H•y,q(x,T•y))]])
        where cs=round(eval(exp,env));
```

The release statement *release facility* allows a transaction to release a previously seized facility. If there are no further requests for the facility the time it has been in use is added to the *TOTALTIMEUSED* for the facility.

```
release(FACILITY)=exclusive(facility•genv,seq[
     REQ•facility•genv←T•REQ•facility•genv;
     [REQ•facility•genv≡NIL → TOTALTIMEUSED•facility•genv←
          TOTALTIMEUSED•facility•genv+
               (TIME•genv-CHANGETIME•facility•genv);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
TRUE → exclusive(trans,seq[
    [NOINTS•trans≡1 → WAITTIME•trans←
        WAITTIME•trans+(TIME•genv-INTTIME•trans)];
    NOINTS•trans←NOINTS•trans-1])
    where trans=TRANS•H•REQ•facility•genv]]);
```

The go to statement *go to labellist, exp* causes control to be transferred to a specified label. If n is the rounded value of the expression *exp* then the label is the n^{th} label on list *labellist* or if n is zero control will continue in sequence.

```
goto(LABELLIST,EXP)=[n≠0→NEXT←evgo(getnth(n,labellist),labtab)]
        where [n=round(eval(exp,env));
            getnth(n,x)=[n≡1→H•x; TRUE→getnth(n-1,T•x)]];
evgo(label,labtab)=[label≡H•labtab → H•T•labtab;
                    TRUE → evgo(label,T•T•labtab)];
```

The stop statement *stop* causes the simulation to terminate.

```
stop( )=STOP←TRUE;
```

A procedure is a subroutine used to save coding. Parameters are not allowed. There are local and global procedures and global procedures cannot refer to local variables. A *go to* may not lead out of a procedure.

```
procedure(ID)=evprocedure(T•id•env,
        [id•env≡id•genv → μ₀(GLOBAL,genv); TRUE → env],
        H•id•env);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A routine *evdecl* is used to make all declarations. *Real* and *integer* variables are initialized to 0. Stores have a fixed capacity *CAPACITY*, a queue *REQ* of requesting transactions a time *TIME* which is the last simulated time at which the store changed from empty to nonempty, the amount *INUSE* of the store which is currently in use, the maximum value *MAXUSE* of *INUSE* over a simulated time interval, and a running total *TOTALOCCUPANCY* of the space–time utilization of the store. Facilities have a queue *REQ* of requesting transactions, a time *CHANGETIME* which is the last time at which the facility went from nonbusy to busy, and *TOTALTIMEUSED* which is the total amount of simulated time for which the facility has been busy. Monitored variables must be previously declared and cause the value of the variable to be printed whenever it changes. The procedure body is associated with the procedure name in the current environment.

```
evdecl(dl,env)=sapp[[H•d≡REAL → papp[μ(env,item,0);item;T•d];
     H•d≡INTEGER → papp[μ(env,item,0);item;T•d];
     H•d≡STORE → papp[μ(env,H•x,μ₀(CAPACITY,H•T•x,REQ,NIL,
          TIME,0,INUSE,0,MAXUSE,0,TOTALOCCUPANCY,0));x;T•d];
     H•d≡FACILITY → papp[μ(env,item,μ₀(REQ,NIL,CHANGETIME,0,
          TOTALTIMEUSED,0));item;T•d];
     H•d≡MONITOR → papp[cont(evmon(item));item;T•d]
         where evmon(item)==par[return(UNDEF,caller);
                   when item•env≠item•env then
                   par[print(item,item•env); evmon(item)]];
     H•d≡PROCEDURE → μ(env,T•d:)];d;dl];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The two portions of Sol which we have not described are the input-output and the evaluation of expressions. The input-output operations will be omitted because they do not influence the control structure. The expressions are a subset of the CDL expressions so we will use the CDL expressions and will evaluate them using *eval*. There are however a few predicates unique to Sol and these are defined below. The expressions *id busy*, and *id not busy* are used to interrogate the status of a facility *id*. The expressions *id full*, *id not full*, *id empty*, and *id not empty* are used to test the current usage of a store.

```
busy(facility)=REQ·facility≠NIL;
full(store)=INUSE·store≡CAPACITY·store;
empty(store)=INUSE·store≡0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
solmodel = BEGIN globaldecllist $; processlist END $.
                # SOL globaldecllist processlist NIL
processlist = prcsdesc                  # prcsdesc NIL
     | prcsdesc $; processlist          # prcsdesc processlist
prcsdesc = PROCESS id $; statement # id NIL statement NIL
     | PROCESS id $; BEGIN prcsdecllist $; statlist END .
                # id prcsdecllist statlist NIL
statlist = statement cptail             # append(H•statement,H•cptail)
                  append(T•statement,T•cptail)
     | id $: statlist
                # cons(id,T•statlist,H•statlist) T•statlist
     | IF exp THEN statement cptail
                # append(H•statement,H•cptail)
                  list(IF,exp,append(T•statement,T•cptail))
                  T•cptail
     | IF exp THEN statement1 ELSE statement2 cptail
                # append(H•statement1,H•statement2,H•cptail)
                  list(IF,exp,append(T•statement1,T•cptail))
                  append(T•statement2,T•cptail)
cptail = empty                          # NIL NIL
     | $; statlist                      # statlist
statement = simpstat                    # NIL simpstat NIL
     | BEGIN statlist END               # statlist
     | $( statlist $)                   # statlist
where append(x,y:)=[T•y≠NIL → append(x,append(y:));
                    x≡NIL → y; TRUE → cons(H•x,append(T•x,y))];
```

Figure III-F, Sol syntax.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
simpstat = NEW TRANSACTION TO label # NEWTRANS label NIL
     | CANCEL                       # CANCEL NIL
     | id $← exp                    # REPLACE id exp NIL
     | WAIT exp                     # WAIT exp NIL
     | WAIT UNTIL exp               # WAITUNTIL exp NIL
     | ENTER id                     # ENTER id list(QUOTE,1) NIL
     | ENTER id $, exp              # ENTER id exp NIL
     | LEAVE id                     # LEAVE id list(QUOTE,1) NIL
     | LEAVE id $, exp              # LEAVE id exp NIL
     | SEIZE id                     # SEIZE id list(QUOTE,0) NIL
     | SEIZE id $, exp              # SEIZE id exp NIL
     | RELEASE id                   # RELEASE id NIL
     | GO TO label                  # GOTO list(label) list(QUOTE,1) NIL
     | GO TO labellist $, exp       # GOTO labellist exp NIL
labellist = label                   # label NIL
     | label $, labellist           # label labellist
label = id                          # id
globaldecllist = declaration        # declaration NIL
     | declaration $; globaldecllist # declaration globaldecllist
declaration = vardecl | facilitydecl | storedecl
     | monitordecl | proceduredecl
prcsdecllist = processdecl          # processdecl NIL
     | processdecl $; prcsdecllist  # processdecl prcsdecllist
processdecl = vardecl | proceduredecl | monitordecl
simpstat = STOP                     # STOP NIL
     | id                           # PROCEDURE id NIL
```

Figure III-F continued, Sol syntax.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
vardecl = INTEGER itemlist          # INTEGER itemlist
        REAL  itemlist              # REAL itemlist
facilitydecl = FACILITY itemlist    # FACILITY itemlist
storedecl = STORE storelist         # STORE storelist
storelist = number item             # list(number,item) NIL
        number item $, storelist    # list(number,item) storelist
monitordecl = MONITOR itemlist      # MONITOR itemlist
proceduredecl = PROCEDURE id $; statement
                # PROCEDURE id statement NIL
exp = ...                           (see figures III-A,B,C)
pexp = ...                          (see figures III-A,B,C)
     | id BUSY                          # BUSY id NIL
     | id NOT BUSY                      # NOT list(BUSY,id) NIL
     | id FULL                          # FULL id NIL
     | id NOT FULL                      # NOT list(FULL,id) NIL
     | id EMPTY                         # EMPTY id NIL
     | id NOT EMPTY                     # NOT list(EMPTY,id) NIL
```

Figure III-F continued, Sol syntax.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER IV -

MULTIPLE SEQUENTIAL CONTROL

> "We do not really have a way of describing control, so we cannot declare its regimes." — A.J.Perlis [Pe 67]

The multiple sequential control is a generalization of the subroutine, coroutine, and pseudo-parallel control found in programming languages. Like these control structures which reflect the sequential processing capabilities of the underlying machine by dealing with only one control path at a time, the multiple sequential control is biased toward sequential processing.

A. SEQUENTIAL PROCESSING The main component of the multiple sequential environment is the process. The process is an active entity which represents an activation of a routine. In this respect it is similar to the processes of CDL. Processes in the multiple sequential environment however, can have only one control path. Because there is only one path of control per process it is meaningful to talk about a process as an element which can be controlled within the system. In the previous chapter an activation record represented a static instance in the progress of a control path within a process together with the dynamic state of the environment of that process. Here an activation record will be used to represent the dynamic progress of a process. Thus an activation record will change over time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and contains the current state of the unique control path for the process as well as the current state of the environment of the process.

Processes can be graphically represented by diagrams which depict the progress of their execution. Each process is represented by a box and is assumed to have a finite lifetime. Time progresses from top to bottom within a process box with the top bar representing the moment at which the process was created and the bottom bar representing the moment of termination. Execution of a process takes place in discrete steps along a line drawn from top to bottom in a process box. The line is viewed as discontinuous and composed of a finite number of points. The entire life of a process is represented thus:

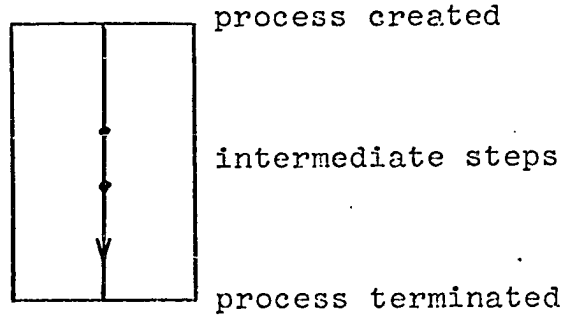

Figure IV-A, process diagram.

We have seen (chapter II) that subroutine, coroutine, and other forms describe relations between processes. The process diagrams can be used to illustrate these relations. A line segment drawn downward between two points in a process box

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

indicates that the processing step at the source (upper) point must be processed before the step at the destination (lower) point. When similar dependencies exist between points in different processes they will be connected by a directed line. If process B acts as a subroutine to process A the relation can be depicted as follows:

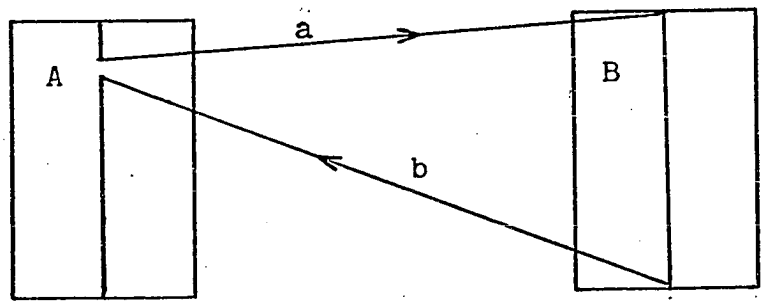

Figure IV-B, subroutine relation.

From the diagram in figure IV-B it can be seen that the entire duration of process B is between two consecutive steps of process A. It can also be seen that no concurrent processing occurs between processes A and B. If A and B acted as coroutines, the diagram would take a different form as shown in figure IV-C. Here control is passed back and forth between the two routines. Process B might be called a subordinate coroutine to process A since the life of B is bounded by two steps of A, but unlike the subroutine the two steps are nonconsecutive.

Other coroutine relations are possible. Process A for example could have begun before process B, act as a coroutine to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

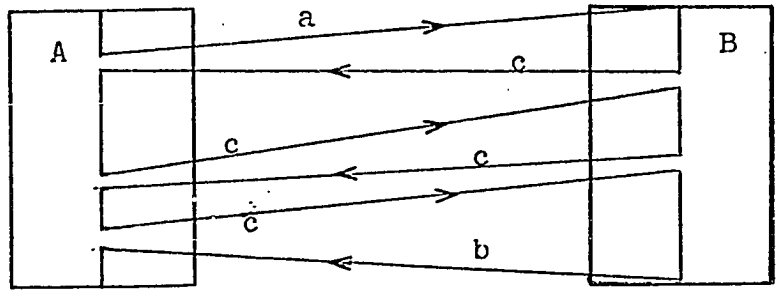

Figure IV-C, subordinate coroutine.

process B, and then terminate before process B. More than two processes can act as coroutines. A three process coroutine relation is shown in figure IV-D where the prccess pairs A-B and B-C each have a coroutine relation. Because A is created before B and also terminates before B they do not hold a subordinate relationship. Process C is however subordinate to process B.

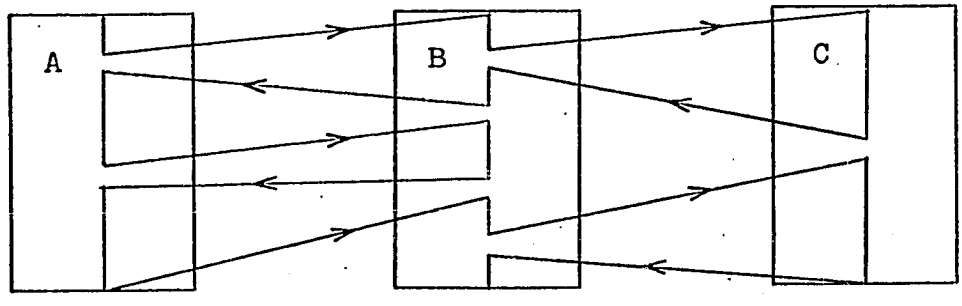

Figure IV-D, double coroutine without subordination between A and B or between A and C.

The double coroutine has two coroutine pairs in which one process (B) is a member of both pairs. Coroutine environments in which control is passed among several processes which are not pair-wise coroutines are also possible. Such an arrangement is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

shown in figure IV-E. There is also a cyclic relation which can occur among processes. Essentially the cyclic relation says that the order of creation of a set of processes is the same as the order of termination. For a two process environment either the subordinate or the cyclic relation holds. In figure IV-D the processes A and B are cyclic while process C is subordinate to B. For more than two processes the subordinate and cyclic relations are mutually exclusive but it may be that neither holds. In figure IV-E the three processes are cyclic with A first, B second, and C last,while in figure IV-D the three processes are together neither subordinate or cyclic.

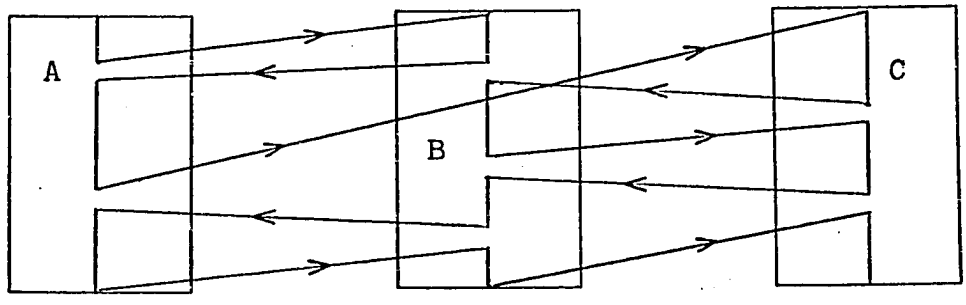

Figure IV-E, cyclic triple coroutine.

B. INTERFACE OPERATIONS The directed lines between processes determine the relations between the processes. They also suggest a set of operations for generating these relations. There is an endless variety of relations which we can have among processes, but there are few varieties of interprocess line seqments in the corresponding process diagram. Consequently there might be one operation corresponding to each type of interface between

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processes. Returning to figure IV-B we see two types of interfaces (labeled a and b). The type a interface causes one process (A) to be suspended, another process (B) to be created, and execution of the latter process to be begun. The type b interface causes one process (B) to be terminated and the execution of another (A) to be resumed. In figure IV-C the type a and b interfaces are apparent, but there is an additional type c interface which causes one process to be suspended and another to be resumed.

The words which we have been using to describe these interfaces (create, suspend, resume, and terminate) can be formalized as operations which combine to form the interfaces. First let us describe the interpreter for such a system. Each process will have its own interpreter. Each interpreter will be concerned with three aspects of processes: their processing status whether suspended, resumed or terminated, their current point or step of processing, and the current state of the associated environment. Each processing step has an associated expression or statement which is to be evaluated at that point and which we will call an instruction. The next processing step will be represented as a list of instructions with the next one as the head of the list. The environment (*env*) provides the context for all processing within a process.

The interpreter for a process returns a representation of the process (actually the local environment of the interpreter)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the interpreter's caller, and whenever the status of the process is *RESUMED* then the interpreter will sequentially evaluate the instructions. The interpreter for a process will be terminated when the status of the process becomes *TERMINATED*. The multiple sequential interpreter is described in figure IV-F where *mseval* evaluates a single instruction.

```
msinterp(status,next,env)==par[return(self,caller);
    until    status≡TERMINATED  do
      when STATUS•self ≠ SUSPENDED then
        [status≡RESUMED → seq[NEXT←T•next; mseval(inst,env)]
                                  where inst=H•next]]
    where mseval(inst,env)= ... ;
```

Figure IV-F, multiple sequential interpreter.

The interface operations can then be described as routines. For definiteness we will say that the routines of which a multiple sequential process is an instance are represented by lists whose heads are the formal parameter specification and whose tail is the instruction list. The *create* operation then has a multiple sequential routine as its first argument and a list of actual parameters to that routine as the remaining arguments. *Create* establishes a new suspended process with its own interpreter. The particular mechanism for parameter passing *pp* is of no concern.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

create(r,x:)=msinterp(SUSPENDED,T•r,pp(H•r,x,μ_o( )))

where pp(fpl,apl,lenv)= ... ;

Terminate(p) permanently halts the execution of *p* instructions of process *p* at the completion of any instruction already in progress. This is mechanized by setting the status of *p* to *TERMINATED* and allowing the interpreter for *p* to terminate itself upon discovering the status. *Suspend* makes the specified process temporarily inactive (*SUSPENDED*) providing the process is not already terminated. *Resume* continues the processing of a suspended process.

terminate(p)=STATUS•p←TERMINATED;
suspend(p)=[STATUS•p≠TERMINATED → STATUS•p←SUSPENDED];
resume(p)=[STATUS•p≡SUSPENDED → STATUS•p←RESUMED];

The various types of interface can now be described using these operations. Returning to the process diagrams we assume that any interprocess line segment represents an interface which was generated by the process at the source of the directed line. Thus *activate* creates a process and then resumes that process without effecting the status of the generating process; a subroutine call suspends the generating process and activates a new process; an intermediate coroutine return suspends the calling process and resumes the specified process; and a terminal return terminates the returning process and resumes the specified process. The latter three operations generate type

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a, c and b interfaces respectively as shown in figure IV-C.

```
activate(r,x:)=resume(create(r,x:));
call(r,x:)=seq[suspend(caller); activate(r,x:)];
coreturn(p)=seq[suspend(caller); resume(p)];
termreturn(p)=seq[terminate(caller); resume(p)];
```

The interface operations are illustrated in figure IV-G. Notice that the use of the *activate*, *suspend*, and *resume* operations permit the creation and control of multiple simultaneously active sequential processes. Parallel paths can be terminated by a lone *terminate*. It may also be necessary to provide parameter passing capabilities with all the interface operations.

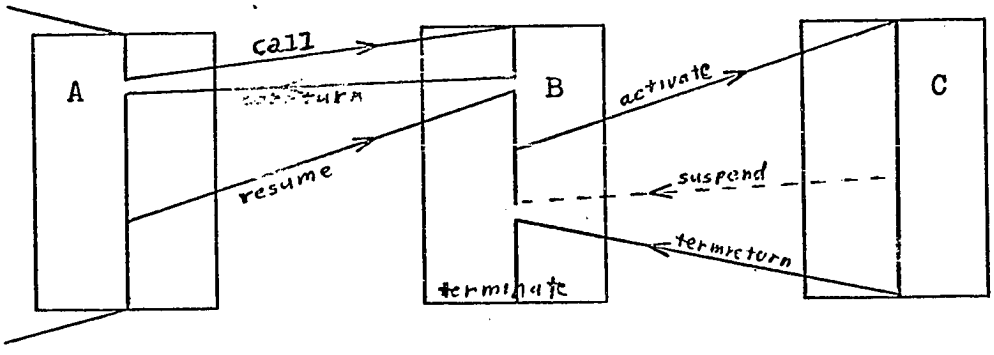

Figure IV-G, interface operations.

The description of the intermediate coroutine interface given here can be contrasted with a description of a coroutine interface in the CDL using the primitive control operations directly. Within each process the symmetry is emphasized with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the interface appearing as a subroutine call from the source process and a subroutine return to the other process. Thus, we do exactly that in the description: the *coreturn* is accomplished by making a subroutine-like call on a dummy process which terminates (returns) to the coprocess.

```
coreturn(p,x)==return(x,p);
```

The process diagrams represent processes or instances of routines. That is, they represent the dynamic control relations which are established for some specific execution of the associated routine. If the interface operations appear within the arms of a conditional then the control relation can vary from execution to execution. Again we see that it is not meaningful to describe routines by monadic predicates like sub-, co-, and parallel-. Instead we should deal with the interface operations which generate the relations and in cases for which certain interprocess relations are quaranteed to remain invarient we may describe them using polyadic predicates which encompass the processes or routines involved.

C. STACK AND QUEUE STORAGE The more general a system the less efficient must be its implementation. Consequently, when special properties are observed in a control environment we look for efficient implementations which take advantage of those properties. Two classes of relations among processes were seen in the process diagrams. One class described the sequencing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

among the processes and included subroutines, coroutines, double coroutines, and multiple process coroutines. The other class of relations were concerned with the relative lifetimes of the processes and included the subordinate and cyclic relations. The relative lifetime of processes is the same as the relative lifetimes of their storage so that we can look for savings in storage management when the subordinate or cyclic relations hold.

It has become common practice to allocate the local storage for subroutines in a stack. That is, a contiguous area of memory is provided for the subroutine storage as shown in figure IV-H. The first process is allocated in region A. If A then creates process B as a subroutine then storage for B is allocated immediately above A. And if B calls C as a subroutine then storage for C is allocated above B. When processing of C is completed and C makes a terminal return to B, the storage for C is released and can be reallocated to other subroutines (D) called by B. When B is completed its storage is released and can be reallocated to other subroutines (E) of A.

There are several reasons for using this first in last out mechanism. It provides a simple means of dynamically allocating the memory with only a single pointer marking the boundry between the allocated storage and the available space. Only those processes which have been created but not yet terminated take storage space, and recursive routines which require several processes which are instances of the same routine to exist

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

simultaneously can be accommodated without further mechanism.

A programming environment with only subroutines has the property that for any pair of processes in that environment, either the subordinate relation holds (one is created before and terminated after the other) or the two processes are time-wise disjoint (one is terminated before the other is created). It is quite possible that these relations hold even without a subroutine environment. This is also the only requirement for stack allocation. The requirement could be restated as: storage for all processes will be allocated at the top of the stack when the process is created, and it must be guaranteed that every process is at the top at the time it is terminated.

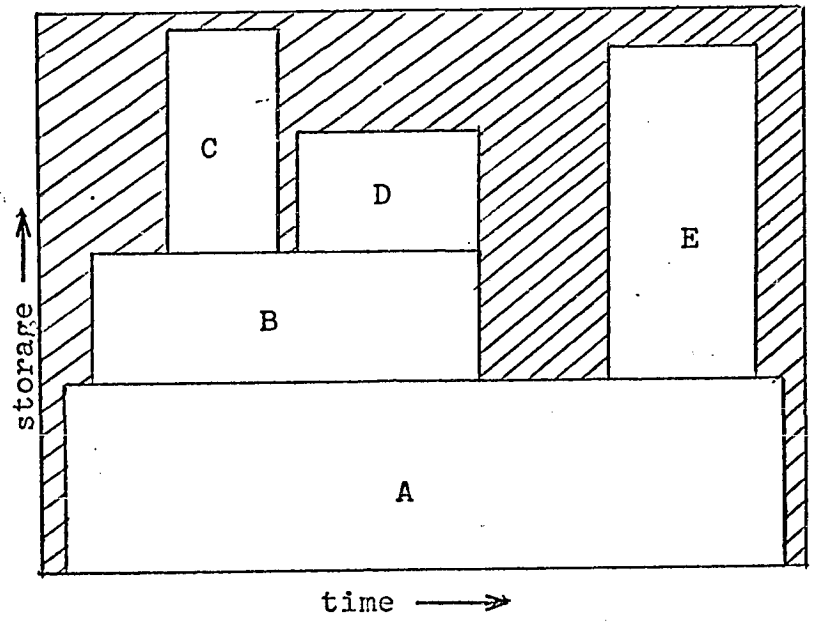

Figure IV-H, stack allocation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For coroutine and parallel routine environments the stacking property cannot in general be guaranteed, but may exist in specific situations. If these can be detected either automatically or are noted by the programmer then stack allocation can be used. One such example is the macro generator. Programming languages sometimes have provision for syntax macros. A macro associates a name with a string of characters. Whenever that name is encountered the associated string is substituted for the name (possibly with provision for parameters). Such a system is described in [Str65]. When, as is usually the case, the final string is being fed to a translator it may be convenient for the macro generator and the translator to act as coroutines. If the recognition of a macro name is made the job of the translator and the macro generator is a routine which successively emits the symbols of a macro definition then the stacking requirements hold. Returning to figure IV-H, A might represent the storage for the translator and B, C, D, and E the storage to generate four specific macros. When A encounters the call on macro B, B is called as a coroutine from A, and when within the coroutine interchange between A and B, A detects an occurrence of macro C then A will call C as a coroutine. Processing between A and C will continue until C is completed then processing between A and B will be resumed. Thus all other processes act as coroutines with A but it is guaranteed that if the existance of any two processes is overlapped then one is created before and terminated after the other. The stack storage model is applicable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Other specialized allocation policies may be useful in other applications. If one were simulating active entities with fixed (real) lifetimes, a process might be used to represent each such entity. Although the processes might be created and terminated at various times throughout the simulation and their interactions may be very complex, the fixed real lifetime guarantees that the order of creation is the same as the order of termination (the cyclic property mentioned earlier). This suggests a queue (first in first out) storage mechanism. For this purpose the memory can be viewed as cyclic. Such an allocation scheme is illustrated in figure IV-I where the processes A, B, C, D, and E are allocated in that order.

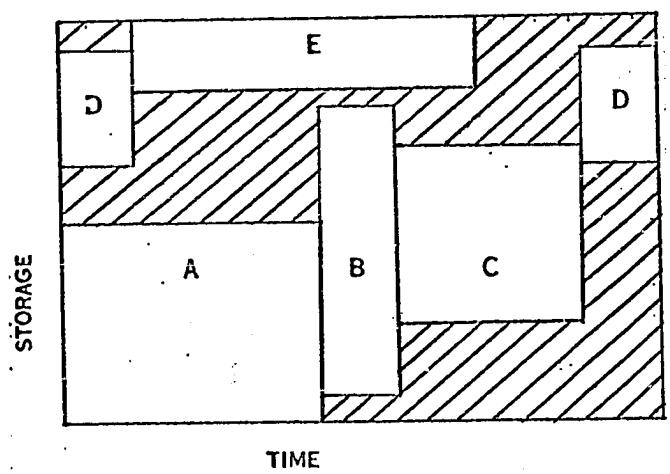

FIGURE IV-1. QUEUE STORAGE ALLOCATION

In a real programming situation we do not expect things to be as simple. The major routines might be cyclic for example, but each of them calls subroutines. In such a case the subroutine stack areas for each major process could be allocated

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cyclically. More likely an environment will be almost cyclic or almost subordinate. In such cases the queue or stack could be used and exceptions taken only when the offending conditions occur. If for example a system can be managed with a stack except that once in a while a process not on top terminates, it may be worthwhile to use a stack and when a process not at the top of the stack terminates then mark it terminated but do not release the storage until it comes to the top.

D. SEMAPHORES We now take a closer look at the semaphore introduced in section II-G. The semaphore is a device to limit the number of processes simultaneously executing a critical section of program or simultaneously using a datum. The semaphore has an integer value which at any time is the difference between the number of processes which can simultaneously enter the critical section and the number of processes (including those in the critical section) wanting to enter the critical section. This is a slight generalization of the semaphore presented in [Ha 67] [Di68a] [Wi 69] in that here the number of processes simultaneously in a critical section can be greater than one. This may be useful when the semaphore is associated with the pool for some resource and is initialized to the size of the pool.

Before entering a critical section, each process executes a *p* operation on the associated semaphore. The operation *p(sem)* will decrement the value of the semaphore *sem* by one and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

continue processing only if the resulting value is nonnegative. Otherwise the process is suspended until another process completes the critical section. When a process completes a critical section, it executes a *V* operation on the associated semaphore. The operation *v(sem)* will increment the value of *sem* and if there are processes waiting to enter the critical section will then resume one of them. The management of the waiting processes is by an unspecified booking scheme. The existence of the booking routines *book(p,sem)* which books the process *p* for later processing in the critical section associated with semaphore *sem* and *unbook(sem)* which unbooks one process waiting for the critical section associated with *sem*, is assumed.

The semaphore operations described above are in the context of a multiple sequential environment and can be described in CDL with the aid of some of the interface operations. The semaphore is represented by a construct with a *VALUE* part.

```
p(sem)=evp(sem,caller) where evp(sem,x)=
     synch(sem,
          [(VALUE•sem←VALUE•sem-1)<0 → par[suspend(x); book(x)],
          evp(sem,x));

v(sem)=synch(sem,
          [(VALUE•sem←VALUE•sem+1)≤0 → resume(unbook(sem))],
          v(sem));
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The CDL definitions point out the two major ingredients of the *P* and *V* operations. First is their indivisibility with respect to other operations on the same semaphore. Thus the most global operation in the definitions of *P* and *V* is *synch*. The second ingredient is the nonbusy waiting provided by the suspension of any process which cannot gain immediate access to the critical section and the later resumption of that process when the section becomes available. Note however that waiting necessary because of the indivisibility of the *P* and *V* operations is a busy wait (the third argument to *synch*).

The *P* and *V* operations can be described in the context of the CDL parallel environment without resort to the multiple sequential operations.

```
p(sem)=[when VALUE•sem≥1 then synch(sem,
        [VALUE•sem≥1 → par[VALUE•sem←VALUE•sem-1; FALSE];
            TRUE → TRUE],
        TRUE)
    → p(sem)];

v(sem)=synch(sem,VALUE•sem←VALUE•sem+1,v(sem));
```

Here the value of *sem* represents the number of processes may enter the critical section at any given time and is never negative. For mutual exclusion the value of *sem* will be zero for section busy or one for section available. *Synch* was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

again used to make the *P* and *V* operations indivisible with respect to a given semaphore, but the nonbusy waiting is provided by the *when* which monitors for a nonzero value in *sem*. If these operations are used for resource allocation they can be extended to permit multiple simultaneous allocations by adding an additional positive integer parameter to both routines and using the value of that parameter instead of each occurrence of one (1) in the above CDL descriptions.

E. OPERATING SYSTEMS Operating systems are large complex programs which are seldom wholly understood by either their users or their implementers. Operating systems act as a bridge between radically different machines. On the one side, the real machine has a hierarchy of memories with various speeds, capacities, and addressing characteristics and usually only a serial processor with only sequential, conditional, and possibly interrupt control facilities. On the other side is an ideal machine with multiprogramming capabilities, virtual memories with uniform characteristics, and virtual devices corresponding to the logical requirements of the system.

What often happens in practice is that the description of the operating system is ad hoc. The function of the bridge and its parts get distributed throughout the system. The task of implementing an operating system is further complicated because the various subtasks which must be dealt with (management of the various physical and logical resources of the system act as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parallel or pseudo-parallel routines and have no hierarchical structure suitable to subroutines. Consequently the implementer for any subtask must be directly concerned with the interface between his and all other subtasks and the ramifications of his actions throughout the rest of the system. When the conceptually useful control structures are recreated at each interface in terms of the sequential control primitives, each interface tends to have its own idiosyncrasies and there is no practical way to predict the total effect of any control interface.

Alternatively one might first create an overall environment in which the control structures suited to the description of an operating system are primitive. This control environment must include parallel or pseudo-parallel processing capacilities and means for interfacing and synchronizing the parallel processes. With a set of precisely defined controls suited to the task (and no undercutting of these controls by direct reference to their underlying sequential implementation), each subtask should have minimal concern with the interface and the effects of any routine should be limited by the control structure and not pervade the entire system.

Machine languages have remained the primary vehicle for writing operating systems. This is understandable for two reasons. First, an important objective in operating system design is efficiency and this can be gotten by going directly

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the underlying machine without intervening software. Secondly the advantages offered by high level languages are primarily in the complexity which can be dealt with as a unit (i.e. arrays, files, and lists for data and subroutines and loops for control). These advantages are lost for control when the control structure of the task is in conflict with the control structure of the language. In fact the imposition of high level sequentially oriented control (subroutines and iterative loops) on an essentially parallel task can at best only confuse the issues. We should not discard high level languages as tools for building operating systems but rather should use high level languages with suitable control.

We are not prepared to suggest a specific language for operating systems, but the desire for both parallel processing and efficiency suggests a multiple sequential control. The multiple sequential control provides a form of parallel processing and can be implemented relatively efficiently on a sequential machine. A number of (possibly all) multiprogramming operating systems have used some form of multiple sequential control but in many cases have not clearly separated it out from the control structure of the underlying machine. One operating system in which there is a strict use of a multiple sequential control with well defined interfaces is the "THE"-multiprogramming system [Di68b]. In this system each system function and each user program is a process. Parallel processes are synchronized using the *P* and *V* operations on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

semaphones. The system was built in a hierarchical manner so that concern with the physical properties of the processor, memories and devices is required at only the lowest levels and at the higher levels one sees only the virtual machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER V

PARSING AS A PROBLEM IN CONTROL

The task of writing recognizers and parsers for context free languages will be taken as an example of how an explicit concern with control structure can simplify and lend insight to programming tasks. As we will see later, the control structures used for parsing are useful in several problem areas. Parsing, however, has some special properties which permit an efficient implementation of these control structures. The general approach to a programming task in terms of its control structure will be: given a task area 1) find a control structure which is well suited to that task, 2) associate with that control structure a concrete syntax (notation) which makes the constant parts of that control implicit, and 3) find an efficient implementation for that control in terms of some given controls.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A. TERMINOLOGY The following terminology will be used in our discussion of recognizers. A language is a set of strings over a finite set of terminal symbols. A context-free grammar is a formal device for specifying which strings are in a language. The grammar uses another set of symbols, the nonterminals which can be thought of as naming syntactic categories. The grammar is specified by a finite set of rewriting rules called productions. Each production specifies that some nonterminal, the left hand side of the production, may be rewritten as a string, the right hand side of the production. If several right hand sides are associated with a nonterminal then they will be called alternatives of that nonterminal. The strings which can be formed by successive application of the rewriting rules to a nonterminal are called derivatives of that nonterminal. Any derivative which contains only terminal symbols is called a terminal derivative. The nonterminal whose terminal derivatives are exactly the strings of the language is called the root of the grammar. For examples of productions see section III-C.

B. A NONDETERMINISTIC CONTROL Productions provide a generative description of a context-free language. To obtain a string of the language we begin with an alternative of the root as the first derivative. For any nonterminal in that derivative we substitute one of its alternatives and repeat this process until no nonterminals remain. The purpose of a recognizer is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the opposite: given a terminal string, determine if it is in the language. We may also (as with a parser) be interested in the derivation which produces the string. The obvious method is to begin with the root and successively apply the productions to produce the given string. When substituting for a nonterminal we must choose that alternative of the nonterminal which will ultimately produce the desired string. Thus we have a nondeterministic control structure which given a grammar and a terminal string, will (beginning at the root) successively apply the productions and at each substitution always make the correct choice so that ultimately the terminal derivative formed will be the given string.

C. SIDETRACKING Since we cannot implement such a control structure on any real machine or with any deterministic control primitives, we will replace it with a deterministic control structure which achieves the same effect. The first step is to systematize the substitution process. At each step we will replace the left-most nonterminal of a nonterminal derivative. This gives a unique order to the productions used to derive a terminal string. For any step of the derivation process, the string of terminal symbols to the left of the left-most nonterminal symbol of the current derivative must exactly match some head of the given string. The problem is to choose an alternative of the left-most nonterminal which will guarantee this condition throughout the derivation. Because the decision

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cannot be made immediately we will take all choices. That is, we will establish parallel paths of control, one for each choice. These paths will then continue the derivations in parallel until a derivative is encountered for which the string of terminal symbols to the left of the left-most nonterminal is not identical to some head of the given string. All such control paths will be terminated. Thus, all and only those derivations which terminate in the given string will be completed.

This control structure will be called sidetracking in contrast with the backtrack control examined in section II-I. The backtrack primitives could be applied to this task so that the alternatives would have been selected one at a time instead of in parallel. Both of these methods can get into trouble with left recursive grammars, grammars in which some nonterminal has a nonterminal derivative with itself as the left-most symbol. When the backtrack *choice* function chooses this alternative it will continue to do so indefinitely. For the sidetrack control it means that some derivations will continue indefinitely. If we further require that parallel paths in the sidetrack control operate in a somewhat uniform rate so that progress is eventually made on all paths, then we guarantee that if there is a derivation it will be found (this is a breadth first search as opposed to the depth first search with backtracking). Actually, we will impose a stronger condition that all paths which do not ultimately lead to a solution will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

eventually terminate (i.e. even in cases of left and right recursion, no path will continue indefinitely). This condition can be guaranteed but the explanation will be postponed until we give an implementation.

Let us look at the progress along any one control path. At any step in the derivation only those terminals at the head (or left most end) of the alternative to be applied need be compared to the given string since all symbols to the right of the substitution will have been matched at the previous step. Thus, at each step we compare the terminal string (if any) at the head of the alternative to the head of some tail of the given string. If they match then they are removed from the head of the given string and the current derivative (for this control path only).

With this control structure the grammar for a context-free language can be interpreted as a program which recognizes terminal strings of that language. The grammar consists of a set of productions each of which will be interpreted as a routine. The nonterminal at the left hand side of the production is the name of the routine, the right hand side is the body. Each routine has a single parameter which is a string of terminal symbols. The function of a routine is to determine whether any initial segment of the given string is a terminal derivative of that nonterminal and if so to return the remainder

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the string after removing that initial segment.

The alternatives to a production indicate parallel paths of control within the body of the routine for a nonterminal. The symbols which comprise an alternative are processed in order from left to right. When a terminal symbol is encountered it is compared to the first (left most) symbol of the given string. If they differ, the control path is terminated. If they are the same the first symbol is removed from the given string and processing continues in order. When a nonterminal is encountered a subroutine call is made on the routine for that nonterminal. If this call finds an instance of the nonterminal at the head of the given string it will return the remainder of the string and processing will continue. When the processing of an alternative is completed, the remaining terminal string is returned to the calling routine.

```
x = A | B x C
```

Figure V-A, production language program

The above grammar for language x uses capital letters to represent terminal symbols and lower case letters to represent nonterminals. The alternatives are separated by "|". The interpretation of the above grammar as a program which recognizes strings of language x, is illustrated by transliterating it into the Algol-like program in figuve V-B where

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

anderson's *and* operator [An 65] is used to separate statements which can be processed in parallel and the *halt* operation is used to terminate control paths. These should be considered as equivalent programs, both recognizing strings of language x, both employing the same control structure, but having different syntax. The strings of language x are all of the form B^iAC^i where $i \geq 0$. In the program below *halt* indicates termination of the control path executing the *halt*, *head* extracts the first symbol from a string, and *tail* returns the rest of a string after the first symbol is removed.

```
string procedure x(s); string s; value s;
parallel begin if head(s)='A' then x:=tail(s) else halt end
     and begin if head(s)='B' then s:=tail(s) else halt;
               s:=x(s);
               if head(s)='C" then x:=tail(s) else halt end;
```

Figure V-B, ALGOL-Like language program.

D. MULTIPLE PARALLEL RETURN Although routines for nonterminals can be called as subroutines, the subroutine return will not suffice. The body of a routine may have several parallel control paths and therefore can have more than one successful completion. Thus there can be several returns to the same point of call. To maintain these separate paths after their return to avoid confusion among them, returns will be made to copies of the calling processes instead of the processes themselves.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We have a multiple parallel return which preserves parallel control paths beyond their return from a process.

The above example shows the advantage of making the repetitive parts of programs implicit in the syntax. Recognition is a very restricted problem area and consequently can use a compact notation. The notation of figure V-A also restricts the generality of the language since all the control operations *if*, *parallel*, sequential processing, *halt* which were available in the ALGOL-like language are accessible only in the restricted forms required for recognition. Because the control environment is more restricted, more efficient implementations should be possible.

E. PRODUCTION LANGUAGE SYNTAX The production language for describing context-free grammars can now be viewed as a programming language. A formal description of the control structure for that language can be given by the machine which interprets a representation of programs in the language. The concrete syntax and corresponding abstract syntax for the production are given in figure V-C. The abstract syntax for a grammar is a construct with one element for each nonterminal. The element for a nonterminal is named by the symbol for that nonterminal while the value represents the right hand side of the production and is a list of the alternatives. Each alternative is a list of the elements which must be concatenated

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to form that alternative. Terminal symbols are represented by a list of two elements, *QUOTE* and the symbol itself. Non-terminals are represented by a single list element, the symbol itself. The variable *g* in figure V-C is assumed to be initialized to an empty construct. The representation of the program in figure V-A for example could be formed by the following CDL expression:

```
μ(g,X,list(list(list(QUOTE,A));
                list(list(QUOTE,B),X,list(QUOTE,C))));
```

```
grammar = production                # g
     | production grammar           # g
production = id $= altlist          # (μ(g,id,altlist))
altlist = alt                       # alt NIL
     | alt $| altlist               # alt altlist
alt = pexp                          # pexp NIL
     | pexp alt                     # pexp alt
pexp = id                           # id
     | capid                        # QUOTE capid NIL
     | $$ symbol                    # QUOTE symbol NIL
```

Figure V-C, syntax for production language

F. INTERPRETER FOR SIDETRACK CONTROL An interpreter for the sidetrack control used as a recognizer is given in figure V-D. The recognizer has three parameters, a grammar *g* represented as in figure V-C, the nonterminal *r* which is the root of that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

grammar, and the string *s* to be recognized. Initially, the recognizer has two parallel control paths. The first (line 1) calls on the routine *production* to find an instance of the root at the head of *s*. Whenever one is found the value of *v* is set to *TRUE*. The second control path (line 2) monitors the many paths which may be generated in the recognition process until all have terminated and then returns with the value of *v*. The variable *v* (line 3) is initially set to *FALSE*. The routine *production* looks for all instances of derivatives of the non-terminal *p* at the head of the string *s* and whenever one is found, returns the remainder of the string and control to its caller. *Production* provides the sidetrack control (line 4) by establishing parallel control paths for each alternative of *p*.

```
1    recognize(g,r,s)==par[seq[production(r,s); V←TRUE];
2        when paths≡1 then return(v, caller)]
3    where [v=FALSE;
4    production(p,s)==papp[alt(a,s,caller);a;p•g];
5    alt(a,s,ar)==[a≡NIL → mpret(s,ar);
6        atom(H•a) → alt(T•a,production(H•a,s),ar);
7        H•H•a≡QUOTE → [H•T•H•a≡H•s → alt(T•a,T•s,ar)]]];
8    mpret(v,ar)==
9        return(v,μ₀(STATE,STATE•ar,GLOBAL,copy(GLOBAL•ar)));
```

Figure V-D, sidetrack recognizer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The routine *alt* processes a single alternative, *a*. If *a* is null then (line 5) all elements of the alternative have been processed and the current string *s* is returned to the original caller, *ar*. Because there may be several paths of control returning to *ar*, this return constitutes the multiple parallel return. If the first element of *a* is atomic (line 6) then it is a nonterminal and production is called to find an instance of that nonterminal; if found then processing of alternative *a* will continue with the string returned by *production*. If the first element of *a* is a terminal symbol then (line 7) it is matched to the first symbol of the string *s*; if they are the same then processing of the alternative continues at the next element. Control paths are terminated in line 7 whenever a terminal symbol does not match and in line 1 whenever a successful derivation is found. The environment of the calling process is copied (lines 7,8) when the multiple parallel return (line 5) to prevent conflicts in the values of variables from several returns.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1    parse(g,r,s)==par[V←H•production(r,s);
2        when paths≡1 then return(v,caller)]
3.   where[V=FALSE;
4    production(p,s)==papp[alt(a,NIL,s,caller,UNDEF);a;p•g];
5    alt(a,tree,s,ar,x)==[a≡NIL → mpret(cons(self,s),ar);
6        atom(H•a) → seq[ X̂←production(H•a,s);
7                alt(T•a,cons(H•x,tree),T•x,ar,UNDEF)];
8        H•H•a≡QUOTE → [H•T•H•a≡H•s→alt(T•a,tree,T•s,ar,UNDEF)]]];
```

Figure V-E, sidetrack parser

The corresponding interpreter for parsing is given in figure V-E. Each node in the parse tree is represented by the local environment of the process which initiated the return from the associated alternative. In each such environment (in fact all *alt* process environments) is a list (called *tree*) of parse trees for the nonterminals of that alternative. These lists are kept in reverse order and are built as the alternative is processed from left to right. The multiple parallel return (line 5) returns the parse tree as well as the remainder of the string. Whenever the string is parsed as a derivative of the root, the value of v is replaced (line 1) by the parse tree. The parse tree for each production is initiated (line 5) to a null list and whenever a nonterminal within an alternative is identified (line 8), it is added to the head of the parse tree (line 7).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

G. AN EFFICIENT IMPLEMENTATION The recognizer as described in figure V-D requires time and space proportional to an exponential in the length of the string being recognized. For each string position there is a possibility of multiplying the number of control paths by a constant proportional to the number of alternatives. If in the recognition process every nonterminal routine were called to look for an occurrence of that nonterminal at every string position, there would be only p×n calls on nonterminal routines where p is the number of nonterminals in the grammar and n is the length of the given string. Consequently, we expect several calls on a nonterminal routine for a given string position, though these occur from different control paths.

This suggests an implementation strategy which is applicable to any control having routines which may be called several times with the same arguments. For each call on such a routine the value of the arguments and the returned value (we assume no side effects) will be saved with the routine. If a subsequent call is made on the routine with identical arguments then the value from the first call will be returned without reevaluating the routine. The merit of this method depends on the relative costs of recording and looking up the arguments versus reevaluating the routine. If the potential arguments to a routine can be ordered such that the argument at any given call always succeeds or is identical to the arguments of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

previous call (i.e. calls with identical arguments are clustered) then the look up time is reduced to a constant and only one argument-value pair need be remembered at a time. This could be extended to remember any constant number of previous arguments. Note also that it is not necessary to know the ordering on the arguments, but only that there is an order.

The sidetrack control does not have this property, but for any trial derivation (control path) the calls on each non-terminal routine will be in the order of the string positions from left to right. Multiple calls with the same string position will be clustered and will occur only for instances of left recursion in the grammar. Consequently, the ordering property can be imposed on the entire collection of control paths by synchronizing their processing to the string position. That is, no control path will progress beyond a given string position until all other paths have completed their processing at that position.

This change is shown in figure V-F. Because the string position is the same for all active processes, one global copy is kept and the string is no longer an explicit parameter for the *production* and *alt* routines. There is also no need (line 5) to return the remaining string. Whenever a control path encounters a terminal symbol (line 7) it waits until the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1   parse(g,r,s)==par[V←production(r);
2      seq[ until paths≡1 do when active≡0 then S←T•s; return(v,caller)]]
3   where [v=FALSE;
4   production(p)==[save(caller,p•g) → papp[alt(a,NIL,ARL•p•g);a;p•g]];
5   alt(a,tree,arl)==[a≡NIL → papp[ mpret(self,ar);ar;arl];
6          atom(H•a) → alt(T•a,cons(production(H•a),tree),arl);
7          H•H•a≡QUOTE → [H•T•H•a≡H•s → when s≠s then alt(T•a,tree,arl)]];
8   save(ar,p)=synch(p,[S•p≠s → par[μ(p,ARL,list(ar),S,s); TRUE];
9          TRUE → par[push(ar,ARL•p); FALSE]], save(ar,p))];
```

Figure V-F, parse with merged nonterminal processes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

string position is stepped for all control paths. A global control path (line 2) updates the string position whenever all paths are waiting. Not only are the calls on each nonterminal procedure clustered, but all calls for a given string position (except nonterminals having the empty string as a derivative) will occur before any return from a call at that string position. Thus, instead of saving the value returned by the first call and returning it to all subsequent callers, we will save the return points (activation records) for the subsequent callers and when a value is later returned by the process for the first caller it will be passed to the others. A list of callers for the current string position (and the string position itself) is kept with each production of the grammar. The list is then passed along for processing by the *alt* routine instead of a single caller. The *save* routine is used (line 4) to add return points to the current list of callers (*arl*). *Save* returns *TRUE* only for the first call on a nonterminal routine at each string position. Because several processes may simultaneously attempt to add to an *arl*, *save* (line 8) uses a *synch* operation to guarantee mutual exclusion when adding to the list. Note that this implementation can handle left recursion in the grammar without an infinite regression. A similar implementation is given by Irons [Ir 63], but he does not permit left recursion.

There is still a possibility for exponential growth,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

because the number of *alt* processes associated with each nonterminal call can be multiplied each time a nonterminal is encountered in the right hand side of its alternatives. If the grammar is linear (all alternatives contain at most one nonterminal) then the number of *alt* processes for the body of one call on a nonterminal routine can be at most proportional to n and the total time for recognition or parsing proportional to pn^2. The *alt* processes are the intermediate states in the processing of a nonterminal routine, one for each alternative and one for each return from a nonterminal call within an alternative. If in the recognition process every nonterminal routine were called for every string position and for each of these calls, every state (*alt* process) were entered for every string position, the total number of states would be only mn^2 where n is the string length and m is the number of nonterminal calls in the bodies of nonterminal routines. There must be identical *alt* processes.

Duplicate *alt* processes can be eliminated by recording the returns at each string position and checking at each return to see if a return has already been made to that position at the current string position and for ultimate return to the same list of callers. Care must be taken in how this is done. One mean would be to associate with each string position and nonterminal in the body of routines, a list of *arl*'s returned from that nonterminal at that string position. At each return the appropriate list could be searched for an identical return list

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This search however requires time proportional to the string length.

The method used here keeps a separate copy of the alternative for each starting string position. The list of callers then contains at most one element and all and only duplicate returns will be saved on the same list. Thus, it is sufficient to associate with each position of the copied alternative, the string position and parse tree at the last return to that position. These changes are incorporated in the program of figure V-G which differs from figure V-F only in line 4 and line 6. A copy of each alternative is made (line 4) as processing is begun of a new alternative and string position. Continuation of processing for an alternative after return from a nonterminal within that alternative is determined (line 6) by the *continue* routine. *Continue* checks whether a return would create identical *alt* processes (same position in the grammar, same starting string position, and same current string position). If not processing of the alternative continues. Otherwise the current parse tree and an ambiguity indicator, *AMBIGUOUS*, are pushed onto the parse tree of the duplicate *alt* process. Note that any such duplicate indicates that there is an ambiguous parse for some head of the given string.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1   parse(g,r,s)==par[V←production(r);
2       seq[ until paths≡1 do when active≡0 then S←T•s; return(v,caller)]]
3   where [v=FALSE;
4   production(p)==[save(caller,p•g) → papp[alt(pdist[x;x;a],NIL,ARL•p•g);a;p•g]];
5   alt(a,tree,arl)==[a≡NIL → papp[impret(self,ar);ar;arl];
6       atom(H•a) → continue(a,cons(production(H•a),tree),arl);
7       H•H•a≡QUOTE → [H•T•H•a≡H•s → when s≠s then alt(T•a,tree,arl)]];
8   save(ar,p)=synch(p,[S•p≠s → par[μ(p,ARL,list(ar),S,s); TRUE];
9           TRUE → par[push(ar,ARL•p); FALSE]], save(ar,p));
10  continue(a,tree,arl)==[synch(a,[S•a≠s → par[μ(a,TREE,tree,S,s); TRUE];
11              TRUE → par[push(AMBIGUOUS,tree,TREE•a); FALSE]], continue(a,tree,arl))
12      → alt(T•a,tree,arl)]];
```

Figure V-G, parser with merged return states.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

H. COMPUTATION OF TIME BOUNDS The above implementation is a variation on Earley's parsing algorithm [Ea 68] and achieves his times. The significance of Earley's algorithm is that it achieves in one algorithm the best times for several classes of grammars. Previously these times were achieved only by special algorithms for each case. The algorithm will parse for any context-free grammar in time proportional to n^3, for unambiguous grammars in time proportion to n^2 and for some classes including LR(k) grammars in time proportional to n.

Let n be the length of the string to be parsed, p be the total number of nonterminals in the grammar, m be the number of nonterminal symbols in the right hand side of all productions, and a be the number of alternatives in the grammar. By constant we mean independent of n, p, m, and a. The time for parsing can be computed by replacing each line of the program (figure V-G) by the time for one execution of that line and then multiplying by the number of times the line is executed. With the exception of lines 2, 4, and 5 all lines require only constant time. The number of times a line is executed is complicated by the multiple parallel return which can cause the latter portion of a line to be executed several times for each execution of the earlier part. This can be corrected by reassociating constant time computations with other parts of the algorithm. In line 6 the constant time required after a multiple return from *production* will be associated with the line 5 return for the *alt* process which made the return.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In line 4 the constant time required before the execution of the parallel application *papp* will be associated with the process which called *production*, either *parse* in line 1 or *alt* in line 6.

The *parse* routine will be executed only once and the loop in line 2 cycles once for each string position, so the time associated with *parse* is proportional to n. The routine *production* (counting only from the *papp* call) will be executed at most once for each production and each starting string position (pn times) because all duplicate calls were eliminated by the *save* routine. The *papp* call in line 4 copies a production and therefore requires time proportional to the size of a production (m/p). The total time associated with the *production* routine is then mn. Because the *continue* routine guarantees no duplicate *alt* processes the number of calls on the *alt* routine can be at most one for each nonterminal in the right hand side of the grammar, each starting string position, and each current string position (total calls proportional to mn^2). Nonfinal *alt* processes (lines 6 and 7) require constant time. The time for a final *alt* process (line 5) will be proportional to the length of the caller list, that is the number of calls on a given production at a given string position which will be at most one for each occurrence of the production in the right hand side of the grammar (m/p) and each associated starting position (n). Only one call in m/a calls on *alt* will be final, so the total time for line 5 is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

man^3/p and the time of *alt* is mn^2+man^3/p. Summing the times for *parse*, *production*, and *alt* the total time to parse a string is at most proportional to $n+mn+mn^2+man^3/p \simeq man^3/p$.

If the grammar is unambiguous then each return (line 5) will cause a new *alt* process to be formed so that the total number of returns throughout the parsing of a string can be no greater than the number of *alt* processes (mn^2). That is, the total time for final *alt* processes is proportional to mn^2, the total time for *alt* processes is mn^2+mn^2, and the time for parsing is $n+mn+mn^2+mn^2 \simeq mn^2$.

If there is a constant bound b at each string position on the number of *alt* processes for a given position of the grammar then the grammar is called bounded state. The number of *alt* processes at each string position will then be at most bm and will be bmn for all string positions. Only bm *alt* processes at a string position means there can be only bm/p return points on a caller list. The time for nonfinal *alt* processes is then proportional to the number of nonfinal *alt* processes (bmn) and the time for final *alt* processes is the product of the number of final *alt* processes (bmn(a/m) = ban) and the time for each, the time for the multiple parallel return (line 4). The time for the return is proportional to the length of the caller list (bm/p), so the time for *alt* processes is $bmn+b^2amn/p$ and the total time for parsing is proportional to $n+mn+bmn+b^2amn/p \simeq b^2amn/p$. It turns out that, except for certain right recursive

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

grammars, LR(k) grammars are bounded state.

I. HYBRID RECOGNIZERS Regardless of the implementation the sidetrack and multiple parallel return controls can be used in forming recognizers and parser. The only fundamental property of the routine for a nonterminal is that they return once for each instance of the nonterminal at the head of the given string (that is, that they make the multiple parallel return). Thus some or all of the nonterminal routines could be replaced by other routines which perform the same function. If the processing is left-to-right in the string (as with our implementation), for example, then some of the nonterminal categories which are operator precedence could be replaced by calls on an operator precedence recognizer [Fl 63]. The implication of this is that hybrid recognizers can be built merging several well known syntax analysis techniques. An obvious simple case of this technique which is used in most translating systems is the lexical analyzer which is called as a coroutine and uses more efficient methods that are available for processing the more complex syntactic categories.

The technique for building hybrid recognizers will be illustrated by combining the sidetrack recognizer of figure V-D with a simple precedence recognizer. We will assume that any special processors to be used within the sidetract recognizer will be described in the CDL. Thus, we need only extend the *alt* routine of figure V-D so that the default case (i.e. the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

case in which the next element is neither a terminal or non-terminal symbol) is interpreted as a call on a CDL routine. The revised *alt* process is shown if figure V-H. Recall that the expression *eval(x,self)* causes the evaluation of the expression *x* by the CDL interpreter in the context of the current environment.

```
alt(a,s,ar)==[a≡NIL → mpret(s,ar);
        atom(H•a) → alt(T•a,production(H•a,s),ar);
        H•H•a≡QUOTE → [H•T•H•a≡H•s → alt(T•a,T•s,ar)];
        TRUE → eval(H•a,self)];
```

Figure V-H, extension for hybrid recognizers.

The precedence recognizer (shown in figure V-I) has four parameters: a *stack* used to temporarily hold operators until an operator of lower precedence is encountered, the string *s* to be recognized, the name *opnd* of the syntactic catagories of the operands to be recognized, and a precedence table *pt* which for each operator contains a numeric precedence (high values for strong binding, low values for weak binding). For the precedence recognizer in figure V-I we assume that all the operators are binary. That is, this recognizer will recognize strings of the form: $x' = opnd \mid x\ bop\ x$ where bop is the set or binary operators and opnd is the form of the operands. The recognition of opnd may require the full sidetrack recognizer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1    pr(stack,s,opnd,pt) ==
2       seq[ S←production(opnd,s);
3       until (H•s)•pt≡UNDEF do
4           seq[until (stack≡NIL)∨((H•s)•pt>(H•stack)•pt) do
5                   STACK←T•stack;
6           S←production(opnd,s)];
7       until stack≡NIL do
8           STACK←T•stack;
9       mpret(s,caller)];
```

Figure V-I, precedence recognizer.

Because the precedence recognizer is being used as a recognizer it could be replaced by a simpler routine of the form:

```
pr(s,opnd,pt)==seq[ S←procuction(opnd,s);
   until (H•s)•pt≡UNDEF do
      S←production(opnd,T•s);
   mpret(s,caller)];
```

If we are interested only in recognition then the precedence is of no consequence and the operator stack is not needed. The form in figure V-I, however, indicates how the precedence recognizer can be converted to a parser which will output a representation of the expression as a post-fix string. To obtain the post-fix representation it is only necessary to output (i.e. form them into a list) the operands as they are recognized (lines 2 and 6) and to output the operators when

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

they are removed from the stack (lines 5 and 8). The resulting output string is the post-fix representation.

As a specific example of a hybrid recognizer consider the sidetrack recognizer for Boolean expressions of Algol-60:

Boolexp = simpbool | ifclause simpbool ELSE boolexp
simpbool = implication | simpbool $≡ implication
implication = boolterm | implication $⊃ boolterm
boolterm = boolfactor | boolterm $∨ boolfactor
boolfactor = boolsecondary | boolfactor $∧ boolsecondary
boolsecondary = boolprimary | $¬ boolprimary
where *ifclause* and *boolprimary* are also described for the sidetrack recognizer. The Boolean expression of Algol could be recognized by the precedence recognizer if the Boolean secondary is taken as the operands to which the binary operators apply. The hybrid recognizer is formed by rewriting the above productions in the following form:

boolexp = simpbool | ifclause simpbool ELSE boolexp
simpbool = pr(NIL,s,BOOLSECONDARY,μ_0($≡,1, $⊃,2, $∨,3, $∧,4))
boolsecondary = boolprimary | $¬ boolprimary

It should now be clear that any special purpose recognizer could be substituted for nonterminals within a sidetrack recognizer. The only problem of concern occurs when the syntactic category to be replaced is not terminal relative to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the special recognizer, that is, the special recognizer is not capable of recognizing all the constituant nonterminals of the replaced syntactic category. For the lexical analyzer this was not the case, but with the Boolean expression the nonterminal *boolsecondary* could not be recognized by the precedence recognizer. Consequently, the sidetrack recognizer was called from within the precedence recognizer to recognize occurrences of *boolsecondary*, and because the sidetrack calls might return several times (i.e. multiple parallel returns) for each call it was necessary (line 9 of figure V-I) to make a multiple parallel return from the precedence recognizer.

J. AN EXTENSION USING A DELAYED EXECUTION CONTROL The notation used here for describing a context-free grammar is similar to the notation used in chapter III to describe the syntax of the CDL. The difference being that there we simultaneously described two grammars (i.e. the concrete and the abstract syntax) in order to specify their correspondence. We would like an interpretation of the syntax specifications which would not only give the correspondence between the concrete and abstract syntax, but would also act as a translator which when given a string in the concrete language would emit the corresponding representation in the abstract syntax.

The abstract syntax is specified by a generative grammar whose primitive operations (cons instead of concatenate) are those of the CDL. Consequently, to generate a "terminal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

string" it is necessary only to execute the productions beginning at the root. For each occurrence of a nonterminal some alternative of that nonterminal must be generated. Now however the proper choice can be determined by the parse tree for the concrete syntax. The interpretation of the abstract syntax for any given alternative follows: The abstract syntax for an alternative is a list of simple expressions in the CDL. These expressions are to be each evaluated and the resulting values "consed" together in the order given to form the corresponding "terminal string". When evaluating the expressions, names of nonterminals will be encountered (as atoms), these reference the representation for the corresponding nonterminal in the parse tree of the string of the concrete language. The generation of the representation for any alternative should be delayed until it is known that that alternative will appear in the final parse tree of the concrete language string, that is until the return for the call on the root routine is made.

For a translator both the concrete syntax and the abstract syntax for a nonterminal form the body of the routine for that nonterminal. Each alternative is a list. The earlier elements on an alternative list are either atoms standing for nonterminals or quoted atoms standing for terminal symbols. The later list elements are expressions of CDL. The two sections of an alternative are separated by the atomic list element "#". As with parsing, the elements of the alternative lists are executed in order from left to right, but the return from the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parse now occurs when the "#" is encountered. This return acts to delay execution until the root is parsed. This return then is not only a multiple parallel return but also a coroutine return. If the returned node of the parse tree is in the parse tree when the root is recognized then it will be resumed to generate the corresponding representation prescribed by the abstract syntax. Processing of an alternative beyond the "#" takes place in two steps. First, each of the returned coroutines are again called to generate the corresponding representation for the nonterminals appearing in this alternative. These representations are then paired with their nonterminal name in the local environment of the alternative. In the second step the rest of the alternative is evaluated as if it were a list of arguments to a *cons* routine.

The syntax for syntax specifications is given in figure V-J. The syntax for the nonterminal *pexp* was given in figures II-A and III-C. The interpreter for the abstract syntax of the syntax specification language is given in figure V-K. This interpreter is in fact a translator for the described language. The interpreter of figure V-K differs from the parser of figure V-E only in the additional call on generate (line 2) to build the representation in the abstract syntax, the return (line 5) from a nonterminal process upon encountering an "#" instead of *NIL*, and the pairing of the nonterminal name with its parse tree (line 6). The parameter *e* to translate (line 1) is the global environment of the grammar.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
syntax = syndef                 # g
     | syntax syndef            # g
syndef = id $= altlist          # (μ(g,id,altlist))
altlist = alt                   # alt NIL
     | alt $| altlist           # alt altlist
alt = concrete $# abstract      # concrete $# abstract
     | concrete alt             # concrete alt
concrete = id                   # id
     | capid                    # QUOTE capid NIL
     | $$ symbol                # QUOTE symbol NIL
abstract = pexp                 # pexp NIL
     | pexp abstract            # pexp abstract
```

Figure V-J, syntax for syntax specifications.

```
1    translate(g,r,s,e)==par[V←H•production(r,s);
2        when paths≡1 then return(generate(v),caller)]
3    where [v=FALSE;
4    production(p,s)==papp[alt(a,NIL,s,caller);a;p•g];
5    alt(a,tree,s,ar)==[H•a≡$# → return(cons(self,s),ar);
6        atom(H•a) → alt(T•a,cons(list(H•a,H•x),tree),T•x,ar)
7                where x=production(H•a,s);
8        H•H•a≡QUOTE → [H•T•H•a≡H•a → alt(T•a,tree,T•s,ar)]];
9    generate(alt)=seq[papp[μ(lenv,H•x,generate(H•T•x);x;TREE•alt];
10       cons(pdist[eval(x,lenv);x;T•A•alt]:)]
11       where lenv=μ₀(GLOBAL,e)];
```

Figure V-K, interpreter for translator.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

K. ERROR CORRECTION For a recognizer to be of practical value it must not only be able to recognize a correct string, but must be able to detect errors in the string and continue the recognition process beyond the errors. Again the approach will be to first determine the control structure appropriate for error correction and to then look for specific error correction techniques within the context of that control.

How do we detect errors when recognizing strings? From the sidetrack recognizer in figure V-D we see that an error in the given string will cause all the parallel control paths of the recognizer to terminate. That is, failure occurs when all control paths terminate without completing the recognition. Some one of these paths would have continued to the correct recognition had not the string had an error. Thus, the control structure for error corrections will permit the string to be corrected on the path which will lead to a correct recognition and should continue that path beyond the point of error.

There is of course no way of knowing what the programmer intended when he wrote a syntactically incorrect string, so the control structure should continue all the failure paths with one or several trial corrections on each path. In the sidetrack recognizer all failure paths terminate when a terminal symbol of the grammar fails to match the symbol at the current string position (see line 7 of figure V-D). Thus, for error recovery

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the control path should not terminate when the match fails, but should wait until all other paths fail (if they do) and then try some corrections and continue processing. The change required for line 7 of figure V-D follows:

```
H•H•a≡QUOTE → [H•T•H•a≡H•s → alt(T•a,T•s,ar);
               TRUE → when active≡0 do correction]]];
```

This control structure will cause an error correction on each failure path exactly when all paths have encountered an error. Consequently, at any point within the recognition, if some active path has made n corrections in its history, then all other active paths will also have encountered n errors and corrected n errors while all waiting paths will have encountered n+1 errors and correct the first n of them. This means that the first recognition of the given string will have the fewest possible number of corrections which could be used to convert the given string into a correct string. One aspect of this error correction control which differentiates it from most others is that this control permits corrections to be made anywhere (including to the left) within the given string and not just at the right most position beyond which all paths fail. Also, because it is associated with a recognizer for any context-free language it is not dependent of the properties of any particular grammar (e.g. identification of statement boundries to limit the context of errors in sequential languages).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

What error correction methods should be used? Because the failures occur when a symbol of the grammar fails to match a symbol of the given string, we might suppose that the symbol of the given string was in error (i.e. it should not be present). Thus, correction might be the expression *alt(a,T•s,ar)* which deletes the current symbol from the given string and continues processing. Alternatively, we might suppose that a symbol (in fact the current symbol of the grammar) is missing from the given string and continue processing with the given string at the next position of the grammar. Condition would then be the expression *alt(T•a,s,ar)*. Our personal preference is to try both of these corrections in parallel. That is, we try both deleting a symbol from the given string and inserting a new symbol into the given string. This also means that if two consecutive failures occur the trial corrections will include not only a double deletion and a double insertion, but also replacements (i.e. insertion and deletion at the same point). The point to be made is not that any particular correction scheme is best, but that with an error correction control structure many different schemes can be tried.

Next we might ask if this correction control can be embedded in a recognizer which is implemented with Earley's algorithm instead of the full sidetrack control. The answer is no. To achieve Earley's time bounds we required that the *alt* processes be processed in order of the string position. The above control resumes processing on all paths regardless of the string

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

position. This may explain why most error correction schemes do not make corrections to the left of the string position which causes the last remaining path to fail. If we restrict the correction control in this manner (i.e. continue processing only at the right most column at which a failure is found) then the correction control can also be implemented within the context of Earley s algorithm. Actually, this restriction is not as severe as it might appear. First, in our grammars the terminal symbols represent units with semantic meaning (e.g. identifiers rather than characters, or even expressions in the case of hybrid recognizers) so that each correction has semantic as well as syntactic significance. Secondly, single isolated errors will be processed the same regardless of the restriction, while paired errors will still be corrected in one of the several possible ways (e.g. with the restriction a left parenthesis without a corresponding right parenthesis would be corrected by inserting a right parenthesis while without the restriction it would also be corrected by deleting the left parenthesis).

There are other advantages to error correction in a control context in which similar return states are merged. When error correction is necessary during a recognition the number of possible corrections which will produce a correct string using a minimum number of changes may be quite large. Because return states are merged we can detect ambiguities (see line 11 of figure V-G). If an anbiguity arises from insertion of several

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

different single symbols at the same point then the ambiguity can be resolved by restating the correction as an insertion of the nonterminal of which the inserted terminals were an example. Because processing is by string position rather than by the number of corrections, the number of corrections in the history of a control path will vary. When ambiguous recognitions arise from alternative corrections, however, they can be resolved in favor of the path with the fewest (if they are different) corrections in its history.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER VI

CONTINUOUSLY EVALUATING EXPRESSIONS

The continuously evaluating expression can be viewed as an analogue device which continuously meters the value of an expression. Once the expression is activated it acts as if it were evaluated continuously. If the value of any free variable of the expression changes then the value of the continuously evaluating expression is immediately recomputed to reflect the change.

A. SOME USES AS AN EXPRESSION Such a control structure is needed for monitoring in programming languages. Monitoring facilities have been variously referred to as *"when"*, *"wait until"*, and *"on condition"*. Each argument to a *monitor* operation is a rudimentary form of continuously evaluating expression. In any case, each of the above requires a continuously evaluating Boolean expression to monitor some condition and trigger the designated response when the condition becomes true.

Expressions used as Algol call by name parameters can also be viewed as continuously evaluating expressions, where the actual parameter defines the continuously evaluating expression in the environment of a procedure call and each occurrence of the corresponding formal parameter name within the procedure body reference the current value of the expression.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The possibility of race conditions arises when a continuously evaluating expression has side effects which change the value of the variables to the expression. Generally, continuous evaluation would be restricted to expressions without side effects or the side effects although present would not be continuous. For a given control environment rules can be given to provide for discrete times at which the side effects occur. In Algol the side effects produced by the evaluation of call by name parameters are effective only at those times when the value of the parameter is referenced. That is, the actual parameter can be viewed as a continuously evaluating expression, but the side effects of any evaluation become effective only at the time the value of the expression is referenced. In hardware the periodic clocking of information into registers provides the same function.

B. SOME IMPLEMENTATIONS There are several ways in which the continuously evaluating expression might be implemented. One method is that commonly used to implement Algol call by name parameters: reevaluate the expression each time its value is used. This method can also be used for monitoring in single control path environments because interrupts can only occur at discrete points in a computation. Even in machine languages, the hardware interrupts become effective only at the completion of instructions. In higher level languages the "instructions" will generally be more complex and thus reduce

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the frequency at which the condition must be tested.

Alternatively, the expression might be reevaluated only when the value of one of its free variables changes. An evaluator for this interpretation of continuously evaluating expression is given in figure VI-A where *v* is a list of the free variables, *exp* is the expression to be evaluated, *env* is the environment of the evaluation, and *t* is an expression which is used to terminate the continuously evaluating expression. The routine *evcee* returns once initially and then once for each change in one of the free variables of the expression *exp*.

```
evcee(v,exp,env,t)== until eval(t,env) do
   par[return(eval(exp,env),caller);
       monitor(pdist[list(x,env,$≠,x•env);x;v]:)];
```

Figure VI-A, reevaluate on change.

The routine *evcee* might be called using the form:

```
X ← cont(evcee(v,exp,env,t));
```

where *x* is a variable always containing the current value of the continuously evaluating expression and the operation *cont* is used to guarantee that reevaluation of the expression can be completed between any two consecutive steps of the calling program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C. A TREE REPRESENTATION The method proposed here represents the continuously evaluating expression as a tree and borrows from both of the above methods. The tree for the expression (a-b)x(a-c)+5 where a=3, b=3, and c=2 is shown in figure VI-B. Each terminal node is either a constant or a free variable of the continuously evaluating expression together with its value. Each nonterminal node of the tree is a pair having an operator of the expression and a value obtained by applying the associated operation to the values of its descendant nodes. The value at the root node is the value of the continuously evaluating expression.

When changes occur in the values of the free variables of a continuously evaluating expression, the new value can be computed by propagating the change up the tree from the terminal node whose value changed. It is not necessary to reevaluate every node, in fact the number of expressions to be evaluated will be proportional to the logarithm of the number of operators in the expression. If the initial tree is as shown in figure VI-B and the value of b is changed to zero (0) then the change will be propagated as shown in figure VI-C.

Several changes can occur simultaneously in the values of terminal nodes 1) when a variable appears more than once within the continuously evaluating expression such as variable a in figure VI-B, 2) when the language permits simultaneous assign=

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ments such as $b, c \leftarrow c, b$ meaning exchange the values of b and c, and 3) when several assignments occur within one "instruction".

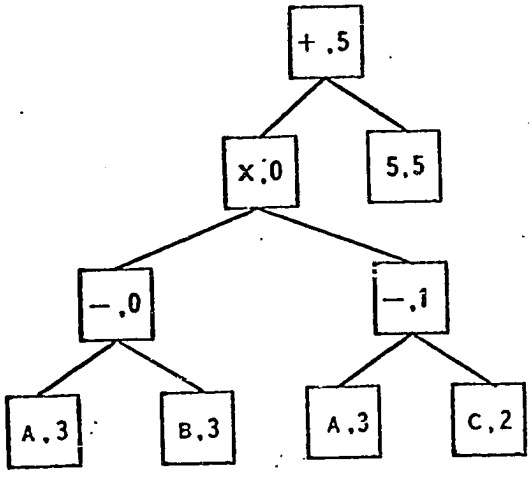

FIGURE VI-B. TREE FORM

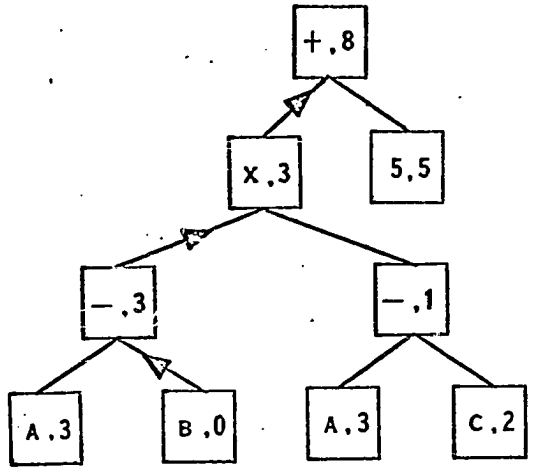

FIGURE VI-C. CHANGE PROPAGATION

When several terminal node values have changed, the order of propagation becomes important. Consider an initial tree as shown in figure VI-B. If the variable a is assigned the value 2 then the tree might be updated by first propagating the change from the left a node (figure VI-Da) and then from the right a node (figure VI-Db). This order introduces a transient change in the value of the root node and the value of the expression.

Changes need be propagated only as long as they cause the value at each node to change. If any descendant of a node changes then it is necessary to reevaluate that node, but if that reevaluation does not change the value of the node then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

its ancestor nodes need not be reevaluated. Thus, if changes are propagated in order of tree level, that is all changes at the i^{th} level from the root are processed before any at the $i-1^{st}$ level, transient changes and other changes which do not alter the value of the value of the expression will not propagate to the root node. Figure VI-E shows the same changes as figure VI-D but with propagation by level. Strictly, the propagation order need only be such that all descendants of a node are reevaluated before that node.

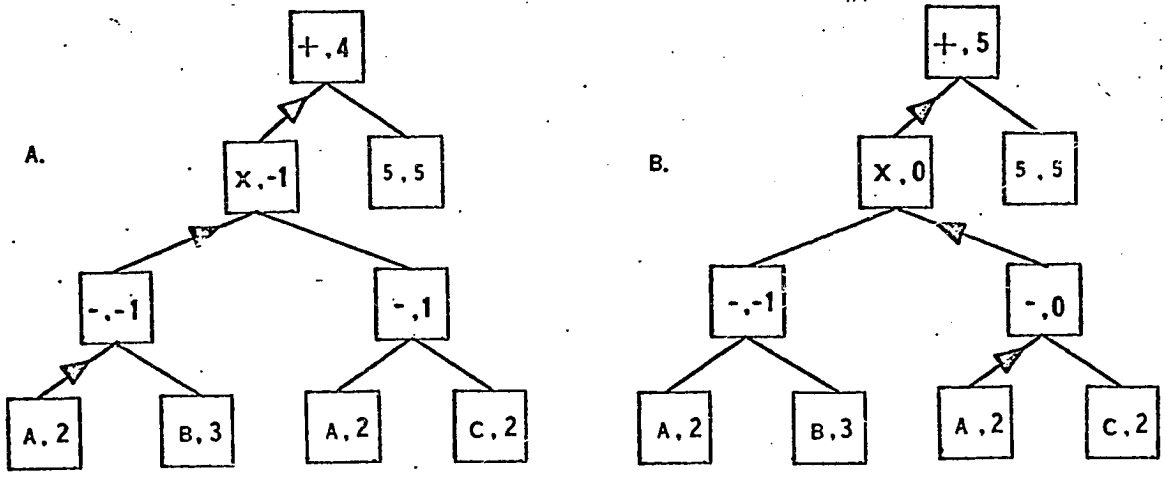

FIGURE VI-D. SEQUENTIAL PROPAGATION.

If an expression is evaluated each time its value is used then there will be unnecessary computation whenever a value is used several times without intervening changes in the values of the variables of the expression. On the other hand, if an expression is evaluated each time the values of its variables change then there is unnecessary computation whenever

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the values of variables change several times without intervening use of the value of the expression. Consequently, two conditions will be required before changes will be propagated through the tree of a continuously evaluating expression: 1) at least one of the terminal nodes must have changed value since the expression was last evaluated, and 2) there must be a request for the current value of the expression. Whenever the value of a free variable of the expression changes, the tree for the expression will be marked to indicate that a change has occurred but the change will not be propagated. All changes since the last value request will be propagated when the next value request is generated. If a variable changes several times between requests for the value, only the latest change will be propagated. If several requests for the value occur between changes, the same value is used over and over.

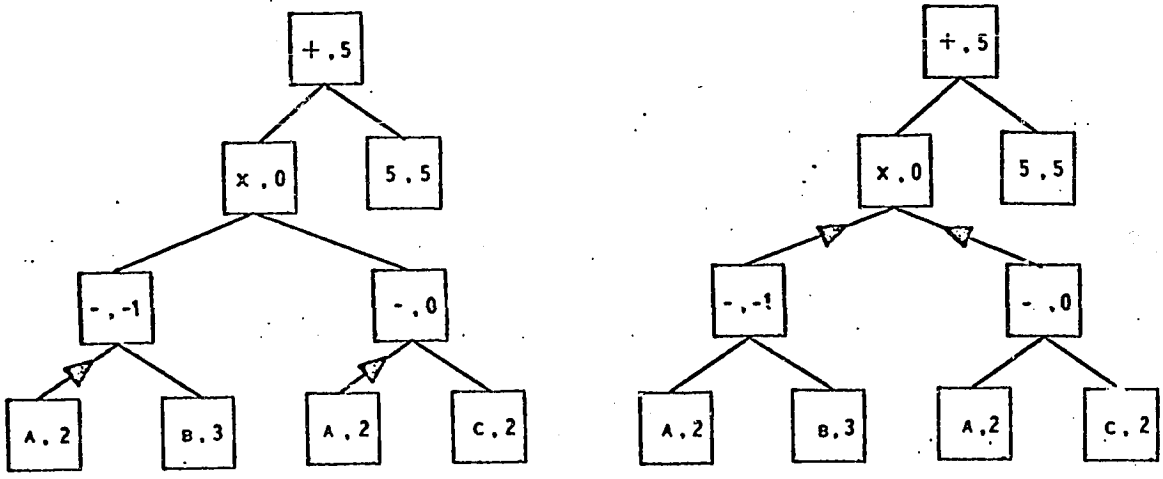

FIGURE VI-E. PROPAGATION BY LEVEL

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

With each nonterminal node of the tree will be associated an indicator which, when set, indicates that the node must be reevaluated. When the value of a free variable of a continuously evaluating expression changes, the indicators for the direct ancestors of the terminal nodes for that variable will be set. The nonterminal nodes will be ordered so that each node precedes its ancestors, for example the increasing numeric order of nodes as shown in figure VI-Fa.

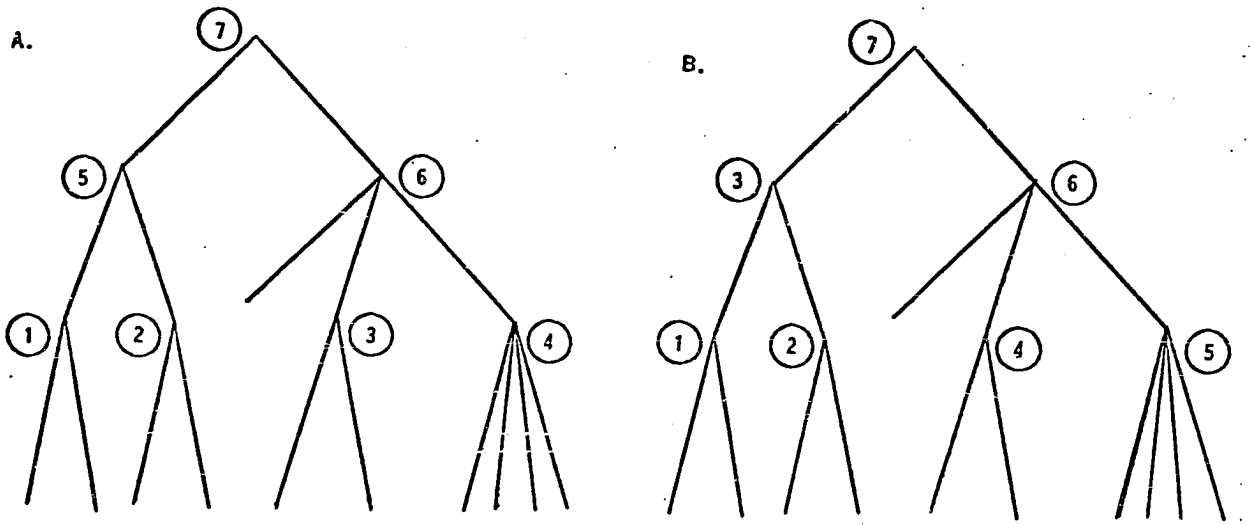

FIGURE VI-F. ORDER OF NONTERMINAL NODES.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

D. COMPILATION OF TREE FORM EXPRESSIONS. Code for a sequential machine can then be compiled to propagate changes. For each node *i* the following code where node *a* is the ancestor of node *i* will be compiled:

```
test(i):    if (indicator on node i is not set)
                then go to test (i+1);
Compute(1): reset indicator for node i:
            compute value for node i;
            if value of node i is unchanged
                then go to test(i+1);
            set indicator for node a;
            go to test(i+1);
```

Figure VI-G, compiled sequential code.

Execution begins at *test(1)* and is completed at *test(n+1)* where *n* is the number of nonterminal nodes in the tree. If the computation of node *i* causes a change and *a=i+1*, then there is no need to set the indicator for node *a* and immediately test it. That is, for each node *i* for which *a=i+1*, such as node 6 in figure VI-Fa, the last two lines of code above can be replaced by: *go to compute(a)*. The latter savings can be maximized by numbering the nonterminal nodes so that each node *i* which has a nonterminal descendant will always have node *i+1* as one of its descendants. The descendant of node *i* which is assigned *i+1* should be that descendant whose value is most likely to change. For example, the tree in figure VI-Fa might

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be renumbered as shown in figure VI-Fb.

The propagation of changes as described, requires that the indicator for each nonterminal node be tested. In some machines this can be done economically by storing the indicators as bits left-to-right in a word and using a leading one's detector to determine the first one set. Alternatively, one might keep an ordered list of the nodes needing reevaluation instead of an indicator for each nonterminal node. When the value of a node changes its ancestor will be added to the list.

The method proposed here will be of value only if the values of variables of a continuously evaluating expression change infrequently relative to requests for the values of the expression or if the time for the computations bypassed when changes are propagated is greater on the average than the time required for the additional bookkeeping. The latter will be the case if the operations associated with nonterminal nodes are comparatively lengthy, such as matrix or vector operations. In other cases, the bookkeeping costs can be reduced by combining several operations to form a single complex operation, a binary tree of sums might be formed into a single node having a summation operator. In some cases, the root node might be the only nonterminal node, but the method could still be used to advantage if several variable changes occur between requests for the value of the continuously evaluating expression

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

containing those variables, or if several value requests occur between changes in the values of the variables.

E. CONTINUOUSLY EVALUATING SYSTEMS There is no need to restrict the node of a continuously evaluating expression tree to simple operations. They might also be expressions or programs. These expressions or programs could contain *monitor* operations which would serve to delay propagation within certain branches of the tree until prescribed conditions arise. Once branches have a provision for delay it becomes meaningful for nodes to have sideeffects (i.e. to change the environment). Loops can also appear within the continuously evaluation expression tree.

This more general view of continuously evaluating expressions can be taken at the environmental control structure for describing whole systems. The tree nodes are then the constituent parts of the system while the connecting arcs are data flow paths and indicate the interdependencies among the various parts of the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER VII

FURTHER STUDY AND CONCLUSIONS

This investigation has been a first look at control structures. The control structure of a programming language is its sequencing or interpretation rules and as such provides a programming environment which encompasses any task within that language. A control regime influences our approach to problems, it establishes a view point and an environment for attacking problems within the context of that view. Many languages limit us to single step by step processes which reflect the underlying sequential machines. This may force awkward simulations of nonsequential processes on us.

If the primary control structure of a language were the continuously evaluating expression then our view would be one of combining continuous simultaneous computing mechanisms into circuits which represent the desired computation (this is a world view shared by the hardware circuit designer.) If the programming environment were a multiple sequential control then one would naturally divide a task into several semi-independent processes each which is thought of as a number of sequential steps. This division may differ from the hierarchical divisions of most current languages because with the multiple sequential control it is just as natural to divide along an axis (coroutine structure) perpendicular to hierarchical (subroutine) structure.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The control regime of our language for describing control has neither the concurrent processing bias of the continuously evaluating expression or the sequential processing bias of most programming languages. Instead both parallel processing and sequential processing are used as they conform to our conceptual view of the control structure. Rather than finding a representation of a control in terms of parallel networks or in terms of sequential step by step processes, the problem is to isolate our conceptual view of the control. This means that in developing a control structure appropriate to a task area one must look for solution methods for problems in that area without the initial imposition of a particular control regime, and then abstract the control structure from those methods.

The formal device proposed here for describing control is a language in which one describes the machine which interprets a control structure either directly or by extension. This device differs in two important ways from similar facilities for the formal description of the semantics of programming languages. First, it includes a set of primitive control operations which, as far as we have been able to determine, constitute a basis for the mechanisms underlying control structures. Secondly, although our formal description of a language is an interpreter for the language, we take a different view of the semantics for a language and therefore ascribe a different meaning to the interpreter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Formal languages used for describing interpreters of languages have themselves been programming languages with the standard control structures: sequential control, a conditional, iteration, and recursion. Consequently, the interpreter is a device which simulates a given control structure or language in terms of the sequential primitives. In fact there is an implicit assumption when one says that an interpreter provides the semantics of a language. Namely, the semantics of a language is its behavior when viewed as a black box. The interpreter describes the semantics by giving a formal explication of what that behavior will be in each case, but not necessarily how it is accomplished.

Our view of semantics includes not only the behavior which can be observed but the way in which that behavior is achieved. If a language has a parallel processing capability then we would claim that no interpreter which simulates that language using sequential control can provide an adequate description of the language. That is, no pseudo-parallel scheduling algorithm will ever convey the idea of concurrency to the reader of that description. Concurrency is not a concept which can be composed from the sequential processing primitives, and so a description of control structures which involves concurrent execution can be given only in languages which have some form of concurrent execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Carrying this view further, the semantics of a language is taken to include not only the effect of any program of the language when viewed as a black box, but also the means by which that effect is achieved. This imposes a programming convention when describing the interpreter for a language: the interpreter will be written so that whenever a program calls for a control action, the same control action will be carried out in the interpreter as part of the interpretation process. The control structure of the interpreter will for each program be the same as the program being interpreted.

The value of a formal device for describing control comes not from the device in isolation, but in combination with facilities for describing other language components. In this thesis the goal has been the investigation of control structures in their own right. Other descriptive facilities have been introduced only as required. Even so we had to introduce a meta-system for describing syntax and to use a general purpose data structure in lieu of a data definition facility.

One obvious area for further research is in extensible languages. A practical means for extending the control structure of a language as well as the data and syntax could have a significant impact on the usefulness of extensible languages. Current extensible languages either lack facilities for extending control or permit extension only in the form of a sequential interpreter for the desired control. Qualitive

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

changes in languages arise from the introduction of new environmental control structures. If the desired control is similar to what is available then there is little value in change, but if it is much different (i.e. nonsequential control) then it must be simulated using sequential control whether in an extensible language or some other language. Thus, there is little value with respect to control to a user of most proposed or available extensible languages.

The importance of data and syntax should not be underrated. Arguments for variability in data structures similar to those for control can also be given. In fact, the approach in language design should probably be to first determine the data structures suited to a task and then let them dictate the sequencing rules. On the other hand, it may be that the facilities for data definition have themselves been biased by a view whatever the sequencing rules they will be essentially sequential. A reexamination of data definition facilities may be worthwhile.

Another hope for extensible languages is that formal machine independent specifications of languages will lead to greater portability of languages from machine to machine. Another kind of portability may be possible when a language includes control extension facilities: efficient implementations for controls and optimizing (or debugging, or verifying) interpreters can be transferred among languages as control extensions as long as the languages share the same controls.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The study of control and of means for incrementally extending control is important for parsimony in both programming languages and their implementations. Control structure tends to be invariant over large sections of programs and therefore notations can be used within the language to make the constant parts of the control implicit. An ability to modify and extend the control means that the interpreter can also take advantage of the constancy in control to provide more efficient implementations. This process is aided by explicit specification of cases where execution order is of no concern so that an actual order can be determined by the interpreter on economic grounds rather than by the user as an arbitrary choice, and by structured descriptions (although not always hierarchical) of the control and interpretation which allow changes to be made without starting over each time a new technique is to be added. Some examples of this procedure were seen in chapter V.

There is nothing sacred about the form of our device for describing control, the choice of primitive control operations, or the various views taken of control. There are however some general rules which can be suggested: any facility for describing control should have an environmental control structure of its own which permits the descriptions to be decomposed into logical units (remembering that the logical structure is not necessarily hierarchical), the set of primitive control operations should provide at least the functions of those proposed here, and it should be possible to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

take several alternative views of control and thereby get different perspectives of a problem.

This dissertation has attempted to demonstrate our thesis: complexity diminished and clarity increases to a marked degree if algorithms are described in a language in which appropriate control structures are primitive or easily expressible. A number of control structures extant in programming languages and software systems are catalogued and then used as a guide to develop a programming language which has a control extension facility. This language not only has the mechanical necessities for control extension, but also has primitive control operations for sequential processing, parallel processing, alternative selection, monitoring, synchronization, and relative continuity. These operations were the source of the clarity and simplicity of the control descriptions because they span our conceptual notion of control and because they can be easily composed to form more specialized control structure.

The thesis was demonstrated by using the control description language to give formal descriptions of itself, the simulation language Sol, and a variety of specific control structures. Some nonstandard control structures (including sidetracking, continuously evaluating expressions, and multiple sequential control) were invented to show the clarity and simplification which can result from a conscious examination

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of control. It was also shown that the use of nonstandard controls in some cases (e.g. parsing and continuously evaluating expressions) leads naturally to efficient implementation.

This investigation has only scratched the surface, but it is hoped that it has added to our understanding of control, illustrated the clarity and simplification that can result from a conscious examination of control, and demonstrated the large, although unrealized, potential for variability in control. In conclusion, we would like to point out a few candidates of other useful nonstandard control structures. The fuzzy control, filters, and reversible subroutines outlined below have not been fully worked out, but suggest lines for further work and may be of philosophical interest.

Fuzzy control. The fuzzy set [Za65a] [Za65b] is an imprecise collection of objects whose imprecision arises from there being no sharply defined boundaries. Instead fuzzy sets have grades of membership between full membership and nonmembership. Such things as "bald men", "numbers much greater than 10", and "Algol-like languages" are fuzzy sets.

Many problems dealt with in machines and in the real world involve fuzzy concepts in the above sense. The artificial intelligence area employs fuzzy "algorithms" to select heuristics appropriate to a task. The heuristics themselves are often imprecise, although they may be made artificially

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

precise to incorporate them into programs. The "algorithm" for parking an automobile is fuzzy, it must deal with a number of fuzzy sets including the dimensions of the parking place and the position of the car.

The fuzzy control structure is then a control which permits the execution of programs which are incompletely specified because they employ fuzzy sets in their description. Such a control would require a representation of fuzzy sets. When operations are encountered the paths to be pursued may be determined by the value of a fuzzy set, that is no precise decision can be made. In the latter sense the fuzzy control is similar to nondeterministic control where each of several paths must be pursued because the selection criterion is not available at the time the decision must be made.

The difference between fuzzy control and a nondeterministic control such as backtracking is that the fuzzy control is not expected to produce a precise result. Thus, although many control paths must be pursued the cost of each can be reduced by approximating the computation instead of carrying each out in detail. A fuzzy control structure might interpret an iterative loop as follows: determine the approximate effect of a single iteration of the loop, determine approximately the number of iterations, and then multiply these to obtain an imprecise description of the effect of the loop.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fuzzy control could be used for program composition and debugging in a conversational computing situation. During the composition and debugging phases parts of a program may be imprecisely defined or missing entirely. Some conversational systems [Iv 62] [MPV68] execute incomplete programs until an undefined item is encountered, that item must then be defined before processing can continue from that point. With a fuzzy control structure an approximate specification of the effects of the undefined item could be supplied, or the item might be ignored entirely (possibly with warning to user). Even if a program has been completely specified, a fuzzy control structure can be of value for debugging problems which arise from the logical complexity of a program and not from the computational complexity of individual steps. Yet the costs of a debugging run are determined primarily by the computational complexity of the individual steps. Consequently, the fuzzy control will permit the logic of a program to be checked without incurring expense for computations whose results are not important to the task at hand (i.e. debugging).

An important ingredient in making the fuzzy control effective is feedback. In parking an automobile one uses very imprecise calculations to determine the direction and distance to move at each step, but there is constant visual feedback and once in a while feedback from a bumper touching in front or in back or from a tire touching the curb. In the debugging

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

situation the user can limit the fuzziness of computations and guide the selection of alternative paths for analysis.

It can also be argued that fuzzy control processes occur in the brain. Von Neumann [vN 58] argues the importance of numerical procedures in the brain and the high precision arithmetic that would be required. The necessity for numeric procedures results from the requirements for constancy of temperature and pressure and chemical isostasy in various parts of the body. With methods similar to those used in computers ten to twelve decimal digits precision in computations would be required to adequately reduce the errors multiplied by the many logical levels of a computation. He also points out that data transmission frequencies in the brain make such precisions implausible. Data is transmitted by periodic or nearly periodic trains of pulses and this suggests a statistical behavior with high reliability but low precision. We would add that it also suggests a pipeline system in which logical depth of a computation could be great but because the data varies on a nearly continuous basis the changes at any given stage of the pipeline are minimal. Also since it is a system with constant feedback, extremely fuzzy calculations can be used with direction and magnitude of changes determined by the feedback rather than precise prediction of effects. The above must remain an exercise in fuzzy thinking until we have a more concrete handle on fuzzy control structures.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Filters. A filter is a device which is placed between two other systems to limit the flow between them. A person at a console might place a filter (in the form of a program) between himself and the standard console handling system to intercept any system-to-user signals which are not currently desired or which can be processed by a constant algorithm without direct intervention by the user. A filter in the other direction would be a program which talks to a console without passing all the user requests on to the underlying system.

Filtering is also an important characteristic of operating systems. An operating system intercepts the many machine interrupts and passes along only those of direct concern to the user. In multiprogramming systems the operating system also isolates each user from the other so that in many cases each program acts as if it were the only program in the system.

Reversible subroutines. Some unusual classes of subroutines have been suggested [RF 65] in the context of a single address machine which can execute instructions in a forward or backward direction. When a sequence of instructions are executed either the given instructions or their conjugates can be the interpretation. It is then possible to write palindromic subroutines whose instruction pattern is the same whether viewed in the forward or backward direction, reversible subroutines whose transpose (reverse execution) is its own inverse, and Hermitian subroutines which are identical to their transposed conjugate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These can be classified as little more than curiosities.

If examples could be readily found their value would be questionable, but in practice even the simplest examples must be contrived. The major difficulty is that many instructions either do not have conjugates or are not very useful when executed in reverse. This is particularly true for assignment and control operations, for they have neither conjugates nor useful backward interpretations. Both the assignment and go to cause an irrecoverable loss of information (previous value of the variable and location of the go to) and therefore have no conjugate. As for backward execution: what possible meaning can be associated with programs which first use the value of a variable and then assign the value? Or, how can the arms of a conditional be meaningfully processed before the condition is examined?

A useful system which can run programs backwards is Balzer's XDAMS [Ba 69]. This system is used for debugging and keeps a history tape of all changes to memory and of all transfers of control. These changes in memory and control are then used to generate views of the program as it runs either backward or forward.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

[An 65] Anderson, J.P. Program Structures for Parallel Processing. *Comm. ACM 8* (December, 1965), 786-788.

[Ba 57] Backus, et al. The FORTRAN Automatic Coding System. *Proc. WJCC 11.* 1957. 188-198. Also in [Ro 67]. 29-47.

[Ba 69] Balzer, R.M. EXDAMS - EXtendable Debugging and Monitoring System. *Proc. SJCC 1969, 34.* May, 1969. 567-580.

[BBH63] Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., and Strachey, C. The Main Features of CPL. *The Computer Journal 6* (July, 1963), 134-143.

[Be 69] Bell, J.R. Transformations: The Extension Facility of Proteus. In [CS 69]. 27-31.

[Be 66] Bernstein, A.J. Analysis of Programs for Parallel Processing. *IEEE Trans. 15.* October, 1966. 757-763.

[Bö 66] Böhm, C. The CUCH as a Formal and Descriptive Language. In [Ste66]. 179-197.

[BJ 66] Böhm, C., and Jacopini, G. Flow Diagrams, Turing Machines and Languages with only two Formation Rules. *Comm. ACM 9* (May, 1966), 366-371.

[BM 62] Brooker, R.A., and Morris, D. A General Translation Program for Phrase Structure Languages. *Journal ACM 9* (January, 1962), 1-10.

[CL 63] Cheatham, T.E., Jr., and Leonard, F. An Introduction to the CL-II Programming System. *Computer Associates Inc., Report CA-63-7-SD.* August, 1963. Also in [Ro 67]. 582-597.

[Ch 65] Cheatham, T.E., Jr. The TGS-II Translator Generator System. *Proc. of IFIP Congress.* 1965. 592-593.

[CS 69] Christensen, C., and Shaw, C.J. (Ed.). Proceedings of the Extensible Languages Symposium. *SIGPLAN Notices 4.* August, 1969.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Co 62] *An Introduction to COMIT Programming.* The M.I.T. Press: Cambridge, Mass. 1962. 60.

[Co63a] Conway, M.E. Design of a Separable Transition-Diagram Compiler. *Comm. ACM 6.* (July, 1963), 396-408.

[Co63b] Conway, M.E. A Multiprocessor System Design. *Proc. FJCC 24.* 1963. 139-146.

[DN 66] Dahl, O., and Nygaard, K. SIMULA - an ALGOL-Based Simulation Language. *Comm ACM 9* (September, 1966), 671-678.

[De 68] Dennis, J.B. Programming Generality, Parallelism, and Computer Architecture. *IFIP Congress 1968.* 1968. c1-c7.

[DV 66] Dennis, J.B., and Van Horn, E.C. Programming Semantics for Multiprogrammed Computations. *Comm. ACM 9* (March, 1966), 143-155.

[Di65a] Dijkstra, E.W. Programming Considered as a Human Activity. *Proc. IFIP Congress 1965, 1.* 1965. 213-217.

[Di65b] Dijkstra, E.W. Solution of a Problem in Concurrent Programming Control. *Comm. ACM 8* (September, 1965), 569.

[Di68a] Dijkstra, E.W. Go To Statement Considered Harmful. *Comm. ACM 11* (March, 1968), 147-148.

[Di68b] Dijkstra, E.W. The Structure of the "THE"- Multi-programming System. *Comm. ACM 11* (May, 1968). 341-346.

[Ea 68] Earley, J. An Efficient Context-Free Parsing Algorithm. *Doctoral Dissertation, Carnegie-Mellon University.* August, 1968.

[FGP64] Farber, D.J., Griswald, R.E., and Polonsky, I.P. SNOBOL, A Sting Manipulation Language. *Journal ACM 11* (January, 1964), 21-30.

[Fe 66] Feldman, J.A. A Formal Semantics for Computer Languages and its Application in a Compiler- compiler. *Comm. ACM 9* (January, 1966), 3-9.

[FG 68] Feldman, J., and Gries, D. Translator Writing Systems. *Comm. ACM 11* (February, 1968), 77-113.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Fi 68] Fikes, R.E. A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures. *Doctoral Dissertation, Carnegie-Mellon University.* November, 1968.

[Fi 67] Fisher, D.A. Program Analysis for Multiprocessing. *Burroughs Corp. Tech. report TR67-2.* May, 1967.

[Fl 63] Floyd, R.W. Syntactic Analysis and Operator Precedence. *Journal ACM 10* (July, 1963), 316-333.

[Fl67a] Floyd, R.W. Nondeterministic Algorithms. *Journal ACM 14* (October, 1967), 636-644.

[GP 67] Galler, B.A., and Perlis, A.J. A Proposal for Definitions in Algol. *Comm. ACM 10* (April, 1967), 204-219.

[Gi 58] Gill, S. Parallel Programming. *The Computer Journal 1* (April, 1958), 2-10.

[Go 61] Gorn, S. Specification Languages for Mechanical Languages and Their Processors. *Comm. ACM 4* (December, 1961), 14-17.

[Go 66] Gosden, J.A. Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers. *Proc. FJCC 29.* 1966. 651-660.

[Ha 67] Habermann, A.N. On the Harmonious Co-operation of Abstract Machines. *Doctoral Dissertation, Technische Hogeschool Eindhoven.* October, 1967.

[Ho 59] Holland, J.A. A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously. *Proc. EJCC 16.* 1959. 108-113.

[Il 66] Illiac-IV System Study Final Report. *Burroughs Corp., University of Illinois No. 09852-B.* December, 1966.

[Ir 63] Irons, E.T. An Error-correcting Parse Algorithm. *Comm. ACM 6* (November, 1963), 669-673.

[Ir 69] Irons, E.T. The Extension Facilities of IMP. In [CS 69]. 18-19.

[Iv 62] Iverson, K.E. *A Programming Language.* John Wiley and Sons, Inc.: New York. 1962.

[Jo 69] Jorrand, P. Some Aspects of BASEL. The Base Language for an Extensible Language Facility. In [CS 69]. 14-17.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Kn 66] Knuth, D.E. Additional Comments on a Problem in Concurrent Programming Control. *Comm. ACM 9* (May, 1966), 321-322.

[Kn 67] Knuth, D.E. The Remaining Trouble Spots in ALGOL 60. *Comm. ACM 10* (October, 1967), 611-618.

[Kn 68] Knuth, D.E. The Art of ComputerProgramming. *Fundamental Algorithms 1.* Addison-Wesley: New York. 1968. 190-196.

[KM64a] Knuth, D.E., and McNeley, J.L. SOL - A Symbolic Language for General Purpose Systems Simulation. *IEEE Trans. on EC.* August, 1964. 401-408.

[KM64b] Knuth, D.E., and McNeley, J.L. A Formal Definition of SOL. *IEEE Trans. on EC.* August, 1964. 409-414.

[La 66] Landin, P.J. The Next 700 Programming Languages. *Comm. ACM 9* (March, 1966), 157-166.

[Le 70] Leathrum, J.F. A Language Design System. *Burroughs Technical Report.* January, 1970.

[Le 66] Leavenworth, B.M. Syntax Macros and Extended Translation. *Comm. ACM 9* (November, 1966), 790-793.

[Le 68] Leavenworth, B.M. Programmer-Defined Control Structures.

[Le 69] Leavenworth, B.M. The Definition of Control Structures in McG360. *IBM Research Report no. RC2378.* February, 1969.

[LLS68] Lucas, P., Lauer, P., and Stigleitner, H. Method and Notation for the Formal Definition of Programming Languages. *IBM, Vienna Laboratory. Technical report TR25.087.* June, 1968.

[Ma 69] MacLaren, M.D. Macro Processing in EPS. In [CS 69]. 32-36.

[Mc 60] McCarthy, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Comm. ACM 3* (March, 1960), 184-195.

[Mc 64] McCarthy, J. Definitions of New Data Types in ALGOL X. *ALGOL Bulletin 18* (October, 1964).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[MPV68] Mitchell, J.G., Perlis, A.J., and VanZoeren, H.R. LC^2: A Language for Conversational Computing in Interactive Systems for Experimental Applied Mathematics. *Proceedings of the Association for Computer Machinery Inc.* Klerer, M., and Reinfelds, J. (Ed.). Academic Press: Washington, D.C. August, 1967. 203-214.

[Na 63] Naur, P. (Ed.). Revised Report on the Algorithmic Language ALGOL-60. *Comm. ACM 6* (January, 1963), 1-17.

[Ne 61] Newell, A. (Ed.). *Information Processing Language-V Manual.* Prentice-Hall Inc.: New York. 1961.

[NN 66] Nivat, M., and Nolin, N. Contributions to the Definition of Algol Semantics. In [Ste66]. 148-159.

[Op 65] Opler, A. Procedure Oriented Language Statements to Facilitate Parallel Processing. *Comm. ACM 8* (May, 1965), 306-307.

[Pa 67] Parnas , D.L. Sequential Equivalents of Parallel Processes. *Carnegie Institute of Technology.* February, 1967.

[Pe 67] Perlis, A.J. The Synthesis of Algorithmic Systems. *Proc ACM National Conference.* 1966. 1-6. And also *Journal ACM 14* (January, 1967), 1-9.

[Pe 68] Perlis, A.J. Lecture Notes on Seminar on Extensible Languages. *Carnegie-Mellon University.* Fall, 1968.

[PI 64] Perlis, A.J., and Iturriaga, R. An Extension to ALGOL for Manipulating Formulae. *Comm. ACM 7* (February, 1964),127-130.

[PIS66] Perlis, A.J., Iturriaga, R.,and Standish, T.A. A Definition of Formula Algol. *Internal Publication. Carnegie Inistitute of Technology.* 1966.

[Pe 66] Petri, C.A. Communication with Automata [a translation of "Kommunikation mit Automaten", (Bonn, 1962)]. *Supplement 1 to Technical Report no. RADC-TR-65-377 1.* January, 1966.

[PL 66] PL/1 - Language Specifications, IBM System/360 Operating System, Form C28-6571-3, *IBM Corp. Prog. Syst. Publications,* New York (July 1966).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[RF 65] Reilly, E.D., Jr., and Federighi. On Reversible Subroutines and Computers that Run Backwards. *Comm. ACM 8* (September, 1965), 557-558,578.

[Re 69] Reynolds, J.C. GEDANKEN: A Simple Typeless Language Which Permits Functional Data Structures and Coroutines. *Argonne National Laboratory report ANL-7621.* September, 1969.

[Ro 67] Rosen, S. (Ed.). *Programming Systems and Languages.* McGraw-Hill: New York. 1967.

[SBM62] Slotnick, D.L., Borck, W.C., and McReynolds, C. The SOLOMON Computer. *Proc. FJCC 22.* 1962. 97-107.

[St 67] Standish,T.A. A Data Definition Facility for Programming Languages. *Doctoral Dissertation, Carnegie Inistitute of Technology.* May 1967.

[St 68] Standish, T.A. A Preliminary Sketch of Polymorphic Programming Language. *Unpublished, Centro de Calculo Electronico, Universidad Nacional Autonoma de Mexico.* June,1968.

[St 69] Standish, T.A. Some Features of PPL, A Polymorphic Programming Language. In [CS 69]. 20-26.

[Ste66] Steel, T.B. (Ed.). *Formal Language Description Languages for Computer Programming.* North-Holland Publishing Co.: Amsterdam. 1966.

[Str65] Strachey, C. A General Purpose Macrogenerator. *The Computer Journal 8* (October, 1965), 225-241.

[Str66] Strachey, C. Towards a Formal Semantics. In [Ste66]. 198-220.

[TL 66] Teichroew, D., and Lubin, J.F. Computer Simulation - Discussion of the Technique and Comparison of Languages. *Comm. ACM 9* (October, 1966), 723-741.

[vN 56] von Neumann, J. *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components.* Yale University Press: New Haven, Conn. 1956.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[vN 58] von Neumann, J. *The Computer and the Brain.* Yale University Press: New Haven, Conn. 1958.

[We 68] Wegner, P. *Programming Languages, Information Structures and Machine Organization.* McGraw-Hill: New York. 1968.

[Wi 66] Wirth, N. A Note on "Program Structures for Parallel Processing." *Comm. ACM 9* (May, 1966), 320-321.

[Wi 69] Wirth, N. On Multiprogramming, Machine Coding, and Computer Organization. *Comm. ACM 12* (September, 1969), 489-498.

[Za65a] Zadeh, L.A. Fuzzy Sets and Systems. *Proc. Symposium on System Theory, Polytechnic Inistitute of Brooklyn.* April, 1965. 29-37.

[Za65b] Zadeh, L.A. Fuzzy Sets. *Information and Control 8* (June, 1965), 338-353.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.