MAC TR-87

A MODEL FOR PROCESS REPRESENTATION AND SYNTHESIS

Robert H. Thomas

June 1971

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

A MODEL FOR PROCESS REPRESENTATION AND SYNTHESIS

Robert H. Thomas

June 1971

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

ACKNOWLEDGEMENTS

I wish to thank my advisor, Professor Arthur Evans Jr., for his guidance and for his interest in me. He has contributed significantly to the successful completion of this dissertation. I am indebted to Dr. Daniel G. Bobrow of Bolt Beranek and Newman Inc. for his enthusiasm for and contributions to the research. I thank Professor Michael J. Fischer for his suggestions and criticisms which have influenced both the course of the work and its presentation.

For his encouragement and interest in the research I express my gratitude to Dr. William R. Sutherland of Bolt Beranek and Newman Inc.

Finally I thank my wife Elaine for being Elaine.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under the Office of Naval Research Contract Nonr-4102(01); and, in part by the National Aeronautics and Space Administration grant NGR-22-009-393.

A MODEL FOR PROCESS REPRESENTATION AND SYNTHESIS*

Abstract

This dissertation investigates the problem of representing groups of loosely connected processes and develops a model for process representation useful for synthesizing complex patterns of process behavior. There are three parts to the dissertation. The first part isolates the concepts which form the basis for the process representation model by focusing on questions such as: What is a process; What is an event; Should one process be able to restrict the capabilities of another? The second part develops a model for process representation which captures the concepts and intuitions developed in the first part. The model presented is able to describe both the internal structure of individual processes and the "interface" structure between interacting processes. Much of the model's descriptive power derives from its use of the notion of process state as a vehicle for relating the internal and external aspects of process behavior. The third part demonstrates by example that the model for process representation is a useful one for synthesizing process behavior patterns. In it the model is used to define a variety of interesting process behavior patterns. The dissertation closes by suggesting how the model could be used as a semantic base for a very potent language extension facility.

---

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Doctor of Philosophy, May 1971.

TABLE OF CONTENTS

CHAPTER 1

A Model For Process Synthesis

1.1 Introduction

The notion of sequential process has proven to be a useful conceptual device for dealing with a number of situations which arise in computing. The process notion has seen use as a tool for understanding and designing operating systems [Sa66] [Di68b], as a vehicle for investigating control aspects of programming languages [Fi70], and as a building block for simulation of discrete event systems [Da66].

It is usefully employed in situations that can be divided into two or more parts which operate independently except for occasional interaction. The observation that in such situations certain sequences of actions follow one another naturally and are, for the most part, independent of other such sequences is the intuitive basis for the process notion. Dijkstra [Di68a] uses the phrase "loosely connected" to describe such processes.

This dissertation investigates the problem of representing groups of loosely connected processes. The goal of the investigation is to develop a method for process representation useful for synthesizing complex patterns of

process behavior. It is important that the method be capable of describing both the internal structure of individual processes and the "interface" structure between interacting processes.

The major part of the dissertation is concerned with the development of a model which captures the essential aspects of loosely connected processes. The model provides a synthetic tool for describing situations that involve changing numbers of processes, interactions between processes, interruption of process activity and periods of process inactivity. It can support detailed specification of two aspects of process behavior: the internal aspects, having to do primarily with the independent activities of a process and, the external aspects, having to do primarily with interactions of a process with other processes. Process behavior patterns such as those exhibited by subroutines, coroutines, backup programming and various kinds of parallel processing can be formulated in an intuitive way in terms of the model.

The model derives much of its descriptive power from its treatment of the notion of process state. Each process is "aware" of its state to the extent that it can manipulate it in much the same way as it can any other data object.

The model for process representation and synthesis developed in this dissertation has a number of potential applications. The remainder of this section discusses some of

them.

The model provides a conceptual framework in terms of which ideas concerning the process notion can be concisely formulated. Used in this way it can bring into sharp focus ideas which might otherwise remain obscure and intuition bound. One can use the model as a gedanken device for experimenting with ideas and for communicating them to others.

Contemporary extensible languages, such as BASEL [Che68], GPL [Gar68] and EL1 [Weg70], have been relatively successful in providing for extension in the area of data types. Using such a language one can, without much difficulty, define and use new data types and operations. Such languages display an almost total absence of facilities for describing new patterns of flow of control. As a result, there is little that can be done with them to specify extensions in the area of control. For example, none include means of sufficient potency to describe control patterns such as those required to construct coroutines, simulation primitives or parallel processing. For this reason one is led to conclude that contemporary extensible languages are not very strongly extensible but are, in fact, only slightly perturbable. To realize a truly extensible language a semantic base capable of supporting descriptions of control patterns, in addition to specifications of data types, is required. The ability of the model to describe both the internal and external aspects of process activity that are necessary to synthesize a wide

variety of process behavior patterns makes it attractive for such an application.

These are interesting and practical situations which can most naturally be thought of as involving more than a single locus of control. For example, imagine a fully computerized solution to the air traffic control problem involving ground and airborne computers, all working together on a single distributed computation: the scheduling and controlling of aircraft maneuvers. The participants in the computation would continually change as aircraft enter (takeoff) and leave (land) the system. Because conventional programming languages largely ignore the possibility of concurrency, they are ill-suited for programming in situations which naturally involves multiple, interacting loci of control. Its concern with concurrency, interactions and dynamically changing numbers of processes makes the model an ideal candidate for the semantic base of a programming language capable of coping with concurrency.

1.2 Related Work

This section surveys previous work that is relevant to this dissertation. Although most of the work mentioned has to do with external aspects of process behavior, it is important to remember that the process representation method developed in the dissertation is for both internal and external aspects of process behavior. The external aspects receive more

attention here because they are less well understood.

For this survey it is convenient to separate the related work into four categories:

1. language definition work which seeks techniques for formally defining programming languages;
2. theoretical work which seeks a fundamental understanding of phenomena associated with processes, concurrency and interactions;
3. linguistic work which attempts to provide facilities for specifying parallelism in programs; and
4. operating system work which makes use of the process notion as a means for coping with complexity.

Work representative of each category is considered in turn.

1.2.1 Language Definition Work

The primary goal of research in the area of language definition is development of methods for formally defining programming languages. Motivations for this research include:

1. providing the language implementer with a complete and concise definition of the language he is implementing;
2. providing the language designer with a framework for design and comparison of languages;
3. providing the programmer with a reference he can consult whenever the usual language primers and manuals provide insufficiently clear answers to his questions; and

4. providing a basis for making proofs about properties of programs and correctness of implementations.

As Section 1.1 notes, most programming languages ignore the possibility of concurrency. Consequently language definition techniques, for the most part, have little to say about external aspects of process behavior.

A considerable amount of research has been reported in the area of language definition. Only the techniques most relevant to this dissertation are discussed. More complete surveys of the area are to be found in a paper by deBakker [deB69] and a book edited by Steele [Ste66].

The language definition techniques of interest here are those which use an interpreting machine to assign meaning to language features. Generally, the interpreting machines operate on abstract representations of programs rather than directly on the programs themselves (concrete representations). Because an abstract representation of a program can better reflect its semantic structure, the language definitions that result are less complex than they would be if a concrete representation were used. Typically the peripheral issue of parsing is side-stepped by defining a mapping from abstract to concrete representation and by assuming that the inverse mapping exists and is well defined. Most techniques of this type are based on work by Landin and McCarthy.

As part of an effort directed toward developing a language definition method based on the λ-calculus of Church [Chu51], Landin [Lan64], [Lan65] developed a mechanism for evaluating expressions. His expression interpreter, called the SECD interpreter, evaluates "applicative expressions", expressions constructed from "known" constants by functional application and function abstraction. It consists of a stack (S), whose items are intermediate results awaiting subsequent use, an environment (E), made up of name-value pairs, a control (C), which is a representation of the expression being evaluated, and a dump (D), used in evaluating functional applications. Wozencraft and Evans [Wo70] have shown how to extend the SECD mechanism to interpret programs containing imperative features such as assignment and transfer of control.

McCarthy [Mc66] uses the notion of state as the basis for a language definition technique. The semantics of a language L are defined in terms of a state vector function F_L which specifies how programs in L transform a state vector ξ. The meaning of a program P can be deduced by evaluating $F_L(Pa,\xi)$ which is the state vector that results from applying Pa, the abstract representation of P, to ξ. F_L acts as an interpreter for the program Pa. Embedded in it are the rules necessary for interpreting L programs. Components of the state vector include "data" which provides an environment in which to carry out interpretation of L programs. For example, a definition

of Landin's applicative expression language would consist of a function F_{AE} describing the operation of the SECD interpreter. The state vector ξ would include S, E and D; Pa would correspond to C.

The result of perhaps the most extensive effort in the area of language definition is the method developed by the IBM Vienna Laboratory [Lu68a]. The so-called Vienna method was devised as part of an effort directed toward preparing a completely formal definition for PL/I. It provides a metalanguage and a basic abstract machine for constructing language definitions. The metalanguage includes the propositional calculus, conditional expressions, function composition, and operators for manipulating structured objects. A set of states $\{\xi_i\}$ and a state transition function Λ define the basic machine. When applied to a state the transition function produces a set of possible successor states from which one is chosen as the actual successor. A sequence of states, $\xi_0, \xi_1, \ldots, \xi_n$, where ξ_0 = an initial state and $\xi_i \in \Lambda(\xi_{i-1})$, defines a computation. The nondeterministic nature of the basic machine allows sets of actions to be performed in an unspecified order. This makes it possible to make language definitions which, where appropriate, leave open certain aspects of a language, such as the order in which the operands of binary operators are evaluated. It can also be used to simulate concurrency.

The definition of a particular language by the Vienna method includes detailed specification of the state structure and definition, in the form of a set of instructions, of how the basic machine transforms states. To define PL/I the metalanguage is used to describe an abstract representation for PL/I programs, the translation from abstract representation to concrete representation, the translation from abstract representation to initial machine states and a set of instructions for the basic machine. Each syntactically correct concrete program defines an initial machine state, part of which includes instructions for the basic machine. The behavior of the basic machine when "started" from an initial state provides the meaning for the corresponding PL/I program.

The Vienna method has been used to define other languages including ALGOL 60 [Lau68], APL [Ger70] and BASIC [Lee69]. In addition, it has been exploited to investigate certain implementation issues concerning PL/I. Lucas has used it to show the equivalence of two interpretations of the PL/I block concept [Lu68b] and to uncover a subtle bug in a PL/I compiler [Lu71].

The work in language definition has influenced the development of the method for process representation in two respects. The conceptual basis for the notions of process state and process state transition rule, as they appear in the model, is to be found in McCarthy's work. And, parts of the

model which deal with internal aspects of process behavior are similar in some respects to Landin's SECD mechanism.

1.2.2 Theoretical Work

The desire to develop a design methodology for hardware constructed from asynchronously operating components has been motivation for much theoretical work. Typically, the approach taken is to devise a model that can be used to investigate, at a fundamental level, situations involving interactions between asynchronous operations. The models generally consist of a collection of computing and memory elements. The computing elements operate according to rules which, depending upon the particular model, permit various degrees of concurrency. The memory elements are read and written by the computing elements for input and output. The investigations proceed by placing constraints on the models in attempting to answer questions concerning desirable behavioral properties. One such property is determinacy. A collection of computing elements is said to be determinate if the results of computations it performs are independent of the relative speeds of asynchronous operations. Typically, constraints which disallow more than a single computing element to change the same memory element at any time are placed on the models. The results of such investigations are usually conditions which are sufficient to insure a particular property or under which certain equivalence questions are decidable.

Van Horn [VH66] uses a model called "machines for coordinated multiprocessing" to investigate constraints which insure that the sequence of values in each memory element is a unique function of the initial state of the model. Systems which display such behavior are said to exhibit complete functionality. Constraints which insure output functionality, behavior in which only a subset of the memory elements need contain unique sequences of values, are the subject of Luconi's work [Luc68]. Slutz [Slu68] and Karp and Miller [Ka68] have developed models for representing and studying algorithms containing parallelism; the emphasis of their work is on decision procedures for properties such as equivalence and determinacy.

In his dissertation Haberman [Ha67] studies situations involving collections of cooperating abstract machines in which one machine performing a task can generate tasks for others. In his model the input for each machine is the output tape of another; a machine may proceed no further than the availability of its input permits. Haberman looks for conditions which insure that all machines in such a system eventually return to their "homing positions".

For the models cited above, the computing elements have no internal structure and can be treated as "black boxes" in analysis. Furthermore, the computations performed by the individual computing elements are left unspecified. That is, no interpretation is assigned to them. The only assumption

made is that each computes a fixed but unspecified function. Such models are said to be uninterpreted. Some work, such as that of Slutz and of Karp and Miller, uses such uninterpreted models to avoid the undecidability results associated with interpreted algorithms. In Haberman's work the internal activities of the individual abstract machines are ignored because they are considered to be irrelevant to the question of harmonious cooperation. In any event, the tasks and the manner in which they are accomplished are not representable in terms of these models. Hence, while they are well suited for investigating conditions which insure certain general behavioral properties, these models are inadequate for synthesizing specific behavioral patterns.

1.2.3 Linguistic Work

The linguistic work has been concerned primarily with proposals for language features which allow a programmer to indicate that sections of program are to be executed in parallel. These features are generally suggested as extensions to existing languages such as FORTRAN or ALGOL. Because most of the proposals have not been included in complete language designs they are difficult to evaluate.

Conway [Con63], Anderson [An65] and Opler [Op65] propose and illustrate the use of statements for initiating and terminating parallel execution paths. The statements "fork", "join" and "terminate" are representative of their proposals.

Separation of a single locus of control into two (or more, depending upon the proposal) loci can be specified by "fork"; "join" specifies the merger of several loci into a single one. A parallel path can be terminated by the "terminate" statement.

Wirth [Wi66] proposes use of "and" to specify the possibility of parallel execution; the decision as to whether or not execution is, in fact, parallel would be left to the implementer. Thus the phrase

S1 and S2 and S3

indicates that S1, S2 and S3 can be executed either in parallel, or sequentially in any order, or even in some fashion in which parts of the execution of each are interleaved. Dijkstra [Di68a] proposes "parbegin" and "parend" to bracket statements that are to be executed in parallel. The entire construction between the brackets is regarded as a single compound statement whose execution is completed when the execution of all its constituents is completed. Thus

```
begin
        S1;
        parbegin  S2; S3; S4  parend;
        S5
end
```

indicates that after completion of S1, the statements S2, S3, and S4 are to be executed in parallel and only after all of them are finished is S5 to be executed.

Some, but not all, of the proposals discussed above include mechanisms for enabling a single path to obtain and release the sole use of variables accessible to several paths.

The tasking facility of PL/I [Be70], [IBM69] provides means to specify two or more concurrent program executions (tasks). PL/I allows tasks to be created under program control and provides facilities for synchronizing tasks, terminating tasks and testing for task completion. An interrupt handling mechanism is provided by PL/I's ON condition facility which allows a programmer to specify actions taken when certain interrupt conditions hold. Both tasking and ON conditions in PL/I display anomalies. For example, the lifetime of a task may not exceed that of the block which initiated it. Such anomalies are probably, in part, the result of the language having been designed with a particular implementation and operating environment in mind.

The features discussed above represent first attempts at introducing concurrency into programming languages. Using them one can, indeed, write programs which exhibit multiple loci of control. However, the kinds of interactions between such loci that can be described conveniently are rather limited. This is due, in part, to limited goals and, in part, to the fact that the features have been added as afterthoughts to already existing language designs. The realization of languages well suited for describing multiple, interacting loci of control requires that their design be based on a

fundamental understanding of processes and interactions between them.

The language SIMULA [Da66] is representative of a number of simulation languages which make use of the process notion. It uses processes as the basis for decomposing discrete event systems into separately describable components. The actions and interactions of collections of processes are taken to represent the behavior of such systems. Although SIMULA processes can be thought of as evolving concurrently, SIMULA provides a basically quasi-parallel environment in which the programmer explicitly schedules the running of processes in order to simulate the behavior of a particular system. Such a quasi-parallel environment is useful for certain simulation applications. However, because the programmer must concern himself with issues such as scheduling in order to achieve the effect of concurrency, SIMULA only weakly supports the notion of concurrently evolving, interacting processes. In all fairness, it is necessary to add that the ability to synthesize groups of loosely connected processes is not a stated design goal for SIMULA and languages of which it is representative.

1.2.4 Operating System Work

The process notion is frequently used as a tool in the design and implementation of operating systems. It provides a basis for discussion of the behavior of complex computer

systems. Saltzer [Sa66] pioneered its use in discussing the design of an operating system.

Dijkstra [Di68b] describes an operating system which is arranged as a society of sequential processes. In that system a process corresponds to each user program and to each peripheral device. Dijkstra claims that the use of the process notion combined with a hierarchical structure made proof of the correctness of the operating system by "discrete reasoning" possible.

The MULTICS [Cor65] operating system associates a process with each logged-in user. In addition, it associates processes with certain system provided services. User processes can make requests of such processes. For example, to obtain listings a user requests service from the process corresponding to the line printer. A system provided backup service is performed by a "daemon" process that periodically writes user files onto tapes.

Many operating systems incorporate the process notion in one guise or another. However, in most the properties of processes are obscured by implementation details which are peculiar to the particular system. In addition, few operating systems make the process structure available to the user of the system. A notable exception in this respect is the TENEX [BBN70] operating system developed for the PDP-10 which permits a user to create and destroy processes as he sees fit.

1.2.5 Other Work

Before concluding this discussion it is important to note two other efforts, neither of which fit neatly into the above categories.

A proposal by Leavenworth [Lea69] has influenced the treatment of the notion of state in the process representation method developed in this dissertion. Leavenworth considers a programming language which includes as data objects states of its own interpreter. The language he considers, McG [Bur68], is similar to that proposed by Landin as ISWIM [Lan66] and implemented as PAL [Ev68]; its interpreter is similar to the SECD interpreter. Although the specific mechanism proposed is rather clumsy, it is possible for a programmer to define functions which transform the current state of the interpreter. In addition, it is possible to create, from scratch, data objects which can subsequently be used as interpreter states. Leavenworth's paper includes several examples which illustrate how this mechanism can be used to define sophisticated control structures.

In a thesis which investigates control structures of programming languages and proposes several new and interesting ones, Fisher [Fi70] attempts to isolate primitive control operations from which more complex control structures can be built. The explanation of his primitive operations makes heavy use of the process notion. Neither a definition nor a

discussion of the term "process" is included in his work. As a result, the meanings of the more interesting control primitives, those concerned with monitoring, synchronization, and relative continuity are only as precise as the reader's intuitive notion of process.

Fisher describes a control definition language which includes his primitives. That language can support descriptions of a wide variety of behavioral patterns. There is a significant difference between the approach Fisher takes and the one taken here. Fisher's approach is an axiomatic one. The question his work addresses is: Independent of implementational considerations, what is an intuitively appealing set of primitive operations for describing control? The approach taken here is mechanistic and addresses the question: How can processes be represented in order to facilitate synthesis of complex process behavior patterns? The answer comes in the form of a model which can be used as a base for synthesis of a wide variety of control patterns. In Section 6.6 of this dissertation Fisher's primitives are discussed further and defined in terms of the model.

1.3 Plan For the Dissertation

Conceptually, this dissertation is divided into three phases. The first phase isolates the concepts which are to form the basis for the process representation technique. The second phase develops a model for process representation and

synthesis which captures those concepts. The third phase demonstrates, by example, that the model developed is useful for describing process behavior patterns.

Chapter 2 (part of the first phase) examines the process notion and characterizes the class of processes of interest. It considers constraints placed on the process representation method by the requirement that it be able to describe both internal and external aspects of process behavior. The model for process representation and synthesis is defined in Chapters 3 and 4 (parts of the second phase). Chapter 3 considers the model in overview. In Chapter 4 it is presented in more detail. Chapter 4 notes a weakness in the model, to be corrected in Chapter 7, concerning its ability to describe situations involving processes with different capabilities. Chapters 5 and 6 (parts of the third phase) are concerned with using the model to describe process behavior patterns. A simple programming notation for the model is defined in the first part of Chapter 5. The remainder of Chapter 5 and all of Chapter 6 illustrate use of the model to define sophisticated patterns of process behavior. In Chapter 7 (part of all three phases) the problem of controlling the capabilities of processes is considered and extensions are made to the model (as described in Chapters 3 and 4) which make it possible to do so. Chapter 8 completes the dissertation by suggesting areas for extending the work presented in it.

CHAPTER 2

Motivation For The Model

2.1 Introduction

This chapter discusses considerations which form the intuitive basis for the model for process representation. It begins by examining the process notion and by characterizing the class of processes of interest. The chapter continues by considering requirements placed on the model by the ability to describe groups of loosely connected processes. External aspects of process behavior such as those related to interactions between processes, interruption of process activity and changing numbers of processes are considered. Discussion turns next to internal aspects of process behavior, such as the binding of identifiers and the universe of discourse for processes, and then to a preview of the treatment the notion of process state receives in the model. The chapter concludes by presenting a list of specific questions the model addresses.

2.2 The Process Notion

An important difference between computation and traditional mathematics is captured by the notion of process. The realm of traditional mathematics is infinite and timeless.

It deals with collections of values which are often infinite. Operations are defined as mappings from one set of values to another. There is little more to be said about such operations beyond demonstrating that they have certain properties. On the other hand, the realm of computation can be characterized as finite and dynamic. Its concern is with finite collections of value representations. Operations are performed for the effect they have. They generate new value representations from existing ones. The process notion captures the idea of continual change.

As our starting point we take an intuitive definition for the term process.

> A process is an activity comprised of a time-ordered sequence of actions. A process is an abstract entity and therefore can not be directly observed. The evidence for the existence of a process is change.

Making this definition more precise and thereby achieving a deeper understanding of the process notion is a primary goal for the remainder of this chapter. The model for process representation described in Chapters 3, 4 and 7 provides a very precise definition for the term process which captures the concepts and intuitions developed in this chapter.

It is useful to think of there being both a passive member and an active agent associated with a process. As the process evolves the condition of the passive member changes in response to the actions of the active agent.

The condition of the passive member is described by the process state. At any given time the state of a process, together with the "rules" used by the active agent to change it, represents all there is to be known about the process. The state describes all that is accessible to the active agent. An important idea has been introduced:

> The extent of the changes resulting from the activity of a process is isolated; the effect the actions of a process can have is limited to that which is accessible from its state.

The active agent is called the processor. Although the process representation method to be presented is capable of describing the time multiplexing of processors, this dissertation is not concerned with such issues. Consequently each process is assumed to have its own processor. The periods of process inactivity the model is capable of describing do not result from the non-availability of physical processors but rather occur whenever a process temporarily runs out of things to do and awaits an occurrence that will enable it to continue.

The class of processes under consideration is limited to those for which the actions taken by a processor depend only upon the process state. For such processes it makes sense to talk of a state transition rule and to view the processor as that which changes the process state according to the state transition rule. Furthermore, in structuring the process

state it is useful to separate out part of it, a program component, which is interpreted by the processor. The program component is a specification of the future behavior of the process. The specific actions taken when the program component is interpreted may, in general, depend upon information (data) found in other parts of the process state.

The possibility of (at least part of) the state of a process being changed by other than the processor associated with the process has not been excluded. Indeed, interactions between processes require that one process be able to change information accessible to another. That is, to interact, it is necessary that one process be able to change (at least part of) the state of another.

It is appropriate at this point to summarize:

1. A process is the activity of a processor interpreting and changing a state in accordance with a state transition rule.
2. The effect the actions of a process can have is limited to that which is accessible from its state.
3. The state of a process includes a specification of its future behavior. The process evolves as its processor interprets that specification in the environment provided by the remaining parts of its state.
4. Part of a process state may change in response to the actions of other processes.

It is important to remember that a process is neither the

processor nor the process state but rather the activity of the processor as it changes the process state. This view of process is consistent with that proposed by Dennis and Van Horn [Dns66] of an "abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor".

2.3 The Role of Memory

The term "memory" frequently brings to mind physical storage devices such as core, machine registers, disc units, etc. In this dissertation the term memory is used in a more abstract sense to mean a collection of passive elements which are capable of holding information.

The passive member or state associated with a process "resides" in memory. As the process evolves, its state and, therefore, the condition of memory change. The state of a process defines the memory elements accessible to the process. It is those parts of memory which are interpreted and changed as the process carries on its activity.

An important notion has been introduced:

Processes have memory requirements.

This view of the relation of processes and memory is consistent with the practice of identifying a process with an address space [Sa66] [Lam68].

As implied above, the model does not deal directly with physical memory devices. Rather, it deals with an abstract or virtual memory. Virtual memory can have properties significantly different from those exhibited by physical memory devices. For example, a virtual memory may provide the illusion that memory is much larger than the available core memory, that access to memory is limited, or that memory is organized in a particular way. Examples of virtual memory include: the MULTICS segmented memory [Cor65] for which addresses are two-dimensional and access to a memory element may be constrained to be read only, write only or instruction fetch only; PAL memory [Ev68] for which memory elements are stretchable and can contain arbitrarily large amounts of information; and LISP memory [Mc62] for which memory elements have two parts: car and cdr. Of course, an implementation of a virtual memory must transform virtual memory operations into physical memory operations. This dissertation is not concerned with the problems of implementing virtual memories. The reader interested in such issues is refered to the tutorial paper by Denning [Dng70] on that subject.

Virtual memory can be thought of as a parameter for model description. By varying it models with very different properties can be obtained. It is appropriate to note here three properties of physical memory which are preserved in the model virtual memory:

1. the possibility of running out of memory exists

(finiteness);

2. when a memory element is simultaneously presented with two or more storage or retrieval requests it handles them sequentially rather than concurrently (arbiting ability); and
3. a distinction is made between memory elements, names of memory elements and "contents" of memory elements.

Details of the particular virtual memory incorporated into the model are described in Section 4.2.

2.4 Controlled Interactions

The possibility of interactions between processes suggests that process behavior has two aspects. It has external aspects, concerned primarily with interactions with other processes. And, it has internal aspects, concerned primarily with activity which is independent of other processes. The structure of process states in the model reflects this distinction. Certain state components of the state are used mostly in connection with internal actions while others are used primarily in connection with external actions. Although this distinction proves to be useful, it is not a rigid one because external actions are frequently undertaken to achieve the internal goals of a process.

Interactions between processes can occur only through memory that is shared by the processes. It is useful to view systematic interaction as occuring by way of the exchange of

messages in shared memory.

Before two previously non-interacting processes can begin to interact certain prerequisites must be satisfied. Spier and Organick [Sp69] have noted that these prerequisites must be satisfied externally to the two processes. The prerequisites are:

1. each process must be aware of the other's existence;
2. the processes must have access to common memory; and
3. conventions must be established in order that the processes can detect the occurrence of interactions and interpret their meaning.

These prerequisites place requirements on the model. Since more than two processes may exist simultaneously a means for identifying processes is needed. That is, the model must include in some form the notion of process identifier. There must be a way to achieve shared memory between two processes. This could be accomplished by organizing the process state such that part of it is accessible by way of process identifier to all (authorized) processes or by providing operations which make it possible to achieve overlap in the memory that is accessible from separate process states.

Process which interact are no longer isolated from one another, for to interact each must sacrifice some of its independence. The means for cooperative interaction is also potentially the source of destructive interference. It is

important to provide along with the means for achieving interactions the means for controlling them.

The key to controlled interactions is the notion introduced in Section 2.2 that there are limitations to the effect the actions of a process can have. In particular, that there are limitations to the effect one process can have on another. The key is to relax process isolation sufficiently to permit interaction but not enough to allow interference. Virtual memory plays an important role in achieving this kind of process isolation. An obvious minimal requirement for the virtual memory is that it deny free access to arbitrary memory elements. (This point is discussed further in Section 4.2, Section 4.7 and Chapter 7.)

The distinction proposed earlier between internal and external actions can be refined. Within a process, internal actions are those whose effect can be guaranteed to be limited to that process' state. External actions are those which, because they effect other process states, require close monitoring to insure that the process stays within its limitations.

2.5 Events

The occurrence of something of interest to a particular process is an event. The discussion of events in this section is largely from the point of view of the interested process.

An event can occur only as the result of the action of some process. Frequently, but not always, an event is the result of the action of a process other than the interested process and, therefore, represents part of an interaction between processes. The occurrence of an error situation and the arrival of a message from another process are examples of typical events.

Although events are often anticipated, they typically occur at unpredictable times. Consequently, it is necessary for a process to monitor for the occurrence of events of interest. The monitoring may occur either explicitly or implicitly. Explicit monitoring occurs at the direction of the process and appears as actions encoded in the program component of its state. Implicit monitoring occurs automatically as part of the activity of the process. Events detected by implicit monitoring are usually called interrupts. (Most hardware processors automatically monitor for the occurrence of certain events, such as the appearance of a voltage level on a particular bus.) While at some level monitoring is necessary, it makes a significant difference in synthesizing interacting processes whether monitoring can be assumed to occur automatically or whether processes must explicitly check for event occurrence.

Implicit monitoring for certain kinds of events is incorporated into the model. In addition, it is possible to explicitly monitor for others.

Dealing with events detected by explicit monitoring presents little that is new. A process merely checks for and responds to the occurrence of events whenever it sees fit to do so.

On the other hand, the possibility of interrupt events places interesting requirements on the model. Because a process has little control over when an interrupt event will be detected it must be prepared to respond to the occurrence of an interrupt event at all times. Its response to an interrupt event includes suspension of its current activity, with the possibility of later resuming it, in order to perform an action appropriate to the particular event. The ability to respond in this manner requires means for remembering the interupted activity and, in addition, a "programmed" response for each anticipated interrupt event.

Because interrupt events are caused by processes, the possibility of one process interrupting another to the point of interference exists. If interrupt events are to be allowed, it is important that a process have the means to control whether its current activity is interrupted whenever a particular interrupt event occurs. Ideally, if continuing uninterrupted is more important to a process than responding to the occurrence of a particular interrupt event, it should be able to postpone its response. Such an ability requires that a process have both the means to specify the importance of continuing its current activity relative to that of

responding to various interrupt events and the means to remember that an interrupt event has occurred until it chooses to respond to it.

When a number of interrupt events occur in quick succession the "current activity" may well be the response to an earlier interrupt. Consequently, a method for specifying the relative urgency of particular interrupt events is desirable. Furthermore, it is frequently the case in such situations that the order in which the events occur is important; hence it is desirable that not only the occurrence of interrupt events but also their order of occurrence be remembered. There is, of course, a limit to the number of interrupt events that can be remembered at any time.

2.6 Changing Numbers of Processes

At the request of existing processes new ones can be created. The creating process must, of course, specify an initial state for the new process. As far as the creating process is concerned, process creation is an atomic act. However, from the previous discussion it is clear that process creation includes:

1. fulfilling the memory requirements of the new process;
2. insuring the isolation of the new process from existing processes;
3. associating a processor with the new process; and
4. associating a process identifier with the new process.

The first two actions involve the virtual memory-physical memory interface and, the third, the multiplexing of physical processors.

Because a process requires resources, whenever it completes its activity or whenever its actions are deemed no longer necessary it should cease to exist. How should process destruction be initiated? There are at least three possibilities:

1. suicide: a process ceases to exist as a result of its own actions;
2. murder: a process is forced out of existence by another; and
3. orphanage: a process ceases to exist because its creator has been destroyed.

All but suicide represent potential sources of interference between processes. The model provides suicide as the only means of initiating process destruction. The effect of murder can be achieved when one process commits suicide at the request of another. Destruction by orphanage can be achieved in a similar way.

The possibility of process creation and destruction gives rise to interesting questions concerning the amount of control (if any) one process can exert over another and the capabilities of processes with respect to one another. For example, it seems reasonable that a process should be able to exert some degree of control over those it creates. Questions

such as these are considered further in Section 2.9.1, Section 4.7 and Chapter 7.

2.7 Internal Aspects of Process Behavior

The discussion so far has focused on the requirement that the method for process representation support descriptions of interactions between processes. It is equally important that it be able to describe how individual processes structure their sequences of actions to achieve their internal goals.

The way a process actually evolves is an important part of its behavior. As Section 2.2 notes, the actions a process performs are encoded in the program component of its state. The processor traces a path through the program focusing first on one encoded operation and then on another. The information needed to control the evolution of that path must be part of the process state. After completing a state transition, in order to initiate the next one, the processor needs to know

1. what the next operation is; and
2. whether or not it is to be performed now (recall that interruption of process activity is possible, as are periods of process inactivity).

The process state must include this information since it includes all there is to be known about a process.

As a process evolves, it produces results or values which are of only short term or temporary interest. Once used they

can be discarded. However, until they are used such intermediate results must be held somewhere. What constitutes an intermediate result depends upon the sequence of actions which are of interest. For example, when the sequence of actions under consideration are those comprising the execution of a single hardware machine instruction, the contents of storage buffer and storage address registers can be considered intermediate results. On the other hand, if those comprising the execution of a block of instructions are of interest the contents of index registers and accumulators represent intermediate results. The point to be noted is that processes require a mechanism for retaining intermediate results. Pushdown stacks provide a very simple mechanism for temporary storage. As intermediate results are produced they can be pushed onto a stack and as they are used, popped from it.

The specific operations processes can perform represent an important aspect of their behavior. From previous sections it is clear that there are operations to cause interrupt events, to create other processes, to initiate process destruction and to interact with virtual memory. Another class of operations, of particular interest when considering internal behavior, has to do with the kinds of objects processes can directly deal with. That class of operations defines a universe of discourse (Ω) for processes. Integers, for example, are considered to be part of the universe of discourse if arithmetic operations are included in the

operation repetoire but are not if arithmetic operations are absent, even if they can be implemented with sequences of other operations. The universe of discourse for the model is described in detail in Chapter 4. Like virtual memory, it can be thought of as a parameter for model description. Models of quite different properties result from varying it.

To be a useful device for synthesis the model should include features which make it easy to build complex behavior patterns from basic operations. That is, the model should include features making it attractive to program. Two such features, used to great advantage in programming languages, are included in the model. One is the ability to use identifiers of one's own choosing to denote particular values and the other is a data structure facility.

The relation between an identifier and the object it denotes is called a binding. When such a relation exists between an identifier and a value, the identifier is said to be bound to its (the) value. Conventional hardware is incapable of coping with identifier-value bindings. As a result it can not directly mechanize programs involving them. Two approaches are commonly used to handle identifier-value bindings. One approach is to bind identifiers to memory locations by systematically replacing references to identifiers by references to memory locations. The programming languages FORTRAN, ALGOL, and PL/I are usually implemented by "compiling" identifiers in this way. The other

approach is to implement a virtual machine or interpreter capable of dealing with identifier-value bindings by augmenting the hardware with software. The dynamic linking capability provided by the MULTICS operating system requires such a virtual machine. The LISP and PAL interpreters are other examples of virtual machines which can handle identifier-value bindings.

Because issues concerned with the compilation of identifiers are not of interest in this dissertation, the model incorporates the second approach. That is, a process is capable of interpreting identifiers appearing in the program component of its state. This capability requires that the state include a record of identifier-value bindings. As a process evolves its identifier-value bindings may change. Consequently, operations for creating and deleting bindings are required.

The data structure facility of the model provides construction operations for building structures from collections of parts and selection operations for accessing components of structures. Usually, a distinction is made between structures on the basis of how components are selected. There are those structures, usually called vectors, tuples, or arrays, whose components are selected by integer or subscript. And, there are those, usually refered to simply as structures, whose parts are selected by name.

Structure operations supported by programming languages are usually implemented by mapping them into physical memory operations by a compilation process. In a sense, compilers for such languages implement virtual memories. To avoid getting involved with questions concerned with how mappings from structure operations to physical memory operations are accomplished, structures are included in the universe of discourse of the model.

2.8 The Process State as a Data Object

Many interesting behavioral patterns exhibited by processes are conveniently described directly in terms of state transformations. Consider, for example, how subroutine behavior exhibited by a single process could be explained in terms of operations which extract, manipulate and set its state. The subroutine pattern consists of a call, a save, and a return. When a process performs the call and save it transforms its state to prepare for the subroutine execution. It sets its program component to correspond to the subroutine "code" and the other parts of its state to include the bindings for the formal parameters of the subroutine and the values certain components of its state had at the point of call. It accomplishes the return by restoring the values of those components of its state which it saved at the point of call.

The process state is a natural vehicle for describing the interrelation of the internal and external aspects of process behavior. For example, the behavior of a process subsequent to an externally caused interrupt event depends, in part, upon its internal activity. The occurrence of the event and the reaction of the process to it can be described conveniently and intuitively in terms of the state of the process, just prior to the event, and subsequent changes to it.

The process representation method includes means to describe behavioral patterns directly in terms of state transformations. The model allows processes to directly modify their own state. The values of components of the current process state can be "extracted" and operated upon as ordinary data objects. Furthermore, a process can perform operations on data objects which it can subsequently use as components for its own state. It can, for example, construct a structure which represents identifier-value bindings and by an appropriate operation specify that the structure be used as the identifier-value bindings component for its state. The subroutine pattern, as described above, involves this kind of state transformation.

2.9 Other Issues

2.9.1 On the Independence of Processes

The issues considered in previous sections, concerning initiation of process destruction, isolation of processes, and

preventing interruption of the current activity of a process, all relate to process independence. How much of it and in what ways should it be relinquished to permit useful interactions between processes? The question follows: Can one process be forced by another to do something? That is, within what bounds can one process control another?

In the model a process is, in principle, independent to the extent that after its creation it determines its own destiny. This is consistent with the goal that the model describe groups of processes which operate independently except for occassional interactions between them. Hence, one process can not force another to do something; it may merely call the other's attention to its wishes.

The situation is not nearly so anarchic as it might seem. When a process creates another it specifies an initial state for the new process. The creating process has, in effect, control over the behavior of the new process. It sets the program component for the new process and specifies how it is to respond to interrupt events. Therefore, although one process can not directly force another process to do something, processes can be created such that they respond obediently to the wishes of other processes. Furthermore, although the model insures that processes are normally isolated from one another, it includes mechanisms enabling a process to relinquish a large amount of its isolation by granting another access to what normally only it could access.

2.9.2 The World External to the Model

It is not intended for groups of processes synthesized by the model to exist completely independently of the outside world. For the model to be capable of describing situations of practical interest it must be able to describe interactions with the outside world so that processes can obtain input and report output.

As far as any process is concerned there is little to distinguish its interactions with the outside world from those with other processes, nor is there much to distinguish interrupt events caused by the outside world from those caused by other processes. Consequently, the outside world can be adequately represented by a process or group of processes.

In order for a process to interact with another it is unnecessary for it to understand the other's behavior in detail. Therefore, there is no need to specify the detailed behavior of processes representing the outside world. Whenever it is necessary to discuss the interactions of processes with the outside world, the approach taken in the sequel is to make no more assumptions concerning the behavior of the process (or processes) representing the outside world than necessary to permit processes to interact with it (them).

2.9.3 On the Traditional Problems Arising From Concurrency

The ability of a group of processes to evolve concurrently is a manifestation of their relative independence. Because interaction between processes can occur, their relative independence is not total.

It is frequently important to coordinate the behavior of such processes. For example, processes are typically coordinated in order to avoid, or at least to resolve, race conditions in which two processes attempt to change the same data item at the same time or in order to prevent one process from accessing data which temporarily is in an inconsistent state because another is modifying it. Achieving such behavior is variously called coordination, synchronization, or mutual exclusion.

The arbiting ability of memory plays an important role in achieving coordinated behavior. The model copes with coordination at two levels:

1. Coordination is involved as part of some of the built-in operations a process can perform. Such operations and the coordination required are discussed in Chapters 3 and 4.
2. The model can be used to define groups of processes which exhibit coordinated behavior. That is, it includes means to synthesize coordinated behavior.

Typically, coordination techniques require processes to adhere to certain conventions regarding the use of some type of locking mechanism. The semaphore and P and V operations introduced by Dijkstra [Di68a] provide an elegant mechanism for achieving coordination. Section 6.4 illustrates how the model can be used to describe processes coordinated by semaphores.

The locking mechanism which makes coordinated behavior possible unfortunately also makes the following kind of situation possible:

> Process A has item 1 locked and will not unlock it until it uses item 2 which is currently locked by Process B. Similarly Process B will not unlock item 2 until it uses item 1.
>
> Neither process can proceed.

Such deadlock or deadly embrace situations are almost always undesirable and are usually catastrophic. Haberman [Ha69] and Dijkstra [Di68a] have investigated strategies for preventing deadly embrace. The attitude taken in this dissertation is that a model with sufficient constraints to insure that deadly embrace situations can not arise would be too strongly constrained to be of practical interest. This does not imply that deadly embrace can not be avoided. Indeed, programmers willing to discipline themselves appropriately can successfully avoid describing such situations. In this regard deadly embrace is somewhat analogous to looping. A

programming language sufficiently constrained to prevent specification of programs with endless loops is of little practical interest. However this does not prevent programmers from writing terminating programs in practical languages.

2.9.4 How the Notion of an Executive Fits In

The notion of an executive is relevant when discussing how the model might be implemented on a conventional digital computer.

In an implementation a single "atomic" process action might, in reality, be the result of a number of executive actions. The executive actions would, of course, be invisible to the process. That is, processes would be unaware of the existence of an executive.

Among the responsibilities of the executive would be:

1. Associating with each process a processor.
   This involves multiplexing hardware processors.
2. Implementing the state transition rule.
   This includes implementation of operations which are not built into the hardware processor and, in addition, "monitoring" for the occurrence of interrupt events.
3. Providing the illusion of a virtual memory by managing the allocation of physical memory and transforming virtual memory operations into physical memory

operations.

4. Insuring that processes are isolated from one another. This includes "monitoring" the external operations undertaken by a process.

The executive would deal directly with hardware and operating system software (if any) to accomplish these things.

2.10 Toward a Particular Model

The issues raised in this chapter can be reformulated as a list of specific questions which the model addresses. The questions are conveniently divided into two groups: those related primarily to internal aspects of process behavior and those related primarily to external aspects.

The specific questions related to internal aspects are:

I1. How are the operations a process performs represented? That is, how is the program component of the process state represented?

I2. How does a process keep track of which operation it is to do next?

I3. How does a process store intermediate results until they can be used? That is, how is temporary storage accomplished?

I4. How does a process keep track of identifier-value bindings?

I5. What is the universe of discourse? That is, what classes of values are directly manipulable by

processes?

I6. What operations can a process perform? That is, what is the repertoire of the processor?

The specific questions related to external aspects are:

E1. How can processes achieve shared memory?

E2. How can a process indicate the relative importance of its current activity?

E3. How does a process know when to interrupt its current activity in order to respond to an interrupt event?

E4. How is monitoring for interrupt events accomplished? How often is it performed?

E5. How are the relative urgencies of interrupt events represented?

E6. How can a process find information about a particular interrupt event so that it may properly respond to it?

E7. Can a process accept more than a single request to interrupt its current activity? If so, how?

E8. How does a process know the proper response to a particular interrupt event?

E9. How can a process remember an interrupted activity so that it may resume it after responding to an interrupt event?

These questions should be kept in mind as the model is presented in Chapters 3, 4 and 7. Answers are to be found in the organization of the process state, the state transition rule, and the virtual memory processes deal with.

CHAPTER 3

The Model In Overview

3.1 Introduction

The presentation of the model for process synthesis is made in three parts. This chapter represents the first part. It describes the organization of process states and the process state transition rule. Chapter 4 is a more detailed discussion of the model. It presents the virtual memory, the universe of discourse and the treatment of process states as data objects. The details of aspects of the model discussed in overview in the present chapter are to be found in Chapter 4. Chapter 5 defines a simple programming language for using the model to synthesize patterns of process behavior. Some of the notation to be described in Chapter 5 can be used to advantage in the present and the following chapter and, therefore, is introduced informally whenever it is convenient to do so.

3.2 Organization of Process States

Process states are structured as collections of state components. Some of the information that must be included in process states was discussed in Chapter 2. The way I have chosen to organize that information is, in part, a reflection of the kinds of operations I expect will frequently be

performed on it.

The components of a process state are:

1. program (prog):

The prog component is a collection of "instructions" which defines the actions comprising the current activity of a process. Each instruction defines a state transition, the details of which may, in general, depend upon the values of other state components. The state of a process undergoes transition as the instructions of its prog component are interpreted in the environment provided by its remaining state components.

2. program counter (pc):

The pc indicates the instruction of the prog component to be interpreted next. As a process evolves, its pc component, in effect, defines a locus through its prog component.

3. level:

The activity of a process is interpretation of its prog component. The level component is a measure of the importance a process places on its current activity. Its value is an integer between one and lmax where lmax is a fixed (for all processes) integer greater than one; $1 \leq level \leq lmax$, $1 < lmax$. An exact value for lmax is not given. However, none of the examples appearing in later chapters requires it to be greater than 7.

4. active flag (aflag):

The aflag indicates whether or not a process is currently evolving.

5. stack:

The stack component is a pushdown store used for temporary storage. Items enter and leave the stack at its "top".

6. program identifiers (prog-id):

The prog-id component is a record of identifier-value bindings. A process uses its prog-id to interpret identifiers appearing in its prog component.

7. process identifiers (proc-id):

Like prog-id, the proc-id component contains bindings for identifiers. It represents a means by which a process can tailor its state structure to match the requirements of specific behavior patterns. (This point is discussed more completely below.)

8. reserve program (rp):

The rp component is a collection of instructions suitable to serve as the prog component. It is to be interpreted whenever the pc component is undefined. One way for the pc component to become undefined is for a process to "complete" interpretation of its prog component (see Sections 3.3 and 3.5).

9. handler programs (hp):

The hp component is an ordered collection (1,2,...,lmax) of

"programmed" responses for anticipated interrupt events. Each component of hp is itself suitable to serve as the prog component. Each is a collection of instructions defining a sequence of actions to be taken whenever a particular interrupt event occurs.

10. the queues (q):

The q component is an ordered collection (1,2,...,lmax) of first-in-first-out (FIFO) queues. Items enter a queue at one "end", called its back, and leave it at the other "end", called its front (see Section 4.2). Each component of q can collect requests for the process to interrupt its current activity.

11. the dump:

Whenever a process responds to an interrupt event the dump component is used to remember the interrupted activity. The dump is an ordered collection (1,2,...,lmax) of "areas", each capable of holding information about an interrupted activity sufficient to allow it to be resumed at a later time.

12. the seized flag (sflag):

The sflag of a process indicates whether it has been "seized" by another process. After a process has been seized it can neither continue its current activity nor respond to interrupt events until it is "released". A process can not directly set its own sflag.

More detailed discussion of each state component is deferred until Chapter 4. However, the existence of two

components, prog-id and proc-id, both of which record identifier-value bindings requires further comment. Both are included in the process state because they are used for quite different purposes. The prog-id component is used by a process to interpret identifiers which appear in its prog component. Its use is discussed further in Section 3.5. The proc-id component provides a means for a process to achieve variability in its state structure. Identifiers bound in it function as "extra" state components which enable a process to exhibit specific behavior patterns. Reconsider the subroutine pattern as described in Section 2.8. An important part of the state for a process able to engage in subroutining is the part which holds values for the state components to be saved at points of call. That part of the state can be thought of as a special "component" useful for the specific pattern of subroutining. It is special because not all processes require it. The proc-id component represents a means for realizing such "special purpose" state components. The difference in the way the prog-id and proc-id components are used is reflected by the amount the identifier-value bindings they record can be expected to change as a process evolves. The bindings recorded by the prog-id component can be expected to be quite dynamic. For example, in the case of subroutine behavior they would change on each call and return. On calls, bindings for formal parameters would be added to the prog-id and on returns, removed from it. On the other hand, the identifier-value bindings recorded by the proc-id component can be expected to

be more static. The proc-id component is used extensively in the examples to be found in Chapters 5 and 6.

Certain groups of state components are frequently used in combination. The prog and pc components form one such combination called the control. The control components identify the instruction which describes the next state transition to be taken by a process as part of its current activity. The combination called the status is made up of the prog, pc, level, and aflag components. It defines the current activity of a process. The stack, prog-id, proc-id and rp components comprise the environment combination. In the absence of interactions with other processes, the activity of a process is the interpretation of its control components with respect to its environment components.

3.3 The State Transition Rule

Each action in the sequence of actions making up the activity of a process is a state transition. Each transition occurs in accordance with the state transition rule. In general, the states of several processes can undergo transition concurrently. No commitment is made with respect to the relative speeds with which such transitions occur.

The parenthesized numbers appearing in the following discussion refer to parts of Figure 3.1, a flow diagram for the state transition rule. That figure and the discussion to

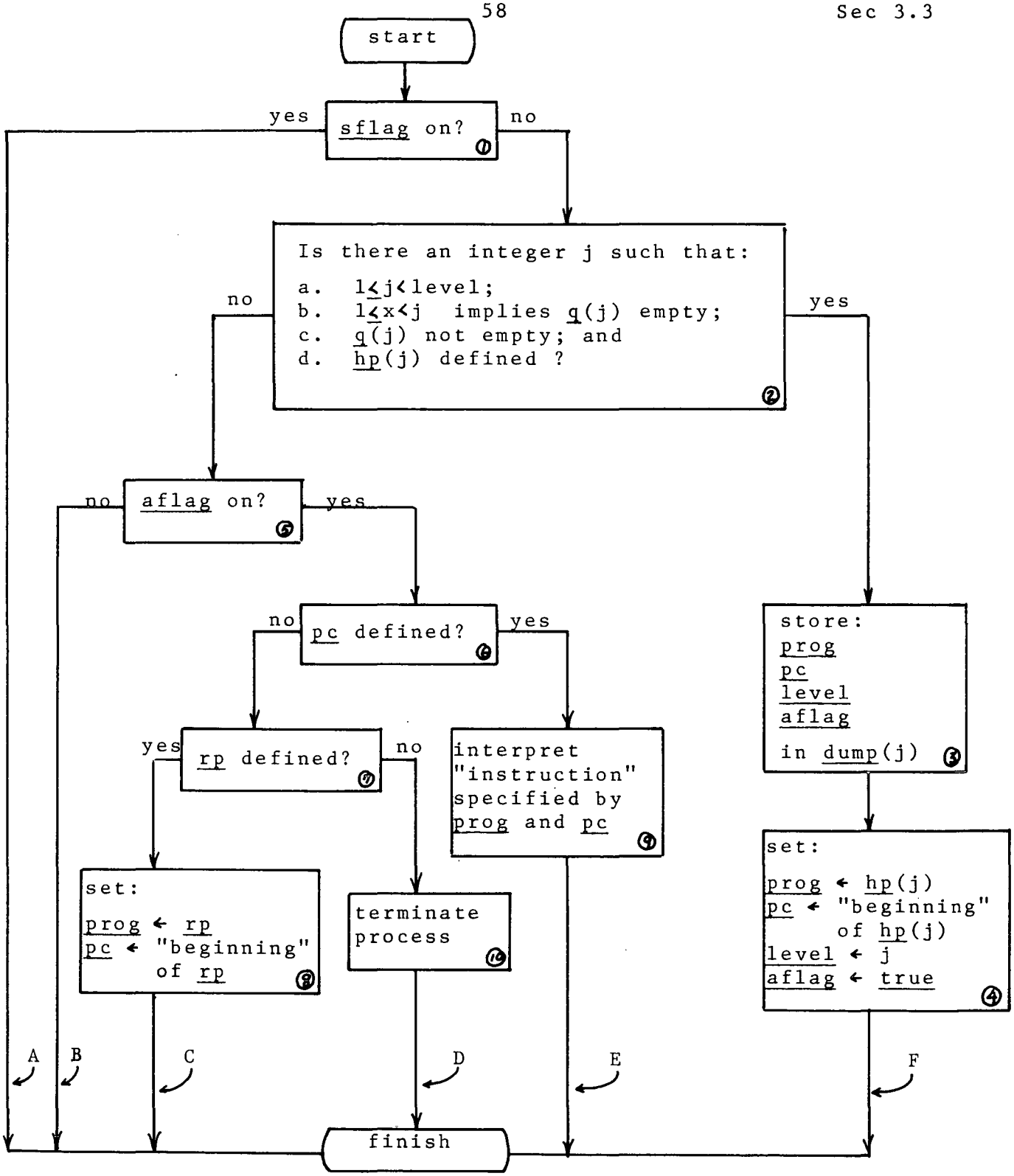


Figure 3.1

The State Transition Rule

follow use q(j) to denote the jth queue of the queues component. Similarly, dump(j) and hp(j) are used for denoting the jth components of the dump and handler programs components.

Each state transition begins with a check to see whether the process has been seized by another (1). If it has been seized the state "transition" is completed with no further action being taken.

If the process has not been seized, the state transition continues with a check to see if the current activity should be interrupted (2). It is to be interrupted only if both:

a. an interrupt event more important than the current activity has occurred (2.a,b,c); and
b. a programmed response has been defined for such an event (2.d). (The sense in which "defined" is used is explained in Section 4.3.)

If there is to be an interruption, the status components of the process state are saved in the dump (3) and the state transition is completed by initiating the response to the interrupt event (4). If an event has occurred for which no response has been defined, the event is ignored.

If no sufficiently important interrupt events have occurred and if the process is active (5), the process continues with its internal activity. If the pc component is defined (6), interpretation of the operation specified by the control components (9) completes the state transition. Section

3.5 discusses this part of the state transition rule in more detail. If the pc component is undefined and if the rp component is defined, the state transition is completed by initiating interpretation of the rp component (8). Should both pc and rp components be undefined the process is terminated (10). If the process is inactive (5) the state "transition" is completed with no changes having been made to the state.

The distinction made earlier between external and internal aspects of process behavior is reflected by the six possible routes through the state transition rule. Some paths, A and F, represent possible reactions to the influence of other processes. And others, B, C, D and E, describe how a process accomplishes its activity independent of the influence of other processes. As Section 2.4 notes, this distinction is not a rigid one. The action taken as the control components are interpreted (9) may involve other processes.

Assuming processes evolve independently of one another except for occassional interactions, path E describes how the majority of state transitions can be expected to occur. The state transition that occurs when the operation specified by the control components is interpreted (9) results in changes to pc and, depending upon the particular operation, to various other state components. Chapter 4 details operations processes can perform.

Paths A and B describe the behavior, or more accurately lack of it, exhibited by seized and inactive processes. Note that there is a difference between a process which has been seized and an inactive process which has no interrupt events to respond to. Neither produces change to its process state. However, an inactive process can respond should an important interrupt event occur, while one which is seized can not. Note also that once a process becomes inactive it can become active again only as the result of the actions of another process. Similarly, a process which is seized can be "released" only by another process. The role process seizure plays in the model is discussed in Section 3.6.

Because the state transition rule repeatedly checks a seized process to see whether it is seized (1) and an inactive one to see whether it is inactive (5), it might appear at first glance to waste a computing resource that could be used for other useful computation. Recall, however, that each process has a processor associated with it. When a process is inactive there is no other computation for its processor to perform. Hence, there is no waste. Similarly, when a process is seized, another process is keeping it from proceeding. Again, there is no computation its processor can perform and so, there is no waste. On the other hand, an implementation of the model is free to use techniques such as maintaining lists of inactive and seized processes to help achieve the effects of steps (1) and (5) of the state transition rule. An implementation that

repeatedly tests seized and inactive processes would be intolerably wasteful.

The rp component and its role in state transition path C requires comment. First, note that if ever the pc becomes undefined while the rp component is being interpreted, interpretation of rp is re-initiated. The possibility of looping exists. For example, completion of interpretation of rp has the effect of causing pc to become undefined (see Section 3.5). As a result, the next state transition re-initiates interpretation of rp. Such potential loops can be avoided by including explicit terminate or state component setting operations (see Section 4.6) in rp.

A second comment concerns the necessity of the rp component: It is unnecessary. Without it path C would disappear from Figure 3.1 and a process would terminate when its pc becomes undefined. The motivation for the rp component is that a typical process, through the course of its existence, can be expected to use a number of different programs as its prog component. An important aspect of such a process is the action taken when it completes interpretation of the program that is currently its prog component. That aspect is made explicit by the rp component. The rp can be regarded as the response defined for a special, internally generated interrupt event: completion of the prog component. In initiating the response to that "event" (8) it is unnecessary to save the status components; prog has been completed and the aflag and

level components are not changed. The subroutine pattern as described in Section 2.8 is an example of a process behavior pattern for which an important part is the action taken by a process upon completion of its prog. When a process finishes interpreting the prog corresponding to a subroutine it is to perform the return. As another example, consider the behavior patterns implied by Dijkstra's "parbegin" proposal (see Sections 1.2.3 and 6.4). A process which corresponds to a parallel path generated by a parbegin statement is no longer needed after it completes interpretation of its path and may, therefore, terminate. However, before it does so, it should report its completion. The rp component can be used to accomplish the subroutine return and the parbegin completion report and subsequent termination. Detailed examples of how rp can be used in synthesizing behavior patterns are to be found in Sections 5.3.3, 6,3, 6.4, 6.6 and 6.7.

3.4 External Aspects

The state components level, q, hp and dump determine whether, when and how a process responds to interrupt events.

An interrupt event occurs when an item appears in one of the queues of the q component of a process state. The presence of such an item represents an interrupt request. The item itself may be any member of the universe of discourse. The particular item constituting an interrupt request represents information about the interrupt event.

Monitoring for the occurrence of interrupt events is accomplished by checking the queues of the q component for the presence of interrupt requests (2,c). Because it is part of each state transition, the monitoring action is invisible to the processes being monitored. That is, processes need not concern themselves about it for it is automatically part of the activity of every process.

The value of the level component of a process state is a measure of the importance of the current activity of the process: the smaller its value, the more important the activity. The queue in which an interrupt request appears is a measure of the importance of the request relative to the current activity of the process. Hence, a request in q(3) is more important than one in q(4) but is less important than the current activity when the value of level is 2. Only an interrupt request which is more important to a process than its current activity can cause a response. That is, a request in q(j) can cause a response only if j<level (2,a). Less important interrupts requests are ignored until the value of the level component is appropriately changed. The state transition rule insures that when several important interrupt requests are present the most important one is responded to (2,b).

The way a process responds to interrupt events is defined by the hp component of its state. The hp(j) component defines its response to an interrupt request of importance j. Should

hp(j) be undefined, the process does not expect interrupt requests to appear in q(j). In the absence of a defined response interrupt events are ignored (2.d). This is consistent with the prerequisites for process interactions discussed in Section 2.4. Before meaningful interaction can occur between two processes conventions for the interactions must be established. If the interactions are to include interrupt events, the conventions must include the queues in which interrupt requests are to appear and the responses the requests are to trigger. The meaning of an undefined hp(j) is that the process has not agreed to respond to interrupt requests appearing in q(j). Such a request remains in q(j) until it is explicitly removed. Its presence in q(j) continues to represent an interrupt request and is capable of triggering a response, should one subsequently be defined.

When an interrupt event for which a response is defined occurs, the response is initiated in two steps. The first step (3) stores the status components of the state, which represent the current activity of the process, in dump(j). The second step (4) redefines the current activity by:

a. setting the control components so that interpretation of hp(j) begins on the next state transition;
b. setting the aflag to indicate that the process is active (recall that inactive processes can be interrupted); and
c. setting the level component to j to insure that the

response can be interrupted only by more important interrupt requests.

After actions appropriate to the interrupt event are taken, the interrupted activity can be resumed by restoring the status components, prog, pc, level and aflag, stored in dump(j). This can be accomplished using the operation

restore-dump (j)

It is possible for the response to an interrupt event to change the values of the status components saved in dump(j). Such behavior might be expected in response to an interrupt event signalling the occurrence of an error situation (Chapter 7).

An interrupt request in q(j) remains until it is explicitly removed. If the interrupted activity is to continue upon completion of the response to the request, the response should remove the request from q(j). If it does not, the presence of the request could re-initiate the response, resulting in a loop. Generally, when the interrupted activity is to continue after the interrupt request is handled, hp(j) will consist of the following sequence of actions:

```
        .                    } actions in response
        .                    } to the interrupt event
        .                    }
advance (q(j));              } remove request from q(j)
restore-dump (j)             } restore the interrupted
                             } activity
```

The advance and restore-dump operations could alternatively

appear in an appropriately specified rp component. In such a case when the "actions in response to the interrupt event" were completed, the resumption of the interrupted activity would appear to be automatic.

Processes cause interrupt events to occur. To cause an interrupt event a process must identify the process that is to receive the interrupt request. When created, each process has associated with it a unique identification tag or process designator. The effect of the operation

interrupt (pd, n, v)

is to place v at the end of the q(n) component of the state of the process designated by pd. As a result, process pd receives an interrupt request of importance n.

The preceding paragraphs have introduced additional notation:

a. operators are underscored;
b. semicolons separate operations appearing in a sequence;
c. parentheses separate operands from operators; and
d. when an operator takes more than a single operand, commas are used to separate the operators.

Because processes can evolve concurrently the possibility exists that several might simultaneously attempt to interrupt the same process at the same level. The arbiting property assumed for memory elements (see Section 2.3) prevents the loss of interrupt requests that could result from "races" between

several processes trying to deposit items in the same queue. It insures that whenever two or more processes simultaneously attempt to deposit items in the same queue, the items are deposited sequentially in an arbitrary order.

It is appropriate at this point to summarize the main ideas presented in this section. This can be accomplished nicely by reviewing how the q component of the process state is used:

a. It provides shared memory required for interactions between processes. A process can directly access its own q component; others can access it indirectly through the process designator of the process by using the interrupt operation.
b. It holds requests for the process to interrupt its current activity.
c. Because it is an ordered collection of queues, it can define the relative importance of each interrupt request it holds.
d. By holding requests associated with interrupt events it remembers the occurrence of events until the process can respond to them.
e. Because each queue is managed in a FIFO manner, the q component remembers the order in which interrupt events of the same relative importance occur.

3.5 Internal Aspects

This section discusses the part of the state transition rule concerned with interpreting the prog component (i.e., part 9 of Figure 3.1). It is primarily concerned with how the prog, pc, stack and prog-id components function together. In many respects their behavior is similar to that of the SECD evaluator described by Landin [Lan64]; the control components, prog and pc, correspond roughly to Landin's control (C), the stack to his stack (S), and the prog-id component to his environment (E).

It is useful to visualize the prog component as a directed graph with labeled arcs and to think of the pc component as identifying a node within that graph. Such a graph is referred to as a p-graph (for prog graph). The exact representation for the prog and pc components is presented in Section 4.5.

A p-graph is a directed graph for which

1. each arc is labeled with a label from the set next,true,false ; and
2. each node must have departing from it either:
   a. no arcs, or
   b. a single arc labeled next, or
   c. two arcs, one labeled true, and one labeled false.

The nodes in a p-graph represent "instructions" for the processor. The pc component indicates which node of the prog

p-graph is to be interpreted. Arcs departing from p-graph node n serve to specify the node to be interpreted following interpretation of node n. As a process evolves its pc component traces a path through its prog component by following p-graph arcs. Figure 3.2 illustrates some p-graphs.

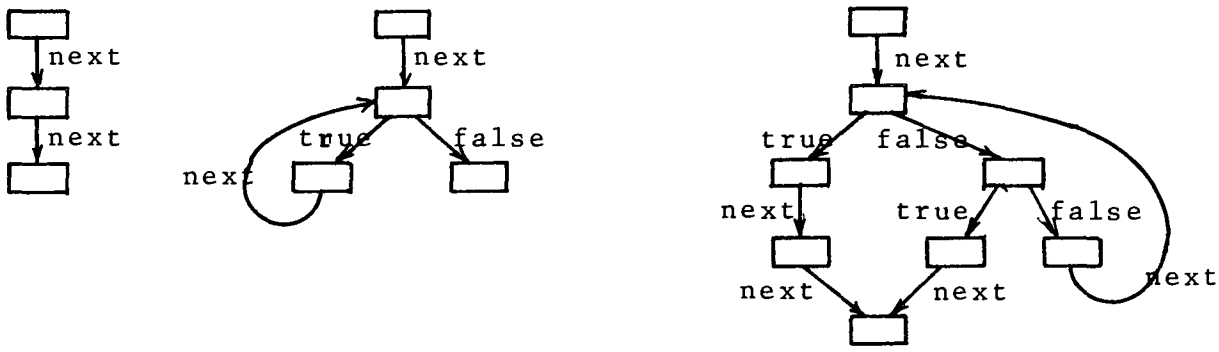


Figure 3.2
Examples of p-graphs. Nodes are represented by ▭'s.

P-graphs represent sequences of actions in postfix form. Together the prog, pc and stack components function as a postfix evaluator. The motivation for the choice of a "stack" evaluator for the model is that a single state component, the stack, can fulfil the temporary storage requirements of the state transition rule. Stacks represent a very simple mechanism for temporary storage. All operands come from the top of the stack. And, the values produced by all operations are pushed onto it. Their last in-first out property makes

stacks well matched to the temporary storage requirements of nested expression evaluation.

Figure 3.3a expands part 9 of the state transition rule displaved in Figure 3.1. It describes in more detail how interpretation of the control components is accomplished. The parenthesized numbers, (9.1) through (9.8), appearing in the following discussion refer to parts of Figure 3.3a.

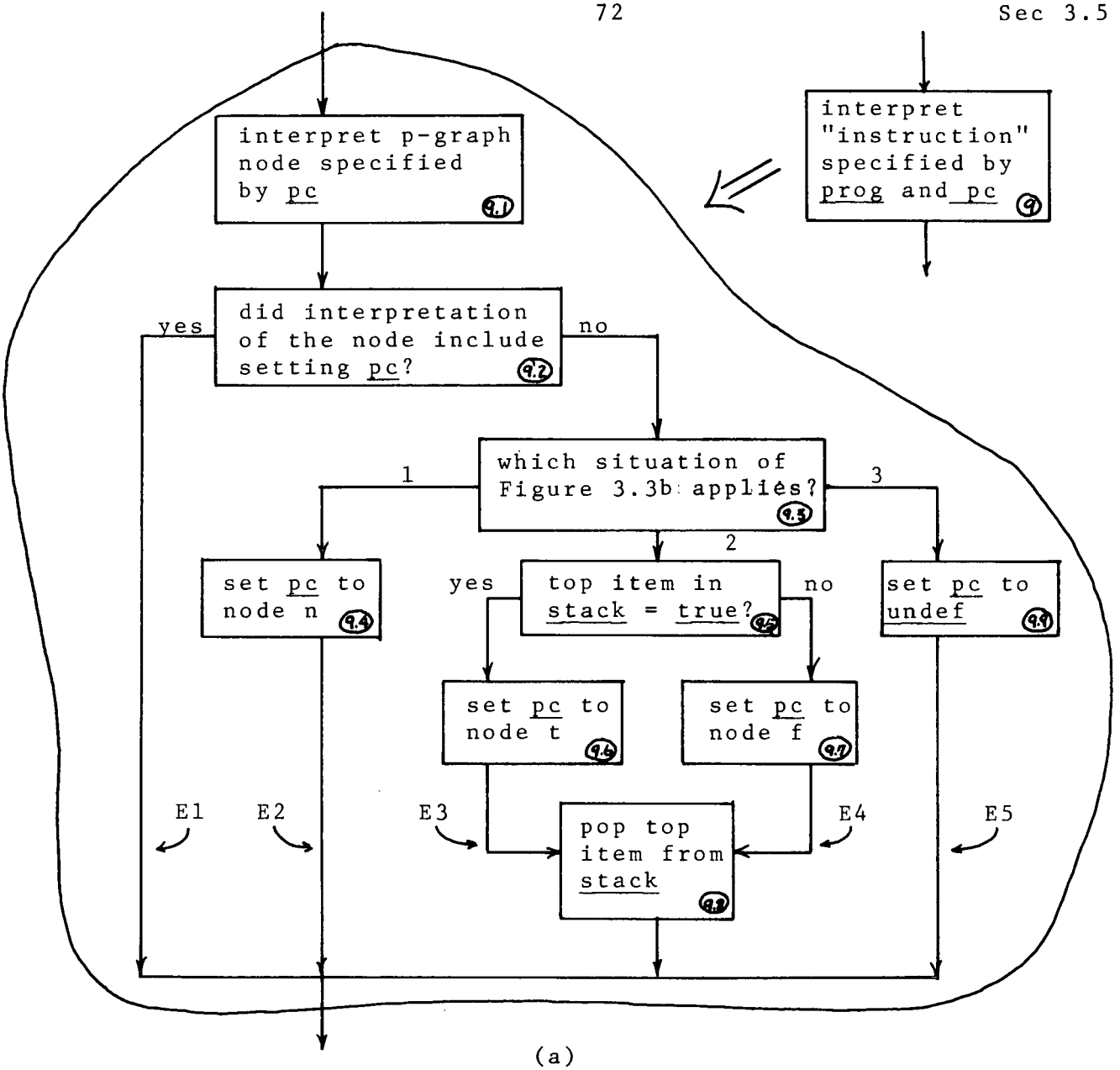
Conceptually, there are two parts to the interpretation of a p-graph node:

1. the state transition specified by the node is made (9.1); and
2. the pc component is set to designate the node to be interpreted on the next state transition (9.2-9.8).

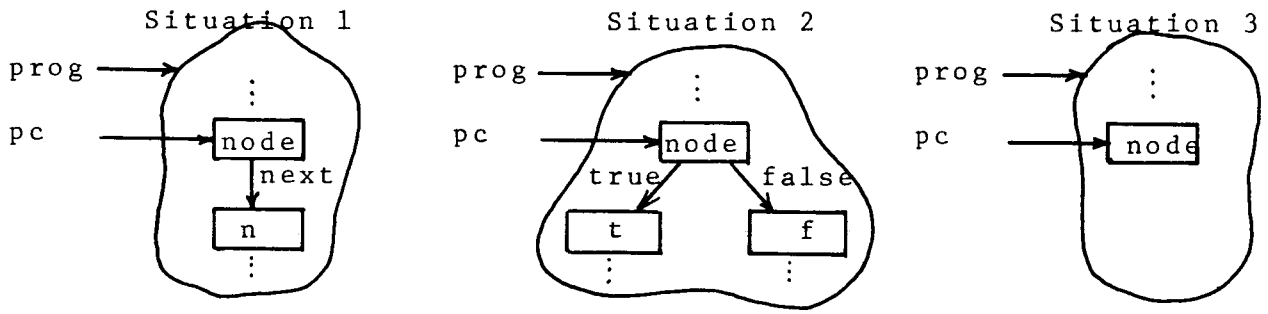
To explain the state transitions that result from interpreting p-graph nodes (9.1) it is useful to partition the nodes into three classes:

1. prog-items;
2. identifiers;
3. other members of the universe of discourse (Ω) (Ω includes prog-items and identifiers).

Prog-items represent operators. Several prog-items, among them restore-dump, advance, and interrupt, have already been introduced in Section 3.4. Identifiers and other members of Ω appearing as p-graph nodes represent operands. When a prog-item is to be interpreted, the operation denoted by it is



(a)



(b)

Figure 3.3

Part 9 of the state transition rule (See Figure 3.1)

performed. The operands, if any, required by the prog-item are taken from the top of the stack component and the results, if any, produced by the operation are pushed back onto the stack. If the p-graph node to be interpreted is an identifier, the value to which the identifier is bound by the process prog-id is pushed onto the stack. Finally, if the node is any member of Ω other than an identifier or a prog-item, the action taken is to push its value onto the stack.

When the action specified by a prog-item includes setting the pc component no further action is taken and the state transition is completed (9.2). (This case includes degenerate situations in which the value to which the pc is set is its current value.) Otherwise, pc must be set to complete the state transition. What pc is set to depends upon the arcs departing from the p-graph node. If the node has a "next" arc which terminates on node n (situation 1 of Figure 3.3b), pc is set to node n (9.4). If it has "true" and "false" arcs terminating on nodes t and f respectively (situation 2), then the new pc setting depends upon the top item of the stack. If it is the value true, pc is set to t (9.6), otherwise pc is set to f (9.7); in either case, the top item is popped from the stack. A p-graph node with no departing arcs (situation 3) represents a "terminal" node of the p-graph. After it is interpreted, the pc component is set to the value undef (9.9) to indicate that the prog has been completely interpreted. (For a discussion of undef see Section 4.3.) In such a case,

the next state transition either initiates interpretation of the rp component, if it is defined, or, if it is not, terminates the process (see Figure 3.1).

At this point an example is in order. Consider a process whose prog and pc components are as indicated in Figure 3.4.

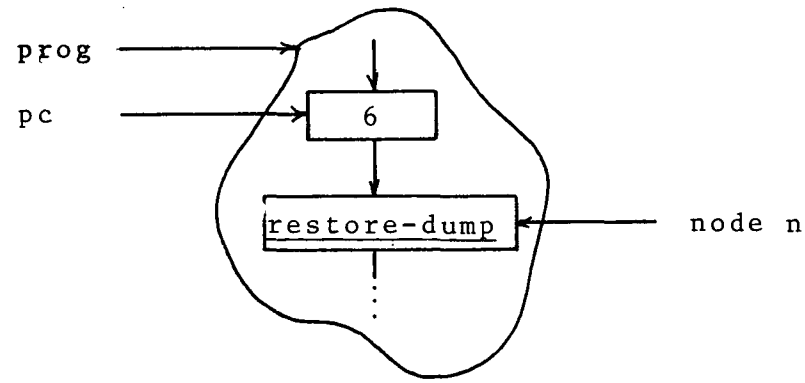


Figure 3.4
The prog and pc state components for a process (see text).

Assuming that its current activity is not interrupted, its next two state transitions are:

transition 1 - path E2

(refer to Figures 3.1 and 3.3a)

the integer 6 is pushed onto the stack component;

the pc component is advanced to node n

transition 2 - path E1

the prog, pc, level and aflag components are set from the values saved in the 6th component of the dump;

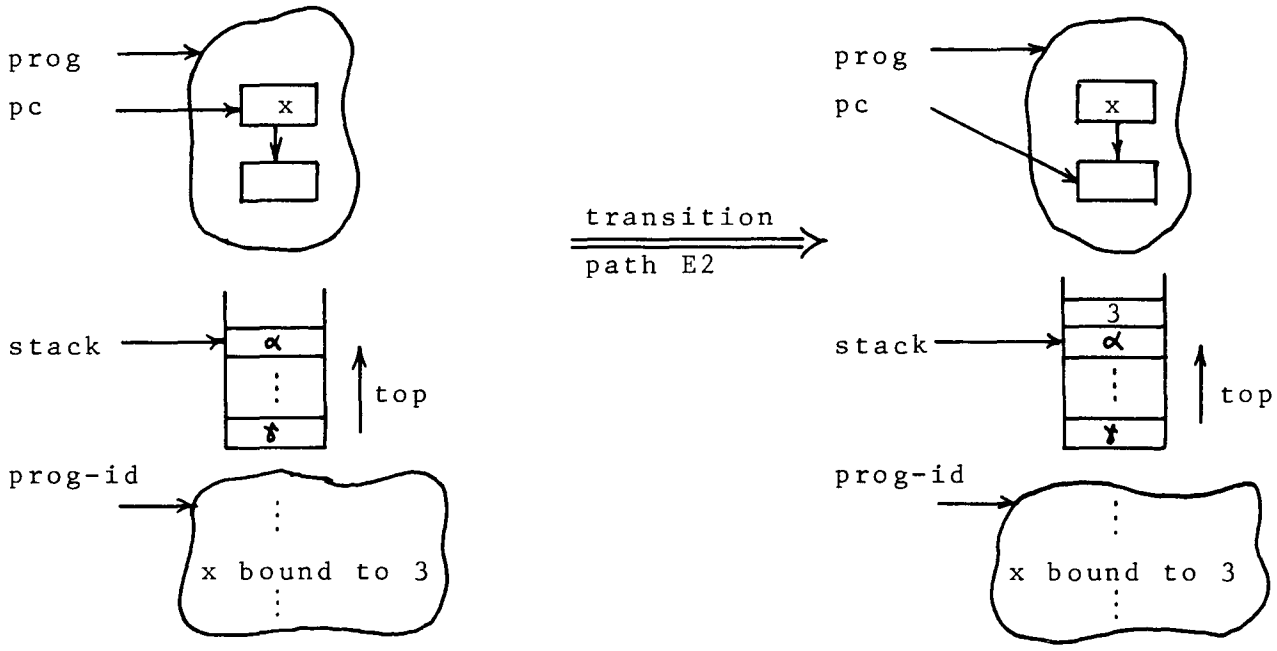
the top item (the integer 6) is popped from the stack component.

As noted earlier both in this section and in Section 2.9, identifiers can appear in the prog component. The prog-id component is used to interpret such identifiers. It is a record of identifier-value bindings. Note that there is a difference between an identifier and the value to which it is bound. Conventional programming practice dictates that an identifier appearing in a program means the value to which it is bound. That practice is followed in the model. When the p-graph node to be interpreted is an identifier, the identifier is interpreted to be the value associated with it by the prog-id component. The following actions are taken (as part of (9.1)) to interpret such a node (see Figure 3.5a):

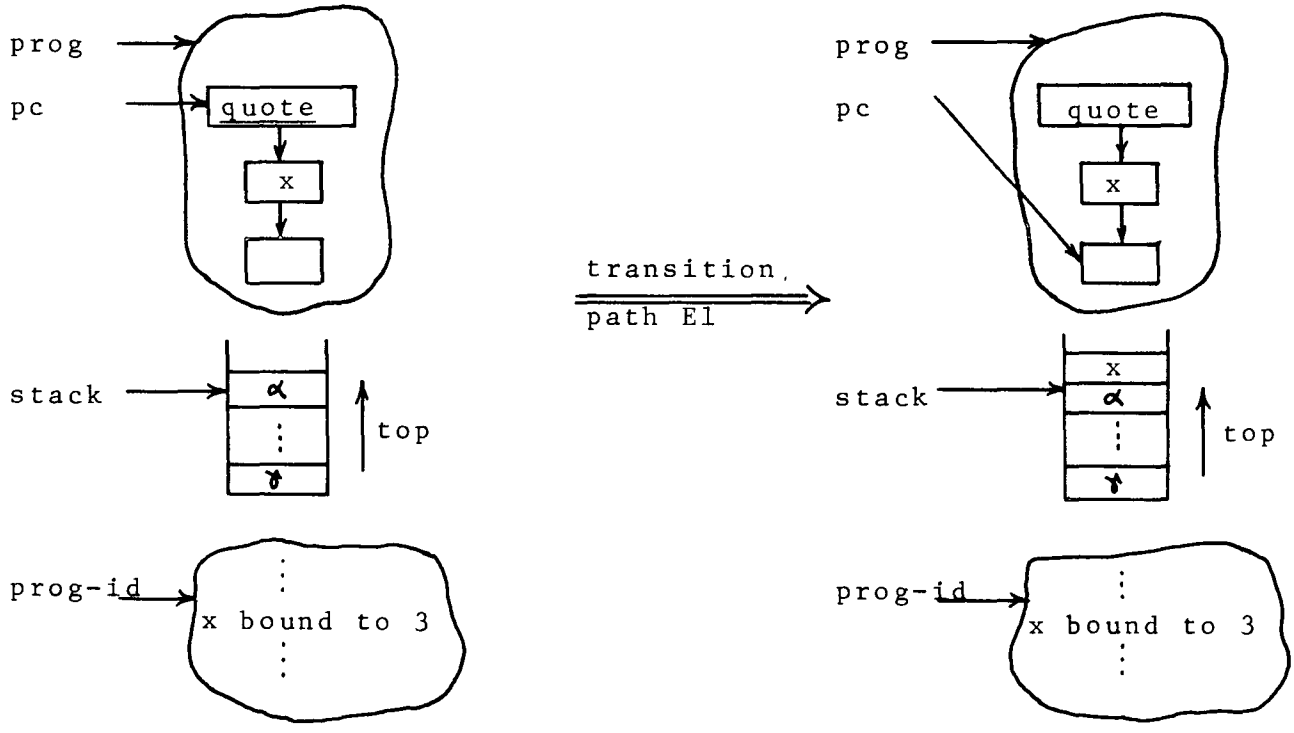
a. the prog-id component is searched for a binding for the identifier;

b. if a binding for the identifier is found, the value to which it is bound is pushed onto the stack; otherwise, the value undef is pushed onto the stack, to indicate that the identifier is currently unbound.

There is occassion to deal with identifiers as objects themselves (see Section 4.4). The prog-item quote can be used in such a case to prevent an identifier from being interpreted with respect to the prog-id component. When the p-graph node being interpreted is quote, the action taken is to push the "next" p-graph node item onto the stack and advance the pc two



(a)



(b)

Figure 3.5

a. Interpretation of the identifier x.

b. Interpretation of quote (x).

nodes. Figure 3.5 compares the interpretation of an identifier with that of a "quoted" identifier. A complimentary prog-item binding interprets a specified identifier with respect to a specified prog-id to produce the value to which the identifier is bound in that prog-id (see Section 4.5).

3.6 Manipulating the Process State

A process has the ability to extract and to set both the values of its own state components and those of other processes. Each operation a process performs is accomplished by making changes to its state. However, a useful intuitive distinction can be made between "component setting" operations and "ordinary" operations. The state transition required to accomplish a component setting operation is explicit in the operation. The state components to be changed and the values they are to be changed to are explicitly stated as part of the operation itself. On the other hand, ordinary operations, such as arithmetic or structure manipulating operations, represent more implicit changes to the state in the sense that they do not detail the state changes required to accomplish them. Ordinary operations are largely independent of the process state structure, whereas component setting operations are not.

State component setting operations are useful for making extraordinary changes to the activity being performed by a process. The components set determine the degree of change: setting pc is equivalent to a local goto (i.e., within the same

prog); setting level changes the class of interrupt requests to which a process responds; setting parts of hp redefines responses interrupt requests receive; setting prog-id redefines identifier-value bindings and may have a significant effect upon the behavior of a process. When a process creates another it can initialize the state of the new process using state component setting operations.

Whenever a process manipulates its own state it is engaged in potentially tricky business because the program and data that describe the manipulations are part of the state being manipulated. Consider, for example, a process attempting to make a "copy" of its stack component. To do so it could obtain a previously unused stack (see Section 4.2) and begin pushing items held in its stack component onto it. The potential confusion arises from the fact that all operations involved in making the copy take their operands from the stack being copied. An example in Section 5.3.1 shows how such a copy can be made. Whenever a group of components to be set includes the prog and pc, care must be taken to insure that prog and pc are the last set, since they describe the setting operations. The situation is less confusing when one process sets components of another's state because the program and data describing the operations are not part of the state whose components are being changed.

Because a process can set the state components of another, the situation could occur that several processes simultaneously

attempt to change components of the same process state. For example, process P1 might attempt to set components of process P2's state while P2 itself is changing them as part of its internal activity. Or, P1 and a third process P3 might both attempt to set components of P2's state. In either case P2's state could be left in an inconsistent condition.

The model insures that at any time only a single process can be engaged in setting the state components of process P2. That process may be P2 itself or it may be another process. A prerequisite for another process P1 to set components of P2's state is that P1 have P2 "seized". Process seizure is a kind of locking operation. No other process can seize P2 while P1 has it seized. Consequently while it is seized by P1 no process other than P1 can change the components of P2. P2 is unable to because it is seized (part 1 of Figure 3.1) and others, because to do so they must first seize P2. Because the sflag is tested only once as part of each state transition and because that test occurs before any changes to the state are initiated (see Figure 3.1), a process can not be stopped "mid-way" through a state transition as the result of being seized by another process. In particular, once P2 initiates an operation to set its own components the operation is guaranteed to be completed.

The prog-item t-seize (for test and seize) is used to seize a process. It is a predicate with a side effect. When

is interpreted an attempt is made to seize the process designated by pd. If that process is not already seized, the attempt succeeds and the value of the predicate is true; otherwise the attempt fails and the value of the predicate is false. The t-seize operation is indivisible in the sense that no other process can seize the process between the test and the seizure. Furthermore, if t-seize succeeds the sflag of the designated process is set immediately but the value true is not "returned" to the seizing process until the designated process completes its current state transition. This guarantees that the seizing process can not set the state of the seized process while the seized process is completing a state transition. A process can "release" processes it has seized. Interpretation of

release (pd)

causes the process designated by pd to be released (i.e., to be no longer seized).

CHAPTER 4

The Model In Detail

4.1 Introduction

The description of the state transition rule started in Section 3.3 and continued in Sections 3.4 and 3.5 is complete except for the part that reads

interpret the p-graph node specified by pc

(i.e., part 9.1 of Figure 3.3a). To complete the description this chapter discusses each prog-item and the state transition that results from its interpretation. In the course of doing so it defines the virtual memory for the model and its universe of discourse. In addition, it presents the way in which the model treats state components as data objects. A summary of all prog-items may be found in Appendix 1 which is included to serve as a reference source.

This chapter also discusses the extent to which processes are isolated from one another and ways in which interactions between them can occur. That discussion reveals a weakness in the model, to be corrected in Chapter 7, concerned with controlling capabilities of processes. The chapter ends with a summary of the process representation method in which the questions stated in Section 2.10 are re-examined.

Before proceeding, it is appropriate to introduce some conventions to be used in this chapter. As Section 3.5 notes, prog-items take their operands from the top of the stack component. The expression

pi (r1, ..., rn)

is used to denote application of prog-item pi to operands r1,...,rn. The p-graph fragment corresponding to that expression is such that when pi is interpreted the top item in the stack component is the value of r1, the second item the value of r2, etc. For example, the p-graph fragment corresponding to

interrupt (pd, n, v)

is displayed in Figure 4.1. The phrase

the value of E is V

where E is an expression is frequently used in the sequel. Its meaning is

after the p-graph fragment corresponding to E is interpreted the top item in the stack is V.

All prog-items taking operands remove those operands from the stack. Unless it is explicitly noted that a particular prog-item produces a value, the only effect its interpretation has on the stack component is to remove its operands (if any) from the top of it. For example, the effect interpretation of interrupt has on the process stack is to pop the top three items from it.

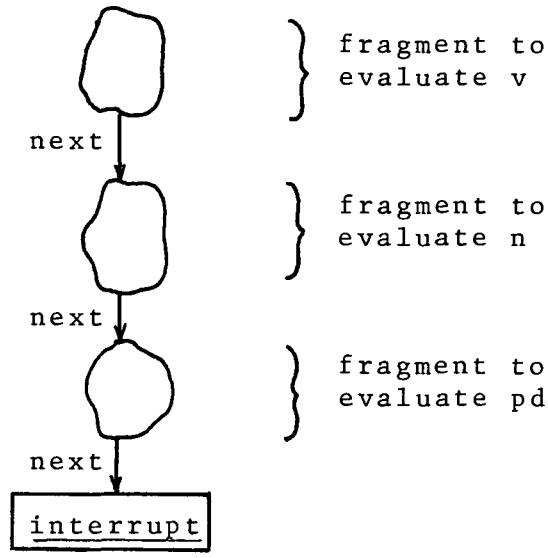


Figure 4.1
The p-graph fragment corresponding to
interrupt (pd, n, v)

4.2 Virtual Memory for the Model

The virtual memory provides three kinds of memory elements:

1. stacks;
2. queues; and
3. cells.

Stacks and queues have been discussed in connection with process states and the state transition rule. Although they behave in different ways, the three kinds of memory elements have four properties in common:

1. storage property: memory elements are used by processes to hold values. A memory element can hold

any member of the universe of discourse.

2. retrieval property: processes can retrieve values stored in memory elements; retrieval has no effect on the values stored in a memory element.
3. allocation property: a process can request and obtain use of a "new" memory element guaranteed inaccessible to other processes (within the constraint that memory is finite); until it chooses to "share" them with other processes, a process has "exclusive" use of memory elements it allocates.
4. arbiting property: attempts by several processes to simultaneously access the same memory element (for storage or retrieval) result in sequential access.

How memory elements can be guaranteed inaccessible to certain processes is discussed in Section 4.7.

A process can use stacks in addition to the one which is the stack component of its state. The values held in a stack are kept in a linear list. The storage operations make insertions to and deletions from one end of the list, called the top of the stack. Each stack has associated with it a stack designator which uniquely identifies it.

Like a stack, a queue holds the values stored in it in a linear list. As with stacks, the storage operations are insertions to and deletions from that list. However, for queues insertions occur at one end of the list, called the back of the queue, and deletions occur at the other end, called the

front of the queue. Associated with each queue is a queue designator which uniquely identifies it.

A cell, unlike a queue or stack, can hold only a single value. Each cell has associated with it a cell designator which uniquely identifies it. The terms l-value and r-value, coined by Strachey [Str67], are used in the sequel when referring to cells. The r-value of a cell is the value held by it; the l-value of a cell is its cell designator.

Cells, stacks and queues are collectively referred to as memory elements and cell designators, stack designators and queue designators, as memory designators. Note that a distinction between memory elements, memory designators and the values held by memory elements is made by the virtual memory.

The prog-items new-cell, new-stack and new-queue are the allocation operations. They are used to request the use of new memory elements. The value of

new-cell (value)

is the cell designator of a new cell whose r-value is initialized to value. The value of

new-stack

is the stack designator of a new stack and that of

new-queue

the queue designator of a new queue. New stacks and queues are initially empty.

The storage operation for cells is store;

store (cd, value)

sets the r-value of the cell designated by cd to value. The insertion and deletion operations for stacks are push and pop respectively. Interpretation of

push (sd, value)

causes value to be inserted at the top of the stack designated by sd, and that of

pop (sd)

has the effect of removing a single value from the top of the designated stack. For queues the storage operations are enqueue (insertion) and advance (removal). When

enqueue (qd, value)

is interpreted, value is inserted at the back of the queue designated by qd. The value at the front of the designated queue is removed by

advance (qd)

Stacks and queues have finite capacity. Stacks can hold smax items and queues qmax. An attempt to make an insertion to a full stack or queue results in an error situation (see Section 7.6).

The retrieval operations for memory elements are rval, length, and index. The r-value of the cell designated by cd is

rval (cd)

The value of

length (sd)

is the number of items currently held in the stack designated by sd. Similarly, the number of items currently held in the queue designated by qd is

length (qd)

The prog-item index is used to retrieve values held in stacks and queues. The nth item from the top of the stack designated by sd is

index (sd, n)

(by convention, the "first item from the top" of a stack is the top item). Similarly

index (qd, n)

is the nth value from the front of the designated queue. Should less than n items be in the stack or queue, the value produced by index is undef.

With respect to storage operations queues behave in a strictly FIFO manner. Similarly, stacks exhibit strictly LIFO behavior with respect to storage operations. Note, however, that the retrieval operation index permits the value of any item in a stack or queue to be accessed. This property of stacks and queues with respect to the retrieval operation represents a departure from the ususal restriction that only the item at the top of a stack or the front of a queue be accessible.

An additional operation for cells provides a basis for building locking mechanisms. It is t-set (for test and set), a predicate with a side effect. When the predicate

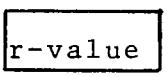
t-set (cd)

is interpreted the r-value of the cell designated by cd is examined. If it is 0, it is set to 1 and the value of the predicate is true; otherwise it is left unchanged and the value of the predicate is false. The t-set operation is "indivisible" in the sense that no other process can set the r-value of the cell between the test and the set. An example in Section 5.3.2 uses t-set to build a locking mechanism.

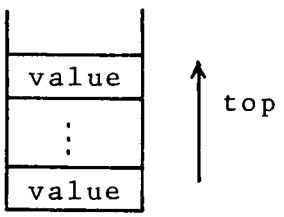
It is frequently useful to be able to display memory elements graphically. The following conventions are used to graphically represent memory elements and memory designators:

a. memory designators are drawn as arcs pointing to memory elements; arcs that point to the same memory element represent the same memory designator;

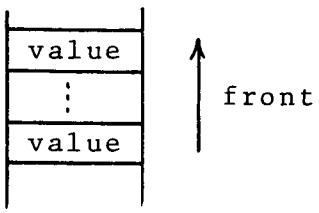
b. cells are drawn



c. stacks are drawn



d. queues are drawn



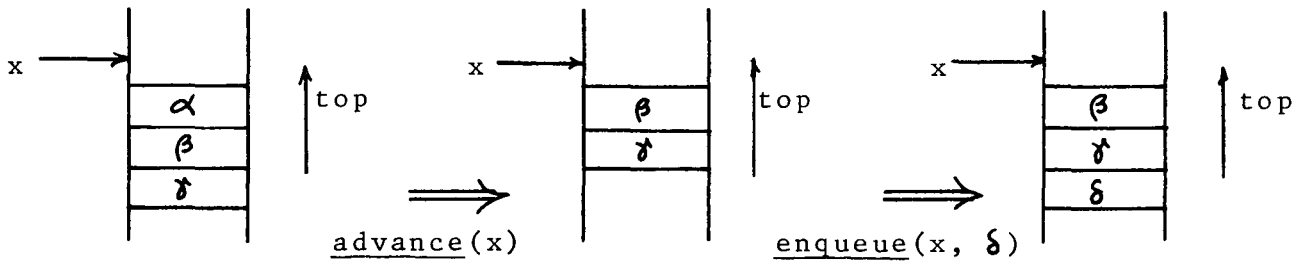
Consider the queue designated x shown in Figure 4.2a. It holds the values α, β and γ. For that queue the following are identities:

length (x) = 3

index (x, 1) = α

index (x, 3) = γ

index (x, 4) = undef



(a) (b) (c)

Figure 4.2

The effect of the queue storage operations advance and enqueue.

Figures 4.2b and 4.2c show the queue after interpretation of

advance (x)

and then of

enqueue (x, δ)

4.3 The Universe of Discourse

The universe of discourse for processes, Ω, is the set of objects processes can deal with. Ω is partitioned into classes called types. Different types have different properties. Practically speaking, one type is distinguished from another by the operations that can be performed on it. For example, the operation t-set can be performed on l-values but not on integers; l-values and integers are different types.

Section 2.8 develops an operational definition for Ω which can be paraphrased

> anything that can appear as an item in the stack component is in Ω.

Together with the discussion of previous sections this definition implies that Ω includes

process designators	(Section 3.4)
cell designators (l-values)	(Section 4.2)
stack designators	(Section 4.2)
queue designators	(Section 4.2)
identifiers	(Section 3.5)
prog-items	(Section 3.5)
integers	(Section 3.4)
truthvalues	(Section 3.5)
undef	(Section 3.5)

In addition Ω includes two kinds of structure designators: designators for rows and for structs (to be discussed in Section 4.4). Note that designators for memory elements,

rather than memory elements themselves, are members of Ω.

The universe of discourse for processes is such that given a value it is possible to determine its type. Interpretation of a prog-item includes checking its operands for type. If an operand is not of the proper type an error situation results (see Section 7.6). Such behavior is commonly called dynamic type checking.

There are explicit type checking predicates for each type. If the operand of a type checking predicate is of the type being tested, the predicate produces the value true; otherwise, it produces the value false. The type checking predicates are: is-proc, is-lval, is-stack, is-queue, is-prog-item, is-ident, is-int, is-truthval, is-undef, is-row, and is-struct.

The prog-item eq is the equality predicate; it tests any two members of Ω for equality. The value of

eq (v1, v2)

is true only if the value of v1 is equal to the value of v2.

There are two operations of interest for objects of type prog-item: quote and do. As it does for identifiers, quote prevents prog-items from being interpreted (see Section 3.5). When do is interpreted the top item of the stack is expected to be a prog-item. The state transition that results is the same as would result if the prog-item were popped from the stack and used in place of do as the prog-item being interpreted.

The usual operations are included for integers and truthvalues. The operations for integers are plus, minus, times and divide; the predicates are greater than (gr), less than (ls), greater than or equal (ge), less than or equal (le) and the general equality predicate (eq). The truthvalues are denoted true and false. The operations for truthvalues are and, or and not.

The value undef is special in two respects:

1. all type checking predicates produce true when applied to it; and
2. with the exception of predicates, each value-producing operation produces undef if any of its operands is undef. (e.g., 2 + undef = undef, but eq(2, undef) = false)

Although it "propogates" in the same way as an undefined value would (i.e., 2 + undef = undef), undef is a specific, testable value (there is a predicate is-undef) and therefore, is not, strictly speaking, undefined. The pc, rp and hp(i) components are said to be undefined (see Figure 3.1) whenever their values are undef.

Note that a number of types usually found in programming languages, such as reals and strings, are not included in Ω. Additional types such as these could certainly be added to Ω with little more than a perturbation to the model resulting. I would consider as faithful an implementation of the model that includes, in addition to the prog-items described in this

dissertation, prog-items whose only effect is upon the top part of the process stack. Thus, an implementer is free to include strings and string operators in his implementation if he chooses. He is free, also, to introduce additional operators for existing types.

4.4 Structures

A structure is an organized collection of values. Each structure has a structure designator associated with it when it is constructed. A particular member or component of the collection can be accessed by "applying" a selector to the structure designator. Any member of the universe of discourse, including a structure designator, can be a component of a structure.

Processes can deal with two kinds of structures: rows and structs. Rows are sequential structures for which integers are used to select components. For structs, identifiers are used as selectors. Neither structs nor rows are required to be homogeneous.

The null structure, nil, is the empty collection of values and is considered to be both a row and a struct. Thus

is-row (nil) = is-struct (nil) = true.

The predicate for the null structure is is-nil.

The row designated by row designator rd has

length (rd)

components. The integers 1,2,...,length(rd) serve as selectors for it. Providing that 1≤n≤length(rd), the value of the expression

index (rd, n)

is the nth component of the row designated by rd; otherwise it is undef.

The prog-item row is a constructor for rows. The value of

row (n, vl, ..., vn)

is the row designator for a row having n components, vl, ..., vn.

Recall that "unquoted" identifiers appearing as p-graph nodes are interpreted with respect to the prog-id component. Identifiers to be used as selectors for structs should be quoted to prevent their interpretation with respect to the prog-id. In the discussion of structs that follows id-exp, id-expl, ..., id-expn are used to denote identifier-valued expressions and id, idl, ..., idn to denote the values of those expressions.

The value of

selectors (sd)

is the row designator for a row whose components are the selectors for the struct designated by sd. No commitment concerning the order in which the selectors for the struct appear in the row of selectors is made. Successive applications of selectors to the same struct designator are

guaranteed to produce the same row designator. If id is a component of selectors(sd), the value of

select (sd, id-exp)

is the "id component" of the designated struct; otherwise its value is undef. The expression id-exp can be arbitrarily complex. Selection of the "id component" of a struct can be accomplished simply by

select (sd, quote (id))

Structs can be constructed using the prog-item struct. The value of

struct (n, id-exp1, v1, ..., id-expn, vn)

is the struct designator for a struct having the n components v1, ..., vn, selected respectively by the selectors id1, ..., idn. When the prog-item struct is interpreted, the top item in the process stack, which is the value of n, indicates the number of items to be taken from the top of the stack to build the struct.

The following notational conventions permit the quote operation to be omitted when referring to selectors, making descriptions of struct manipulations somewhat more readable:

1. sd.id is equivalent to select (sd, quote(id));
2. ⟦id1:v1, ..., idn:vn⟧ is equivalent to

   struct (n, quote(id1), v1, ..., quote(idn), vn)

There are analogous conventions for rows:

1. rd[n] is equivalent to index(rd, n);
2. ⟦v1,...,vn⟧ is equivalent to

row (n,vl,...,vn)

Rows and structs have identity independent of their structure. That is, the value of

eq (sdl, sd2)

where sdl and sd2 are structure designators, is true only if sdl is the same structure designator as sd2. Thus, for example

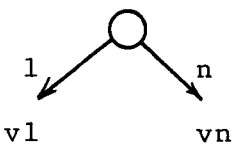
eq (row (2, 1, 2), row (2, 1, 2)) = false

because the operands of eq designate different, although structurally identical, rows.

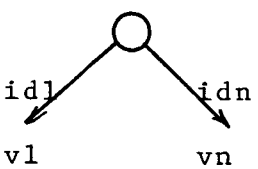
As with memory objects it is frequently useful to graphically display structures and structure designators. Rows and structs are displayed using the following conventions:

a. nil is drawn

b. The row ⟦vl, ..., vn⟧ is drawn



c. The struct ⟦idl:vl,...,idn:vn⟧ is drawn



d. Structure designators are drawn as arcs pointing to structure circles. Arcs that point to the same circle represent the same structure designator.

The row

$$[\![4,\ \underline{\text{nil}},\ \underline{\text{new-cell}}(\underline{\text{new-cell}}(6))]\!]$$

and the struct

$$[\![\text{a}:[\![\underline{\text{undef}},\ 7]\!],\ \text{b}:5,\ \text{c}:\underline{\text{new-cell}}(\underline{\text{new-stack}})]\!]$$

are displayed in Figure 4.3. These two expressions are equivalent to

$$\underline{\text{row}}\ (3,\ 4,\ \underline{\text{nil}},\ \underline{\text{new-cell}}(\underline{\text{new-cell}}(6))\ )$$

and

$$\begin{aligned}&\underline{\text{struct}}\ (3,\ \underline{\text{quote}}(\text{a}),\ \underline{\text{row}}(2,\ \underline{\text{undef}},\ 7),\\ &\qquad\qquad \underline{\text{quote}}(\text{b}),\ 5,\\ &\qquad\qquad \underline{\text{quote}}(\text{c}),\ \underline{\text{new-cell}}(\underline{\text{new-stack}})\ )\end{aligned}$$

respectively.

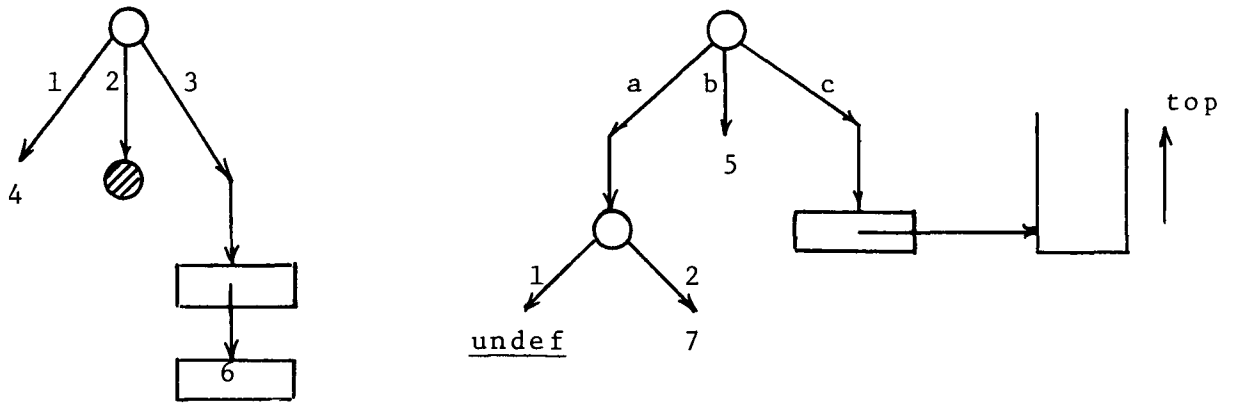
There are two additional constructors for structures. Each produces from a given structure a new one identical in all respects to the original with the single exception that the new structure has one component more than the original. The value of

$$\underline{\text{aug-row}}\ (\text{rd},\ \text{v})$$

is the row designator td, such that

a. $\underline{\text{length}}\ (\text{td}) = 1 + \underline{\text{length}}\ (\text{rd})$

b. $$\underline{\text{index}}(\text{td},\text{n}) = \begin{cases} \text{v} & \text{for n} = \underline{\text{length}}(\text{td}) \\ \underline{\text{index}}(\text{rd},\text{n}) & \text{otherwise} \end{cases}$$



(a) (b)

Figure 4.3

A row and a struct produced by the constructors row and struct, respectively (see text).

The value of

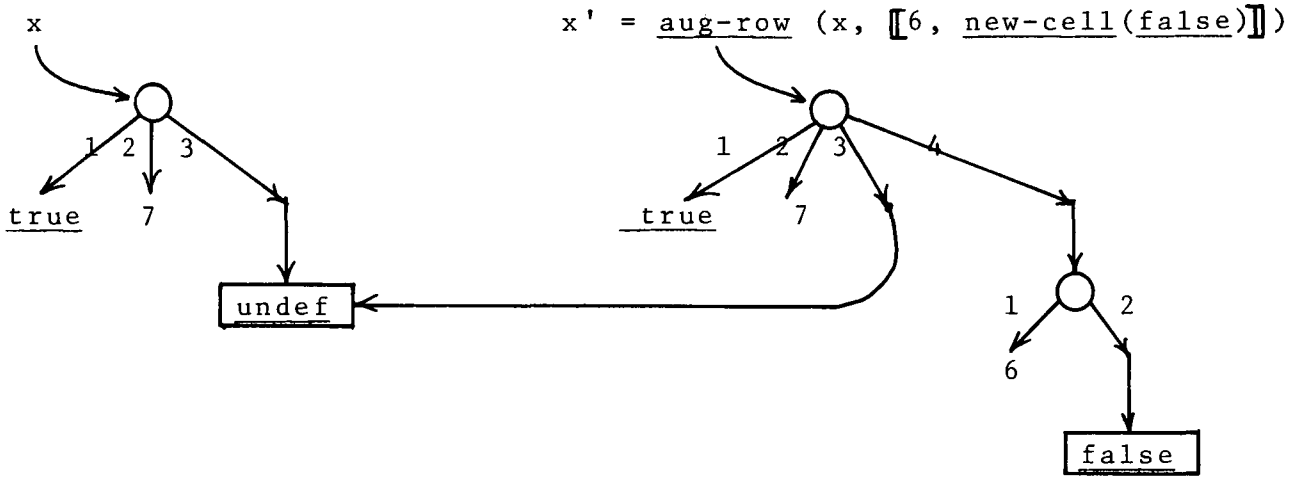
aug-struct (sd, id-expl, v)

is a struct designator td such that

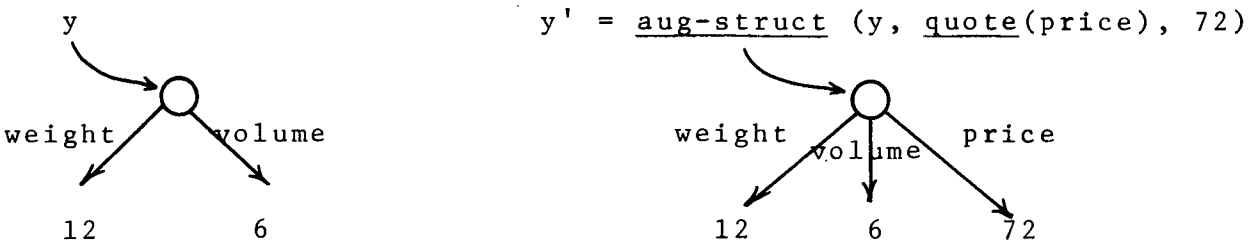
a. selectors (td) = aug-row (selectors(sd), id-expl)

b. $$\text{select}(td,id\text{-}exp) = \begin{cases} v & \text{if } id = id1 \\ \text{select}(sd,id\text{-}exp) & \text{otherwise} \end{cases}$$

Figure 4.4 illustrates structures produced by aug-row and aug-struct. Note that x[3] and x'[3] "share" the same l-value. Any operation which has the effect of changing rval(x[3]) also has the effect of changing rval(x'[3]). Sharing is discussed further in Section 5.3.3. Operations complimentary to aug-row and aug-struct for constructing from a given structure another



(a)



(b)

Figure 4.4

Examples of structures produced by the constructors aug-row and aug-struct. Note that x[3] and x'[3] "share" the same l-value.

one with one fewer component can be expressed in terms of length, index, aug-row, selectors, select and aug-struct. As remarked in Section 4.3, an implementor is free to build prog-items for such operations into his implementation.

4.5 State Components as Members of Ω

There are operations for directly manipulating components of process states. Hence, by the operational definition given in Section 4.3, state components are members of Ω.

prog, rp, hp:

The prog, rp and each of the lmax components of hp are p-graphs. There are two ways the model could deal with p-graphs as values:

1. a new type, p-graph, could be introduced and included as part of Ω; or
2. an existing type could be used to represent p-graphs.

The second approach is used for the following reasons:

1. it allows the pc component to be particularly simple;
2. it requires no additions to Ω;
3. it appears to be no more complex than the first approach.

Rows are used to represent p-graphs. The components of a row representing a p-graph are designators for structs which correspond to the nodes of the p-graph. Each struct is of the form:

[[item:~~ , next:~~]]

The integer selector for the struct corresponding to a specific p-graph node is referred to as the index of the node.

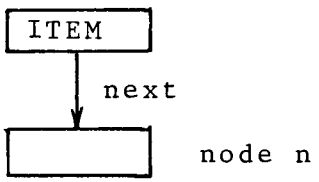
Three cases must be considered:

1. The p-graph node has no departing arcs:

Recall that ITEM can be a prog-item, an identifier or any other member of Ω. The struct corresponding to such a p-graph node is:

[[item:ITEM, next:undef]]

2. The p-graph node has a single departing arc:

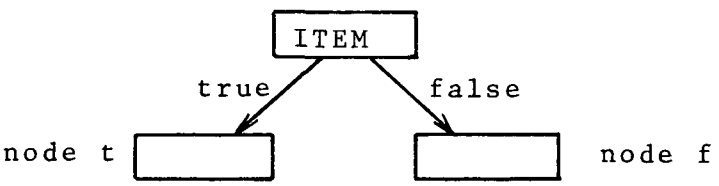


The corresponding struct is:

[[item:ITEM, next:INDEX_n]]

where INDEX_n is the index of node n.

3. The p-graph node has two departing arcs:



Such a node is represented by a struct of the form

⟦item:ITEM, next:⟦INDEX_t, INDEX_f⟧⟧

where INDEX_t and INDEX_f are respectively the indices of node t and node f.

The p-graph shown in Figure 4.5 can be represented by the row

```
⟦  ⟦item:x,       next:2⟧,
   ⟦item:length,  next:3⟧,
   ⟦item:10,      next:4⟧,
   ⟦item:eq,      next:⟦5, 8⟧⟧,
   ⟦item:10,      next:6⟧,
   ⟦item:x,       next:7⟧,
   ⟦item:index,   next:undef⟧,
   ⟦item:z,       next:9⟧,
   ⟦item:x,       next:10⟧,
   ⟦item:push,    next:1⟧   ⟧
```

Note that there are a number of other rows which also represent the structure of that p-graph. For example, a row whose third, fourth and fifth components are respectively

⟦item:10, next:5⟧

⟦item:10, next:6⟧

⟦item:eq, next:⟦4, 8⟧⟧

and which otherwise is identical to the one above also represents it.

pc:

The value of the pc component is an integer which is to be

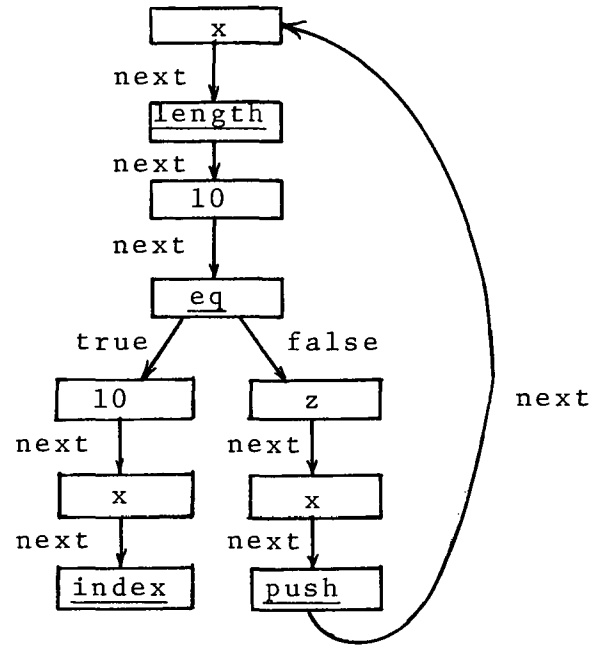


Figure 4.5
A p-graph fragment.

---

interpreted as the index of a node in the prog p-graph. The instruction

prog[pc].item

describes the next state transition to be taken by a process as part of its current activity. By convention, 1 is the index for the node for a p-graph to be interpreted first. Therefore, the value of pc is set to 1 by parts 4 and 8 of the state transition rule as described in Figure 3.1.

level, aflag:

The value of the level component is an integer j such that $1 \leq j \leq lmax$. The remaining status component, aflag, is a truthvalue.

stack:

The stack component is a stack designator. The prog-item spop can be used to explicitly pop items from the stack component. When it is interpreted a single item is popped from the stack component.

prog-id:

The prog-id component is a struct. It is either nil or it is a struct of the form

[[top:struct, rest:p]]

where p is itself a struct suitable for use as a prog-id component. The individual structs making up the prog-id component are called id-layers. Hence, the prog-id component is either nil or a list of id-layers. Two structs which can be used for the prog-id components of process states are shown in Figure 4.6.

When a process has a prog-id component that is nil none of the identifiers appearing in its prog component is bound.

Each id-layer describes a set of identifier-value bindings. The identifier x is bound in id-layer L if x is a selector for L; in such a case, the value to which x is bound is L.x. For the prog-id shown in Figure 4.5b, identifiers x, y and z are bound in the top layer, and identifiers a, x, and q, in the next id-layer. When a p-graph node that is an identifier is interpreted, the id-layers of the prog-id component are searched in order (i.e., first prog-id.top, then

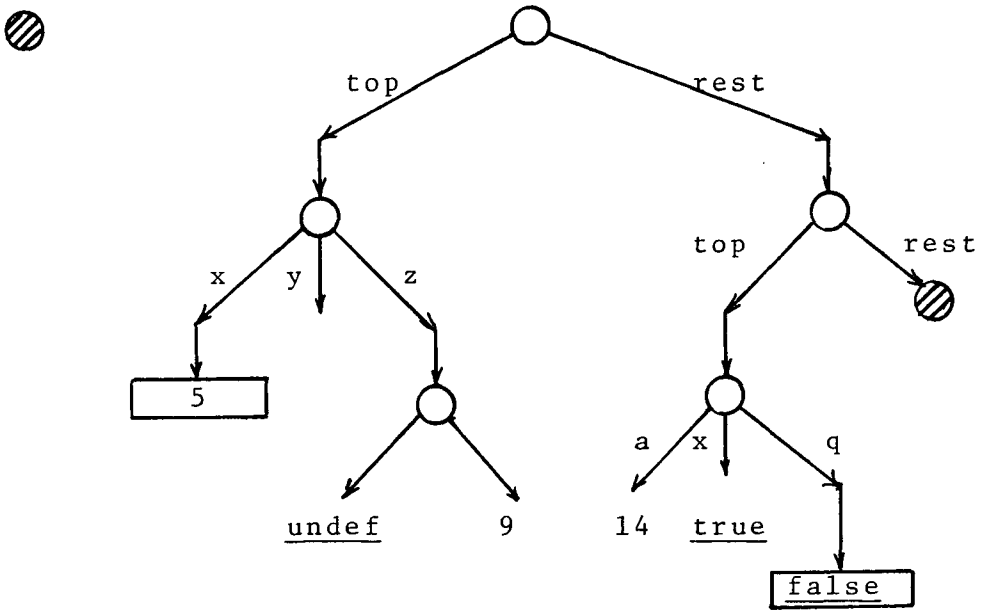


Figure 4.6

Structs suitable for use as prog-id state components.

---

prog-id.top.rest, etc.) until either the identifier is found to be bound in an id-layer, in which case the value to which it is bound is pushed onto the stack component, or the id-layers are exhausted, in which case undef is pushed onto the stack.

Identifiers can be bound to any member of Ω (including undef). The prog-items bind and unbind create and delete identifier-value bindings. The effect of

bind (id-exp, value)

is to bind id (the value of id-exp) to value in the top id-layer (prog-id.top) of the prog-id component. Figure 4.7 illustrates the effect interpretation of

bind (quote(w), 2)

has upon the prog-id component illustrated in Figure 4.6b.

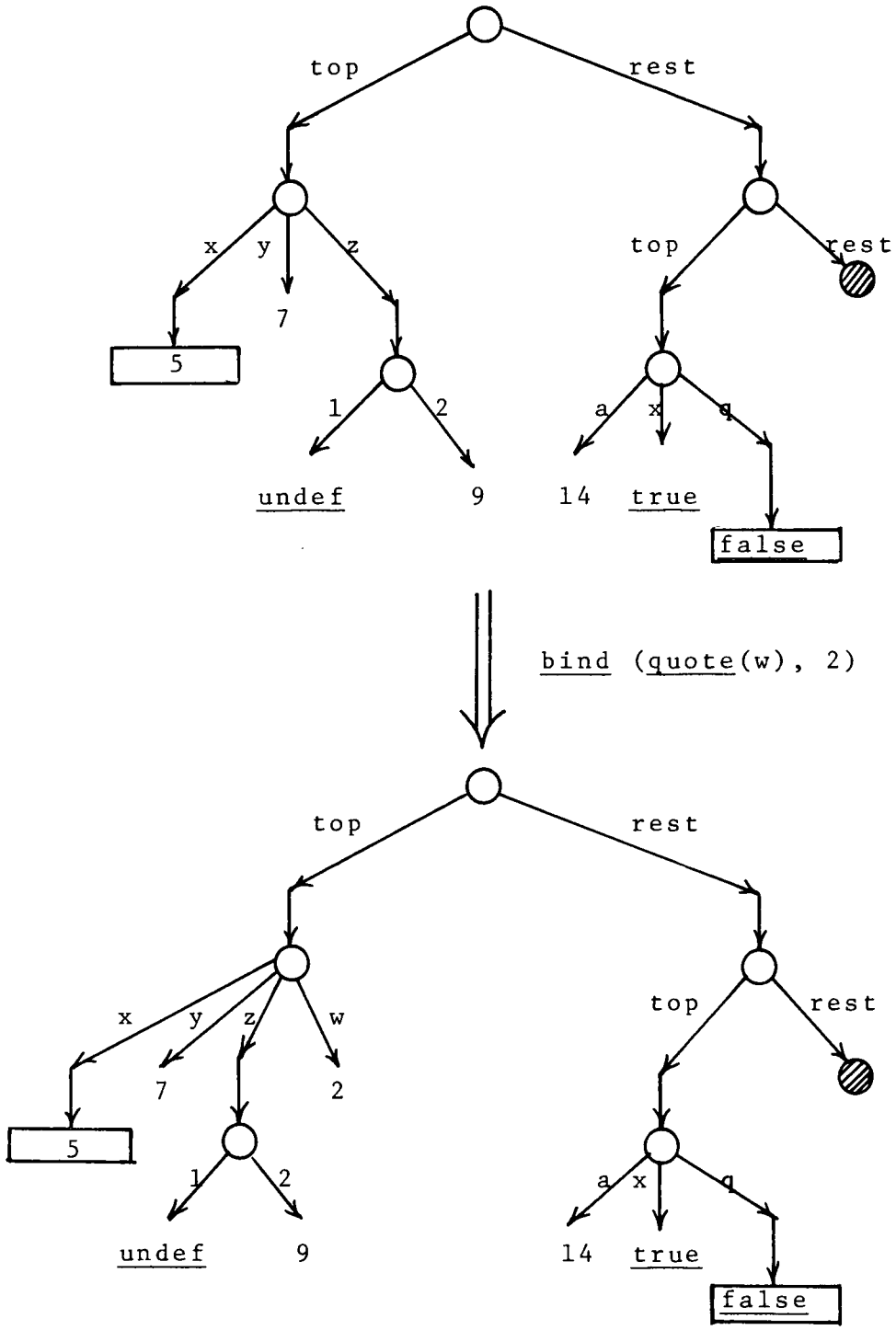

top
rest
x
y
z
top
rest
5
7
1
2
a
x
q
undef
9
14
true
false
bind (quote(w), 2)
top
rest
x
y
z
w
top
rest
5
7
2
1
2
a
x
q
undef
9
14
true
false

Figure 4.7

When

unbind (id-exp)

is interpreted the selector id is removed from the top id-layer of the prog-id component, thereby "unbinding" id in that id-layer. Identifier-value bindings can also be changed by directly setting the prog-id component (see Section 4.6). The value of the expression

binding (id-exp, p_id)

where p_id is the designator for a struct suitable for use as a prog-id and the value of id-exp is the identifier id, is the value to which id is bound in p_id.

proc-id:

The remaining environment component, proc-id, is a struct. The selectors of the struct define the identifiers that are bound by the proc-id.

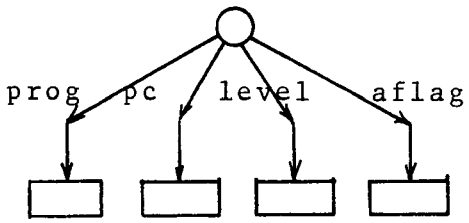
queues, sflag:

Each component of the queues component is a queue designator. The value of the sflag component is either nil or a process designator. When its value is a process designator, the sflag of a process indicates that the process is currently seized by the process designated. Therefore, the test specified by part 1 of Figure 3.1 could be accomplished by either

is-nil (sflag) or is-proc (sflag)

dump:

Each component of the dump component is the designator for a struct of the form



capable of holding the status components of an interrupted activity. Because the struct is composed of l-values, the response to an interrupt event can, when appropriate, change the status components of the interrupted activity before allowing it to continue.

4.6 Process Creation and State Components as Operands

The operation new-proc creates a new process. The value of

new-proc

is the process designator of a newly created process. When a process comes into existence its state components have the values:

prog = undef

pc = 1

level = lmax

aflag = false

stack = designator of a new stack

prog-id = nil

proc-id = nil

rp = undef

hp(1),...,hp(lmax) all = undef

q(1),...,q(lmax) each = designator of a new queue

dump(1),...,dump(lmax) each = struct of form

[[prog:new-cell(undef),
pc:new-cell(undef),
aflag:new-cell(undef),
level:new-cell(undef)]]

sflag = designator of creating process

Note that a process comes into existence seized by its creator. The creating process can use the component setting operations to initialize the states of processes it creates.

In the model, one process can not explicitly terminate (destroy) another. The only way a process can cease to exist is by self destruction (see Section 2.6). If its rp component is undefined when it completes interpretation of its prog component, a process terminates (see part 10 of Figure 3.1). To explicitly cause its termination a process can use the prog-item terminate.

Most state component setting operations require the process designator of the process whose components are to be set as an operand. That designator must be either that of the

process performing the operation or that of one it has seized. A process can use the prog-item proc to obtain its own designator.

State components can be set singly or in combination. Interpretation of

set-prog (pd, rd)

sets the prog component of the process designated by pd to the row designated by rd. The prog-items set-pc, set-level, set-aflag, set-stack, set-prog-id, set-proc-id and set-rp work in an analogous manner. The components of the queues, dump and handler programs components are set individually. The effect of

set-q (pd, n, qd)

is to change the queue which accepts interrupt requests of importance n for process pd to the queue designated by qd. When

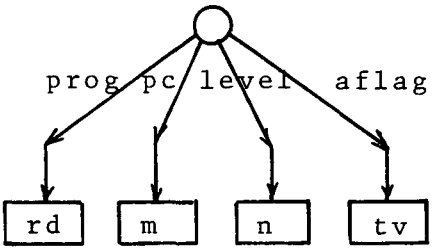
set-hp (pd, n, rd)

is interpreted the nth component of the hp component for the designated process is set to the row designated by rd. The set-dump operation works in an analogous manner.

The operations for setting combinations of state components are set-control, set-status and set-env which respectively set the control, status and environment component combinations (see Section 3.2). The effect of

set-status (pd, sd)

where sd is the designator for a struct of the form



rd = row designator
m, n = integers
tv = truthvalue

is to set the prog, pc, level and aflag components of the designated process to rd, m, n and tv, respectively. The prog-items set-control and set-env work in an analogous way. Note that

restore-dump (j)

is equivalent to

set-status (proc, dump(j))

A process can simultaneously set its own level and aflag components using the prog-item set-level-inactive. When a process performs

set-level-inactive (n)

its level component is set to n and its aflag to false. This operation is useful in situations which require a process to wait until it receives and handles a particular interrupt request expected to appear on a higher level (for examples see Sections 5.3.4 and 6.4). It enables the process to increase its level and set itself inactive to await the interrupt event. In such situations the sequence

set-level (proc, n);
set-aflag (proc, false)

is inadequate because the anticipated interrupt event might have occurred before the set-level operation or it might occur after the set-level but before the set-aflag. Both cases require the process to respond to the interrupt request before it can set its aflag. As a result, upon completion of the response, the process would set its aflag to await the request. By setting its aflag and level simultaneously, a process insures that it does not respond to the interrupt event before becoming inactive.

As is the case with setting components, a process can access both the values of its own components and those of other processes. The operations for accessing the values of state components are analogous to those for setting them with the exception that they produce values rather than having effects. To access components of another process, a process must first have the other process seized.

The value of

prog-of (pd)

is the row designator of the prog component for the process designated by pd. The prog-items pc-of, level-of, aflag-of, stack-of, proc-id-of, and rp-of work in an analogous manner. Components of hp, q, and dump are accessed individually. The value of

hp-of (pd, n)

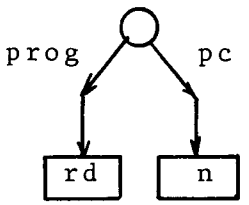
is the row designator of the nth component of the handler programs for the designated process. The prog-items dump-of

and q-of work in a similar way.

The value of

control-of (pd)

is the designator for a struct of the form



rd = row designator
n = integer

where rd and n are respectively the prog and pc components of the state of the designated process. The prog-items status-of and env-of work analogously. Because the structs produced by these operations contain l-values a process can use the operations to extract values for a collection of state components, some of which it can subsequently change by assignment, and then use the corresponding component setting operation to restore the modified collection.

An additional set of operations enables a process to access the values of its own state components without specifying its own process designator. When

prog

is interpreted it is as if

prog-of (proc)

were being interpreted. Similary the prog-items pc, level, stack, prog-id, proc-id, rp, hp, q, dump, control and env work in a manner analogous to their counterparts discussed above, with the exception that a process designator is not specified.

The process is understood to be the one performing the operation. These operations are included for convenience.

Note that the sflag component is treated differently from other state components with respect to component setting and accessing. There are no operations for it analogous to the set-~~~ and ~~~-of operations for other components. The operations t-seize and release are the only means for setting and extracting its value. This is consistent with the sflag component's function as a lock. A "set-sflag" operation would defeat its purpose. There is no need for a process to ever examine its own sflag for the fact that it can proceed implies that the value of its sflag is nil.

4.7 Isolation and Interaction in the Model

With the exception of the relation existing between a process and those processes it creates, the model insures that processes are normally isolated from one another. Consider the ways two processes could interact. (As Section 2.4 notes, all interactions between processes must occur through shared memory.) In the model there are just three:

1. by way of shared memory elements (cells, stacks, queues);
2. by way of the interrupt operation in which case the memory shared is the relevant queue of the q state component;
3. by way of component setting operations in which case

the state components set represent the shared memory; Each requires use of either the process designator of the other process or a memory designator known to the other process.

The only operations that generate process designators or memory designators are new-proc, new-cell, new-stack and new-queue. Processes can not arbitrarily create process or memory designators. In particular, a process can not generate a value which it can subsequently use as a process designator or as a memory designator. Because a process can not generate designators of existing processes or memory designators held by other processes, it can "have" such a designator only if it has been "given" it. In this way the model insures control of isolation of processes.

It is clear why two non-interacting processes, initially isolated from one another, can not interact unless "helped" externally. If they do not have the required memory or process designators there is no way they can obtain them on their own.

For interactions to be possible there must exist a mechanism for relaxing the isolation of processes from one another. In the model interactions are made possible by the new-proc operation. When a process creates another it obtains the process designator of the new process. Hence, it is possible for a process to interact with processes it creates in any of the ways listed above. Furthermore, because it can interact with them, it can arrange for them to interact with

each other by providing them with the required process or memory designators.

Consider, for example, how a process could arrange for two processes it has created to interact by way of interrupt events. To do so it must provide each with the other's process designator. In addition, it must establish conventions for the interactions. It can establish the necessary conventions by setting the handler programs component of each's state. These arrangements having been made, the prerequisites, noted in Section 2.4, are satisfied and the two processes are free to interact:

1. each process has been made aware of the other and has been provided with the other's designator;
2. the q components for the processes represent the shared memory;
3. the monitoring action of each state transition detects interactions when they occur; interpretations for the interactions are defined by the hp state components of the processes.

The ability to control interactions between processes is based on the fact that values, in particular process designators and memory designators, can not be arbitrarily generated by processes but rather can be created only in certain restricted ways. Unfortunately this permits only very coarse control to be exerted. Whether or not it is sufficient depends upon how hostile processes are to one another. In any

environment but a very friendly (and debugged) one it is likely to be insufficient. There is, for example, no way to control how a process uses the designator of another once it obtains it. It can interrupt the other process; it can seize it and change its state in any way it sees fit. The possibility of interaction degenerating to interference is very real. This represents a weakness in the model. It is a consequence of too great a relaxation of process isolation. Finer controls for process interactions are desirable.

The following hypothetical situation illustrates another aspect of this weakness:

> Process P is to create processes upon request from initial state specifications. In addition, P is to act in a supervisory manner toward the processes (slaves) it creates. That is, it is to have some control over them. For example, P should be able to cause a slave to terminate. The specifications for processes it is to create originate externally to P.

This situation is not an unrealistic one. Operating systems exhibit behavior similar to P's. An operating system receives requests originating externally to it to create processes and it is important that the system be able to control processes it creates.

A technique P might use to control its slaves is to reserve an important interrupt level, say level 1, for interactions with them. P could do this by setting the hp(1)

component of the state of each process it creates such that whenever a slave receives an interrupt request of importance 1 from P the slave responds as P wishes. Unfortunately, because processes are able to set their own state components, this technique will not always work. P has no control over the process specifications it receives. Therefore, P can not be sure that a slave will not set its level component to ignore P's interrupt requests or even redefine its hp(1) conponent to react as it wishes to P's requests.

As an alternative technique, whenever it wishes to exert control over a slave, P could seize the slave and force it to behave in a particular way by setting its state components. This technique is unsatisfactory for at least two reasons:

1. It is just barely workable. Should a slave obtain P's designator it could seize P. The result could be catastrophic.
2. While it solves this problem, it is not applicable to the solution of the more general problem of which this is a specific instance. Suppose, for example, that slaves create other processes. P could not use this technique to control descendents of its slaves because there is no way to force its slaves to reveal designators of processes they create.

The general problem is that of controlling the abilities of processes. As the model currently exists all process are equally capable. There is no way to restrict the capabilities

of a particular process. If there were, the first technique proposed could provide a workable solution to the specific problem. P could restrict the way slaves set their level component and could prevent them from tampering with their hp(1) components. There is no way P can do this as the model presently exists, short of responding to each request to create a process by creating, instead, an "interpreter process" which interprets the specified prog component.

No attempt is made in the present chapter to correct the weakness noted above. Rather, further discussion of it is deferred until Chapter 7. Chapters 5 and 6 present a series of examples in which the model is used to describe some non-trivial process behavior patterns. My reason for organizing the dissertation in this way is to give the reader a chance to familiarize himself with the model by seeing it used before presenting the additional features which permit process capabilities to be controlled. The reader may, if he wishes, read Chapter 7 before Chapters 5 and 6 with little loss in continuity.

4.8 The Model in Perspective

The important aspects of the model for process representation which have been presented so far are summarized in this section. The approach taken is to reconsider the questions posed in Section 2.10, indicating for each how it is addressed in the model. To aid the reader the questions are

repeated and relevant section numbers noted.

I1. How are the operations a process performs represented?
The operations are represented by the prog component of the process state. The prog component is a structured collection of prog-items, identifiers and other members of the universe of discourse which, taken together, define the actions comprising the current activity of a process (Sections 3.2, 3.5, 4.5).

I2. How does a process keep track of which operation to do next?
The pc component of the process state indicates the operation of the prog component to be interpreted next (Sections 3.2, 3.3, 3.5, 4.5).

I3. How does a process store intermediate results until they can be used?
Temporary storage is accomplished by the stack component of its state (Sections 3.2, 3.5).

I4. How does a process keep track of identifier-value bindings?
Identifier-value bindings are recorded in the prog-id component of its state. A process has the ability to add and delete bindings as it wishes.

I5. What is the universe of discourse?
Ω includes integers, truthvalues, process designators,

memory designators, structure designators, identifiers and undef (Section 4.3).

I6. What operations can a process perform?
Sections 3.4, 4.3, 4.4, 4.5 and 4.6 describe the operation repertoire. Appendix 1 is a summary of all prog-items.

E1. How can a process achieve shared memory?
The q component of a process state is directly accessible to the process and indirectly accessible to others by way of the interrupt operation (Sections 3.2, 3.4, 4.6, 4.7). Furthermore, processes can share cells, queues, and stacks; Section 4.7 discusses how this can be accomplished.

E2. How can a process indicate the relative importance of its current activity?
The value of the level component of a process state indicates the relative importance the process places on its current activity (Sections 3.2, 3.3, 3.4).

E3. How does a process know when to interrupt its current activity?
Assuming that the value of the level component of its state is lev, a process is to interrupt its current activity whenever an item appears in q(j) of its q component for j<lev (Sections 3.3, 3.4).

E4. How is monitoring for interrupt events accomplished?
Part of each state transition checks the q component of the process state for the presence of interrupt requests (section 3.3).

E5. How are the relative urgencies of interrupt events represented?
The importance of an interrupt request relative to other interrupt requests and the current activity of a process is defined by the queue it appears in (Section 3.4).

E6. How can a process find information about a particular interrupt event so that it may properly respond to it?
Associated with each interrupt request are two pieces of information: its relative importance, which is defined by the queue it appears in, and its value, which is the actual value that is in the queue and constitutes the request. A process can access values in the queues of its q component (Section 3.4, 4.6).

E7. Can a process accept more than a single request to interrupt its current activity?
Yes, it can accept as many requests as its queues can hold (Section 3.4, 4.2).

E8. How does a process know the proper response to a particular interrupt event?
The handler programs component of its state defines the responses for all interrupt requests (Sections 3.2, 3.3,

3.4).

E9. How can a process remember an interrupted activity so that it may resume it after responding to an interrupt event? The interrupted activity is automatically remembered as part of the state transition initiating the interrupt response. It is saved in the dump component of the process state (Sections 3.2, 3.3, 3.4).

The definition of the model is completed in Chapter 7 which considers the problem of controlling the capabilities of particular processes. In that chapter the model as described in Chapters 3 and 4 is extended to enable control to be exerted over certain external aspects of process behavior.

CHAPTER 5

A Programming Notation For Using the Model

5.1 Introduction

This chapter has two parts. The first part presents, in an informal way, a programming language for describing p-graphs, parts of which have already been introduced in Chapters 3 and 4. The second part is a collection of examples intended to illustrate both the language and some features of the model.

5.2 PGL - A Language for Describing P-graphs

The relatively simple and intuitively appealing explanation of the model state transition rule (see Figures 3.1 and 3.3 which are reproduced in Appendix 2) is due, in part, to the austere structure of p-graphs. The use of p-graphs as a programming notation has two serious flaws:

1. most people find postfix notation unnatural; and
2. each program in p-graph form represents the solution of a graphical layout problem which for large programs can be quite formidable.

Consequently, it is very tedious to express any but the most trivial programs directly in terms of p-graphs. From the point of view of programming the row representation for p-graphs is,

if anything, worse than p-graphs themselves. This section presents a simple language, called PGL (for p-graph language), for describing p-graphs.

The purpose of a descriptive language is to suppress aspects of descriptions which are constant to permit attention to be focused on aspects of a particular description which are variable. The design of such a language is concerned largely with deciding which aspects are likely to be constant and should therefore be suppressed and which are likely to be variable and, therefore, emphasized. The goals of a language provide the basis for making such decisions.

PGL has a single, simple goal. It is to serve as a vehicle for demonstrating the descriptive power of the model. It should, therefore, be simple and unsophisticated and should contain no more features than "necessary" to make it easy to describe p-graphs. Two important considerations have influenced the design of PGL:

1. all features of the model must be accessible from PGL;
2. the correspondence between PGL constructs and the p-graph fragments they represent should be obvious.

While PGL itself is relatively unsophisticated, it can be used to describe quite sophisticated patterns of process behavior (see Chapter 6).

As programming languages go, there is little in PGL that is unusual. For the most part, the meanings of various

constructs are obvious from their syntax. PGL has the following capabilities:

1. It is unnecessary to use postfix notation. Nested expressions are permitted.
2. It is unnecessary to draw "next" arcs. Sequencing can be described in several ways.
3. There are a variety of ways to express conditional and iterative execution.
4. There is a declarative or automatic binding facility which makes it unnecessary to explicitly use the bind operation.
5. Infix notation is permitted in a number of situations.
6. There is a comment convention.
7. There is a macro facility.

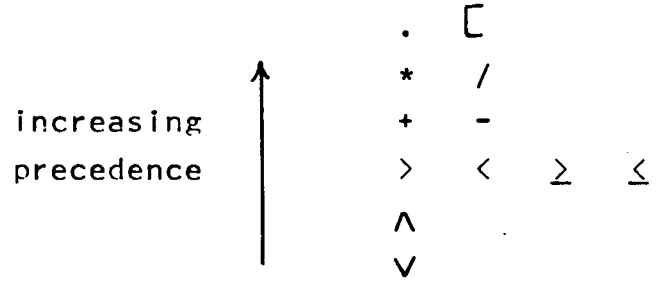
The presentation of PGL is relatively informal. Each feature is explained by exhibiting a correspondence either between it and a fragment of p-graph or between it and other features previously explained.

Figure 5.1 defines the syntax for PGL. The grammar presented for it in Figure 5.1a is ambiguous. To disambiguate it the precedence relations shown in Figure 5.1b and the convention that binary infix operators (OP's) associate to the right are adopted.

The remainder of this section discusses, in turn, each of the features of PGL.

```
P   ::=   let ID = E { ; ID = E }* in B  |  B

B   ::=   S { ; S }*

S   ::=   iff E do E  |  until E do E  |  if E then E else E
        |  while E do E  |  unless E do E  |  for ID = E to E do E
        |  ID => S  |  nextis ID  |  E := E  |  E

E   ::=   E [ E ]  |  E OP E  |  - E  |  E . ID  |  prog-item
        |  prog-item ( E { , E }* )  |  [[ ID : E { , ID : E }* ]]
        |  [[ E { , E }* ]]  |  §  P  §  |  ( P )  |  null
        |  ref ( ID )  |  LIT

LIT ::=   undef  |  nil  |  true  |  false  |  INT  |  ID

OP  ::=   +  |  -  |  *  |  /  |  <  |  >  |  ≤  |  ≥  |  ∧  |  ∨
```

(a)



(b)

Figure 5.1

a. Grammar for PGL. A PGL program is regarded as a continuous stream of characters rather than a sequence of lines, in that, with two exceptions, the transition from one line to the next has no significance in the language. The two exceptions are: the newline character is treated as a space; and, newline terminates comments (see Section 5.2.6). ID denotes an arbitrary identifier and INT an arbitrary integer. $\{\nu\}^*$ represents 0 or more repetitions of the string ν.

b. Precedence relations which, with the rule that OP's are right associative, disambiguate the grammar in (a) for PGL.

5.2.1 Nesting

Application of prog-item pi to operands r1, ..., rn is denoted by the expression

pi (r1, ..., rn) (5.1)

where the ri may themselves be such expressions. The p-graph fragment described by (5.1) is its postfix representation and is displayed in Figure 5.2a. When pi is interpreted the prog, pc and stack components are as shown in Figure 5.2b; the top item of the stack is the value of r1, the second item the value of r2, ... etc.

The single exception to the above rule occurs when pi is quote. For quote (x) to have the desired effect (see Sections 3.5 and 4.3), quote must be interpreted before x. The p-graph fragment corresponding to quote (x) is shown in Figure 5.2c.

5.2.2 Sequencing

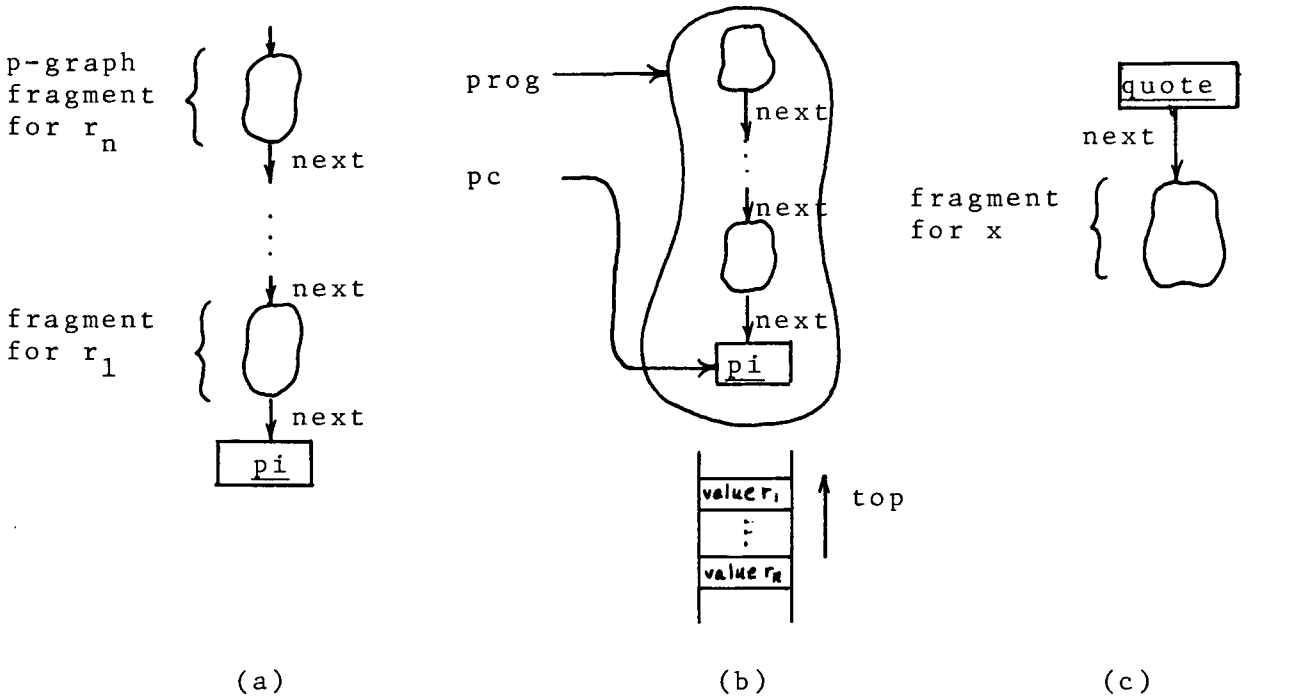
Semicolons (;) are used in place of "next" arcs to specify sequencing. The p-graph fragment corresponding to the expression

a ; b (5.2)

is displayed in Figure 5.3a. Although postfix notation need not be used, PGL permits its use. Hence, the expression

rn ; ... ; r1 ; pi

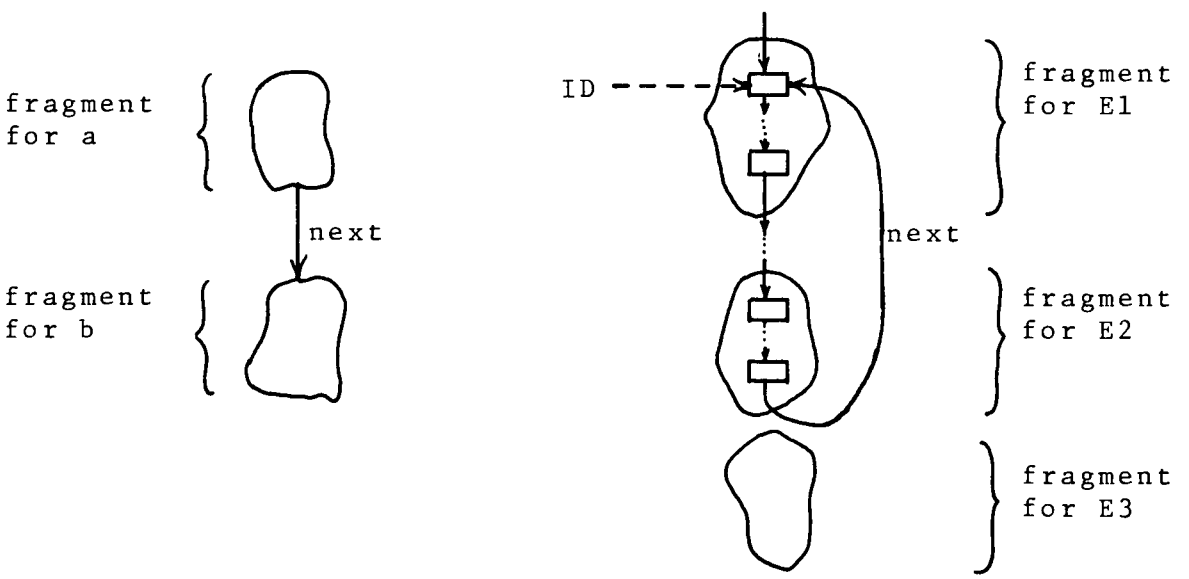
is equivalent to (5.1).



(a) (b) (c)

Figure 5.2

a. P-graph fragment described by (5.1).
b. The prog, pc and stack when pi is interpreted.
c. P-graph fragment for quote (x).



(a) (b)

Figure 5.3

a. P-graph fragment described by (5.2).
b. P-graph fragment described by (5.3).

In PGL sequencing can be specified by explicitly naming successor expressions. The construct

ID => E

associates the name ID with the expression E; and

F ; nextis ID

specifies that the successor to expression F is the one named ID. Figure 5.3b illustrates the p-graph fragment described by

```
          .
          .
          .
ID => E1;                                    (5.3)
          .
          .
          .
          E2;
          nextis ID;
          E3
```

To translate the above PGL program into the corresponding p-graph an association is made between ID and the "entry" node of the p-graph fragment which corresponds to E1. That is, the integer which is the index of the relevant p-graph node (see Section 4.5) is associated with ID; stated somewhat differently, the "value" of a PGL "label" is an integer. Of course, the association between ID and the node index is not part of the p-graph that results from the translation. Note that the semicolons that surround the nextis construct do not represent "next" arcs.

The PGL operator null is a placeholding operator intended for use with the expression-naming and nextis constructs. Figure 5.4 illustrates the p-graph fragment corresponding to

```
        .
        .
        .
        E1;
ID =>   null;                                   (5.4)
        E2;
        .
        .
        .
        E3;
        nextis ID;
        E4
```

Note that null is not a prog-item but rather is a PGL operator.

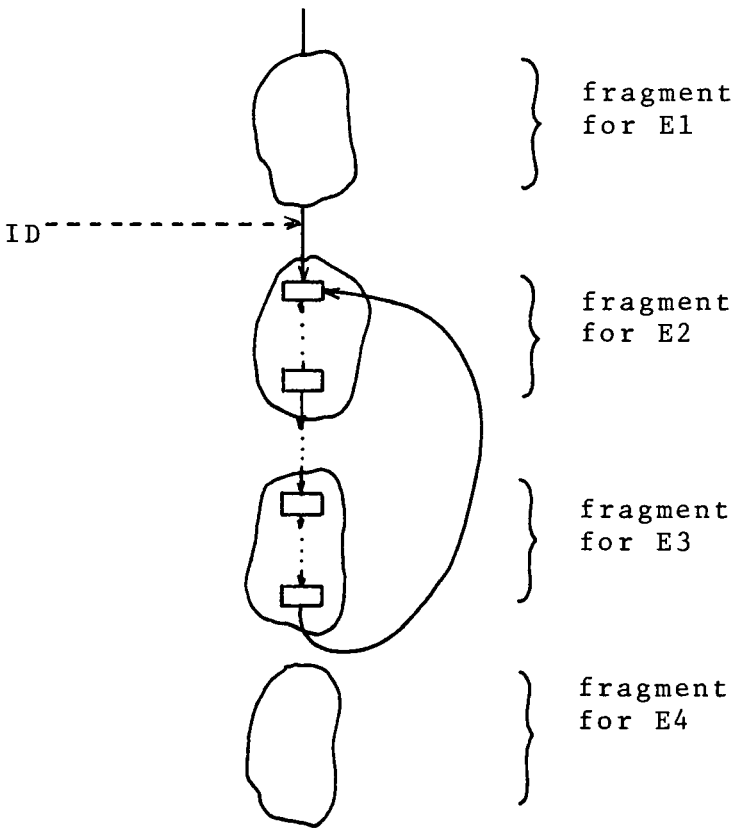


Figure 5.4
P-graph fragment described by (5.4).

5.2.3 Conditionals and Iteration

PGL includes a variety of ways for expressing conditional and iterative execution.

The "one-armed" conditional expression

iff a do b ; c (5.5)

specifies that a is to be interpreted first. If its value is true, b is to be interpreted next, followed by interpretation of c; otherwise, c is to be interpreted immediately after a. The effect of

unless a do b ; c (5.6)

is identical to that of

iff not(a) do b; c

Figure 5.5 illustrates the p-graph fragments corresponding to (5.5) and (5.6). The p-graph fragment described by the "two-armed" conditional expression

if a then b else c ; d (5.7)

is shown in Figure 5.6.

The iterative expression

while a do b ; c (5.8)

specifies that b is to be repeatedly interpreted as long as the value of a is true. Its effect is equivalent to that of

```
L => iff a do
       ( b ; nextis L );
     c
```

where L is an identifier not found elsewhere in the program. Each iteration begins with interpretation of a. If its value

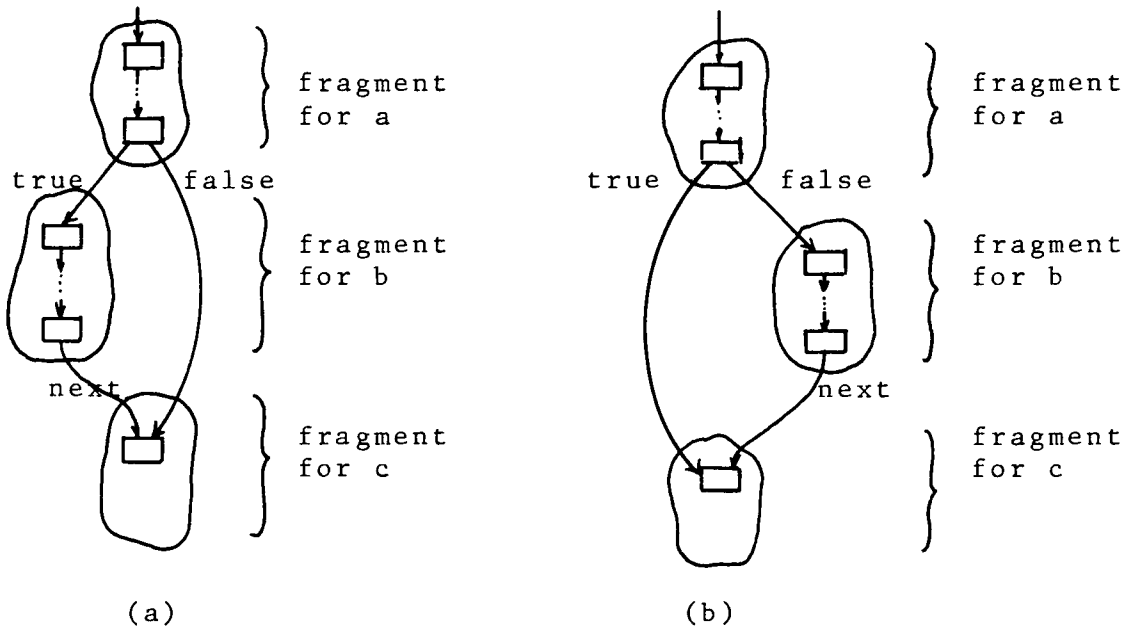


Figure 5.5

a. P-graph fragment corresponding to (5.5).
b. P-graph fragment corresponding to (5.6).

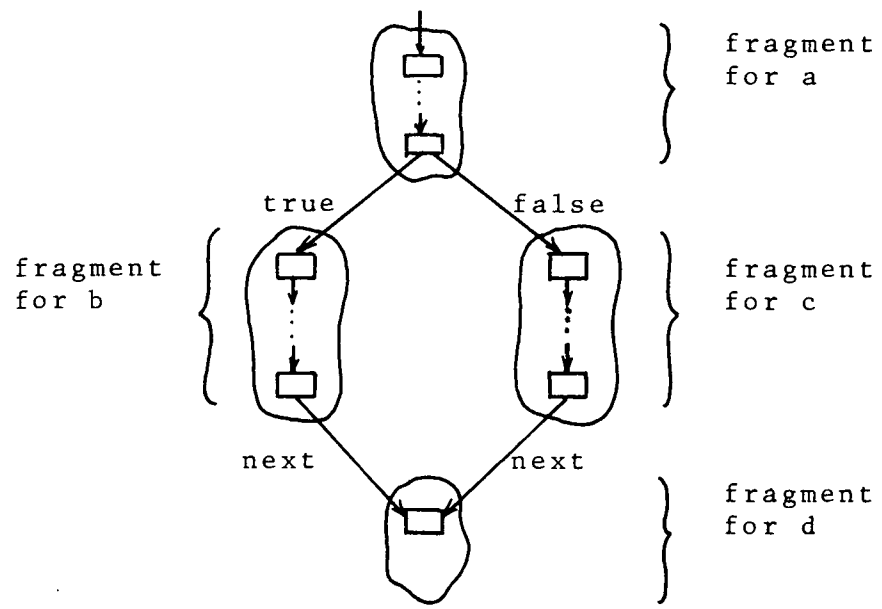


Figure 5.6

P-graph corresponding to (5.7)

is true, b is next interpreted, followed by another iteration; otherwise, the iterations are ended and c is next interpreted. Thus, if a is initially false, b is not interpreted at all. The effect of

until a do b ; c (5.9)

is equivalent to that of

while not(a) do b ; c

The p-graph fragments corresponding to (5.8) and (5.9) are displayed in Figure 5.7.

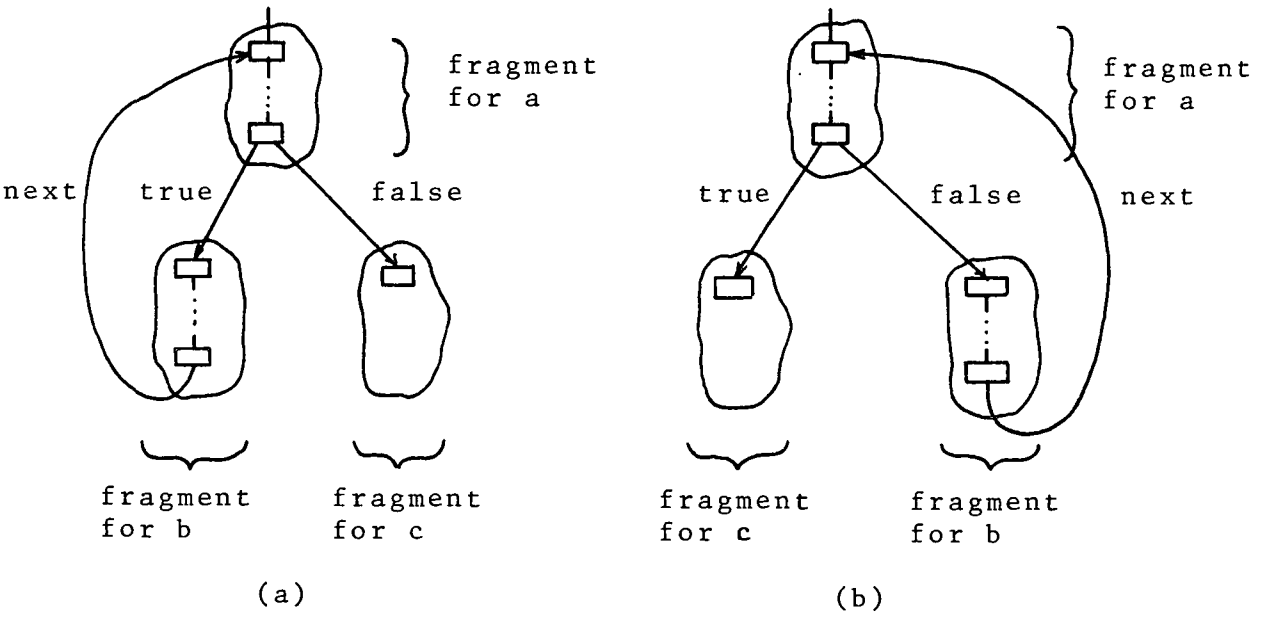


Figure 5.4

a. P-graph corresponding to (5.8).

b. P-graph corresponding to (5.9).

The iterative expression

for i = a to b do c ; d

is equivalent to

```
store (i, a);
until gr(rval(i), b) do
    ( c' ;
      store (i, rval(i) + 1) );
d
```

where c' is obtained from c by replacing each occurrence of i by rval (i). Note that with this rule for generating c' it is not possible to directly update i within c because

store (i, value)

in c would appear in c' as

store (rval(i), value)

This treatment of the iteration variable is based on the feeling that within c most situations call for its r-value. Therefore, PGL suppresses this constant aspect, by allowing omission of the rval operator. There are, however, situations in which the l-value of i and not its r-value is desired within c. To accommodate such situations, PGL includes the operator ref which can be used to prevent insertion of the rval operator when c' is generated. When ref(i) appears within the body of a for construct (i.e., within c), it is replaced by i when c' is generated. Appearances of the ref operator which are not within the body of a for construct have no effect on the p-graph being described. Note that ref, like null, is not a prog-item but rather is a PGL operator.

5.2.4 Declarations

PGL provides means to bind identifiers (i.e., declare variables) without explicitly referring to the prog-id

component. The expression

```
let x1 = v1 ; x2 = v2 ; ... ; xn = vn
in
A
```

where the xi are identifiers, is equivalent to

```
set-prog-id (proc, [top:nil,
                    rest:prog-id]) ;
bind (quote(x1), v1) ;
bind (quote(x2), v2) ;
     .
     .
     .
bind (quote(xn), vn) ;
A ;
set-prog-id (proc, prog-id.rest)
```

Upon "entry" to a let construct a new id-layer, which is to bind the "declared" identifiers, is "pushed onto" the existing prog-id component. On "exit" from it, that id-layer is "popped from" the prog-id component. The let construct provides block structure scoping rules for identifiers. Note, however, that PGL does not require that the model be used in a block structured way. The prog-items bind, unbind and set-prog-id can be used in whatever way one see fit.

A caveat concerning the let construct is in order. Entry to and exit from expression A can occur as the result of operations which set the prog or pc components (e.g., set-prog, set-pc). Whenever such an "abnormal" entry to or exit from A occurs, the prog-id component is not changed. The block structure effect is achieved only when entry to and exit from the let construct is "normal": that is, "through" the set-prog-id operations which bracket the p-graph corresponding to A.

5.2.5 Infix Notation

PGL permits departure from the prefix notation described in Section 5.2.1 in a number of situations, several of which have previously been noted in Chapters 3 and 4. Infix equivalents are provided for the following operators.

1. store

   a := b is equivalent to store (a, b).

2. select

   if id is an identifier, a.id is equivalent to

   select (a, quote(id)).

3. index

   a[n] is equivalent to index (a, n).

4. plus, minus, times, div, gr, ls, le, ge, and, or

infix representation	prefix equivalent
a+b	plus(a, b)
a-b	minus(a, b)
a*b	times(a, b)
a/b	div(a, b)
a>b	gr(a, b)

infix representation	prefix equivalent
a<b	ls(a, b)
a≥b	ge(a, b)
a≤b	le(a, b)
a∧b	and(a, b)
a∨b	or(a, b)

5. row

   ⟦v1, ..., vn⟧is equivalent to

   row (n, v1, ..., vn)

6. struct

   if all the idi are identifiers, ⟦id1:v1, ..., idn:vn⟧

   is equivalent to

   struct (n, quote(id1),v1, ..., quote(idn),vn)

Furthermore, the expression

§ E §

is used in PGL to denote the row that represents the p-graph corresponding to expression E. Thus, for example the row corresponding to the p-graph shown in Figure 4.5 on page 103 can be denoted in PGL by

```
§ until eq (10, length(x)) do
    ( push (x, z) ) ;
  x [10]  §
```

5.2.6 Comments

Comments can appear in PGL. All characters between a double slash mark (//) and the end of a line are regarded as comment and have no effect on the p-graph being described.

5.2.7 Macros

There is a macro facility associated with PGL. A macro definition is made by associating with a macro name a specific string of PGL text. Whenever a macro call appears in a p-graph description, the macro named is replaced by the string of text which is its definition. Macros can have parameters, in which case the string replacement includes substitution of the actual parameters appearing in the macro call for the formal parameters appearing in the macro definition.

The syntax for macro definitions in PGL is:

MACRO_DEF ::= MACRO : name P ENDMACRO
| MACRO : name (ID { , ID }*) P ENDMACRO

MACRO and ENDMACRO are terminal symbols which serve as delimiters for macro definitions. The macro being defined is name and the string of PGL text to be associated with it as its definition is P (see Figure 5.1). The first alternative is used to define macros with no formal parameters and the second, to define macros with one or more parameters.

Macro calls in PGL are of the form

MACRO_CALL ::= name | name (E { , E }*)

where the two alternatives represent, respectively, a call for a macro with no parameters and one for a macro with parameters. In PGL macro calls belong to syntactic class E (see Figure 5.1).

The following macro definition defines increment, a macro which increments its first parameter by its second parameter:

```
MACRO: increment (x, y)
     x := rval (x) + y
ENDMACRO
```

When the macro call

increment (ALPHA, 7)

is encountered, it is replaced by

ALPHA := rval (ALPHA) + 7

Macros are used extensively in the examples which follow. In all cases, they are used in such a straightforward way that

any "reasonable" interpretation of macro expansion should not lead to confusion as to what is intended. For this reason, I have chosen not to belabor the reader with further explanation of how macro expansion works in PGL. (A programmer's manual for a PGL implementation would, of course, detail rules for macro expansion.)

5.3 Using PGL: Examples

This section contains four examples which illustrate the use of PGL. The examples are:

1. making a copy of the process stack component;
2. a locking mechanism;
3. a LISP like Eval operation; and
4. copying arbitrary members of Ω.

The examples each make use of the macro facility described in Section 5.2.7.

5.3.1 Making a Copy of the Stack Component

There are two parts to this example. The first part shows how a process can make a copy of a stack other than its own stack state component. A macro stack-copy, which has a single parameter, is defined. The value of

stack-copy (s)

is to be the designator of a new stack containing the same items as the stack designated by s.

The second part attacks the trickier problem of how a process can copy its own stack component. A macro stack-comp-copy, which has no parameters, is defined. The value of

stack-comp-copy

is to be the designator of a new stack which holds the same items currently held by the process stack component. That is, after the p-graph corresponding to stack-comp-copy is interpreted, the top item of the process stack is to be the designator of a stack containing the same items as the rest of the process stack component.

1. stack-copy

The strategy for stack-copy is relatively simple. A new stack, C, is allocated and the items contained in the stack s are pushed onto it. The definition for stack-copy is

```
MACRO: stack-copy (s)
     let S = s;
         C = new-stack;
         i = new-cell (length(S))
     in
     until eq(rval(i), 0) do
        ( push (C, S[rval(i)]) ;
          i := rval(i) - 1  ) ;
     C
ENDMACRO
```

Comments:

1. When interpretation of stack-copy is completed, the top item of the stack component is the stack designator to which C, the copy, was bound.
2. The actual parameter of stack-copy may be an arbitrarily complex expression. Using S within the

body of the macro rather than s insures that s is evaluated only once. Because macro expansion involves nothing more than string substitution, if S were not bound to s, s would be evaluated for each appearance of S in the definition of stack-copy.

2. stack-comp-copy

To see why it is tricky business for a process to copy its own stack component consider the effect of stack-copy (stack), assuming that the length of the process stack is n. First, consider what happens when new-cell (length(S)) is interpreted. After S is interpreted, the process stack contains n+1 items. Thus, the new cell is initialized to n+1. Next, consider what happens with the first interpretation of S[rval(i)]:

rval(i): causes n+1 to be pushed onto the stack increasing its length to n+2;

S: causes its own designator to be pushed onto it, increasing its length to n+3.

index: accesses the (n+1)st item in the stack component, which, clearly, is not its "bottom" item.

The macro stack-comp-copy avoids these difficulties by

a. saving the designator of the stack component,
b. temporarily setting the stack component to a new stack,
c. making a copy of the "saved" stack using stack-copy,
d. pushing that copy onto the "saved" stack, and
e. restoring the "saved" stack.

The definition for stack-comp-copy is

```
MACRO: stack-comp-copy
     let S = stack                    //Save designator
     in                               //of stack component.
     set-stack (proc, new-stack) ;        //Switch stacks.
     push (S, stack-copy (S)) ;           //Make copy.
     set-stack (proc, S)                  //Restore original
ENDMACRO                                  //stack.
```

Comment:

Each use of stack-comp-copy requires the allocation of a new stack, to be used temporarily while the copy is being made. This potentially wasteful stack allocation can be avoided by using the same stack each time. This could be accomplished by using the proc-id component to bind an identifier, say T-STACK, to a stack designator and replacing the first set-stack operation in stack-comp-copy by

set-stack (proc, proc-id.T-STACK)

5.3.2 A Locking Mechanism

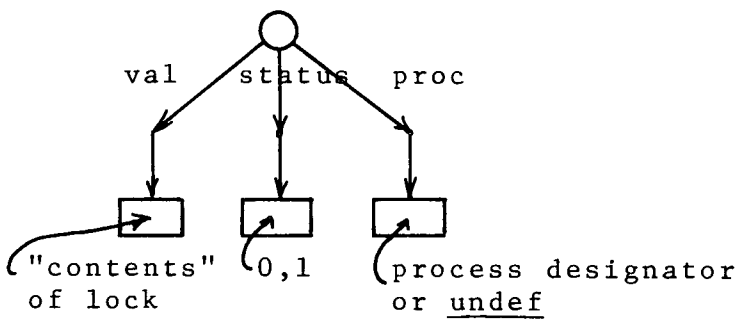
This example describes a simple locking mechanism which a process can use to insure that it has sole access to a particular memory element. In effect, the locking mechanism provides a new kind of memory element called a lock. Before a process can read or write the "contents" of a lock, it must first lock it. Five macros provide the locking mechanism:

1. new-lock is the lock allocation operator. The value of new-lock(v) is to be the designator of a new lock whose "contents" are initialized to v.
2. t-lock is a predicate with a side effect. When t-lock(d) is interpreted an attempt is made to lock the

designated lock. If the lock is not currently locked, the attempt succeeds and the value of the predicate is true; otherwise, it fails and the value is false.

3. unlock unlocks a designated lock, provided the process performing unlock currently has the lock locked.
4. lock-val is the retrieval operator for locks. It produces the "contents" of a designated lock, provided the process performing it currently has the lock locked.
5. lock-store is the storage operation for locks. It changes the "contents" of a designated lock, provided the lock is currently locked by the process performing it.

A lock is represented by struct of the form



The status component indicates whether the lock is locked and the proc component, the process, if any, which currently has the lock locked. The t-lock macro performs a t-set operation on the status component. If the t-set succeeds, the proc component is set to the designator of the performing process. The unlock, lock-val and lock-store macros each check the proc

component of the designated lock to see whether it is currently locked by the performing process. There is no way to prevent processes from directly accessing the components of a lock. Therefore, to insure that the locking mechanism works properly, processes must agree to access locks only by way of the five lock macros.

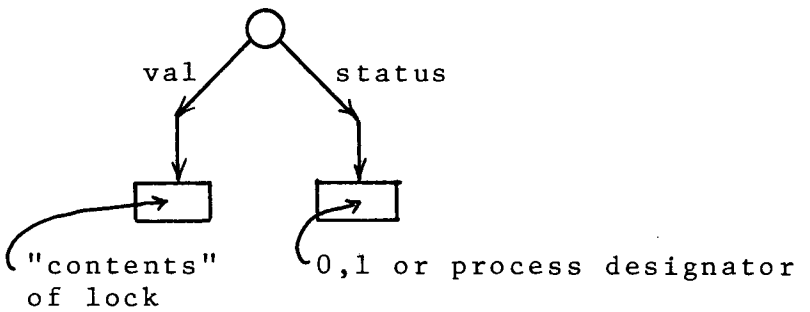
The definitions for the lock macros are:

```
MACRO: new-lock (v)
     [[ val:new-cell(v),
        status:new-cell(0),
        proc:new-cell(undef) ]]
ENDMACRO


MACRO: t-lock (d)
     let L = d
     in
     if  t-set (L.status)
     then  (L.proc := proc ; true)
     else  false
ENDMACRO


MACRO: unlock (d)
     let L = d
     in
     if  eq (rval(L.proc), proc)
     then  (L.proc := undef ; L.status := 0)
     else  ERROR1
ENDMACRO


MACRO: lock-val (d)
     let L = d
     in
     if  eq (rval(L.proc), proc)
     then  rval (L.val)
     else  ERROR2
ENDMACRO
```

```
MACRO: lock-store (d, v)
     let L = d
     in
     if  eq (rval(L.proc), proc)
     then  L.val := v
     else  ERROR3
END'MACRO
```

Comments:

1. ERROR1, ERROR2 and ERROR3 are left unspecified. For a particular application appropriate error handling actions could be specified.
2. A lock could be represented more efficiently by a struct of the form



Such a change in representation would, of course, require that the macros be redefined appropriately.

5.3.3 A LISP-like Eval Operation

For this example a macro eval, similar in effect to the eval function of LISP [Mc62], is defined. The value of

eval (x, p)

where x is a p-graph and p is a prog-id (i.e., x is a row representing a p-graph and p, a struct suitable for use as the prog-id component for a process state), is to be the result of

interpreting x in an environment with prog-id p. The eval macro is used in Sections 5.3.4, 6.3 and 6.7.

The definition for eval makes use of the rp and proc-id state components. The strategy used to define it is the following:

1. When eval(x,p) is performed values for the prog, pc and prog-id components, for use after x has been evaluated, are saved. Next, the prog, pc, and prog-id components are set to initiate evaluation of x with respect to prog-id p.

2. "Return" upon completion of the evaluation of x is to be accomplished by the rp component. (Recall that rp is interpreted when interpretation of the prog component is completed; see Figure 3.1.) When interpretation of x is completed and rp is initiated, the top item in the process stack is the value of x. The rp component performs the return by setting the prog, pc and prog-id components to the values saved by eval.

3. The identifier EVAL_STACK is bound in the proc-id component. It is bound to the designator for a stack which holds values, saved by eval, for the prog, pc and prog-id components.

4. The auxilary macros make-control and top-from are used in the definition of eval. Interpretation of

make-control (x, n)

where x is a p-graph and n is a node of x (i.e., x is a row and n is an integer), produces a value which when used as the operand for set-control results in interpretation of p-graph x beginning at node n. Since the value of a label in PGL is an integer (see Section 5.2.2), n may be a PGL label. The effect of

top-from (S)

is to pop the top item from the stack designated S, producing it as its value.

The macro definitions are:

```
MACRO: top-from (S)
  S[1];
  pop (S)
ENDMACRO

MACRO: make-control (x, n)
   [[prog:new-cell(x), pc:new-cell(n)]]
ENDMACRO

MACRO: eval (x,p)
   push (proc-id.EVAL_STACK,                  //Save prog and pc
         make-control(prog, EXIT));           //for return.
   push (proc-id.EVAL_STACK,                  //Save prog-id for
         prog-id);                            //return.
   set-prog-id (proc, p);
   set-control (proc, make-control(x, 1));
   EXIT ⇒ null                                //Define EXIT.
ENDMACRO
```

To use eval a process must have a proc-id component of the form:

```
[[      .
        .
        .
   EVAL_STACK:new-stack
        .
        .
        .       ]]
```

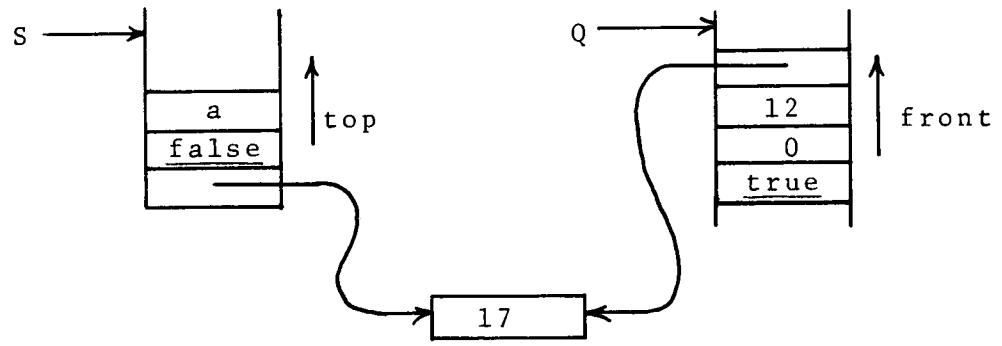
The rp component which accomplishes the return is

```
§ iff eq(0, length(proc-id.EVAL_STACK))//If no place to
    do terminate;                       //return to, terminate.
  set-prog-id (proc,                    //Restore prog-id.
               top-from(proc-id.EVAL_STACK));
  set-control (proc,                    //Restore prog and pc.
               top-from(proc-id.EVAL_STACK)) §
```

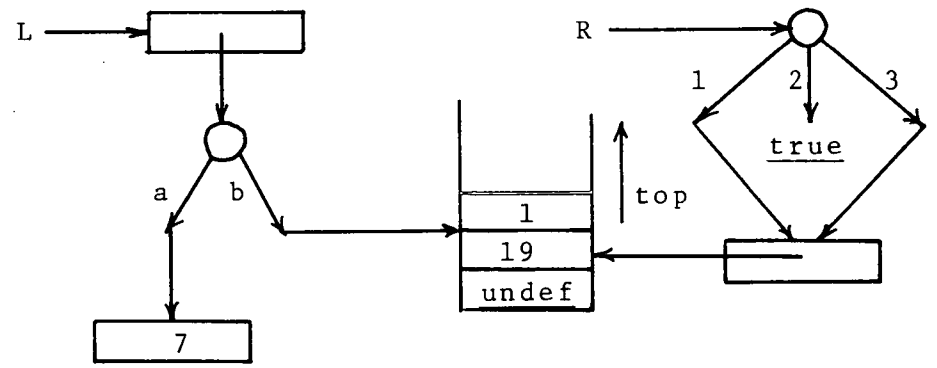
5.3.4 Copying Arbitrary Members of Ω

This example shows a way to "copy" an arbitrary member of the universe of discourse. The macro copy, to be defined, is used in Sections 6.4 and 6.5.

The notion of sharing is useful in discussing copying. Sharing is said to occur between two items which have one or more memory designators in common. (The statement "A and B have memory designator C in common" means that designator C can be obtained from A by performing an appropriate sequence of operations, and can also be obtained from B by performing an appropriate, but possibly different, sequence of operations.) Some examples of sharing are shown in Figure 5.8. Sharing occurs between the stack and the queue of Figure 5.8a; S[3] and Q[1] share the same cell designator. Similarly, the cell designated L and the row designated R in Figure 5.8b exhibit sharing; rval(L).b and rval(R[1]) share the same stack designator. Note that the first and third components of R also share.



(a)



(b)

Figure 5.8

Examples of Sharing

a. S and Q share; S[3] and Q[1] are the same l-value.

b. L and R share; rval(L).b and rval(R[1]) are the same stack designator. Note that R[1] and R[3] also share.

The basis for interactions between processes is sharing between process states (see Sections 2.4 and 4.7). A consequence of sharing between two items is that whenever a storage operation is performed on a memory designator held in common both items are affected. Sharing can not occur between integers, truthvalues, prog-items or identifiers. Such values are said to be atomic. On the other hand, sharing can occur between items which have "parts", such as rows, structs and memory elements (queues, stacks and cells). Such objects are said to be non-atomic. For this example process designators are considered to be atomic.

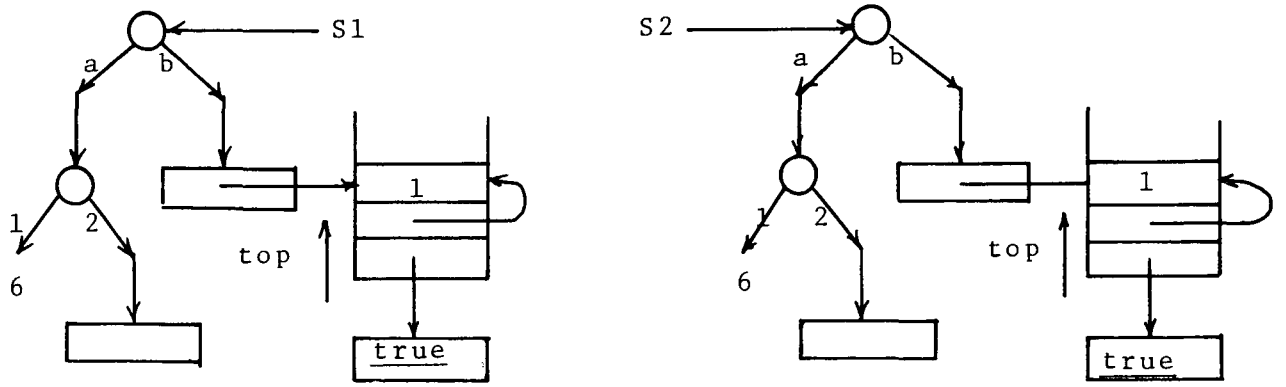
The copy of a non-atomic item I1 is another non-atomic item I2 which exhibits certain well defined sharing relations with I1 and certain well defined "structural" similarities to I1. The "copy" of an atomic item I1 is I1 itself. Copies can be classified on the basis of the nature of the sharing relations and structural similarities exhibited by I1 and I2:

1. For complete copies I1 and I2 are structurally identical and no sharing exists between them. The term "structurally identical" is taken to mean that the sharing relations between "components" of I2 (i.e., items accessible from I2 using rval, select and index) are identical to those between corresponding "components" of I1.
2. For partial copies sharing may exist between I1 and I2, and I1 and I2 need not be structurally identical.

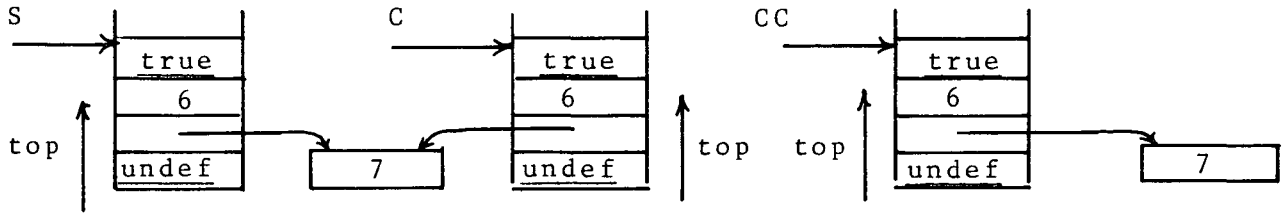
Some examples should help clarify the distinction between complete and partial copies. A struct S1 and its complete copy S2 are shown in Figure 5.9a. There is no sharing between S1 and S2. Note also that S1 and S2 are structurally identical; for example, note that in S1, S1.b and rval(S1.a[2]) share the same l-value, and that in S2, so also do S2.b and rval(S2.a[2]). In general, the copies made by the macro stack-copy, defined in Section 5.3.1, are partial copies of stacks. Figure 5.9b shows a stack S, a copy C of S produced by stack-copy and a complete copy CC of S. Partial copies such as C are sometimes called "one-level" copies. Figure 5.9c exhibits another kind of partial copy. Row R2 is a partial copy of row R1 produced using a "copy rule" that requires one-level copies of stacks and complete copies of all other non-atomic values. Row R3 is a complete copy of R1. Note that there is sharing between R1 and R2. Furthermore, note that R1 and R2 are not structurally identical; R1[2] and R1[1][1] share the same l-value whereas R2[2] and R2[1][1] do not.

The macro copy defined in this example makes complete copies. The following is the strategy used:

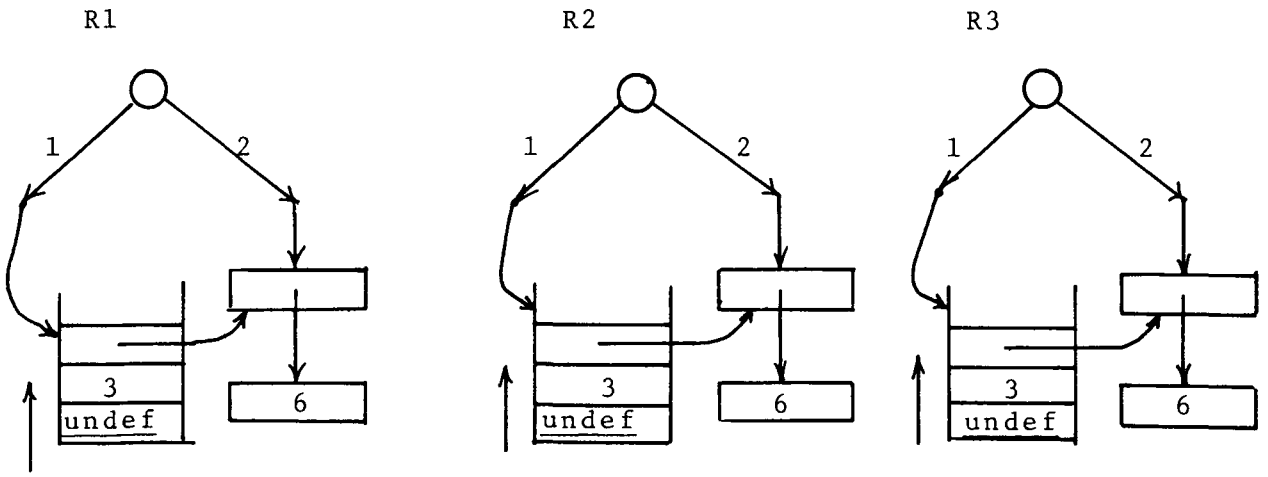
1. Copies are made by a dedicated process C which accepts requests for its services from other processes. A process P requests C to copy item S by using the macro copy (S). While P waits (with its aflag set to false), C responds by making S', the requested copy. C expects requests for copies to appear in its q(2) component and to be of the



(a)



(b)



(c)

Figure 5.9

Examples of complete and partial copies.

form

⟦requestor:P, original:S⟧

When S' is ready, P expects it to appear in its q(lmax-1) component.

2. C makes S' using the "function" COPY. (COPY is in reality a p-graph which is "applied" using the macro eval defined in Section 5.3.3.) COPY builds S' by "walking over" S. It adds to S' a copy of each non-atomic item of S it encounters on its walk. To insure that components of S' exhibit the same sharing properties as those of S, COPY maintains a table, named TABLE, whose entries are copies of the non-atomic items of S encountered on its walk. For each non-atomic item I of S it encounters COPY first checks TABLE to see if I has been encountered previously. It then works as follows:

   A. if I has not been previously encountered then

      1. if it is a queue designator then

         a. a queue designator I' is obtained using new-queue;

         b. an entry is made for I and I' in TABLE;

         c. for each item I[i] for $1 \leq i < \text{length}(I)$

            i. if I[i] is atomic, it is added to I' as its ith item.

            ii. otherwise, a copy of I[i] (made by COPY) is added to I' as its ith item.

         d. the copy of I is I'.

2-5. analogous actions for the cases stack designator, row designator, struct designator, and cell designator.

B. if I has been encountered previously then there must be an entry for it and its copy I' in TABLE. The copy of I is I'.

3. TABLE is a stack to which entries are made using push.

4. The following auxilary macros are used in the definition of copy:

a. lookup(I), which searches TABLE for an entry for I; if an entry is found the value of lookup(I) is I', the copy of I, otherwise, its value is undef.

b. enter(I, I'), which makes an entry for I and its copy I' in TABLE.

c. is-atom(I), a predicate for atoms.

d. copy1(I), which "calls" the "function" COPY using eval.

e. top-stack, whose value is the top item of the process stack component.

When process C is created its status and environment components are initialized as follows:

```
prog = undef        stack = new-stack
pc = 1              prog-id = nil
level = lmax        rp = rp defined in section 5.3.3 for eval
aflag = false       proc-id = [[COPY:p-graph defined below,
                              TABLE:new-stack,
                              EVAL_STACK:new-stack ]]
```

C's hp(2) component, whose job it is to respond to requests for copies, is

```
§ until eq(0, length(proc-id.TABLE)) do      //Clear TABLE from
          pop (proc-id.TABLE);               //the last request.
  interrupt (q(2)[1].requestor, lmax-1,      //Make the copy
             copy1 (q(2)[1].original));      //and return it.
  advance (q(2));
  set-level-inactive (lmax) §                //Wait for next
                                             //request
```

The definition for copy is

```
MACRO: copy (S)
   level;              //place current level in stack for "return".
   interrupt (C, 2                //Request copy
              [[requestor:proc, original:S]]);
   set-level-inactive (lmax)  //and wait for it.
ENDMACRO
```

The job of the hp(lmax-1) component for a process requesting a copy is to accept the copy from C; a suitable p-graph for hp(lmax-1) is

```
§ dump(lmax-1).level := top-stack;         //Retrieve level
  spop;                                    //from the stack.
  dump(lmax-1).aflag := true;
  q(lmax-1)[1];                            //Push the copy
  advance (q(lmax-1));                     //onto the stack.
  set-status (proc, dump(lmax-1)) §
```

Definitions for the auxilary macros follow.

```
MACRO: copy1 (x)                    //Macro to "call" COPY
   eval (proc-id.COPY, [[top:[[quote(S):x]], rest:nil]])
ENDMACRO
```

```
MACRO: lookup (I)                    //Macro to search TABLE
   let T = proc-id.TABLE;            //for entry I.
       A = new-cell (undef);
       i = new-cell (0)
   in
   for i = 1 to length(T) do
     ( iff eq (I, T[i].item) do
           ( A := T[i].copy,
             nextis EXIT ) );
   EXIT => rval (A)
ENDMACRO

MACRO: enter (S, C)                  //Macro to make entry
   push (proc-id.TABLE,              //in TABLE for S.
         [[item:S, copy:C]])
ENDMACRO

MACRO: is-atom (x)
    is-int (x) V
    is-ident (x) V
    is-proc (x) V
    is-prog-item (x) V
    is-truthval (x)
ENDMACRO

MACRO: top-stack     //When index is performed the top
   stack [3]         //2 items in the stack are the
ENDMACRO             //designator for the stack and 3.
```

The p-graph for COPY is written assuming it will be evaluated in an environment in which the item to be copied is bound to S. COPY is defined as follows:

```
§ if is-atom (S)                     //Is S an atom?
  then S                             //Yes, S is its own copy.
  else                               //No, has S been
   ( let x = lookup (S)              //previously encountered?
     in
     if is-undef (x)                 //No, for S an lvalue
     then      if is-lval (S) then
                 ( let C = new-cell (undef)
                   in
                   enter (S, C); C := copy1 (rval(S));
                   C )
```

```
else if is-queue (S) then     //For S a queue
      ( let C = new-queue;
            i = new-cell (0)
        in
        enter (S, C);
        for i = 1 to length(S) do
             enqueue (C, copy1(S[i]));
        C)
else if is-stack (S) then     //For S a stack
      ( let C = new-stack;
            i = new-cell (length(S))
        in
        enter (S, C);
        until  eq (0, rval(i)) do
           ( push (C, copy1(S[rval(i)]));
             i := rval(i) - 1 );
        C)
else if is-row (S) then       //For S a row
      ( let C = new-cell (nil)
            i = new-cell (length(S))
        in
        until eq(0, rval(i)) do  //Leave copies of
          ( copy1 (S[rval(i)]);  //components on
            i := rval(i) - 1);   //stack for row.
        C := row (length(S));    //Build row from
        enter (S, rval(C));      //copies on stack.
        rval (C) )
else                             //S must be a struct
  ( let C = new-cell(nil)
        SEL = selectors(S);
        i = new-cell (length(SEL))
    in
    for i = 1 to length(SEL) do   //Leave copies of
      ( copy1 (select(S, SEL[i]));//components on
        SEL[i]);                  //stack for struct.
    C := struct (length(SEL));    //Build struct from
    enter (S, rval(C));           //copies on stack.
    rval (C) )

else x §                          //S has been previously
                                  //encountered and is
                                  //in TABLE.
```

CHAPTER 6

Using The Model: Examples

6.1 Introduction

This chapter illustrates by example how sophisticated patterns of process behavior can be expressed in terms of the model. Six examples are presented, each presentation having three parts. The first part of each explains the behavior patterns of interest; the second part discusses the strategy to be used to synthesize the behavior in terms of the model; and, the third part presents the PGL code for the description. The casual reader may choose to skim the third part of each example.

Several further remarks concerning these examples are in order. It is not the intent of this chapter to teach basic programming skills. Consequently, PGL code for the details of tasks such as building and manipulating tables is frequently omitted. Because the examples are intended to be illustrative, the programs that appear in this chapter are written to be efficient pedagogically. No attempt to optimize in any other way is made. In particular, an effort is made to avoid using programming "tricks" to minimize the "run time" or "storage space" the programs would require if run. Each example specifies process state components which enable processes to

exhibit the particular behavior pattern of interest. No attempt is made in any of the examples to collect the specifications together into a single formal specification.

6.2 Blocks and Secret Variables

The notion of secret variables for block structured languages was first proposed by Mitchell [Mi70]. The essence of the idea is that there are two kinds of variable declarators, new and secret. Variables declared using new obey the usual block structure scoping rules, the scope of a "new" variable being the block in which it is declared and all nested blocks in which it is not redeclared. The scope of a variable declared using secret is limited to the block in which it is declared. Such a variable is not accessible from and hence kept "secret" from blocks nested within the one in which it is declared.

This example has two parts. The first defines block structure and ordinary (new) variable declaration. The second part modifies the definitions for block structure and variable declaration to include secret variables.

1. Block structure and new variables.

The macros block and new are defined. The effect of

block (x)

is to be equivalent to the ALGOL fragment

begin x end

Variables are to be declared using new. The effect of

new (x)

is to bind identifier x in the top layer of the prog-id component to a cell initialized to undef. The definitions for block and new are:

```
MACRO: block (x)
   set-prog-id (proc, [[top:nil, rest:prog-id]]);
   x;
   set-prog-id (proc, prog-id.rest)
ENDMACRO

MACRO: new (x)
   bind (quote(x), new-cell (undef))
ENDMACRO
```

2. Secret variables.

The macro secret, the declarator for secret variables, is defined in this part. The addition of secret variables requires that new and block be redefined. The strategy used is the following:

1. New variables and secret variables are bound in separate id-layers of the prog-id component. Secret variables for a block are to be bound in the top id-layer and new variables, in the next id-layer (see Figure 6.1a).

2. Upon entry to a block B nested within block A, the id-layer binding A's secret variables is removed from the prog-id component and saved until exit from B. Next, a new id-layer is added to prog-id for the duration of processing B's declarations (see Figure 6.1b). The only identifiers bound in that layer are NLAYER and SLAYER. As variable

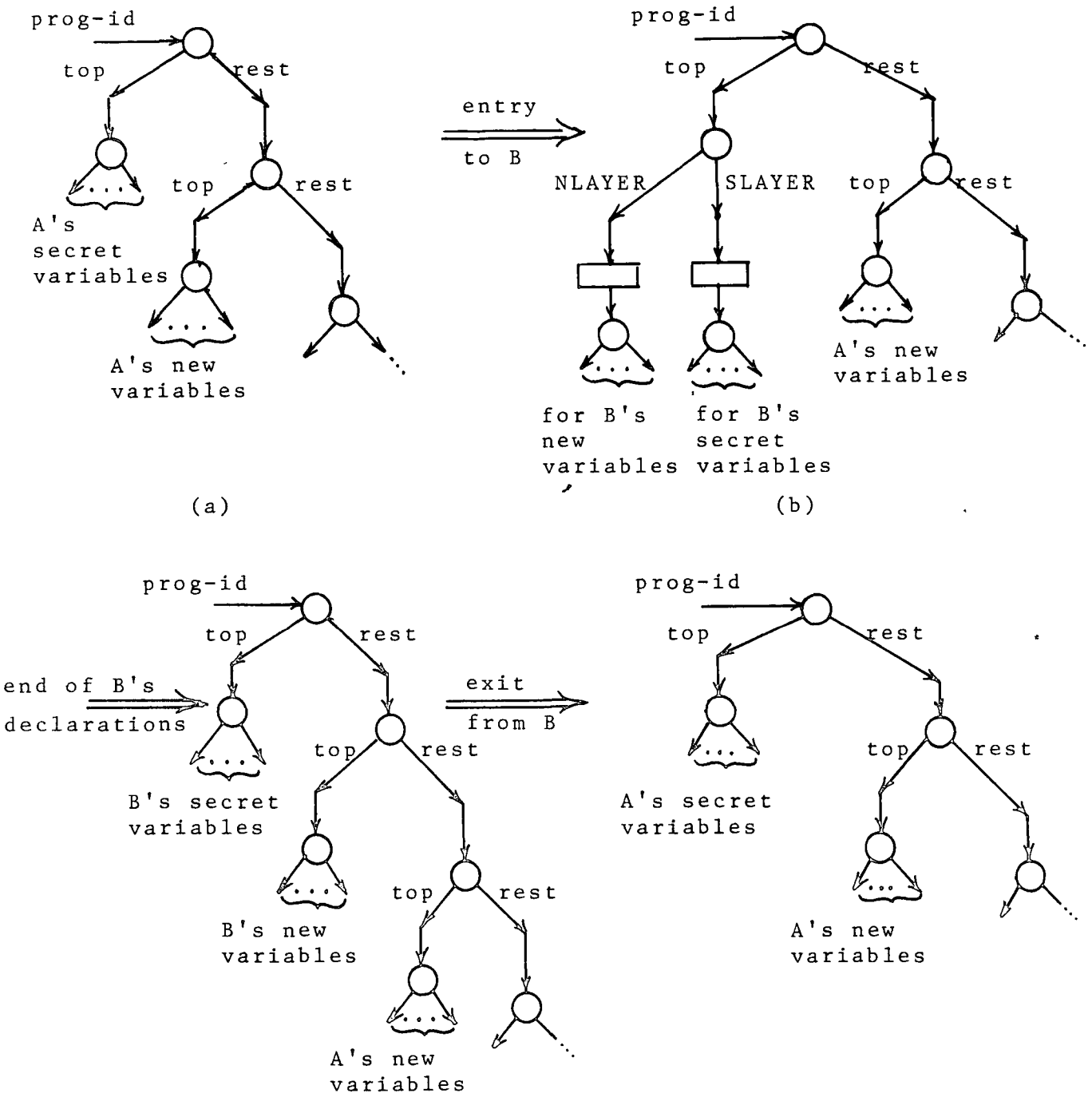


Figure 6.1

Strategy for manipulating the process prog-id component on block entry and block exit in order to realize secret variables.

declarations are encountered new variables are added onto the id-layer (struct) being built in NLAYER and secret variables, onto the one being built in SLAYER. After all of the declarations in B have been processed, prog-id is set such that top two id-layers in it bind, respectively, B's secret and new variables (see Figure 6.1c). At block exit prog-id is restored to as it was prior to block entry: the two layers binding B's variables are discarded and the layer binding A's secret variables is restored (see Figure 6.1d).

3. This strategy requires that declarations in a block appear before the "statements". This restriction could be relaxed in a more sophisticated system which included a "compiler". The end of the declarations must be signalled to allow the prog-id component to be set appropriately. Again, in a more sophisticated system, the signal could be an implicit one detected by the compiler (see Section 8.2.3). However, no such compiler is assumed for the present situation. Rather, the macro end-dec is used to explicitly signal the end of declarations.

4. The identifier SEC_ID is bound to a stack designator by the proc-id component. It is used to "save" the secret variable bindings which are temporarily discarded at block entry.

5. The purpose of this example is to demonstrate how the model can be used to realize the scoping rules for secret variables. Because processes can directly access their own state components a process could "cheat" and examine proc-id.SEC_ID to peek at variables intended to be kept secret from it. As the model has been defined, there is no way to prevent a process from doing that. The general problem of limiting the capabilities of a process is considered in Chapter 7. Were the model to serve as the base for an extensible language, this problem would not be a serious one. Because all language features would be defined in terms of the model, a language extender could define his extended language such that proc-id.SEC_ID not be accessible directly from it and thereby guarantee that secret variables are truly secret. That is, the compiler for the language would never produce the code required to "cheat".

The definitions for block, new, secret and end-dec follow. The macro top-from is defined in Section 5.3.3.

```
MACRO: block (x)
   push (proc-id.SEC_ID, prog-id);        //Save secret variables.
   set-prog-id (proc,                     //Remove secret variables
                [[top:[[NLAYER:new-cell(nil),  //and set
                       SLAYER:new-cell(nil)]], //prog-id for
                 rest:prog-id.rest]]);         //declarations.
   x;
   set-prog-id (proc, top-from(proc-id.SEC_ID))
ENDMACRO

MACRO: new (x)
   NLAYER := aug-struct (rval(NLAYER), quote(x),
                         new-cell(undef))
ENDMACRO
```

```
MACRO: secret (x)
   SLAYER := aug-struct (rval(SLAYER), quote(x),
                         new-cell(undef))
ENDMACRO

MACRO: end-dec
   set-prog-id (proc,
                [[top:rval(SLAYER),
                  rest:[[top:rval(NLAYER),
                         rest:prog-id.rest ]] ])
ENDMACRO
```

These macros require the proc-id component to be of the form

```
[[  .
    .
    .
 SEC_ID:new-stack
    .
    .
    .     ]]
```

6.3 Functions

Nearly every high level programming language permits definition of procedures which can subsequently be "called". This example describes a facility which allows value-producing procedures, called functions, to be created and subsequently "applied". Macros to create and to apply functions are defined.

Before proceeding further, it is useful to develop some terminology. The expression to be evaluated when a function is applied is called the body of the function. The collection of formal parameters of a function is called the bound variable part (bv-part) of the function. A particular member of the bv-part of a function f is said to be a bound variable of f, or

simply a bound variable when it is clear that the function in question is f. Identifiers appearing in the body of f which are not bound variables of f are said to be free variables of f. Consider, for example, the function

g (x, y) = a*x + b*y

The bound variables of g are x and y; its body is a*x+b*y and its free variables are a and b.

When a function is applied to a collection of actual parameters its body is evaluated in an environment in which its bound variables are bound to the actual parameters. That is, when its body is interpreted, the identifiers comprising its bv-part are bound by the process prog-id component to the actual parameters supplied. Programming languages display a variety of methods for treating free variables appearing in function bodies. This example considers three such methods:

1. Free variables are to be left unbound.

   A compiler for a language with this kind of free variable "binding" could detect function definitions containing free variables and report them. The effect would be equivalent to forbidding free variables in function bodies. The macro functionl (p, b), where p is a row of identifiers and b is a p-graph, is to build a function whose bv-part is p and whose body is b. The macro applyl (f, a), where f is a function created by functionl and a is a row of actual parameters, is to apply f to a, leaving free variables in the body of f

unbound.

2. Free variables are to be bound when the function is applied.

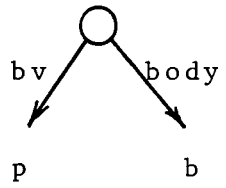
   Unless the programmer explicitly states otherwise, LISP [Mc62] binds free variables in this way. The term "fluid variable" is sometimes used to refer to a free variable bound in this way. When this method of binding is used, a function applied in different places to the same arguments may produce different values, even in the absence of side effects. Consider, for example, the function g defined above. When it is applied to [[1, 1]] in an environment in which a and b are both bound to 1 it produces 2. However, if a is bound to 1 and b to -1 at the point of application, it produces 0. The macros function2 and apply2 create and apply functions using this method of free variable binding.

3. Free variables are to be bound when the function is created (declared).

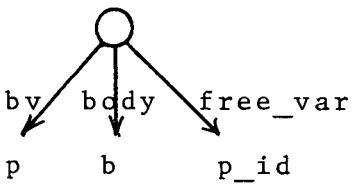
   ALGOL 60 [Na63], PL/1 [IBM69] and PAL [Ev68] bind free variables in this way. When this method of binding is used a function applied in different places to the same arguments produces the same values, in the absence of side effects. The macro function3 creates a function whose free variables are bound as it is created. The macro apply3 is used to apply such a function.

The following strategy is used to define the "function" and "apply" macros:

1. Structs are used to represent functions. Those created by function1 (p, b) and function2 (p, b) are represented by structs of the form



A function built by function3 includes, in addition to its bv-part and body, bindings for its free variables. The function created by function3 (p, b) is represented by a struct of the form



where p-id is the value of the process prog-id when the function is created.

2. The "apply" macros all use the macro eval defined in Section 5.3.3. Each first constructs an id-layer which binds the bound variables of the function to the values supplied for the actual parameters. Then, each uses eval to evaluate the body of the function with the appropriate prog-id. The three "apply" macros differ only in the prog-id they supply to eval. Use of eval requires that the proc-id component include the identifier EVAL_STACK and that the rp conponent be prepared to perform "returns" as

described in Section 5.3.3.

3. The "apply" macros use an auxilary macro make-layer. The value of make-layer (F, A), where F is a row of identifiers and A, a row of values, is an id-layer (struct) in which the identifiers of F are bound to the corresponding components of A.

The macro definitions follow.

```
MACRO: make-layer(F, A)
   let IDL = new-cell (nil);
       i = new-cell (0)
   in
   for i = 1 to length (F) do
      IDL := aug-struct (rval(IDL), F[i], A[i]);
   rval (IDL)
ENDMACRO

MACRO: function1 (p, b)
   [[bv:p, body:b]]
ENDMACRO

MACRO: apply1 (f, a)
   eval (f.body, [[top:make-layer(f.bv, a),
                   rest:nil]])
ENDMACRO

MACRO: function2 (p, b)
   [[bv:p, body:b]]
ENDMACRO

MACRO: apply2 (f, a)
   eval (f.body, [[top:make-layer(f.bv, a),
                   rest:prog-id]])
ENDMACRO

MACRO: function3 (p, b)
   [[bv:p, body:b, free_var:prog-id]]
ENDMACRO

MACRO: apply3 (f, a)
   eval (f.body, [[top:make-layer(f.bv, a),
                   rest:f.free_var]])
ENDMACRO
```

6.4 Dijkstra's Semaphores and Parallel Begin

Dijkstra [Di68a] has suggested an extension to ALGOL 60 to permit description of parallelism of execution (see Section 1.2.3). He proposes use of "parbegin" and "parend" to bracket statements to be executed in parallel. The entire construction between the brackets is to be regarded as a single compound statement whose execution is completed when execution of all its constituents is completed. Thus,

```
begin
  S1;
  parbegin S2; S3; S4 parend;
  S5
end
```

specifies that after completion of S1, statements S2, S3 and S4 are to be executed in parallel, and only after all of them are finished is S5 to be executed.

Dijkstra, in addition, proposes semaphores, together with two primitive operations v and p, as a means for synchronizing processes created by parbegin. A semaphore is a special purpose integer-valued variable whose value is constrained to be non-negative. The effect of the v operation on a semaphore is to increase its value by 1. The v operation is "indivisible" in the sense that two processes simultaneously attempting it on the same semaphore perform it sequentially in an unspecified order. The effect of the p operation on a

semaphore is to decrease its value by 1 as soon as the resulting value would be non-negative. The p operation represents a potential delay. If the value of semaphore S is not positive when a process initiates p(S), the p operation can not be completed until another process perform a v operation on S.

Several processes using the same data base can insure that only one of them accesses it at any time by agreeing to use a binary semaphore b (b's value must be either 0 or 1). Each agrees to perform p(b) before accessing the data base and v(b) upon completion of the access. In an analogous manner a group of processes can use a general semaphore g, whose value is restricted to the range $0 \leq value(g) \leq n$, to insure no more than n of them are engaged in a particular activity at any time. Each process agrees to perform p(g) before starting the activity and v(g) immediately upon completing it.

The macros parblock, semaphore, v and p are defined for this example. The effect of parblock (x), where x is a row of p-graphs, is to be equivalent to

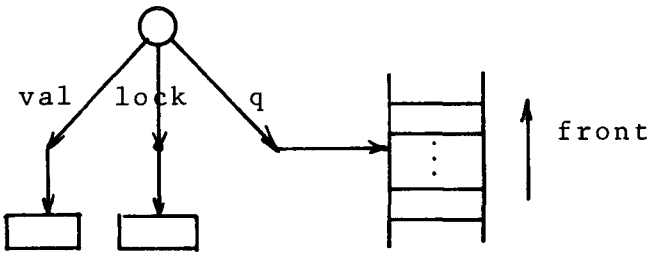
parbegin x[1]; x[2]; ... ; x[length(x)] parend

The macro semaphore (S, n) declares identifier S to be a semaphore with initial value n. The p and v macros correspond to the p and v operations. To provide context for these macros, assume they are to be used with the macros block and new as defined in the first part of Section 6.2.

The strategy to be used in defining the macros is the following:

1. When a process P performs parblock it creates a new process for each of the p-graphs to be executed in parallel. Next, it waits inactive until each process it has created reports completion. When a process created by P completes its task, it informs P and terminates.

2. A semaphore is represented by the designator for a struct of the form



The val component is the "value" of the semaphore. If it is zero when a process attempts a p operation, completion of p is delayed. Designators for processes awaiting completion of the p operation on the semaphore are held in the q component. The lock component is used to insure that only a single process accesses the val component at any time.

3. To perform the p operation on semaphore S, a process first performs t-set on S.lock until it succeeds. If S.val is positive it decrements it by 1, clears S.lock (i.e., sets it to 0) and continues. Otherwise, it places its process designator at the end of S.q, clears S.lock and sets itself inactive to await completion of the P operation.

4. To perform the v operation on semaphore S, a process first locks S.lock (using t-set until it succeeds). Next, if no processes are waiting completion of a p operation on S, it increments S.val by 1. Otherwise, it notifies the process whose designator is the first item in S.q of completion of its p operation and advances S.q. Finally, it clears S.lock.

5. The identifiers NSONS and FATHER are bound in the proc-id component of each process P. FATHER is the designator of the process that created P. NSONS is the number of P's descendents which have not yet reported completion.

6. The conventions extablished for the interactions that occur between processes make use of three levels:

   2: Processes perform their normal activities at level 2; in addition, a process expects completion notifications from its descendents to appear in its q(2) component.

   3: A process expects notification that it can complete a p operation it initiated to appear in its q(3) component.

   4: Processes use level 4 to await the completion of descendents and of p operations.

   When a process must wait, either for permission to complete a p operation or for its descendents to complete, it uses the set-level-inactive operation to simultaneously increase its level from 2 to 4 and set its aflag to false. Use of set-level-inactive prevents races by insuring that the process can not be interrupted by a completion notification

until it becomes inactive (see Section 4.6).

The job of the rp component for each process is to terminate it after reporting completion of its task to its creator; it is:

```
§ interrupt (proc-id.FATHER, 2, nil);      //Report completion
  terminate §                              //and terminate.
```

The hp(2) component is responsible for responding to notification from a descendent that it has completed its task; it is:

```
§ proc-id.NSONS := rval(proc-id.NSONS) - 1;
  iff eq (0, rval(proc-id.NSONS)) do       //If all sons have
      ( dump(2).aflag := true;             //completed, prepare
        dump(2).level := 2 );              //to resume.
  advance (q(2));
  set-status (proc, dump(2)) §
```

Response to notification that a p operation it has initiated has completed is handled by the hp(3) component of a process; it is:

```
§ advance (q(3));
  dump(3).aflag := true;
  dump(3).level := 2;
  set-status (proc, dump(3)) §
```

The macro defintions are:

```
MACRO: parblock (x)
   let X = x;
       N = length (X);
       T = new-cell (undef);
       i = new-cell (0)
   in
   proc-id.NSONS := N;                   //Set number of descendents.
```

```
   for i = 1 to N do                    //Create each descendent.
   ( T := new-proc;
     set-prog (rval(T), X[i]);          //Set state components
     set-level (rval(T), 2);            //for descendent
     set-aflag (rval(T), true);
     set-prog-id (rval(T),              //".rest" because don't want
                  prog-id.rest);        //X,N,T,i in prog-id of T.
     set-proc-id (rval(T), [NSONS:new-cell(0),
                            FATHER:proc]) ;
     set-rp (rval(T), rp);
     set-hp (rval(T), 2, hp(2));
     set-hp (rval(T), 3, hp(3));
     release (rval(T))  );
   set-level-inactive (4)               //Wait for descendents
                                        //to finish.

MACRO: semaphore (S, n)
   bind (quote(S), [val:new-cell(n),
                    lock:new-cell(0),
                    q:new-queue])
ENDMACRO

MACRO: p(S)
   until t-set(S.lock) do null;         //"Lock" S.
   if  eq (0, rval(S.val))              //Is S positive?
     then ( enqueue (S.q, proc);        //No, must wait to
            S.lock := 0;                //complete P.
            set-level-inactive (4) )
     else ( S.val := rval (S.val) -1;   //Yes, decrement S
            S.lock := 0 )               //and continue.
ENDMACRO

MACRO: v (S)
   until t-set (S.lock) do null;
   if  eq (0, length(S.q))    //Processes awaiting P completion?
     then  S.val := rval (S.val) + 1;   //No, increment S.
     else ( interrupt (S.q[1], 2, nil); //Yes, inform process
            advance (S.q) );            //of p completion.
   S.lock := 0
ENDMACRO
```

6.5 Backtracking

In many situations problems arise which can be usefully represented by "search trees" whose non-terminal nodes represent choice points and whose terminal nodes represent

potential solutions. A solution to such a problem can be found by traversing the corresponding tree, beginning at its root, until a satisfactory terminal node is encountered.

Floyd [Fl67] has proposed a programming technique making it easy to write programs to solve such problems. His technique includes an automatic backtracking facility. In effect, Floyd proposes two operations:

choice, which chooses one branch to traverse from the group departing from a non-terminal node in the search tree; and

backup, which "undoes" everything done since the last choice including the choice itself.

For this example macros for choice and backup are defined. The value of the expression

choice (n)

is to be an integer c such that $1 \leq c \leq n$. If, at some time after it is made, a choice is found to be a bad one, it can be "remade" using backup. Interpretation of backup causes "execution" to return to the point of last choice where the choice is remade. Then, execution continues from that point as if the choice had been made for the first time. In particular, storage operations performed after the unsuccessful choice and prior to the backup are "undone". If all choices from the given choice point have been tried, backup causes the choice at the previous choice point (if any) to be redone. That is, choices may be nested.

The macros choice and backup are defined using the following strategy:

1. Interpretation of backup causes the pc, prog, stack and prog-id components to be reset as they were at the previous choice point. Prog and pc are set to the values they had immediately after the unsuccessful choice. The prog-id component is set to a complete copy (see Section 5.3.4) of the prog-id as it was at the choice point. And, the stack is set to a complete copy of the stack component as it was immediately after the unsuccessful choice, with the exception that the top item is the new choice rather than the unsuccessful one. Installating complete copies of the stack and prog-id components insures that all storage operations are undone. It is important that stack and prog-id be copied "together" to insure that the copies exhibit the same sharing properties as the originals. For example, suppose identifier x is bound in the prog-id to be copied to an lvalue which is also the 6th item in the stack to be copied. It be crucial that x be bound in the copy prog-id to the lvalue which be also the 6th item in the copy stack. Note that state components other than prog, pc, stack and prog-id are not restored by backup. An assumption made in defining the choice and backup macros is that processes using them do not change state components other than their prog, pc, prog-id and stack. The macros could, if desired, be defined such that other state components were restored also.

2. The first choice made for choice (n) is n. New choices are made, as often as necessary, by subtracting 1 from the old one, until the last choice, 1, is made.

3. When choice (n) is interpreted preparation for the first backup is made by saving the values of the prog and pc components and by saving complete copies of the stack and prog-id components. Next, n, the first choice, is pushed onto the stack.

4. If not all paths of the tree have been traversed when backup is performed, the following actions are taken:
   a. If the new choice is not the last one for the most recent choice point (i.e., if it is not 1), preparation for the next backup is made. Prog and pc values for the choice point and copies of the stack and prog-id components, as they were at the choice point, are saved.
   b. If the new choice is the last one for the most recent choice point, preparation for the next backup need not be made. Preparation for it has been made at the choice point occurring before the most recent one.
   c. The prog, pc, stack and prog-d components are set to perform the backup.

   If every path of the tree has been traversed, there is no solution to the problem and an error situation arises. In a "real" application appropriate action could be defined.

5. The definitions for choice and backup use the identifiers CONTROLS, STACKS and PROG_IDS which are bound in the process proc-id component. These identifiers are bound to designators for stacks which are used to hold values saved for the control, stack and prog-id components, respectively. The identifiers S, P and C, which are used for temporary storage, are also bound in the proc-id component.

6. The macros top-from, make-control, stack-comp-copy and copy are used. Definition for the first two macros are to be found in Section 5.3.3 and, for the remaining two, in Sections 5.3.1 and 5.3.4, respectively. To use copy the hp(lmax-1) state component must be defined as in Section 5.3.4.

Definitions for the macros follow.

```
MACRO: choice (n)
   if  eq (n, 1)
   then 1
   else
   ( let S = stack-comp-copy;              //Copy stack and prog-id
         T = copy ([STACK:S,
                    P_ID:prog-id.rest])
     in
     push (T.STACK, n-1);                  //Prepare for the
     push (proc-id.STACKS, T.STACK);       //first backup.
     push (proc-id.PROG_IDS, T.P_ID);
     push (proc-id.CONTROLS,
        make-control(prog, EXIT));
     n;                                    //Make first choice.
     EXIT => null )
ENDMACRO
```

```
MACRO: backup
   iff  eq (0, length(proc-id.STACKS)) do ERROR;  //Failure.
   proc-id.S := top-from (proc-id.STACKS);
   proc-id.P := top-from (proc-id.PROG_IDS);
   proc-id.C := top-from (proc-id.CONTROLS);
   unless  eq (1, (rval(proc-id.S)[1])) do
   ( let T = copy ([STACK:rval(proc-id.S),          //Prepare for
                    P_ID:rval(proc-id.P)]))         //next backup.
     in
     pop (T.STACK);
     push (T.STACK, (rval(proc-id.S))[1] - 1);
     push (proc-id.STACKS, T.STACK);
     push (proc-id.PROG_IDS, T.P_ID);
     push (proc-id.CONTROLS, rval(proc-id.C)) );
   set-stack (proc, rval(proc-id.S));               //Backup.
   set-prog-id (proc, rval(proc-id.P));
   set-control (proc, rval(proc-id.C))
ENDMACRO
```

6.6 Non-deterministic Programming

This section suggests an alternative approach to the problem described in Section 6.5. The operations choice and backup, defined in Section 6.5, can be used to traverse a "problem" tree. At each non-terminal node first one branch and then another is followed until a satisfactory terminal node is encountered. Using this approach the tree is traversed in a strictly sequential manner, each path being investigated in turn.

The behavior that results from the alternative approach is reminiscent of that exhibited by non-deterministic automata. Whenever a non-terminal node in the tree is encountered, a separate process is created to follow each branch departing from it. Should a process find that the path it is following leads to an unsatisfactory result, it terminates. If the path

it follows leads to a satisfactory result, the process reports the result and processes working on other paths terminate. This approach, for which paths are investigated concurrently, results in a non-sequential tree search.

The non-deterministic behavior can be described using the macros eval-nd, choice and failure. To indicate that an expression is to be evaluated non-deterministically a process uses eval-nd. After process M performs

eval-nd (e)

it waits for a newly created process P to evaluate e for it. If P successfully evaluates e, it notifies M of the result. Should it be unable to evaluate e (i.e., if all paths in the tree lead to unsatisfactory results), P notifies M that e is undefined. When a path is discovered to lead to an unsatisfactory result, the process following it performs failure. The effect of failure is to first inform the creator of the process and to then terminate the process. When a process Q performs

choice (n)

it creates n new processes C1, ..., Cn, each of which represents a different choice. Each of the Ci then proceeds with the evaluation with a different value for the choice while Q waits for failure notifications from the Ci. If all Ci fail, Q notifies its creator and terminates. Should a process Ci follow a path leading to a satisfactory solution, it notifies P (the process created to evaluate e for M) of its success and

terminates. P then notifies M of the successful result, arranges for processes still following paths to terminate and finally terminates itself. The three macros are defined to allow nesting of eval-nd.

The explanation of the strategy used to define the macros uses M. P. Ci and Q to denote processes. M is used to denote a process performing eval-nd, P to denote the process created to carry out the non-deterministic evaluation for M, and Ci to denote a process created by the choice operation. Q is used, whenever appropriate, to denote an arbitrary process.

The strategy is the following:

1. The conventions established for interactions between processes performing the non-deterministic evaluation make use of five levels:
   - 5: This level is used by processes for awaiting the occurrence of interactions with other processes. M waits for the result of the non-deterministic evaluation. Arbitrary process Q awaits notification from its Ci.
   - 4: Q expects failure notifications from its Ci to appear in its q(4) component.
   - 3: P expects notification from a successful Ci to appear in its q(3) component.
   - 2: This level is used by processes for performing their internal activity. In addition, M expects P to place the result of the non-deterministic evaluation in its

q(2) component.

1: Q expects termination orders to appear in its q(1) component.

To avoid potential race situations processes use the set-level-inactive operation to change from level 2 to level 5 (see Sections 4.6 and 6.5).

2. Should a process receive an order to terminate it must be able to order its descendents to terminate. By creating each Ci such that its q(1) component is the designator for the same queue, a process P can avoid maintaining a table of its Ci. By placing an item in that one shared queue, P can interrupt all its Ci simultaneously ordering them to terminate.

3. To perform eval-nd (e) process M
   a. prepares to receive the result of the non-deterministic evaluation by setting its own hp(2) component;
   b. creates process P to evaluate e and releases it; and
   c. sets itself inactive to await notification from P of its result.

   When P is created its prog component is set to e. Its hp(1) component is set to respond to terminate orders and its hp(3) component, to respond to success notifications from Ci it subsequently creates.

4. To perform choice (n) a process Q
   a. prepares to respond to failure notifications from the

processes it is about to create by setting its own hp(4) component;

b. creates n processes C1,...,Cn and releases them.

c. sets itself inactive to await failure notification from the Ci.

The hp(1) component of each Ci is set to respond to terminate orders.

5. To perform failure a process first notifies its creator and then terminates. Should process Q interpret its prog in its entirety, the path it is following has led to a satisfactory terminal node and the top item in its stack component is the value of e. In such a case, Q's rp component notifies P and then terminates Q.

6. Process Q responds to a failure notification from one of its Ci by decrementing a count it keeps of the number of its Ci which have not yet reported failure. If all of its Ci have failed, the action Q takes depends upon whether the choice which created its Ci corresponds to the root node of the problem tree. If the choice does correspond to the root then Q is P, all paths have been investigated and no solution to the problem exists. In such a case, Q notifies M of that fact and terminates. When Q is not P, it notifies its creator of its failure and terminates.

7. The identifiers SONS_Q, FATHER, NSONS, SPROC and ROOT are bound in the proc-id component of each process Q. SONS_Q

is bound to the designator for the queue used as the q(1) component of Q's Ci. FATHER is the designator of the process that created Q, NSONS, the number of Q's Ci which have not yet reported failure and SPROC, the designator of process P. ROOT indicates whether Q is P.

8. A number of auxilary macros are used:
   a. stack-comp-copy (Section 5.3.1), copy (Section 5.3.4) and top-stack (Section 5.3.4);
   b. hp1, hp2, hp3, hp4 (to be defined) whose values are p-graphs used for the hp(1),...hp(4) components of processes created by choice and eval-nd.
   c. ndrp (to be defined) whose value is a p-graph used for the rp component of processes created by choice and eval-nd.
   d. terminate-sons (to be defined) which is used by a process to order its Ci to terminate.

The macro definitions follow.

```
MACRO: ndrp                    //P-graph for rp component of processes
   § interrupt (proc-id.SPROC, 3, top-stack);//created by choice
     terminate §                              //and eval-nd.
ENDMACRO

MACRO: hp1                     //P-graph for hp(1); responds
   § terminate-sons            //terminate orders.
     terminate §
ENDMACRO

MACRO: hp2                     //hp(2) for M; accepts result from P
   § q(2)[1];                  //Place result on stack.
     advance (q(2));
     dump(2).aflag := true;    //Prepare to become active.
     dump(2).level := 2;
     set-status (proc, dump(2)) §
ENDMACRO
```

```
MACRO: hp3                  //hp(3) for P; responds to success notice.
   § interrupt (proc-id.FATHER, 2, q(3)[1]); //Notify M.
     terminate-sons;                        //Terminate
     terminate   §                          //remaining Ci.
ENDMACRO

MACRO: hp4                          //P-graph for hp(4); responds to
                                    //failure notices from Ci.
   § proc-id.NSONS := rval (proc-id.NSONS) - 1;
     unless  eq (0, rval(proc-id.NSONS) do   //Not the last Ci,
        ( advance (q(4));                    //wait for further
          set-status (proc, dump(4)) )       //notice.
     if  proc-id.ROOT                        //The last Ci.
        then  interrupt (proc-id.FATHER, 2, undef)
        else  failure;
     terminate   §
ENDMACRO

MACRO: terminate-sons               //Order Ci terminate by
   enqueue (proc-id.SONS_Q,         //causing a level 1
            nil)                    //interrupt to occur.
ENDMACRO

MACRO: eval-nd (e)
   let S = stack-comp-copy;
       P = new-proc;                        //Create P.
       T = copy ([STACK:S,                  //".rest" because don't want
                  P_ID:prog-id.rest]) //S,P,T in T's prog-id.
   in                                       //Prepare to receive
   set-hp (proc, 2, hp2);                   //result from P.
   set-prog (P, e);
   set-level (P, 2);
   set-stack (P, T.STACK);
   set-prog-id (P, T.P_ID);
   set-proc-id (P, [ROOT:true
                    FATHER:proc,
                    SPROC:P,
                    NSONS:new-cell (0),
                    SONS_Q:new-queue]);
   set-rp (P, ndrp);
   set-hp (P, 1, hp1);
   set-hp (P, 3, hp3);
   set-q (P, 1, proc-id.SONS_Q);
   release (P);                             //Release P.
   set-level-inactive (5)                   //Wait for result from P.
ENDMACRO

MACRO: failure
   interrupt (proc-id.FATHER, 4, nil);
   terminate
ENDMACRO
```

```
MACRO: choice (n)
   let S = stack-comp-copy;
       PROG_ID = prog-id.rest;
       T = new-cell (undef);
       Ci = new-cell (undef);
       N = n;
       i = new-cell (0)
   in                                    //Prepare for
   set-hp (proc, 4, hp4);                //notification from Ci.
   proc-id.NSONS := n;
   for i = 1 to n do                     //Create the n Ci.
   ( Ci := new-proc;
     set-control (rval(Ci),              //Set Ci's state.
                  make-control(prog, EXIT);
     set-level (rval(Ci), 5);
     T := copy ([[STACK:S, P_ID:PROG_ID]]);
     push (T.STACK, i);
     set-stack (rval(Ci), T.STACK);
     set-prog-id (rval(Ci), T.P_ID);
     set-proc-id (rval(Ci), [[ROOT:false,
                              FATHER:proc
                              SPROC:proc-id.SPROC,
                              NSONS:new-cell(0),
                              SONS_Q:new-queue]]);
     set-rp (rval(Ci), ndrp);
     set-hp (rval(Ci), 1, hp1);
     set-q (rval(Ci), 1, proc-id.SONS_Q);
     release (rval(Ci))  );              //Release Ci.
   set-level-inactive (5);               //Await failure notification.
   EXIT => null
ENDMACRO
```

6.7 Fisher's Control Primitives

Fisher [Fi70] has isolated seven operations which he considers suitable to serve as "control primitives" for constructing larger control structures (see Section 1.2.5). This section defines a set of macros which correspond to his primitives.

Fisher's control primitives are:

1. seq (x1, x2, ..., xn)

   Expressions x1 through xn are to be evaluated in order, from left to right. The value of the "sequence" expression is to be the value of xn.

2. cond (p1, e1, p2, e2, ..., pn, en)

   Beginning with p1 and proceeding from left to right every other operand (i.e., p1, p2, etc.) is to be evaluated until one produces the value true. The value of the "conditional" expression is taken to be the value of the ei corresponding to the pi whose value is true. Should no pi have value true, the value of the entire expression is taken to be undef.

3. par (x1, x2, ..., xn)

   Expressions x1 through xn are to be evaluated concurrently. No commitment is made concerning the relative speeds of their evaluation; hence, there is no commitment concerning the chronological order of their side effects, if any. The value of the "parallel" expression is to be the value of xn and is to be "available" only after all xi have been evaluated.

4. synch (c, x1, x2)

   The synch operation provides a locking mechanism which can be used to achieve coordination, synchronization and mutual exclusion. The expression c is to be evaluated first. Its value must be a "construct", a special kind of structure similar to a struct

(constructs are discussed later). If no other process is currently engaged in a synch operation "on" that construct, x1 is to be evaluated. Otherwise, x2 is to be evaluated. The value of the "synchronization" expression is taken to be the value of whichever expression, x1 or x2, is evaluated.

5. monitor (s, c, r, v, x)

Arguments s, c, r and v are to be evaluated first. The value of s must be a selector (S), that of c, a construct (C) and that of r a relational operator (R). No restrictions are placed on the value (V) of v. The value of the "monitor" expression is the designator for a process that continuously monitors for the condition

R (sel(C, S), V)

where sel is the selection operator for constructs. If ever that condition holds, the monitoring process evaluates x and then terminates.

6. unmonitor (p)

A monitoring process continues until either the condition obtains and x is evaluated or it is explicitly terminated by the unmonitor operation. The effect of unmonitor (p) is to terminate the monitoring process designated p.

7. cont (x)

This operation is based upon the notion of "relative continuity". A process P is said to be <u>continuous with respect to another process</u> Q if all of its actions

> occur between (two) consecutive state transitions of Q. When the "continuous" expression cont (x) is evaluated, the evaluation of x is to be continuous relative to all other processes (not themselves in the midst of a cont).

In the absence of monitor operations the effect of cont is straightforward. Whenever process P performs cont (x) all other processes "pause" while P evaluates x. After x has been evaluated, the processes that paused continue.

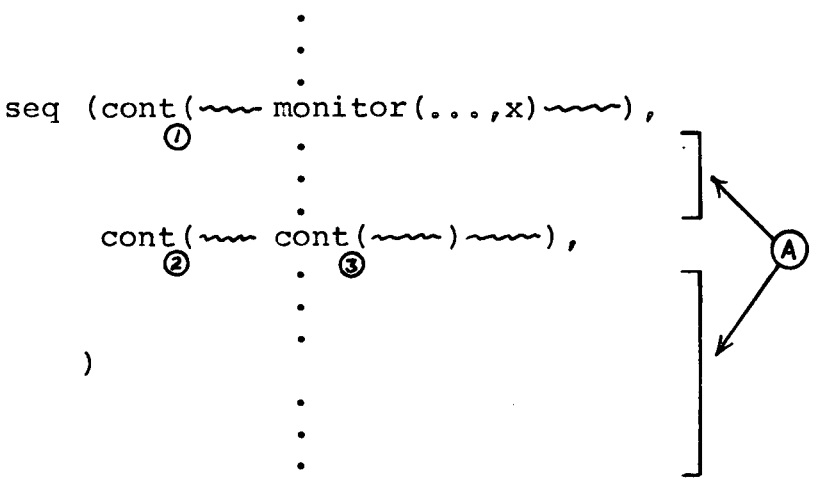
The monitor and cont operations interact in a subtle way. The notion of cont-depth is useful in explaining that interaction. Each process can be thought of as having a cont-depth associated with it. Evaluation of a program written with Fisher's primitives begins with a single process whose cont-depth is 0. Each time a process initiates a cont operation its cont-depth is incremented by 1 and when it completes the operation its cont-depth is decremented by 1. The value for the cont-depth of a newly created process in inherited from its creator. The definition for the cont operation can be reformulated in terms of cont-depth as follows:

> When process P whose cont-depth is D initiates cont (x), its cont-depth becomes D+1 and, simultaneously, all processes with cont-depth less than D+1 pause. Those with cont-depth greater than D are uneffected. After P evaluates x, its cont-depth is

reset to D, at which point the processes that paused resume their activities.

In the absence of the monitor operation all processes which are active must have the same cont-depth.

When a monitor operation is embedded within a cont operation it is poss ble for processes having different cont-depths to be active simultaneously. Consider, for example, a process P which evaluates



and suppose that when the seq operation is initiated its cont-depth is D. The process M created by the monitor operation within cont ① has cont-depth D+1, which is P's cont-depth when M is created. When cont ① is completed, P's cont-depth is reset to D, while M's remains D+1. When P performs cont ② M is uneffected. However, M must pause while P performs cont ③.

The definition of the monitor operation is such that should the condition being monitored become true, all processes

whose cont-depth is less than that of the monitoring process must pause while the expression which is the right most operand of the monitor operation is evaluated. For the above example, assume that the three cont operations shown are the only ones. If P is engaged in evaluating a sub-expression appearing in the parts labeled A when the condition M monitors obtains, P must pause while expression x is evaluated. If, on the other hand, P is engaged in evaluation of cont ② when the condition obtains, it is uneffected. A monitor operation embedded within a cont operation, as in this example, can approximate the effect of an interrupt mechanism. When the monitored condition becomes true, certain processes are "interrupted" while the appropriate expression is evaluated.

For this section a set of macros corresponding to Fisher's primitives is defined. The macros synch, monitor, unmonitor and cont are identical in effect to Fisher's primitives of the same name. Unlike Fisher's counterparts, the macros seq, cond and par take only a single argument. The effects of the macros seq (x), cond (x) and par (x), where in each case x is a row of p-graphs, are to be equivalent to Fisher's

seq (x[1], x[2], ..., x[length(x)])

and

cond (x[1], x[2], ..., x[length(x)])

and

par (x[1], x[2], ..., x[length(x)])

respectively.

In addition to the seven control macros, two macros, cons and sel, for manipulating constructs are defined. Constructs are similar to structs, the difference being that constructs have "locks" (used by synch) associated with them. The macro cons (x), where x is a row whose odd components are selectors and whose even components are values, creates a construct according to specification x. The value of sel (c, s) is to be the s component of construct c.

The cont operation and its interaction with the monitor operation represent the major difficulty in defining these macros. The strategy used is the following:

1. A table, TABLE, accessible to all processes, is maintained. When a process is created (by monitor or par) an entry, consisting of its designator and cont-depth, is made for it in TABLE. When a process terminates (as the result of unmonitor or completion of a parallel path) its TABLE entry is removed. The entry for a process always contains its current cont-depth. To perform cont (x), process P consults TABLE for a list of processes which should pause while it evaluates x; it arranges for each such process to pause, evaluates x, and then arranges for each to resume. It is important that no changes to TABLE be made between the time P begins to generate the list of processes and completes arrangements for them to pause. To insure that, processes access TABLE through a lock. The following auxilary macros are used to manipulate TABLE:

a. add-entry (P), rem-entry (P): P is a process designator. The effect of add-entry is to add the entry (P, D) for process P to TABLE, where D is the cont-depth of the process performing add-entry. The effect of rem-entry is to remove P's entry from TABLE. Both macros access TABLE through its lock.

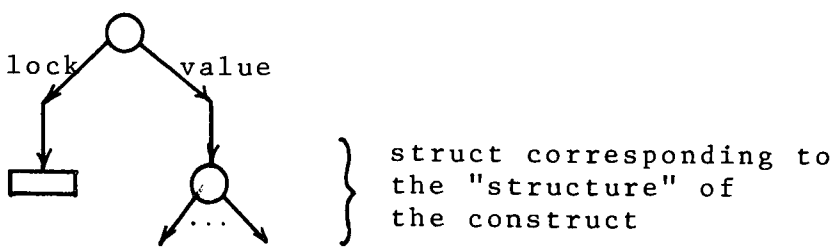
b. incr-depth (n), decr-depth (n): n is an integer. These macros respectively increment by n and decrement by n the cont-depth of the process performing them by changing the entry for the process in TABLE. Both access TABLE through its lock.

c. lock-table, unlock-table: Operations to explicitly lock and unlock TABLE.

The operation to lock TABLE is such that once initiated it is not completed until TABLE is locked. Neither the format of TABLE nor the above six macros are defined beyond the above description.

2. The macros seq and cond are straightforward. Definitions for them use the macro eval (defined in Section 5.3.3).

3. Constructs are represented by structs of the form



To perform synch (c, x1, x2) a process first performs t-set (c.lock). If the t-set succeeds, x1 is evaluated using eval and then c.lock is reset to 0. If it fails, x2 is evaluated using eval.

4. When a process performs par (x) it first creates a separate process for each of the components of x. Then, it waits for those processes to evaluate the components of x. When the process for x[i], for i ≠ length(x), finishes, it notifies P and terminates. When the one for x[length(x)] finishes, it passes P the value it has computed and teminates. After all have completed, P pushes the value of x[length(x)] onto its stack.

5. An auxilary macro contl (x, n) is used in the definitions for both cont and monitor. It represents a generalized cont operation which increments the cont-depth of a process by n rather than by 1. The following actions are taken by a process P with cont-depth D to perform contl (x, n):
   a. TABLE is locked;
   b. P increases its cont-depth to D+n;
   c. P uses the auxilary macro cont-list to build a list of processes with cont-depth less than D+n. The effect of cont-list(S), where S is a stack designator, is to push the designators for such processes onto stack S. (No further definition for cont-list is to be given.);
   d. P interrupts each process in S, requesting it to pause, and then waits for each process so requested to

acknowledge.

e. TABLE is unlocked;

f. x is evaluated using eval;

g. P decreases its cont-depth to D and informs each process whose designator is in S that it may resume.

6. To perform cont (x) a process performs contl (x, 1).

7. To perform monitor (s, c, r, v, x) process P creates another process M dedicated to repeatedly testing the condition, and then pushes M's designator onto its own stack. If ever the condition becomes true, M performs contl (x, 0) (see 5) and then terminates. To perform unmonitor (Q), process P interrupts Q requesting it to terminate.

8. The conventions established for interactions between processes make use of six levels:

1: unmonitor requests appear in q(1);

2: requests to pause while another process performs cont appear in q(2);

3: processes perform their normal activity at level 3; permission to resume after a cont is completed appears in q(3);

4: termination notices from processes created by par appear in q(4);

5: acknowledgements from processes receiving pause requests, indicating that they have paused, appear in

q(5);

6. A process waits for interactions from other processes to occur with its level set to 6.

The macros hp1, hp2, hp3, hp4 and hp5 (to be defined) specify the handler-prog components that respond to requests appearing in the various queues.

9. When a process is created its rp component is set such that
   a. it can use the eval macro (see Section 5.3.3); and
   b. when all nested "evals" have been completed
      i. if created by monitor it removes its entry from TABLE and terminates, and
      ii. if created by par it notifies its creator, removes its entry from TABLE and terminates.

The macro frp specifies the p-graph for such a reserve program.

10. The proc-id component of each process P binds the identifiers TABLE, EVAL_STACK, NPATHS, FATHER and VALUE. TABLE is the table of processes and EVAL_STACK is the stack used by eval. If P is a "monitor" process FATHER is undef; otherwise, it is the designator of the process that created P. NPATHS is the number of processes created by P using par which have not yet completed. P uses VALUE to hold the value passed to it by the "value producing" process created by par while it awaits completion of the other "par" processes. If P itself was created by par its proc-id also includes the identifier LAST. LAST is true if P is the

"value producing" process and is false otherwise.

The macro definitions follow.

```
MACRO cons (x)                    //Macro to build a construct.
   let S = new-cell (nil);
       i = new-cell (1);
   in
   until  gr(length(x), rval(i)) do          //Build "structure"
     ( S := aug-struct (rval(S), x[rval(i)], //part of construct.
                        x[rval(i)+1]);
       i := rval(i) + 2);
   [[lock:new-cell(0), value:rval(S)]]
ENDMACRO

MACRO: sel (c, s)                 //Macro to select component
   select (c.value, s)            //of construct.
ENDMACRO

MACRO: seq (x)
   let i = new-cell (0)
   in
   for i = 1 to length(x) do
       eval (x[i], prog-id.rest)
ENDMACRO

MACRO: cond (x)
   let i = new-cell (1)
   in
   until gr(length(x), rval(i) do
     ( iff eval(x[rval(i)], prog-id.rest) do
           ( eval(x[rval(i)+1], prog-id.rest);
             nextis EXIT);
       i := rval(i) + 2);
   undef
   EXIT => null
ENDMACRO

MACRO: synch (c, e, f)
   if  t-set(c.lock)
   then  (eval (e, prog-id); c.lock := 0)
   else  eval(f, prog-id)
ENDMACRO
```

```
MACRO: init-state (p)                    //Used by par and monitor to
   set-level (p, 3)                      //initialize state components
   set-prog-id (p, prog-id.rest);        //of processes they create.
   set-aflag (p, true);
   set-rp (p, frp);
   set-hp (p, 1, hp1);
   set-hp (p, 2, hp2);
   set-hp (p, 3, hp3);
   set-hp (p, 4, hp4);
   set-hp (p, 5, hp5)
ENDMACRO

MACRO: par (x)
   let N = length (x);
       P = new-cell (undef);
       i = new-cell (0)
   in
   proc-id.NPATHS := N;
   for i = 1 to N do                     //Create N soms.
   ( P := new-proc;
     add-entry (rval(P));                //Make TABLE entry.
     set-prog (rval(P), x[i]);//Set state components.
     set-proc-id (rval(P), [[NPATHS:new-cell(0),
                            FATHER:proc,
                            VALUE:new-cell(undef),
                            LAST:eq(i, N),
                            TABLE:proc-idTABLE,
                            EVAL_STACK:new-stack ]] );
     init-state (rval(P));
     release (rval(P))  );
   set-level-inactive (6)                //Wait for sons to complete.
ENDMACRO

MACRO: unmonitor (p)
   interrupt (p, 1, nil)
ENDMACRO

MACRO: contl (e, n)
   let p_list = new stack;
       p_count = new-cell (0);
       i = new-cell (0)
   in
   lock-table;
   incr-depth (n);                       //Increment cont-depth.
   cont-list (p_list);                   //Form list of processes
```

```
   p_count := length (p_list);      //that are to pause.
   for i = 1 to rval(p_count) do    //Request processes to pause.
         interrupt (p_list[i], 2, proc);
   iff  rval(p_count) > 0  do       //Wait for acknowledgement.
         set-level-inactive (6);
   unlock-table;
   eval (e, prog-id.rest);          //Evaluate x.
   until  eq(0, length(p_list)) do //Allow process to resume.
        (  interrupt (p_list[1], 3, 0); pop (p_list) )
ENDMACRO

MACRO: cont (e)
   cont (e, 1)
ENDMACRO

MACRO: monitor (s, c, r, v, e)
   let M = new-proc                 //Create monitor process.
   in
   add-entry (T);
   set-prog (T, §let VAL = v;       //Set its prog.
                    CONSTRUCT = c
                    S = s;
                    R = r
                in
                until  R(VAL, sel(S, CONSTRUCT)) do null
                contl (e, 0) § );
   init-state (T);
   release (T);
   T
ENDMACRO

MACRO: frp          //p-graph for rp.
   if eq(0, length(proc-id.EVAL_STACK)  //Any evals to finish?
   then ( unless eq(undef, proc-id.FATHER) do     //No,
          ( if proc-id.LAST                       //terminate.
              then  interrupt (proc-id.FATHER, 4, top-stack)
              else  interrupt (proc-id.FATHER, 4, undef));
            rem-entry (proc);
            terminate )
   else ( set-prog-id (proc, top-from(proc-id.EVAL_STACK));
          set-control (proc, top-from(proc-id.EVAL_STACK)))
ENDMACRO

MACRO: hp1                  //P-graph for hp(1).
   § rem-entry (proc);      //Respond to unmonitor order
     terminate  §
ENDMACRO
```

```
MACRO: hp2
   § interrupt (q(2)[1], 5, 0);     //Acknowledge pause request.
     advance (q(2));
     dump(2).aflag := false;        //And, pause.
     set-status (proc, dump(2)) §
ENDMACRO

MACRO: hp3                  //P-graph for hp(3).
   § advance (q(2));        //Continue after cont.
     dump(3).aflag := true;
     dump(3).level := 3;
     set-status (proc, dump(3)) §
ENDMACRO

MACRO: hp4                  //P-graph for hp(4); to accept completion
   § unless  eq(undef, q(4)[1]) do      //notification from sons.
          proc-id.VALUE := q(4)[1];
     advance (q(4));
     proc-id.NPATHS := rval (proc-idNPATHS) - 1;
     iff  eq(0, rval(proc-id.NPATHS)) do//Last son?
        ( dump(4).aflag := true;        //Yes, prepare to resume
          dump(4).level := 3;
          rval (proc-id.VALUE) );       //Place value on stack.
     set-status (proc, dump(4)) §
ENDMACRO

MACRO: hp5                  //P-graph for hp(5).
   § advance (q(5));
     p_count := rval(p_count) - 1;
     iff  eq(0, rval(p_count) do
        ( dump(5).level := 3;
          dump(5).aflag := true );
     set-status (proc, dump(5)) §
ENDMACRO
```

CHAPTER 7

Controlling Process Capabilities and Handling Error Situations

7.1 Introduction

This chapter corrects the weakness in the model noted in Section 4.7 concerning controlled interactions. In addition, it considers the problem of handling error situations in the model.

Section 4.7 observes that although the model does include means to control the effects actions of one process can have on another, the control that can be exerted is very coarse. Extensions to the model which permit finer control to be exerted over the external aspects of process behavior are described in this chapter.

It is useful to distinguish two areas in which control should be exerted: the use of operators which are "inherently external"; and, the use of operands which are "potentially external". Certain of the operations a process can perform are, by their very nature, "external" in that they always affect other processes. The class of inherently external operators includes, for example, the prog-items t-seize,

interrupt and set-level. There are certain other operations a process can perform which may, depending upon the values of their operands, effect other processes. For example, a storage operator with one memory designator as an operand may affect other processes while the same operation with another designator as an operand may not. Such operators are not inherently external. Rather, their operands determine whether their effect is external.

As it is described by Chapters 3 and 4, the model includes no means for restricting the way a process uses particular operators and operands. Section 7.2 describes changes to the model which make it possible to exert control over how a process uses inherently external operators. Section 7.3 discusses an extension which, in effect, makes it possible to restrict the use of potentially external operands. Two examples in Section 7.4 illustrate how, with these extensions to the model, it is possible to specify finer control over process behavior.

Chapters 3 and 4 note certain circumstances in which the actions taken by a process result in error situations. For example, Section 4.2 notes that an error situation occurs whenever a process attempts to add an item to a stack or queue which is already full. Although the possibility of such situations has been noted, nothing has been said that indicates how they are handled when they do occur. Section 7.5 proposes a method for treating error situations.

7.2 Restricted Operators

As the previous section notes, some of the operators processes can perform are inherently external. Let E denote the class of such operators. This section is concerned with the problem of limiting the capabilities of particular processes with respect to members of E. For the present the exact membership of E is unimportant.

The following considerations form the basis for modifications to the model that enable control to be exerted over how members of E are used:

1. There are situations in which some processes require different capabilities than others. For example, in the situation described in Section 4.7 the supervisory process P should be less restricted than its slaves. Therefore, it should be possible to vary, from process to process, the restrictions placed on use of members of E. This suggests that the state of each process define the restrictions placed on use of members of E by the process.
2. It should be possible to place a "continuum" of restrictions on the use of an operator. For example, consider the prog-item set-level. Depending upon the situation, the restrictions placed on its use could range from none at all, to use with operands constrained to be within a certain range (e.g., its second operand, n, might be restricted to $3 \leq n \leq lmax$), to

total prohibition of its use.

3. A question that naturally arises is: Who or what is interested in controlling the capabilities of a process P with respect to E? In the context of the model there is only one answer possible: another process Q. Suppose Q is interested in restricting P's use of an operator $e \in E$. In such a situation an attempt by P to perform e is an event of interest (see Section 2.5) to Q. Q can truly exert control if it is able to intervene whenever P attempts to perform e and, if it chooses, perform e "for" P. This suggests that the interrupt mechanism be used to inform Q of P's attempts to perform e. More generally, it suggests that, for this kind of control to be feasible, attempts by processes to use members of E are events that should be monitored for automatically.
4. It should be possible for a third process R to control Q's use of e. In such a case, an attempt by P to perform e could result in an attempt by Q to perform e ("for" P) which would, in turn, be an event of interest to R.
5. There are situations in which it is useful for a process P to be able to control its own use of e. Suppose, for example, that P is "running" an unreliable program A, perhaps to debug it. If it had the ability to control its own use of certain operators, P could select a set of "critical" operators whose use within

prog A it wished to monitor. P could then run A with the assurrance that whenever one of the critical operators were attempted it could check that it was "safe" to perform the operator before allowing it to be performed. In such situations an attempt by P to perform e is an event of interest to P itself.

6. A natural time to restrict the capabilities of a process is when it is created. Furthermore, it is natural for the creating process to define the restrictions. In addition, it should be possible to modify the restrictions placed on an existing process.
7. It is desirable that, with the exception of the part of its state that specifies restrictions, a process need not be aware of restrictions placed on it. With this property, it is possible to "run" the same program in "debugging" (highly restricted) and "production" (relatively unrestricted) environments without rewriting it.

The specification of restrictions placed on a process' capabilities with respect to E is organized into a new process state component called the restricted operator list (rlist). There is a potential entry in the rlist of a process for each operator $e \in E$. Absence of an entry for e in the rlist component of a process means that use of e by the process is unrestricted. As the model has been described in Chapters 3 and 4, all processes, in effect, have empty rlist components.

The presence of an entry for e means that its use has been restricted. The restrictions placed on e's use are defined by the rlist entry for it which contains two pieces of information: the designator of the process restricting its use and an integer defining an interrupt level.

The restrictions described by the rlist component are enforced by a modification to the state transition rule (see Figures 3.1 and 3.3). The modification is to part 9.1 of Figure 3.3 which reads

interpret the p-graph node specified by pc

The modification is to be described in two parts. The first part describes changes which enable one process to control another's use of particular operators; and, the second part describes additional changes which enable a process to control its own use of particular operators.

The flow diagram in Figure 7.1 illustrates the first part of the modification. Parenthesized numbers (9.1.1 through 9.1.5) appearing in the following discussion refer to parts of Figure 7.1.

Interpretation of a p-graph node n for a process P begins by checking to see whether n is a member of E (9.1.1). If it is not, n is interpreted as before (9.1.2) to complete the state transition. Otherwise, P's rlist component is searched for an entry corresponding to n (9.1.3). If there is no such entry, P's use of n is unrestricted and n is interpreted as

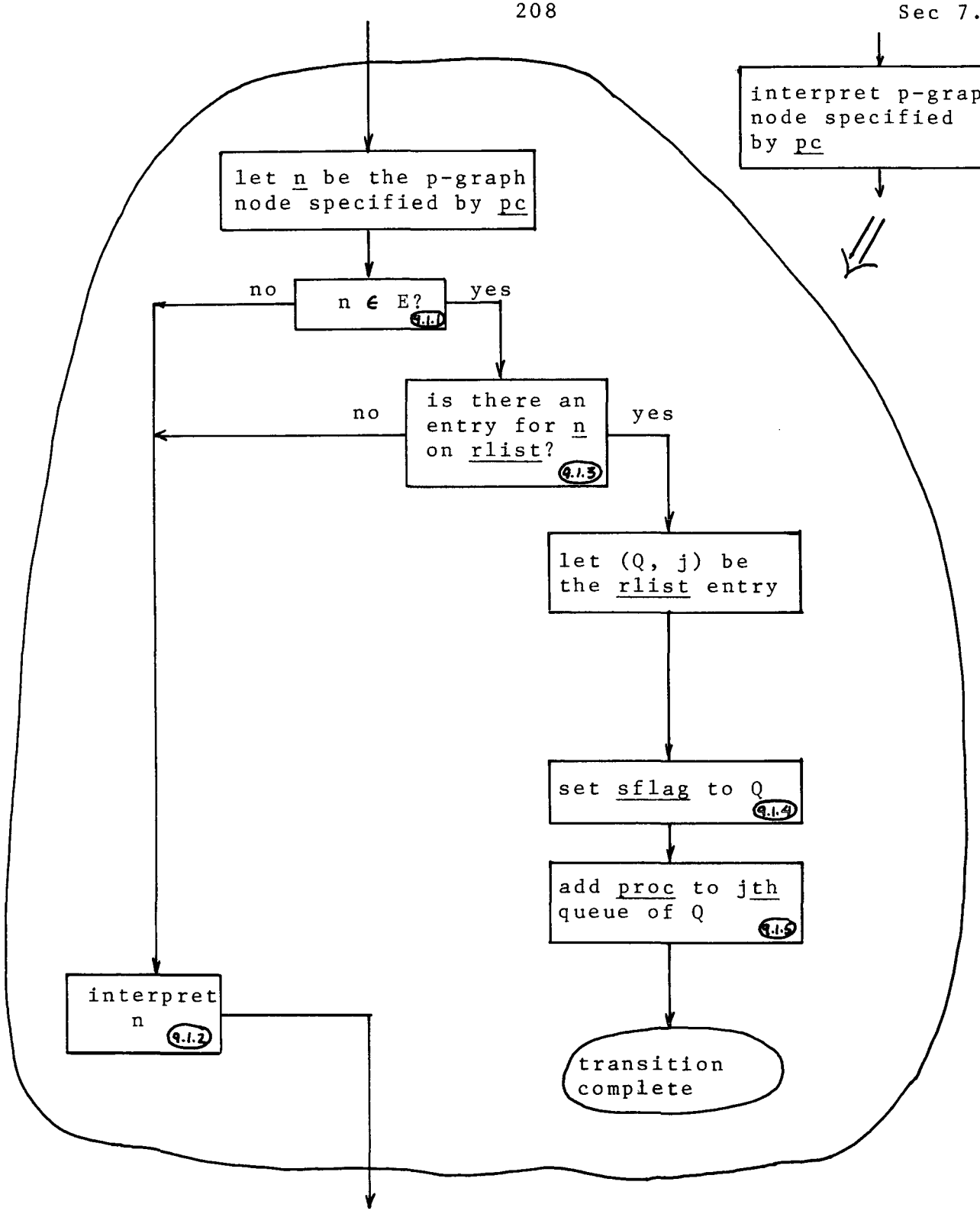


Figure 7.1

Modification to the state transition rule (see Figures 3.1 and 3.3) to enforce the restrictions described by the rlist component. This figure does not include the case Q = proc. Figure 7.2 completes the modification by including that case.

before (9.1.2).

Should an rlist entry for n exist, its use by P is restricted. The entry for n, (Q, j), indicates that process Q controls P's use of n and that an interrupt event of importance j is to occur whenever P attempts n. To complete the state transition, P is seized by Q (9.1.4) and Q receives an interrupt request of importance j consisting of P's process designator (9.1.5).

The seizure of P by Q (9.1.4) serves two purposes:

1. it permits Q to change P's state as it performs n "for" P; and
2. it prevents changes to P's state by processes other than Q until Q completes n for P.

Q acts toward P in a supervisory capacity with respect to operator n. To perform n for P, Q must, of course, be prepared to respond to interrupt requests of importance j; that is, Q's hp(j) state component must be set appropriately. Q can tell which operator P has attempted by examining P's prog and pc.

Note that Q's actions on behalf of P are invisible to P. Therefore, P need not be aware of the restrictions placed on its use of n.

It may be the case that there is an entry (R, k) for n on Q's rlist. In such a case when Q attempts to perform n for P, it is seized by R and R receives an interrupt request of importance k.

A natural extension of the discussion above is to interpret the presence of an entry (P, j) for n on P's rlist as meaning that P is to control its own use of n by way of its own hp(j) component. Suppose P's rlist has such an entry (i.e., in terms of Figure 7.1 suppose Q = P) and consider what happens when P attempts n. The modification to the state transition rule described by Figure 7.1 forces seizure of P (9.1.4) by itself thereby preventing P from proceeding further (part 1 of Figure 3.1). This difficulty can be avoided by checking whether Q is P before setting P's sflag and, if it is, omitting the step that sets it. If j is less than the current value of P's level component, P is interrupted on its next state transition causing interpretation of its hp(j) component to be initiated. However, if j≥level, the presence of an interrupt request in P's q(j) component is unnoticed (part 2 of Figure 3.1) and hp(j), which is to perform n for P, can not be initiated. This suggests that the state transition rule allow the interrupt request to be made only if j<level and take other appropriate action (to be specified momentarily) if it is not. Assuming that j is less than level and therefore that hp(j) is initiated, a problem still remains. The entry for n on P's rlist, which caused hp(j) to be intitiated, prevents hp(j) itself from using n. The solution to this problem is to allow P to perform n when its level is less than or equal to j.

Figure 7.2 completes the modification to the state transition rule. It includes both the changes included in

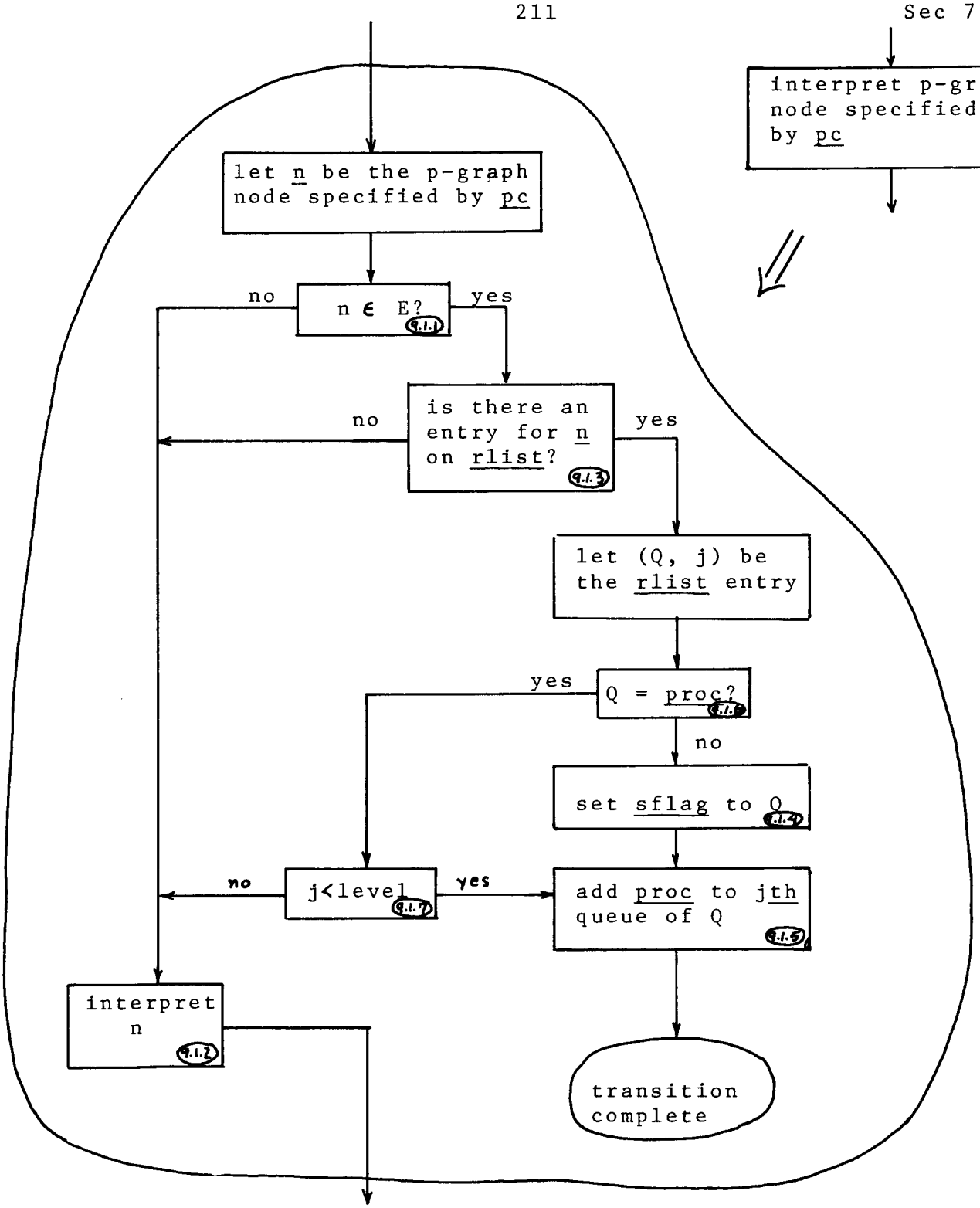


Figure 7.2

Modification to the state transition rule (see Figures 3.1 and 3.3) to enforce the restrictions described by the rlist component.

Figure 7.1 and the changes described above that enable a process to control its own use of particular operators. The presence of an entry (Q, j) for operator n on process P's rlist is interpreted as follows:

1. if Q ≠ P, process Q controls P's use of n; whenever P attempts n, P is seized by Q (9.1.4) and Q receives an interrupt request in its q(j) component (9.1.5).
2. if Q = P, P controls its own use of n; whenever it attempts n
   a. if the value of its level component is greater than j, P receives an interrupt request in its q(j) component (9.1.5) causing its hp(j) component to be initiated.
   b. if its level component is less than or equal to j, it performs n with no interruption (9.1.2).

Figures 3.1, 3.3 and 7.2 together define the model state transition rule. They are reproduced together as Appendix 2.

The ability of a process P to control its own use of an operator n by placing an entry (P, j) for n on its rlist represents a generalization of the notion of master and slave mode frequently found in computer systems. P operates in "master mode" with respect to n when the value of its level component is j or less in the sense that there are no restrictions on its use of n. When its level component is greater than j, it operates in "slave mode" with respect to n and attempts by it to use n are "trapped" by a "master mode

procedure", its hp(j) component, which performs n for it.

In addition to the state transition rule modification, some further changes to the model are required to gracefully accommodate the rlist mechanism.

Process creation, as described previously, is modified slightly to accommodate specification of an rlist component. The new-proc operation is changed to take the rlist intended for the new process as an operand. The value of

new-proc (r)

is the designator of a newly created process whose rlist is r and whose other state components are as described previously in Section 4.6.

It is important that new-proc be a member of E. If it is not, a process could side step restrictions placed on it by using new-proc to create a new unrestricted process to perform its activities.

Because t-seize and release have external effects both should be included in E. However, as defined in Section 3.6, they are incompatible with the restriction mechanism. The source of the incompatibility is that a process can seize and release other processes only for itself. Thus, although process P's rlist has entry (Q, j) for t-seize, Q is unable to perform t-seize for P. The solution to this difficulty is simple: t-seize and release are replaced by t-seize-for and release-for. The effect of

t-seize-for (p1, p2)

is identical to that of

t-seize (p2)

with two exceptions. The first is that if the seizure attempt succeeds the sflag of p2 is set to p1. The second exception has to do with when the sflag of p2 is set (see Section 3.6). If t-seize-for set the sflag immediately, step 9.1.4 of the transition rule could (if p2 is in the midst of attempting a restricted operation) reset the sflag after t-seize-for had set it. To avoid such situations the t-seize-for operation coordinates with p2 such that it does not set p2's sflag or "return" true until after p2 completes its current state transition. Of course, if that state transition should set p2's sflag t-seize-for returns false. To seize p2 for itself a process performs

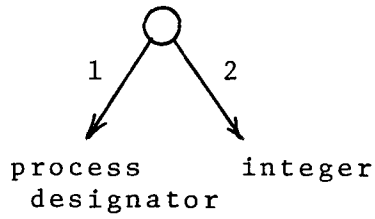
t-seize-for (proc, p2)

The operator release-for works in an analogous manner.

As with the other process state components, processes can directly manipulate rlists. However, effective functioning of the restriction mechanism requires a process be denied arbitrary "write" access to both its own rlist and those of other processes.

It is now necessary to examine in detail the nature of the rlist. The rlist component is the designator for a struct whose selectors are "names" of members of E (i.e., identifiers corresponding to the prog-items of E) and whose components are

designators for rows of the form



Thus, for example,

```
new-proc ( [[ set-level:[[proc,4]],
             set-hp:[[P,4]],
             interrupt:[[Q,6 ]] ])
```

where P and Q are process designators, creates a process with restricted capabilities. Its use of set-level is controlled by the creating process, its use of set-hp by P, and its use of interrupt by Q.

The prog-items add-rlist and rem-rlist, both members of E, strengthen and relax, respectively, the restrictions placed on a process. When

add-rlist (p1, op, p2, n)

is interpreted, the entry [[p2, n]] is added to p1's rlist as the entry for op. As a result process p2 is given control of process p1's use of op. The effect of

rem-rlist (p1, op)

is to allow p1 unrestricted use of op by removing the entry for op from its rlist. For example, the sequence

```
rem-rlist (p1, quote(new-proc));
add-rlist (p1, quote(new-proc), p2, n)
```

causes the entry for new-proc on p1's rlist to be changed to [[p2, n]]. As is the case with the other state component setting operations, for a process to use add-rlist or rem-rlist, p1

must either be its own process designator or the designator of a process it currently has seized. A processs can access the value of its own rlist using the prog-item rlist and the values of other processes' rlist components using rlist-of.

As remarked earlier, the exact membership of E is irrelevant to the mechanism for limiting the capabilities of processes with respect to its members. However, it is clear from the discussion above that, for the mechanism to be effective, E should include

new-proc, rem-rlist, add-rlist

For the purposes of this dissertation E is assumed to include, in addition,

terminate, interrupt, set-level
set-level-inactive, set-status,
restore-dump, set-hp, set-q,
interrupt, t-seize-for, release-for

I feel that use of the state component setting operations not included on the above list can be adequately controlled by the mechanism introduced in the following section.

A different and perhaps more extravagant approach to E's membership would be to allow E to vary from process to process, using the rlist state component to define it. A particular operator would be in E for a particular process if it had an entry in the rlist for that process. If this approach were used, part 9.1.1 of Figure 7.2 could be eliminated.

7.3 Restricted Values

In addition to being able to control the use of inherently external operators, the ability to control the use of potentially external operands is important. For the model the potentially external operands are memory designators (l-values, stack designators, queue designators) and process designators. External effects may result from their use as operands. Section 4.7 observes that whenever the use of such a designator has external effects, the process using it must either have "received" it from or "given" it to another process. Strictly speaking, it is access to the memory elements and processes denoted by such designators which is to be controlled.

The model as it has been described, includes no mechanism to control access to memory elements or processes. This section introduces the notions of restricted type and restricted value and uses them as the basis for a mechanism able to restrict access to memory elements and processes.

Recall that the universe of discourse Ω is partitioned into classes of values called types (Section 4.3). It is possible to associate with each type T of Ω a set of operations O_T whose operands must be members of T. For example, consider the type l-value. Operands for the prog-items rval and t-set must be l-values, as must the first operand of the prog-item store. Therefore, the set associated with type l-value is

$$O_{\text{l-value}} = \{\underline{\text{rval}}, \underline{\text{t-set}}, \underline{\text{store}}{:}1\}$$

where the notation ":1" is used to indicate the relevant operand of store.

To introduce the notion of restricted values, it is useful to think of type T as being defined by the set O_T. A subset of the operation set O_T can then be thought of as defining a class of values which constitute a "sub-type" of T. The class of values T_R defined by the proper subset O_{T_R} of O_T is called a restricted sub-type of T. The members of T_R are restricted values (of type T) in the sense that some but not all operations that can be performed with members of T as operands can be performed with members of T_R as operands. That is, for $v \in T_R$ and $\underline{o} \in O_T$, v can be an operand of $\underline{o}$ only if $\underline{o} \in O_{T_R}$. The subset $O_C = O_T$ defines a sub-type T_C which is T itself. T_C is referred to as the complete sub-type (of T) and members of T_C are called complete values. By convention, complete values of type T and restricted values of type T are collectively referred to as values of type T.

As an example, reconsider the type l-value. The set $O_{\text{l-value}}$, specified above, defines complete l-values, and its subsets

$$O_{\text{l-value}_{R1}} = \{ \underline{\text{rval}} \}$$

and

$$O_{\text{l-value}_{R2}} = \{ \underline{\text{store}}\text{:1}, \underline{\text{t-set}} \}$$

define two restricted sub-types, l-value_{R1} and l-value_{R2}. An l-value of restricted sub-type l-value_{R1} can not be used to "write" the r-value of the cell it denotes. Similarly, one of

sub-type l-value$_{RZ}$ can not be used to "read" the r-value of the cell it denotes. The same cell can be denoted by a complete l-value and by restricted l-values of both sub-types. Each of the l-values defines different access to the cell.

As the universe of discourse has been described, all values are complete. By enlarging Ω to include restricted memory and process designators, it is possible to control access to memory elements and processes. For example, the use of a particular cell by a particular process could be controlled by "giving" that process a restricted, rather than the complete, l-value for that cell. Control of this sort depends upon two properties of the model:

1. there is no means for a process to arbitrarily generate complete values from restricted ones; and
2. operands are checked for type before operations are performed.

The remainder of this section details the restricted sub-types added to Ω and describes operators which generate restricted values from complete values.

Six restricted sub-types are introduced for memory designators, two each for l-values, queue designators and stack designators. They are:

1. read-only l-values defined by the set

   $\{\underline{\text{rval}}\}$

2. write-only l-values defined by the set

   $\{\underline{\text{t-set}}, \underline{\text{store}}{:}1\}$

3. read-only stack designators defined by the set

   {index, length}

4. write-only stack designators defined by the set

   {pop, push:1}

5. read-only queue designators defined by the set

   {index, length}

6. write-only queue designators defined by the set

   {enqueue:1, advance}

Three restricted sub-types are introduced for process designators. They are:

1. read-only process designators, to be used only to "read" values of the state components for processes they designate, defined by the set

   {t-seize-for:2, release-for:2,
   prog-of, pc-of, ...}

2. write-only process designators, to be used only to set the state components for processes they designate, defined by the set

   {t-seize-for:2, release-for:2,
   set-prog:1, set-pc:1, ...,
   add-rlist:1, rem-rlist:1}

3. interrupt-only process designators, to be used only to interrupt the processes they designate, defined by

   {interrupt:1}

The seizure operators appear in the sets defining read-only and write-only process designators because a process must first

seize another before it can "read" or "write" the other's state components.

Clearly, additional sub-types, such as read-interrupt process designators, could be introduced. However, the intent at present is not to discuss which sub-types are most useful, but rather to illustrate how the notion of restricted sub-type can be incorporated into the model. Consequently, no further sub-types are defined.

As before, the prog-items new-proc, new-stack, new-queue and new-cell produce complete values. Restricted memory and process designators are generated from complete ones by the prog-items read-m-des, write-m-des, read-p-des, write-p-des and int-p-des. The value of

read-m-des (m)

where m is a memory designator is a read-only designator for the memory element denoted by m. Similarly, the value of

int-p-des (p)

where p is a process designator is an interrupt-only process designator for the process designated by p. The remaining three prog-items work analogously.

7.4 Examples

7.4.1 Inherited Restrictions

This example illustrates how a process P can insure that its descendents create only processes which are restricted at

least as much as they themselves are. Stated differently, whenever D, a descendent of P, creates a new process E, E can have no more capabilities than D itself has; E "inherits" D's restrictions. D may create processes more restricted than itself. For this example the phrase

E is more restricted than D

is taken to mean

> For each operator o having an entry (Q, j) on D's rlist, E's rlist has a corresponding entry which must be either (Q, j) or (D, k); furthermore, E's rlist may contain entries for operations not found on D's rlist.

There are a number of strategies P might employ. The following one is relatively straightforward and has the advantage that P itself need not perform new-proc for its descendents.

For each "ordinary" process it creates P makes sure that the rlist component contains the entry (S, 2) for new-proc, where S designates a "special" unrestricted process created by P for the sole purpose of performing new-proc for P's descendents. Because P creates S, it can insure that S does nothing but respond to requests to perform new-proc. When S creates a new process for D (a descendent of P) it uses both the rlist specified by D (the top item of D's stack) and D's rlist to build an rlist for the new process which is at least as restrictive as D's. Should the rlist D specifies be less

restrictive than that of D itself, S, nonetheless, creates a process at least as restricted as D. After the new process is created, S sets D's stack, "advances" its pc and releases it.

The macros diff and is-selector appear in the definition of the hp(2) component for S. The value of

diff (R1, R2)

where R1 and R2 designate rows, is the designator for a row whose components are the components of R1 which are not components of R2. The value of predicate

is-selector (x, y)

where x is an identifier and y designates a struct, is true only if x is a selector for the struct designated by y.

The hp(2) component for S is:

```
let D = q (2) [1];              //Proc attempting new-proc.
    DRLIST = rlist-of (D);      //...its rlist.
    DSTACK = stack-of (D);      //...its stack.
    R = DSTACK [1];             //The proposed rlist.
    OPS_R = selectors (R);      //...operators on it.
    DIFF = diff (selectors(DRLIST, OPS_R);
    RLIST = new-cell (nil);     //Rlist to be built
    T = new-cell (undef);
    i = new-cell (0)
in
for i = 1 to length(OPS_R) do  //Add ops on R to RLIST.
  ( if   is-selector (OPS_R[i], DRLIST) ∧
         ne (D, select(R, OPS_R[i])[1])
    then  T := select (DRLIST, OPS_R[i])
    else  T := select (R, OPS_R[i]);
    RLIST := aug-struct (rval(RLIST), OPS_R[i],
                         rval(T))  );

for i = 1 to length(DIFF) do        //Add ops not on R but
 RLIST := aug-row (rval(RLIST),     //on DRLIST to RLIST.
                   DIFF[i], select(DRLIST, DIFF[i])  ;
```

```
T := new-proc (rval(RLIST));  //Perform new-proc
pop (DSTACK);                 //Arrange D's...
push (DSTACK, rval(T));       //...stack
set-pc (D, pc-of(D).next);    //...and pc.
release (D);
advance (q(2));
set-status (proc, dump(2))
```

When S is created, P initializes its status and environment components to

prog = undef	stack = new-stack
pc = 1	prog-id = nil
aflag = false	proc-id = nil
level = 3	rp = undef

and then releases it.

7.4.2 Describing A Supervisory Process

This example illustrates how the situation described in Section 4.7 can be achieved. To review, the situation is:

> Process P creates processes upon request from initial state specifications and subsequently acts in a supervisory manner toward those processes (slaves). Specifications for the processes it creates originate externally to P.

The statement

> P acts in a supervisory manner toward S

is taken to mean

> P is able to exert some degree of control over the actions of S. For example, P can cause S to terminate.

The strategy to be used by P to control its slaves is to reserve interrupt level 1 for interactions with them. And, when it creates a slave, to set the slave's hp(1) component such that the slave responds to its interrupts as P wishes. For this to be a workable strategy P must insure

1. no slave can set its or any other slave's level component to 1;
2. no slave can tamper with its or any other slave's hp(1) component;
3. no slave can remove any of the restrictions placed on it or any other slave by P;
4. no slave can create a process less restricted than itself;
5. whenever a slave creates another process, P is informed (this permits P to act as a supervisor to descendents of slaves); and
6. no slave can seize P.

In addition, although unnecessary, P insures

1. no slave can interrupt another process on level 1;
2. whenever a slave terminates, P is informed.

To implement this strategy P makes an rlist component for each slave it creates which includes the entry

$$[\![\ \text{int-p-des}\ (P),\ 2\ ]\!]$$

for each of the prog-items

new-proc, terminate, rem-rlist, set-level,
set-hp, set-hp-inactive, set-status, restore-dump

This insures that

1. it controls how slaves use these prog-items; and
2. slaves can neither read nor write its state.

P's hp(2) component must, of course, be prepared to perform the above operators for the slaves.

Definitions for fragments of P's hp(2) component corresponding to each of the prog-items follow. The definitions make use of the macros:

1. remove(Q), which removes the process designator Q from the "process table" P maintains. (The "process table" is a record of all slaves and descendents.)
2. npop (S, i), which pops i items from the stack designated by S.
3. seize-from (p1, p2), which seizes process p2, currently seized by p1.
4. reseize-for (p1, p2), which releases process p2 and seizes it for p1.
5. is-on (r, c), whose value is true if c is a component of row L and false otherwise.

and the identifiers:

1. S, the designator of the slave process attempting to perform a restricted prog-item.
2. SSTACK, the stack component of S.
3. OP_LIST, a row whose components are the seven prog_items restricted by P (see above).
4. T, a cell used for temporary storage.

new-proc fragment: same as hp(2) for "special" process S from the previous section with the exception that an entry is made for the new process in the "process table" P maintains.

terminate fragment:

```
remove (S);                                  //Remove S from
                                             //"process table".
rem-rlist (S, quote(terminate));             //Allow S to do terminate.
release-for (proc, S)                        //Release S so that it
                                             //can perform terminate.
```

rem-rlist fragment:

```
unless is-on(OP_LIST, SStack[2]) do          //Unless prog-item
  ( T := eq (SSTACK[1],S)                    //in question is one
    unless rval(T) do                        //restricted by P,
       seize-from (S, SSTACK[1]);            //do the rem-rlist.
    rem-rlist (SSTACK[1], SSTACK[2]);
    unless rval(T) do
       reseize-for (S, SSTACK[1]));
npop (SSTACK, 2);                            //Set S's stack
set-pc (S, pc-of(S).next);                   //and pc.
release-for (proc, S)
```

set-hp fragment:

```
unless eq(1, SSTACK[2]) do                   //Unless attempt is to
  ( T := eq (SSTACK[1], S);                  //set hp(1), do it.
    unless rval(T) do
       seize-from (S, SSTACK[1]);
    set-hp (SSTACK[1], SSTACK[2], SSTACK[3]);
    unless rval(T) do
       reseize-for (S, SSTACK[1]));
npop (SSTACK, 3);                            //Set S's stack
set-pc (S, pc-of(S).next)                    //and pc.
release-for (proc, S)
```

set-level fragment:

```
unless eq(l, SSTACK[2]) do          //Unless attempt is to
   ( T := eq (SSTACK[l], S)         //set level to l, do it.
     unless rval(T) do
       seize-from (S, SSTACK[l]);
     set-level (SSTACK[l], SSTACK[2]);
     unless rval(T) do
       reseize-for (S, SSTACK[l]));
 npop (SSTACK, 2);
 set-pc (S, pc-of(S).next);
 release-for (proc, S)
```

set-level-inactive, set-status and restore-dump fragments are analogous to the set-level fragment.

interrupt fragment:

```
unless eq (l, SSTACK[l]) do          //Unless interrupt is
     interrupt (SSTACK[l],           //for level l, do it.
                SSTACK[2], SSTACK[3]);
npop (SSTACK, 3);
set-pc (S, pc-of(S).next);
release-for (proc, S)
```

Note that whenever a slave S tries to use a restricted prog-item in a way not permitted to it, P merely sets S's state as if it had performed the operation without actually performing it. In a "real" application P would probably inform S that its attempt to do something forbidden to it failed.

7.5 Handling Errors

This section proposes an error handling facility for the model.

Although the number of potential error situations is enormous, all share the property that they occur when a process

attempts something not permitted. In that sense, an error situation is similar to the situation resulting when a process attempts to perform a restricted operation. Both occur when a process attempts something it is unable to do. This suggests that error situations be treated analogously to attempts to perform restricted operators.

A method for handling error situations is to introduce the "fictitious", inherently external operator error. An entry for error in the rlist component of a process defines a process assigned to handle error situations for the process. The occurrence of an error situation is to be regarded as an attempt by the process to perform error. Each possible error situation has a code assigned it.

An error situation occurring in a process P is handled as follows. If P's rlist does not include an entry for error, P is terminated. If there is an entry (Q, j) for error

1. P is seized by Q; and
2. Q receives an interrupt request of importance j consisting of

   [[proc:P, code:n]]

   where n is the code assigned for the particular error.

Q, if it is able to correct the error situation, can set P's state accordingly and release it. If it can not, it can either request assistance from another process or deem the situation hopeless and force P to terminate.

CHAPTER 8

Concluding Remarks

8.1 Summary

This dissertation has investigated the problem of representing groups of loosely connected processes. The goal of the investigation was the development of a method for process representation useful for synthesizing process behavior patterns. The method developed allows specification of both the internal structure of individual processes and the "interface" structure between processes which interact. Presentation of the investigation can be divided conceptually, if not chronologically, into three phases.

The first phase (Chapter 2, Section 4.7, parts of Chapter 7) isolated the concepts which were to form the basis for the process representation technique. In it questions such as

What is a process?

What is an event?

What must processes do to interact?

Should all processes be equally capable?

were considered. Attempting to answer fundamental questions such as these focused attention on aspects intrinsic to processes and interactions among them.

The second phase (Chapters 3, 4, 7) exhibited a specific realization of the concepts developed in the first phase. A model was developed that captures the essential aspects of the process notion. The model is a synthetic one in the sense that it is designed to host descriptions of patterns of process behavior. In the course of defining the model certain decisions concerning the detailed composition of its universe of discourse and the detailed properties of its virtual memory were made. To a large extent the features of the model that result from those decisions are orthogonal to the ones enabling it to synthesize groups of interacting processes. For example, the model's ability to support descriptions involving interactions by way of interrupt events is independent both of the fact that its virtual memory includes cells and of the fact that its universe of discourse has dynamic types. Had such decisions been made differently, the result would have been a different, but possibly equally useful, realization of the concepts developed in the first phase.

The third phase (Section 5.3, Chapter 6, Section 7.4) demonstrated by example that the model for process synthesis is indeed a useful one for synthesizing process behavior patterns. In it the model was used to define a variety of interesting process behavior patterns, including some that have achieved prominence in the literature.

8.2 Areas for Extending the Research

8.2.1 Extensions and Changes to the Model

Many areas touched upon in this dissertation suggest questions which remain unanswered. Perhaps the most significant contribution of the dissertation is the conceptual framework it provides for discussing such questions. Among the more interesting questions which merit further investigation are the following:

1. State structure:

Should all processes have the same state structure? If not, how can the model be modified to cope with processes having variable state structure? Clearly, as the model has been defined the state structure for all processes is the same. The proc-id component, in a sense, represents a small step toward processes with variable state structure. Consider, for example, the proc-id component for processes that use the eval macro (see Section 5.3.3). It is meaningful to think of the identifier EVAL_STACK as a separate state component; its function is to keep track of unfinished eval's.

2. Process seizure:

The notion of process seizure (see Section 3.6) is somewhat heavy handed. Would some sort of "automatic" seizure and release which occurs as part of a set-xxx operation be better? Or, perhaps better yet, would a notion of "partial"

seizure, which effects only specified parts of a process state, be useful? What other changes to the model, if any, would a different treatment of seizure require?

3. Retained objects:

Dennis and Van Horn [Dns66] use the term "retained object" to refer to an item such as a file which normally exists longer than the process which creates it. Operating systems provide various mechanisms for dealing with retained objects. Can the model, as it is, gracefully handle the notion of retained object? If so, how? If not, how should it be extended to do so? The ability to discuss retained objects would enable it to describe groups of processes operating in the environment of a file system. What would be appropriate rules for processes to follow in manipulating retained objects?

4. Ownership:

As the model has been described, the notion of ownership is present in a weak sense. For example, a process which allocates a stack (using new-stack) can be thought of as "owning" it. By passing to some processes the designator for the stack and, to others, restricted designators for it, it can control which processes access the stack and to a certain extent how they access it. However, once another process has a designator for the stack the first process can no longer control how the other process uses it. Would it be meaningful to strengthen the notion of ownership in the model? If so, in what way? In the context of retained objects ownership appears

to be a more meaningful notion. Who or what should "own" retained objects? Ownership should probably not reside in processes since, in general, retained objects outlast the processes which create them. What does ownership of an item mean? What special privileges does the owner have? Can an item have more than a single owner?

5. Protection and security:

The model includes features which make it possible to limit the capabilities of processes with respect to certain operators and certain kinds of operands. Consider the following situation (described by Lampson [Lam69]):

> A and B are competitors. A has a program P which it is willing to let B use for a fee. B has some data D which it would like to use with P. B is willing to pay A for use of P but since A is a competitor, it would like to be sure that A can not use P to "read" either D or the results produced when P runs with D. A is willing to let B use P but would like to be sure that B can not "steal" P and that it can charge B for each use of P.
>
> Provide an environment which satisfies these requirements.

Can the model, as it is, synthesize such an environment? Can such an environment be proven to satisfy the requirements? If such an environment can not be synthesized, what extensions to the model would enable it to be? Vanderbilt [Van69] has

investigated problems of this sort; can the model be gracefully extended to include his results?

6. Storage reclamation:

The model provides for explicit allocation of memory objects from a "storage pool" but includes no means to explicitly return cells, stacks or queues to the pool when they are no longer needed. As the model has been defined, the automatic reclamation of memory objects would probably require a marking garbage collector [Knu68]. All process would probably be forced to "pause" for at least part of the duration of the garbage collection. Is it possible to tell without marking and without forcing processes to pause which memory objects can be reclaimed? How could the model be modified to permit explicit "release" of memory objects? What should happen in the event one process "releases" a memory object also accessible to others? How would an explicit storage reclamation mechanism interact with the notion of ownership?

8.2.2 Relating the Model to Analytic Models

The model has been developed to serve as a synthetic tool. Given a description in terms of the model of a particular pattern of process behavior, it would be useful to be able to analyze the behavior for properties such as determinacy, deadlock freeness and output functionality. The specific properties of interest would probably depend upon the particular behavior pattern under consideration. What would be

required to perform such an analysis? Certainly a description in terms of the model itself would be too complex to lend itself to direct analysis. Could it be "compiled" into a simpler form more tractable for analysis? What would be an appropriate form? Could existing theoretical results be used? Probably not without considerable modification since most such results are obtained assuming an uninterpreted model (see Section 1.2.3). What would be an appropriate model? What are the relevant theorems and decision procedures? The work reported by Van Horn [VH66], in which he partially specifies how a "realistic" computing facility can be related to a simple analytic model that he has developed and studied, exemplifies the sort of work being suggested here.

8.2.3 The Model as the Basis for a Language Extension Facility

The possibility of a language extension facility that uses the model as a semantic base merits investigation. The kind of facility I have in mind would treat the semantics for the base language and extensions to it uniformly. Semantics for a language feature, whether it represents an extension or is part of the base, would be defined in terms of transformations it produces upon a process state or states. The base language would be a collection of pre-defined features. Extensions to the base would be collections of user-defined features, defined using the same formalism used to define the base.

This approach to language extension is to be contrasted with the one usually proposed in the literature which is to specify new constructs in terms of ones existing in the base language. The notion of the state of an evaluating mechanism for the language, if present at all, is only weakly so. The extender can "get his hands on" the evaluating mechanism only indirectly through existing base language features and as a result the extensions he can make are limited to "syntactic sugaring" of existing constructs: hence the term "syntactic extension".

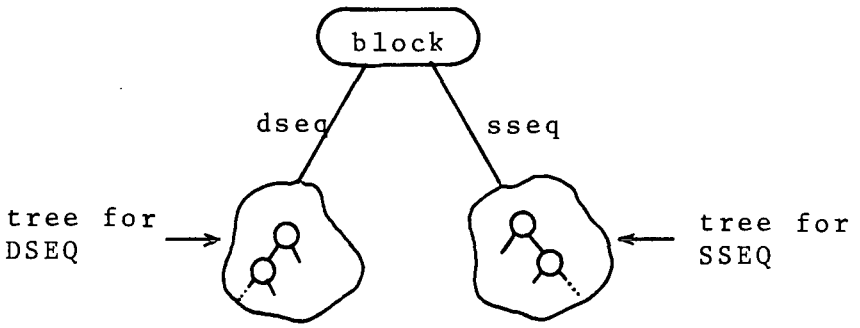
On the other hand, if the extender can make direct reference to the evaluating mechanism and its state in making extensions, the language features he can define are not so severely limited. Chapters 5, 6 and 7 have demonstrated how a variety of linguistic constructs and control patterns, which are not (to the best of my knowledge) expressible in contemporary extensible languages, can be defined in terms of the model.

The following is a sketch of the form such an extensible language facility might take.

Syntactic specification for a language feature would be made using a variant of BNF to describe both concrete (string) and abstract (tree) representations (see Section 1.2.1) for the feature. For example, the "production"

S ::= begin [DSEQ:dseq] ; [SSEQ:sseq] end nodetype block

where S (statement), DSEQ (declaration sequence) and SSEQ (statement sequence) are previously defined non-terminal symbols, describes both the concrete and abstract representations for "block" statements. The concrete representation of a block statement is a string starting with begin, followed by a sequence of declarations (DSEQ), followed by a semicolon (;), followed by a sequence of statements (SSEQ) and terminating with end. The abstract representation is a tree whose root is a node of type block (nodetype block) from which two branches depart. One branch, selected by dseq, leads to the abstract representation for the declaration sequence (DSEQ) and the other, selected by sseq, to the abstract representation for the statement sequence (SSEQ). The tree for a block statement can be graphically displayed:



The method for semantic specification is based on the view that the realization of the computation described by a program is accomplished by translation followed by interpretation. Translation is the creation of an initial process state "corresponding" to the program and interpretation, the repeated transition from state to successor state starting from that

initial state.

The semantic specification of an extension would consist of two parts. One part would describe the contribution an instance of the extension in a program is to make to the prog component of the initial process state. It would be specified by exhibiting a fragment of p-graph corresponding to the extension. The p-graph fragment would be described in terms of the abstract representation for the extension using a PGL-like language (see Chapter 5). The other part would describe the contribution, if any, the extension is to make to other components of the initial state.

As an example, reconsider the "block" statement described above and assume that the extended language is to include both ordinary and Secret variables (see Section 6.2). Part of the semantic specification for the "block" statement would be the p-graph fragment:

```
SEMANTICS (block) =
   push (proc-id.P_ID, prog-id);
   set-prog-id (proc, [top:[NLAYER:new-cell(nil),
                           SLAYER:new-cell(nil)],
                       rest:prog-id.rest]);
   SEMANTICS (block.dseq);
   set-prog-id (proc, [top:rval(SLAYER),
                       rest:[top:rval(NLAYER),
                             rest:prog-id.rest]]);
   SEMANTICS (block.sseq);
   set-prog-id (proc, top-from(proc-id.P_ID))
```

where SEMANTICS (block.dseq) and SEMANTICS (block.sseq) are the p-graph fragments contributed to the initial state by the declaration and statement sequences, respectively. (This

specification is essentially that given in Section 6.2 for the macro block; note that an "enddec" statement is unnecessary.) To complete the semantic specification for this extension it would be necessary to indicate that the proc-id component of the initial state bind the identifier P_ID to a stack designator.

The above discussion has suggested in a very rough way how the model might serve as a semantic base for an extensible language. Many details have been omitted. Aside from the missing details, a number of interesting questions arise. How would the schemes for data type extension found in languages such as BASEL [Che68] fit in with such an approach? It is quite likely that the universe of discourse and the virtual memory for the model described in this disseration would not be entirely satisfactory for the language extension application. What would an appropriate universe of discourse be? What properties would an appropriate virtual memory have? Should other changes to the model be made to make it more suitable for the language extension application? Standish [Sta69] has noted an interesting question in connection with language extension concerning the compatibility of extension packages. Suppose that a facility of the type described above is realized and falls into widespread use. It is likely that a mode of usage will evolve in which users make extensions to their own already extended version of the base language by using extension packages from a library of extensions. What can be done to

ease the problem of incompatibility between different extension packages? As a simple example of the kind of incompatibility that can arise, suppose that a particular programming application requires both the non-deterministic programming extension package (similar to that described in macro form in Section 6.6) and the parbegin-semaphore extension package (similar to that described in Section 6.6). Both packages use level 3 but for different purposes. Is it necessary to start from scratch to define a composite package which resolves this incompatibility? Or, is it possible by some means to automatically or semi-automatically resolve it without completely redefining one or both of the packages?

These questions and others like them deserve careful investigation for the answers to them should lead to languages which are truly extensible.

APPENDIX 1

Summary of Prog-Items

This appendix is an alphabetical list of the prog-items discussed in the dissertation. The entry for each prog-item consists of

1. a description of the operands (if any) taken by the prog-item;
2. a brief description of the effect of the prog-item;
3. reference to sections of the dissertation where a more complete discussion of the prog-item is to be found.

The following conventions are used for specifying operands of prog-items:

1. n, n1, n2 indicate integers;
2. tv, tv1, tv2 indicate truthvalues;
3. id, id1, ..., idn indicate identifiers;
4. pi indicates a prog-item;
5. qd indicates a queue designator;
6. pd, pd1, pd2 indicate process designators;
7. rd indicates a row designator;
8. sd, depending upon the context, indicates either a struct designator or a stack designator;
9. cd indicates a cell designator (l-value);
10. md indicates a memory designator; and

11. v, v1, ..., vn indicate arbitrary values.

add-rlist (pd1, pi, pd2, n):
adds the entry [[pd2, n]] for prog-item pi to the rlist of the process designated by pd1. (7.2)

advance (qd):
removes the item at the front of the queue designated by qd. (4.2)

aflag-of (pd):
the value of the aflag component of the process designated by pd. (4.6)

and (tv1, tv2):
produces the boolean "and" of tv1 and tv2. (4.3)

aug-row (rd, v):
a constructor for rows; produces the designator for a row identical in all respects to the row designated by rd with the single exception that it has one more component which is v. (4.4)

aug-struct (sd, id, v):
a constructor for structs; produces the designator for a struct identical in all respects to the struct designated by sd with the single exception that it has one more component which is v and is selected by id. (4.4)

bind (id, v):
binds id to v in the top id-layer of the process prog-id. (4.5)

binding (id, sd):
the value to which id is bound in prog-id struct designated by sd. (3.5, 4.5)

control:
produces the designator for a struct whose components are the current values of the process prog and pc components. (4.6)

control-of (pd):
produces the designator for a struct whose components are the values of the prog and pc components of the process designated by pd. (4.6)

divide (n1, n2):
produces the quotient n1/n2. (4.3)

do (pi):
causes the prog-item pi to be interpreted. (4.3)

dump (n):
the value of the nth component of the process dump component. (4.5, 4.6)

dump-of (pd, n):
the value of the nth component of the dump component of the process designated by pd. (4.5, 4.6)

enqueue (qd, v):

adds v to the back of the queue designated by qd. (4.2)

env:

produces the designator for a struct whose components are the current values of the process stack, prog-id, proc-id and rp components. (4.6)

env-of (pd):

produces the designator for a struct whose components are the values of the stack, prog-id, proc-id and rp components of the process designated by pd. (4.6)

eq (v1, v2):

a predicate; true if $v1 = v2$, false otherwise. (4.3, 4.4)

ge (n1, n2):

a predicate; true if n1 $\geq$ n2, false otherwise. (4.3)

gr (n1, n2):

a predicate; true if $n1 > n2$, false otherwise. (4.3)

hp (n):

the nth component of the process handler programs (hp) component. (4.5, 4.6)

hp-of (pd, n):

the nth component of the handler programs (hp) component of the process designated by pd. (4.5, 4.6)

index (d, n):

for 0<n<length(d) produces the nth component or item of the row, stack or queue designated by d; otherwise produces undef. (4.2, 4.4)

interrupt (pd, n, v):

causes an interrupt event to occur by placing v in the q(n) component of the process designated by pd. (3.4)

int-p-des (pd):

produces an interrupt-only process designator from the complete process designator pd. (7.3)

is-ident (v):

a predicate; true if v is an identifier, false otherwise. (4.3)

is-int (v):

a predicate; true if v is an integer, false otherwise. (4.3)

is-lval (v):

a predicate; true if v is an l-value (cell designator), false otherwise. (4.3)

is-nil (v):

a predicate; true if v is nil, false otherwise. (4.4)

is-proc (v):

a predicate; true if v is a process designator; false otherwise. (4.3)

is-prog-item (v):

a predicate; true if v is a prog-item, false otherwise. (4.3)

is-queue (v):

a predicate; true if v is a queue designator, false otherwise. (4.3)

is-row (v):

a predicate; true if v is a row designator, false otherwise. (4.3)

is-stack (v):

a predicate; true if v is a stack designator, false otherwise. (4.3)

is-struct (v):

a predicate; true if v is a struct designator, false otherwise. (4.3)

is-truthval (v):

a predicate; true if v is a truthvalue, false otherwise. (4.3)

is-undef (v):

a predicate; true if v is undef, false otherwise. (4.3)

le (n1, n2):

a predicate; true if $n1 \leq n2$, false otherwise. (4.3)

length (d):

for d a row designator, the number of components of the

designated row; for d a stack or queue designator, the number of items in the designated stack or queue. (4.2, 4.4)

level:
the value of the process level component. (4.5, 4.6).

level-of (pd):
the value of the level component of the process designated by pd. (4.5, 4.6)

minus (n1, n2):
produces the difference n1-n2. (4.3)

ne (v1, v2):
a predicate; true if v1 ≠ v2, false otherwise. (4.3)

new-cell (v):
produces the cell designator (l-value) of a newly allocated cell which has been initialized to v. (4.2)

new-proc (sd):
produces the process designator for a newly created process whose rlist component is sd. (4.6, 7.2)

new-queue:
produces the queue designator of a newly allocated queue which is empty. (4.2)

new-stack: produces the stack designator of a newly allocated stack which is empty. (4.2)

not (tv):

produces the boolean compliment of tv. (4.3)

or (tv1, tv2):

produces the boolean "or" of tv1 and tv2. (4.3)

pc:

the value of the process pc component. (4.5, 4.6)

pc-of (pd):

the value of the pc component of the process designated by pd. (4.5, 4.6)

plus (n1, n2):

produces the sum n1+n2. (4.3)

pop (sd):

removes the top item from the stack designated by sd. (4.2)

proc:

the process designator of the requesting process. (4.6)

proc-id:

the value of the process proc-id component. (4.5, 4.6)

proc-id-of (pd): the value of the proc-id component of the process designated by pd. (4.5, 4.6)

prog:

the value of the process prog component. (4.5, 4.6)

prog-of (pd):
the value of the prog component of the process designated by pd. (4.5, 4.6)

prog-id:
the value of the process prog-id component. (4.5, 4.6)

prog-id-of (pd):
the value of the prog-id component of the process designated by pd. (4.5, 4.6)

push (sd, v):
causes v to be inserted at the top of the stack designated by sd. (4.2)

q (n):
the value of the nth component of the process queues (q) component. (4.5, 4.6)

q-of (pd, n):
the value of the nth component of the queues (q) component of the process designated by pd. (4.5, 4.6)

quote (v):
when the prog-item quote is interpreted, the action taken is to push the "next" p-graph item onto the process stack and advance the pc two nodes; if v is an identifier, quote prevents it from being interpreted with respect to the process prog-id; similarly, if it is a prog-item, quote prevents it from being interpreted. (3.5, 4.3)

read-m-des (md):
produces a read-only memory designator from the memory designator md. (7.3)

read-p-des (pd) produces a read-only process designator from the process designator pd. (7.3)

release (pd):
superceded by release-for (see Section 7.2); equivalent to release-for (proc, pd). (3.6)

release-for (pd1, pd2):
releases the process pd2 which is currently seized by process pd1. (7.2)

rem-rlist (pd1, pi):
removes the entry for prog-item pi from the rlist of the process designated by pd1. (7.2)

restore-dump (n):
sets the process status components to the values found in the nth component of the process dump. (3.4)

rlist:
the value of the process rlist component. (7.2)

rlist-of (pd):
the value of the rlist component of the process designated by pd. (7.2)

row (n, v1, ..., vn):

a constructor for rows; produces the designator for a row of length n whose n components are v1, ..., vn. (4.4)

rp:

the value of the process reserve program (rp) component. (4.5, 4.6)

rp-of (pd):

the value of the reserve program (rp) component of the process designated by pd. (4.5, 4.6)

select (sd, id):

produces the "id" component of the struct designated by sd; if the struct has no such component, select produces undef. (4.4)

selectors (sd):

produces the designator for a row whose components are the selectors of the struct designated by sd. (4.4)

set-aflag (pd, tv):

sets the aflag component of the process designated by pd to tv. (4.6)

set-control (pd, sd):

sets the control components of the process designated by pd to the components of the struct designated by sd. (4.6)

set-dump (pd, n, sd):

sets the nth component of the dump of the process designated by

pd to sd. (4.6)

set-env (pd, sd):
sets the environment components of the process designated to the components of the struct designated by sd. (4.6)

set-hp (pd, n, rd):
sets the nth component of the handler programs (hp) component of the process designated by pd to rd. (4.6)

set-level (pd, n):
sets the level component of the process designated by pd to n. (4.6)

set-level-inactive (n):
simultaneously sets the process aflag to false and the level to n. (4.6)

set-pc (pd, n):
sets the pc component of the process designated by pd to n. (4.6)

set-proc-id (pd, sd):
sets the proc-id component of the process designated by pd to sd. (4.6)

set-prog (pd, rd):
sets the prog component of the process designated by pd to rd. (4.6)

set-prog-id (pd, sd):
sets the prog-id component of the process designated by pd to sd. (4.6)

set-q (pd, n, qd):
sets the nth queue of the queues (q) component of the process designated by pd to qd. (4.6)

set-rp (pd, rd):
sets the reserve program (rp) component of the process designated to rd. (4.6)

set-stack (pd, sd): sets the stack component of the process designated by pd to sd.
(4.6)

set-status (pd, sd):
sets the status components of the process designated by pd to the values of the components of the struct designated by sd.
(4.6)

spop:
removes the top item from the process stack component. (4.5)

stack:
the value of the process stack. (4.5, 4.6)

stack-of (pd):
the value of the stack component of the process designated by pd. (4.5, 4.6)

status:

produces the designator for a struct whose components are the current values of the process prog, pc, aflag and level components. (4.6)

status-of (pd):

produces the designator for a struct whose components are the value of the prog, pc, level and aflag components of the process designated by pd. (4.6)

store (cd, v):

sets the r-value of the cell designated by cd to v. (4.2)

struct (n, id1, v1, ..., idn, vn):

a constructor for structs; produces the struct designator for a struct having n components v1, ..., vn selected by the selectors id1, ..., idn. (4.4)

terminate:

causes a process to cease to exist. (4.6)

times (n1, n2):

produces the product n1*n2. (4.3)

t-seize (pd):

superceded by t-seize-for (see Section 7.2); equivalent to t-seize-for (proc, pd). (3.6)

t-seize-for (pd1, pd2):

a predicate with a side effect; an attempt is made to seize

the process designated by pd2; if it succeeds, the sflag of pd2 is set to pd1 and the predicate produces true, otherwise it produces false. (7.2)

t-set (cd):
predicate with a side effect; the r-value of the cell designated by cd is examined; if the r-value is 0, it is set to 1 and the predicate produces true; otherwise the predicate produces false. (4.2)

unbind (id):
causes id to become unbound in the top id-layer of the process prog-id. (4.5)

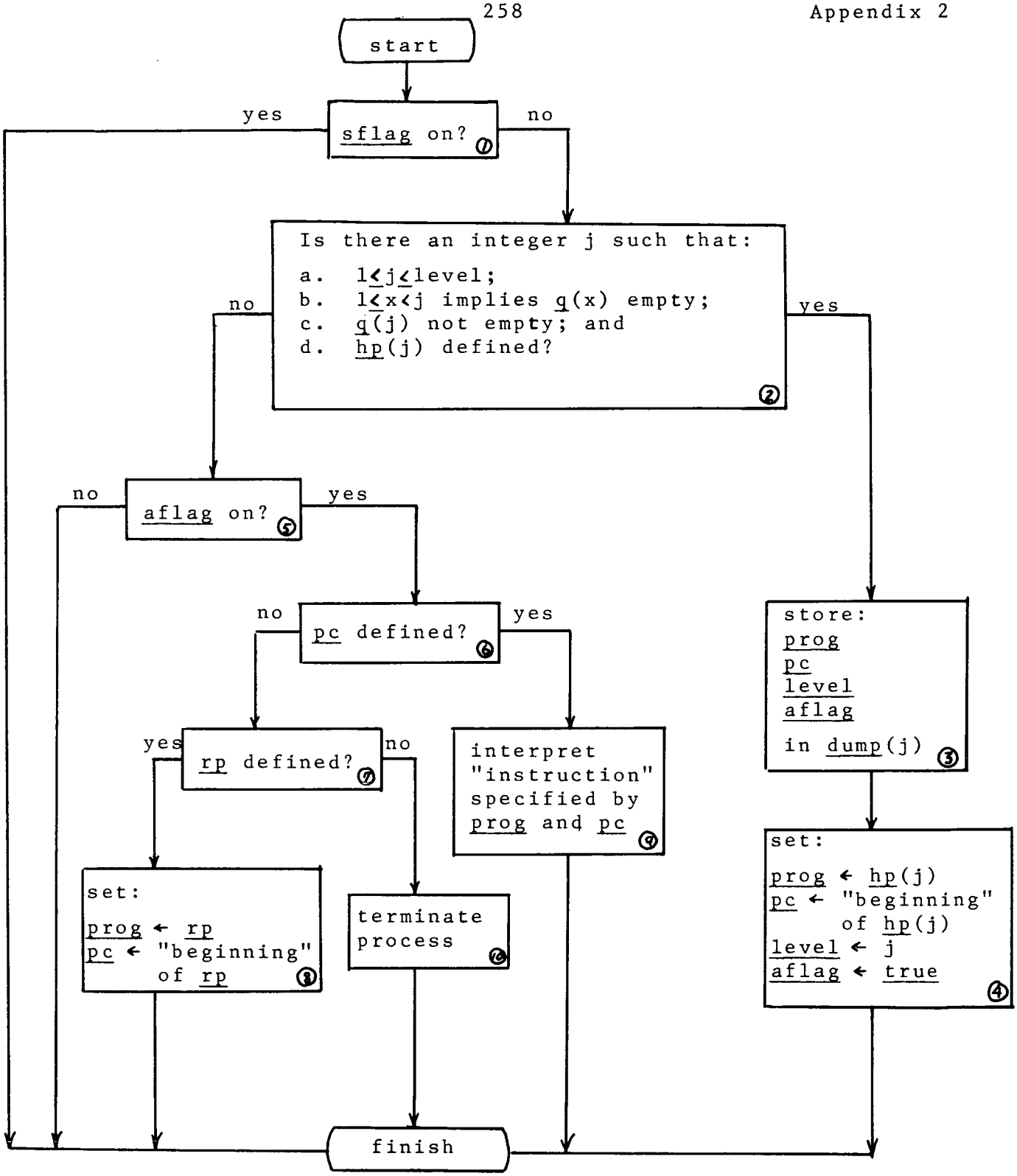
write-m-des (md):
produces a write-only memory designator from the memory designator md. (7.3)

write-p-des (pd):
produces a write-only process designator from the process designator pd. (7.3)

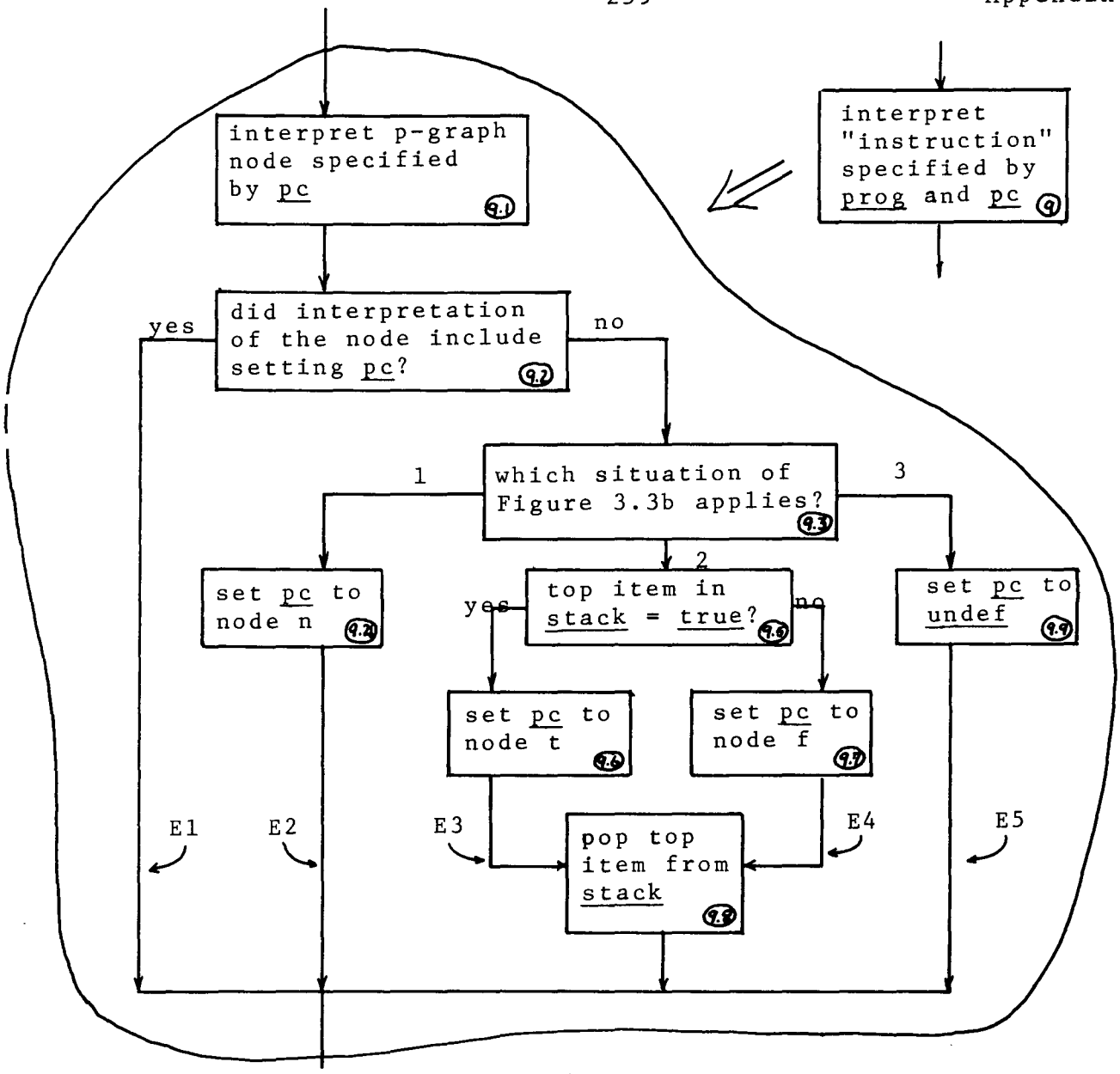
APPENDIX 2

The State Transition Rule

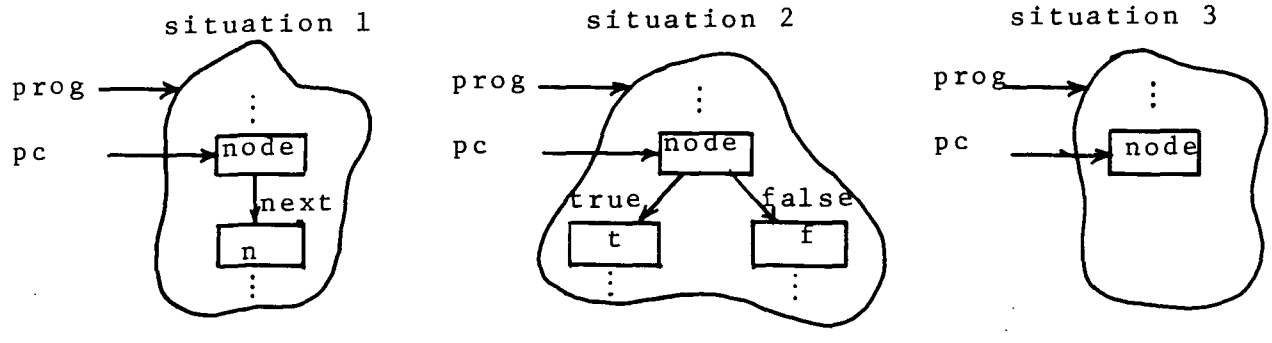
The state transition rule for the model is defined by Figures 3.1, 3.3 and 7.2. Those figures are reproduced together in this appendix.



The State Transition Rule.

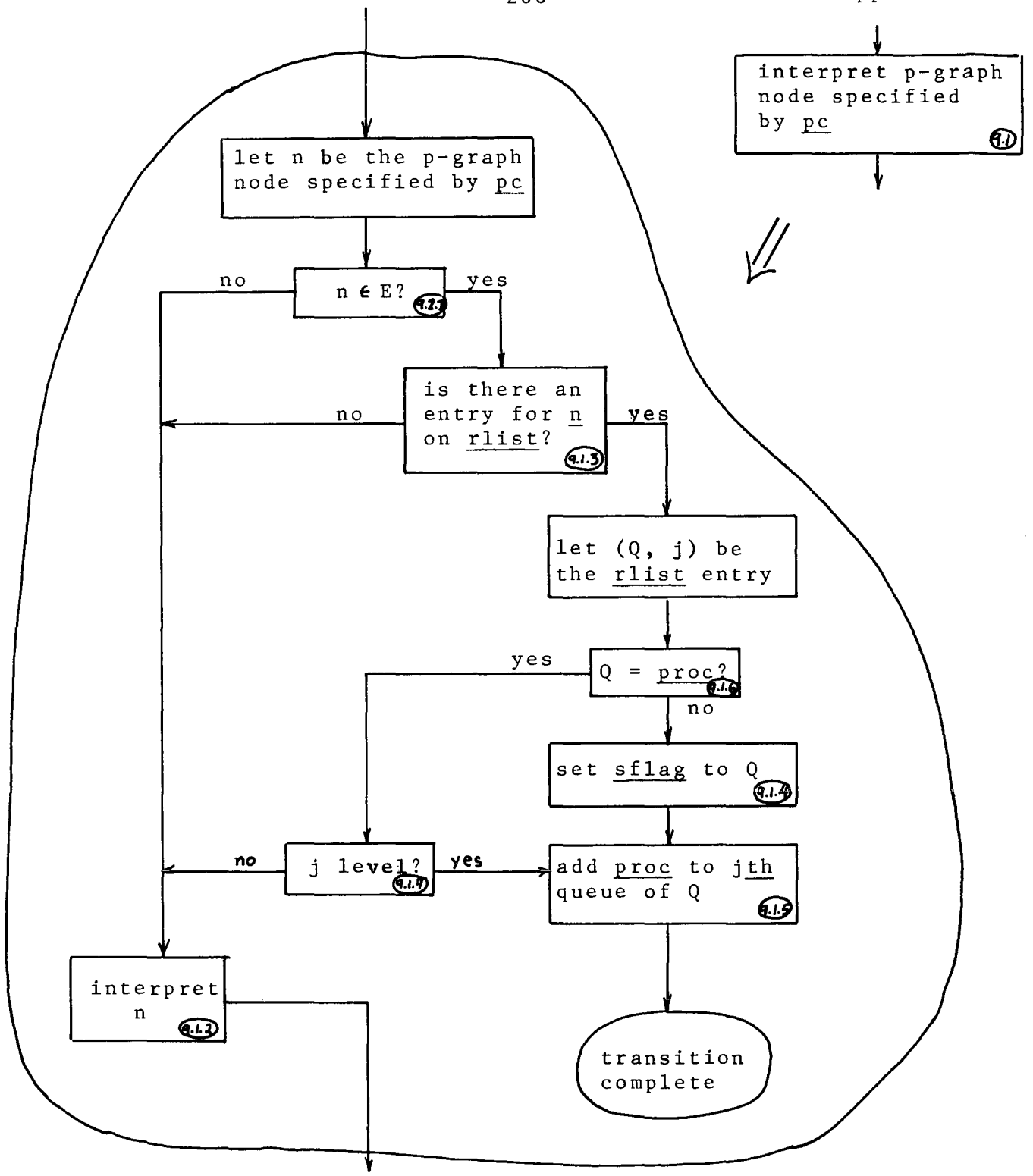


(a)



(b)

Part 9 of the state transition rule (see Figure 3.1)



Part 9.1 of the state transition rule

References

[An65] J.P. Anderson, "Program Structures for Parallel Processing", Comm. ACM Vol 8, No 12 (Dec 1965) pp 786-788.

[BBN70] Bolt Beranek and Newman Inc, "TENEX Technical Manual", Jan 1970.

[Be70] D. Beech, "A Structural View of PL/I", Computing Surveys, Vol 2, No 1 (March 1970), pp 33-64.

[Bu68] W.H. Burge, "McG - A Functional Programming System", Report RC-2111, IBM T.J. Watson Research Center, Yorktown Heights, New York (1968).

[Che68] T. Cheatham, A. Fischer, P. Jorrand, "On the Basis for ELF: An Extensible Language Facility", AFIPS Proc Vol 33 (1968), Fall Joint Computer Conference.

[Chu51] A. Church, "The Calculi of Lambda-Conversion", Annals of Math. Studies, No 6, Princeton University Press, Princeton, New Jersey (1951).

[Con63] M.E. Conway, "A Multiprocessor System Design", AFIPS Proc Vol 24 (1963), Fall Joint Computer Conference.

[Cor65] F.J. Corbato, V.A. Vyssotsky, "Introduction and Overview of the Multics System", AFIPS Proc Vol 27 (1965), Fall Joint Computer Conference.

[Da66] O.J. Dahl, K. Nygaard, "SIMULA - an ALGOL-Based Simulation Language", Comm. ACM, Vol 9, No 9 (Sept 1966) pp 671-678.

[deB69] J.W. deBakker, "Semantics of Programming Languages", appears as Chapter 3 in Advances In Systems Sciences edited by J. Tou, Plenum Press (1969).

[Di68a] E.W. Dijkstra, "Cooperating Sequential Processes", appears in Programming Languages, edited by F. Genuys, Academic Press, New York (1968); also published as Report EWD 123, Department of Mathematics, Technological University, Eindhoven, The Netherlands (1965).

[Di68b] E.W. Dijkstra, "The Structure of THE Multiprogramming System", Comm. ACM, Vol 11, No 5 (May 1968), pp 341-346.

[Dng70] P.J. Denning, "Virtaul Memory", Computing Surveys, Vol 2, No 3 (Sept 70), pp 153-190.

[Dns66] J.B. Dennis, E.C. Van Horn, "Programming Semantics for Multiprogramming Computations", Comm. ACM, Vol 9 No 3 (Mar 1966), pp 143-155.

[Ev68] A. Evans Jr., "PAL - A Language Designed for Teaching Programming Linguistics", Proc 23rd ACM National Conf, (1968), pp 395-403.

[Fi70] D.A. Fisher, "Control Structures for Programming Languages", Ph.D. Thesis, Carnegie Mellon University (1970).

[Fl67] R.W. Floyd, "Non Deterministic Algorithms", Journal ACM, Vol 14 (Oct 1967), pp 636-644.

[Gar68] J.V. Garwick, "GPL, A Truly General Purpose Language", Comm. ACM, Vol 11, No 9 (Sept 1968), pp 634-638.

[Ger70] S. Gerhart, "Formal Definition of APL", unpublished paper, Computer Science Department, Carnegie Mellon University, March 1970.

[Ha67] N.A. Haberman, "On the Harmonious Cooperation of Abstract Machines", Ph.D. Thesis, Technical University, Eindhoven, The Netherlands, 1967.

[Ha69] N.A. Haberman, "Prevention of System Deadlocks", Comm. ACM, Vol 12, No 7 (July 1969), pp 373-378.

[IBM69] IBM Corporation, "PL/I Language Specifications, Form Y33-6003-1, 1969.

[Ir70] E.T. Irons, "Experiences with an Extensible Language", Comm. ACM Vol 13, No 1 (Jan 1970), pp31-40.

[Ka68] R.M. Karp, R.E. Miller, "Parallel Program Schemata", Report No RC-2053, IBM T.J. Watson Research Center, Yorktown Heights, New Center, York, 1968.

[Knu68] D.E. Knuth, The Art of Computer Programming, Vol 1, Addison-Wesley, Reading, Massachusetts, pp 406-420, 1968.

[Lam68] B.W. Lampson, "A Scheduling Philosophy for Multiprocessing Systems", Comm. ACM Vol 11, No 5 (May 1968) pp 347-365.

[Lam69] B.W. Lampson, "Dynamic Protection Structures", AFIPS Proc Vol 31 (1969), Fall Joint Computer Conference, pp 27-38.

[Lan64] P.J. Landin, "The Mechanical Evaluation of Expressions", Computer Journal, Vol 6 (1964), pp 308-320.

[Lan65] P.J. Landin, "A Correspondence Between ALGOL 60 and Church's Lambda Notation", Comm. ACM, Vol 8, Nos 2 & 3 (Feb, Mar 1965) pp 89-101, 150-165.

[Lan66] P.J. Landin, "The Next 700 Programming Languages", Comm. ACM, Vol 9, No 3 (Mar 1966), pp 157-164.

[Lau68] P. Lauer, "Abstract Syntax and Interpretation of ALGOL 60 Programs", "Concrete Representation of ALGOL 60 Programs", Reports LR 25.6.001 & LR 25.6.002, IBM Vienna Laboratory, 1968.

[Lea69] B.M. Leavenworth, "The Definition of Control Structures in McG360", Report RC 2376, IBM T.J. Watson Research Center, Yorktown Heights, New York, 1969

[Lee69] J.A.N. Lee, "The Vienna Definition Language: A Generalization of Instruction Definitions", paper presented at the SIGPLAN Symposium on Programming Language Definition, San Francisco, Calif. Aug 1969

[Lu68a] P. Lucas, P. Lauer, H. Stigleitner, "Method and Notation for the Formal Definition of Programming Languages", Report TR 25.087, IBM Laboratory, Vienna, 1968.

[Lu68b] P. Lucas, "Two Constructive Realizations of the Block Concept and Their Equivalence", Report TR 25.085, IBM Laboratory, Vienna, 1968.

[Lu71] P. Lucas, personal communication

[Luc68] F.L. Luconi, "Asynchronous Computational Sturctures", Ph.D. Thesis, M.I.T., 1968, available as Project MAC Report MAC-TR-49.

[Mc62] J. McCarthy et. al., LISP 1.5 Programmers Manual, M.I.T. Press, 1962.

[Mc66] J. McCarthy, "A Formal Description of a Subset of ALGOL", appears in Formal Language Description Languages, ed. T.B. Steele, North Holland

Pub. Co., Amsterdam, (1966).

[Mi70] J.G. Mitchell, "The Design and Construction of Flexible and Efficient Interactive Programming Systems", Ph.D. Theses, Carmegie-Mellon University, 1970.

[Na63] P. Naur (ed), "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, vol 6, No 1, (Jan 1963), pp 1-17.

[Op65] A. Opler, "Procedure Oriented Language Statements to Facilitate Parallel Processing", Comm. ACM, Vol 8, No 5 (May 1965)

[Sa66] J.H. Saltzer, "Traffic Control in a Multiplexed Computer System", Ph.D. Thesis, M.I.T., 1966, available as Project MAC Report MAC-TR-30.

[Slu68] D.R. Slutz, "The Flow Graph Schemata Model of Parallel Processing", Ph.D. Thesis, M.I.T., 1968, available as Project MAC Report as Project MAC-TR-53.

[Sp69] M.J. Spier, E.I. Organick, "The MULTICS Interprocess Communication Facility", presented at Second ACM Symposium on Operating System Principles, Princeton, New Jersey, Oct 20-22, 1969.

[Sta69] T.A. Standish, "Some Features of PPL, A Polymorphic Programming Language", in Proc. of the SIGPLAN Extensible Languages Symposium, in SIGPLAN Notices,

Vol 4, No 8 (Aug 1969), pp 20-26.

[Str67] C. Strachey, "Fundamental Concepts in Programming Languages", text of lectures given at NATO Summer School on Programming, 1967, to be published by North Holland Pub. Co., Amsterdam.

[Ste66] T.B. Steele (ed), Formal Language Description Languages, Proc IFIP Working Conference 1964, North Holland Pub. Co., Amsterdam, 1966.

[Van69] D.H. Vanderbilt, "Controlled Information Sharing in a Computer Utility", Ph.D. Thesis, M.I.T., 1969, available as Project MAC Report MAC-TR-67.

[VH66] E.C. Van Horn, "Computer Design for Asynchronously Reproducible Multiprocessing", Ph.D. Thesis, M.I.T., available as Project MAC Report MAC-TR-34.

[Weg70] B. Wegbreit, "Studies in Extensible Languages", Ph.D. Thesis, Harvard University, 1970.

[Wi66] N. Wirth, letter on "Program Structures for Parallel Processing", Comm. ACM, Vol 9, No 5 (May 1966), pp 320-321.

[Wo70] J.M. Wozencraft, A. Evans Jr., "Notes on Programming Linguistics", available from M.I.T. Department of Electrical Engineering, Feb 1971