AD-786 721

COPILOT A MULTIPLE PROCESS APPROACH TO INTERACTIVE PROGRAMMING SYSTEMS

Daniel Carl Swinehart

Stanford University

Prepared for:

Advanced Research Projects Agency

July 1974

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-230

STAN-CS-74-412

COPILOT A MULTIPLE PROCESS APPROACH TO INTERACTIVE PROGRAMMING SYSTEMS

BY

DANIEL CARL SWINEHART

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY

ARPA ORDER NO. 457

JULY 1974

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

215

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-74-412	2. GOV'T ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) COPILOT: A MULTIPLE PROCESS APPROACH TO INTERACTIVE PROGRAMMING SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED technical, July 1974
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-74-412
7. AUTHOR(s) Daniel C. Swinehart		8. CONTRACT OR GRANT NUMBER(s) SD-183
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Science Dept. Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA ORDER NO. 457
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/IPT, Attn: Stephen D. Crocker 1400 Wilson Blvd., Arlington, Va. 22209		12. REPORT DATE July 1974
		13. NUMBER OF PAGES 214
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative: Philip Surra Durand Aeronautics Bldg., Rm. 165 Stanford University Stanford, Ca. 94305		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) releasable without limitations on dissemination.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An experimental interactive system, COPILOT, is used as the concrete vehicle for testing and describing methods for adding multiple processing facilities to an interactive language environment.

COPILOT allows the user to create, modify, investigate, and control programs written in an Algol-like language, augmented for multiple processing. Although COPILOT is computer-based, many of our solutions could also be applied to an interpretive system.

Central to the design is the use of CRT displays to present programs, (continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ia

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-230

JULY 1974

COMPUTER SCIENCE DEPARTMENT REPORT
STAN-CS-74-412

COPILOT: A MULTIPLE PROCESS APPROACH TO INTERACTIVE PROGRAMMING SYSTEMS

by
Daniel Carl Swinehart

ABSTRACT: An experimental interactive system, COPILOT, is used as the concrete vehicle for testing and describing methods for adding multiple processing facilities to an interactive language environment.

COPILOT allows the user to create, modify, investigate, and control programs written in an Algol-like language, augmented for multiple processing. Although COPILOT is compiler-based, many of our solutions could also be applied to an interpretive system.

Central to the design is the use of CRT displays to present programs, program data, and system status. This continuous display of information in context allows the user to retain comprehension of complex program environments, and to indicate the environments to be affected by his commands.

COPILOT uses the multiple processing facilities to its advantage to achieve a "non-preemptive" kind of interactive control. The user's terminal is continuously available for commands of any kind: program editing, variable inquiry, program control, etc., independent of the execution state of the processes he is controlling. No process may unilaterally gain possession of the user's input; the user retains control at all times.

The emphasis throughout is on improving the characteristics of the interface between the user and the system.

This research was supported in part by the Advanced Research Projects Agency of the Office of Defense under Contract No. SD-183.

ACKNOWLEDGMENTS

I would like to express special appreciation to my dissertation adviser, Professor Jerry Feldman, for his guidance, and for his unfailing confidence in me, even when my own was waning. I am indebted to Dr. Jim Mitchell, who offered immeasurable assistance with the technical problems, through many conversations and several thorough readings. Professor Robert Floyd contributed helpful, detailed suggestions for improving the style and clarity of this work, as well as several key ideas.

I would like particularly to thank these colleagues who read the drafts, even though they didn't have to: Alan Kay, Jim Low, Andy Moorer, Hanan Samet, Dave Smith, Bob Sproull, Larry Tesler, and Russell Taylor.

The publication of this document would not have been possible without the support of the Stanford Artificial Intelligence Laboratory; nor could I have done without the aid of Larry Tesler, who provided the publication system, and Brian Harvey, who helped me make it work.

My deepest gratitude is reserved for my wife, Ann, who always understood why I was gone, and who will have to put up with me now that I am back.

TABLE OF CONTENTS

FIGURES

TABLES

ABSTRACT

The addition of multiple processing facilities to a language used in an interactive computing environment requires new techniques. This dissertation presents one approach, emphasizing the characteristics of the interface between the user and the system.

We have designed an experimental interactive programming system, COPILOT, as the concrete vehicle for testing and describing our methods. COPILOT allows the user to create, modify, investigate, and control programs written in an Algol-like language, which has been augmented with facilities for multiple processing. Although COPILOT is compiler-based, many of our solutions could also be applied to an interpretive system.

Central to the design is the use of CRT displays to present programs, program data, and system status. This continuous display of information in context allows the user to retain comprehension of complex program environments, and to indicate the environments to be affected by his commands.

COPILOT uses the multiple processing facilities to its advantage to achieve a kind of interactive control which we have termed "non-preëmptive". The user's terminal is continuously available for commands of any kind: program editing, variable inquiry, program control, etc., independent of the execution state of the processes he is controlling. No process may unilaterally gain possession of the user's input; the user retains control at all times.

Commands in COPILOT are expressed as statements in the programming language. This single language policy adds consistency to the system, and permits the user to construct procedures for the execution of repetitive or complex command sequences. An abbreviation facility is provided for the most common terminal operations, for convenience and speed.

We have attempted in this thesis to extend the facilities of interactive programming systems in response to developments in language design and information display technology. The resultant system provides an interface which, we think, is better matched to the interactive needs of its user than are its predecessors.

CHAPTER 1
INTRODUCTION

Interactive, or conversational, computing owes its existence to the development of multiprogramming, or multiple processing, facilities. The scarcity and expense of computing equipment prevented direct, convenient user interaction with the programs he wrote until a way was found for several people to share the resources of a computer system simultaneously.

A process, as we will use it, is "an activity comprised of a time-ordered sequence of actions" [56]. The behavior of a process does not depend on the activity of other processes— except, perhaps, for the time and other resources it requires to execute— unless such interaction is intended. We may therefore treat a process as if it had sole use of its own processor (computer or other active agent). Processes may communicate with each other, through messages or shared data, or they may operate independently.

This multiple process activity can be simulated by a single processor, under control of the appropriate operating system. In such a multiprogramming system, use of the processor (and other resources) is allocated among the competing processes, providing for each a virtual processor somewhat slower than the real one. A time sharing system is a multiprogramming system to which terminal devices (e.g., teletypes or display terminals) have been connected, allowing users to communicate directly with active processes within the system.

Joss [7], Basic [29], LCC [45], APL [26], and BBN-Lisp [53] are examples of language systems which are designed to operate in a time shared environment: they are all **Interactive Programming Systems (IPSs)**. (*) They all allow a user to create a program "on line"; to execute it, examine its state, and modify its definition (to "debug" it); and to supply it with requested data. In the current versions of these systems, the system algorithms and data, along with those created by the user, form a single process within the operating system.

- -

(*) We will examine these and other notable Interactive Programming Systems in Chapter 3.

I.A. THE PROBLEM

A time sharing system can use process structures to provide a totally independent operating environment for each of its users. However, when processes are allowed to communicate and to coöperate with each other, they can become a useful facility for the performance of a single project. The Simula 67 document [14] contains several simple examples of coöperating processes. More recently, other operating systems and language systems have begun providing their users direct access to multiple processing facilities.

Inherent in an Interactive Programming System design is a specification of the role the user plays in its operation: the appearance of the interface between the user and the system. The more sophisticated of the IPSs mentioned above (those which implement the more powerful and complex languages) define a user role which cannot easily be extended to handle the multiple simultaneous control and data environments of a language system which supports multiple processes. We will present arguments to support this contention.

In this dissertation we will address the problem of building Interactive Programming Systems which can contend with multiple processing environments. Instead of treating this endeavor as a burden, we will look for ways to use these facilities to improve the performance of the system, and of the user.

I.B. COPILOT

The bulk of this thesis is a description of an experimental IPS, COPILOT, which we have designed as the concrete vehicle for testing and describing our methods. COPILOT allows the user to create, modify, investigate, and control programs written in an Algol-like language, which has been augmented with facilities for multiple processing. Although COPILOT is compiler-based, many of our solutions could also be applied to an interpretive system.

Central to the design is the use of CRT displays to present programs, program data, and system status. This continuous display of information with some associated context helps the user to retain comprehension of complex program environments, and to indicate the environments to be affected by his commands.

COPILOT uses the multiple processing facilities to its advantage, to achieve a kind of interactive control which we call "non-preëmptive". The user's terminal is continuously available for commands of any kind: program editing, variable inquiry, program control, etc., independent of the execution state of the processes he is controlling. No process may unilaterally gain possession of the user's input; the user retains control at all times.

Commands in COPILOT are expressed as MISLE language statements. This single language policy adds consistency to the system, and permits the user to construct procedures for the execution of repetitive or complex command sequences. A top-level abbreviation facility is provided for the most common terminal operations.

The role of the COPILOT user is that of a global observer and controller, with equal access to all his program and data environments, subject only to protection restrictions imposed by the operating system. We will demonstrate that this view is substantially different from the more local focus provided by the typical single process IPS.

I.C. A BRIEF OUTLINE

The early chapters of this dissertation establish a basis for the study, defining our goals based on observed needs. A survey of existing IPSs follows, provided as a basis for comparison, and to indicate the debt we owe to our predecessors.

Chapter 4 is an overview of the COPILOT design. After describing the basic facilities of the system, emphasizing the achievement of the stated goals, we present a detailed example of system operation. The reader interested in system design may choose to read this chapter first; the references to earlier chapters should not interfere with this procedure.

Subsequent chapters provide detailed user-level descriptions of COPILOT, giving special attention to the facilities for multiple processing, and to our reliance on the use of display devices to enhance these facilities.

We have limited implementation considerations to a brief chapter which concentrates on the structures we have created for representing programs at different levels, or "Tiers", and the means for maintaining the necessary relationships between Tiers.

The final chapter is a compendium of miscellaneous topics, unsolved problems, and suggestions for further research.

CHAPTER 2
HUMAN INTERACTIVE CHARACTERISTICS

2.A. THE BEHAVIOR MATCH

An Interactive computer System (IS) is the hardware and software which allows composition, testing, debugging, and operation of computer programs, enhancing the "ability of the user to initiate, interrupt, and generally interject himself into the control of the system" [44]. In practice, an IS consists of a user console (keyboard and printer), and the set of program and interactive features which are available to it, operating on a digital computer, which is usually time-shared. An Interactive Programming System (IPS) is an IS incorporating a single programming language for all programming and program control.

Most recent emphases in IPS design (†) have been on improved language design, improved debugging facilities, and on the development of "single language" systems, which extend the programming language to include the interactive facilities. Mitchell's thesis [44], itself a significant contribution to Interactive Programming Systems, contains as well a good survey of the leading examples of current systems. His emphasis is is on language design and on implementation considerations (flexibility, efficiency, and portability).

The emphasis of this dissertation is on the user-system interface. It is our desire to provide a convenient, pleasant, intuitive interface between the user and the IPS. We intend to do this by providing a system whose behavior matches as closely as possible the relevant characteristics of the people who use it. Our thesis is that such a system can measurably increase user performance.

There is an intriguing, if not terribly accurate, metaphor to be found in electronic lore: the "impedance match". For maximum efficiency (minimum wasted energy), the impedance of an output from one device must closely match the input impedance of any device to which it is connected. If the impedance mismatch is too great, the connection will fail to perform successfully at all. We will call our IPS analogue a "Behavior Match" – a term which we shall attempt to justify.

- -

(†) Examples are ECL, LCC, and BBN Lisp, all of which we will discuss in the survey of Chapter 3.

To emphasize our conviction of the importance of this Behavior Match concept, and the necessity for some terminology to express it, we offer these informal definitions and terms:

The **Behavior** of an entity is that set of processes which determine the manner in which information can be presented to it, and is presented by it.

A **Behavior Match** has been achieved when the "behavior" of a system complements the behavior of its user, optimizing his performance.

These definitions are clearly subjective, containing as well enough undefined terms and vague semantics to preclude their use for any measurement purposes. Although we hope to clarify these definitions somewhat in the sequel, their major purpose is to provide an intuitive basis for discussion.

The **Behavior Match** diverges from the impedance match example in that user and system behavior need not be identical, or even similar; they need only be "complementary." However, we shall show that the similarity is stronger than one might expect.

At the risk of overloading the "impedance match' analogy, let us point out one additional similarity: the impedance match between communicating devices need only exist at the interface between them. It is possible to design circuits which isolate the main body of a device from its interface, allowing it to employ impedances (and other related characteristics) which are internally convenient. Similarly, many of the internal details of an efficient, powerful IPS must be hidden from the user, since their functions (e.g. compilation, data conversion) are not involved in the problem-solving efforts of the user, nor are their results (binary machine instructions, etc.) likely to be meaningful to him.

2.B. SCOPE OF APPLICATION

The bulk of this dissertation is dedicated to the design of system interface characteristics which will improve the interactive behavior match between system and user. Just as the interface characteristics one chooses for an electronic device place certain constraints on the internal device design, our IPS interface decisions will have an effect on *all* aspects of system design and implementation. However, we should not let our human engineering decisions unduly reduce our range of options in such fundamental areas as: the selection of a programming language; the choice of execution methods (compiled or interpreted); whether the system is intended for the creation of large, "production" programs, or for smaller, "instructional" ones; or whether it is intended chiefly for novice or expert users. We hope to show that the approaches to IPS design which we advocate apply to systems which vary widely in these parameters.

We will present in the course of the dissertation an IPS, COPILOT, as a concrete vehicle for discussing methods for attaining a good **Behavior Match**. Because it is a concrete system, COPILOT exhibits certain choices from the above parameter spectra. Indeed, we think we have made the more difficult, perhaps less inherently flexible choice in nearly every case. This is true in part because of the particular needs of the environment for which we have designed the system, in part because of a desire to demonstrate the versatility limits of our methods. Nevertheless, particularly in these initial chapters, we will attempt to indicate those areas where choices can be made, and those which are heavily constrained by our solutions.

2.C. SPECIFIC ATTRIBUTES

We have chosen for study a set of human interactive attributes which, we believe, an IPS should accommodate in order to achieve a behavior match. This set of characteristics, which follows, was derived in two ways: some are characteristics which we have observed, and which influenced our design – *a priori* observations. The rest are, admittedly, *a posteriori* observations, attributes we have noticed which are fortunate in light of what our methods provide. This fact should not affect their validity.

We do not claim to have isolated all relevant interactive attributes. We have concentrated on these behavioral aspects which relate to "process" and "information transfer". Additionally, these conjectures will have to stand as the opinions of the author– based on his observations of the way he and others use interactive computer systems– used to justify and guide the design of the COPILOT system's behavior.

2.C1 Multiple Activities

The activity of someone engaged in the solution of an intellectual problem can be modelled as a single processor executing a set of coördinated sequential processes (coroutines), in the sense that:

1) He is likely to shift his attention rapidly between different "processes." His reason for doing this may be generated internally (e.g. boredom, inspiration) or externally (the phone rings; or perhaps the part hasn't come in yet).

2) He may retain enough "state" information about an abandoned process to return to it again in time, or he may abandon it entirely.

3) If the alternative is excessive unproductive waiting, he will often turn his attention to some unrelated subject (the processes need not all coöperate), returning to the task at hand when it is again possible.

4) He can carry some state information concerning a previous activity into the next, often correlating the two in order to understand complex relations. After all, he is presumably pursuing some overall goal.

5) Although we have not modelled his internal behavior as true parallel processes (we give him credit for single-mindedness), he can make use of several concurrent external operations (stove burners, machines, computer programs, or whatever), as long as they do not all require constant monitoring.

6) He seldom operates very recursively, or even properly nests operations— the above coroutine-like model is a more accurate one than a simpler recursive model.

2.C2 Single Language

Symbolic communications between people (and between a person and his later self, for that matter) are primarily conducted by means of natural language. The same language base is used for all areas of endeavor, although specialized lexicons (seldom specialized grammars) form dialects for specific topics. All necessary symbolic activities are possible in a natural language.

For efficiency and brevity, people have added to their communication abilities in two major ways:

1) through formal languages (e.g. mathematics) which, though not contained in the base language, nonetheless have a (usually cumbersome) mapping into it.

2) through acronyms, abbreviations, and possibly non-grammatical colloquialisms, often understood by only a small segment of the population ("far out!"). These artifacts clearly map (though not always precisely) into grammatical forms in the base language.

Providing good symbolic communication between the user and his system will be a major goal of this work. We believe that an IPS with a single input language, encompassing all system commands, can enhance this communication. We share an emphasis on the importance of the single language idea with most IPS designers.

2.C3 Non-Preëmption

A request for one's services is not always granted instantly. In fact, it is sometimes not granted at all. At any rate, having noticed such a request, one may respond to it immediately, queue it temporarily until some other task is complete, or ignore it entirely. He is not automatically preëmpted by a "service request"; he can continue what he is doing, or go on to something else entirely; nor must he take care of things in a fixed order.

This non-preëmptive pattern is often thwarted at the user terminal connected to a modern IPS. Much of our attention will be devoted to correcting the situation.

2.C4 Response Time

In contrast, when one requests a service, he would like it to be handled at once. We would like to distinguish between the time required to complete a request, which we call **completion time**, and a potentially different interval, which we call **response time**: the time delay, after submission one request, until that request is acknowledged, and another may be submitted. If there is but one agent for execution of requests, these two quantities will probably be the same. However, in an environment which supports multiple activities, successive requests may call for the initiation of concurrent activities, or they may terminate previous ones. If such activities are possible, then, in order to make maximum use of the concurrent facilities, the response time should be short, independent of the completion time. (In our experience, this time should be short compared to the time required to make the request, and should seldom exceed one or two seconds.) Miller [41] has studied computer system response, determining empirically, for a variety of situations, what kinds of delays people will tolerate. These times range from a second or two, in highly interactive situations, to fifteen seconds or more for complex requests. Miller's report does not make our distinction between completion time and response time. However, in most of the situations he cites in which people will tolerate only short delays, it is rapid response which they seem to be seeking.

Simon, in [51], studied a related time interval, which he called the "minimum human response time". This is the smallest "time slice" which one can efficiently use to work on a task, particularly in the context of waiting for some possibly unrelated activity to complete. In Simon's experience, this time is approximately ten minutes. We do not dispute it, but we do believe that the "minimum human response time" could be reduced, if it were easier to establish the context necessary to switch to a new task. In a computing environment, this requires a system which is both non-preëmptive and responsive.

To summarize, people want to schedule requests for their services (output), but to obtain rapid attention to their own requests (input). This double standard is not always possible in dealings with other people, but we can try to optimize it in an IPS.

2.C5 Minimal (output) Modes

This topic introduces another input/output double standard. People are capable of understanding stimuli which are context-sensitive: whose meaning depends on the environment, or context, in which they are presented. English itself is internally context-sensitive, although normally only in a quite localized fashion-- paragraphs can generally stand alone.

In general, we think it is desirable to reduce the context-sensitivity of what one must say (output) by reducing the number of "states", or "modes", which impose different interpretations on his communications. The single-language criterion also aids us here: a sentence, especially one intended to convey information unambiguously, should always "mean" the same thing. This cannot be true if disjoint (or even worse, partially disjoint) languages are provided for different purposes, since in the latter case a "mode" must be established to determine which language to look for.

We do not mean to imply that the same results will obtain, no matter what the situation (or state), when a given utterance is uttered, or when a given command is typed. There are environmental conditions which influence the interpretation of communications. This context is usually implicit, however, and need not be included in the message.

We do not even intend that every statement be meaningful in every instance. Clearly, there are sentences in nearly any language which are senseless, impossible, or merely silly under some conditions. However, normally one can at least understand such a sentence, to the extent that he can respond that it is senseless, silly, or impossible– and why. We would like to preserve this behavior

We will, therefore, require of our non-preëmptive, single language IPS, that it must allow a user to express anything in that language, at any time– even if it is meaningless in context– a system without excessive "modes".

2.C6 Maximum (input) Context

While one prefers to supply as little explicit contextual information as possible when conveying information (output), one absorbs information (input) most readily when the environment in which it is presented is as completely described as possible. The more one knows about a situation, the more capable he is of handling his part in it. Our goal should be to provide as rich a context as possible, without including irrelevant information which could obscure understanding. Further, it is best if this information is continually present, continually up to date.

When it is possible, we think that contextual information is best presented visually. This sort of presentation can be made to satisfy the "continuously accurate" requirement, without flooding our sensory channels– particularly because visual input also satisfies our non-preëmptive requirement– one need not look at everything all the time, and in fact can select what to look at, and when to look at it.

2.C7 Access to Information

This topic is closely related to the previous one, which requires that the available information be presented as completely and coherently as possible. Now we wish to require, in addition, that as much information as possible be available (accessible). One is clearly more able to deal with a situation or object when all its components are accessible (to see and, hopefully, to change) than when he must treat it as a "black box" (or perhaps "gray box").

2.C8 Non-symbolic operations

Most of the topics we have discussed have dealt with symbolic terms: with language, its uses and effects. But a remarkable number of things people do are not (at least at the conscious "interface") expressed symbolically at all; they are instead "manipulative" activities. We affect things directly by moving them; we sense them directly by touch, sight or smell.

As an example, after one has become experienced at driving a car, he is seldom aware of turning the wheel or manipulating pedals; instead, he turns the car, speeds up, or slows down– another example of levels of internal mapping which involve intermediaries at other than conscious levels. Perhaps a better example is the playing of a musical instrument: one does not (except when learning something difficult) think in terms of plucking strings, pushing keys, or blowing air. He thinks in terms of producing notes, or even melodic phrases, of the desired pitches, amplitudes, durations, and tonal quality.

Examples of these operations for a computer terminal might be functions performed by a single keystroke, perhaps qualified with "control key" modification, or by light pens, function keyboards, etc. The conscious mind is aware only of their effect. This feeling applies especially to those operations which have an immediate and visible effect– for instance, the movement of cursors or the deletion or movement of text on a display screen.

What we are advocating here is that the way in which such repetitive operations have to be performed be made simple enough that one thinks of them (while doing them) only in terms of their effect. In this way they tend to lose any symbolic meaning and to become practically bodily extensions.

Having made the distinction between symbolic and "manipulative" operations, we would like to soften it somewhat. Although we do not normally do it, we can describe nearly any action in words: there is a way to map a given action into an "equivalent" symbolic form. We will find this duality very useful in the sequel.

2.D. THE BEHAVIOR MATCH REVISITED

We have attempted in the preceding section to indicate some characteristics of the IPS user which the IPS must "complement" to achieve an acceptable "Behavior Match". Before we proceed to an analysis of the success of previous systems in this regard, we should attempt to clarify what we mean by "complementary" behavior (recall the definition of Behavior Match in Section 2.A).

Whatever the means of communication, the user does not really "do" any of the things he requests: the computer does them, under the control of the interface routines of the IPS. Thus before he can communicate a message to his system, he must translate that thought, using his own internal model of this interface, into the series of symbols which will accomplish the transmission.

This internal model must adequately represent the real thing, given the low tolerance of most language systems for syntactic errors (‡). In this sense the Behavior of model and system must be quite similar; i.e., their Behavior must match precisely. What we wish to achieve, in these terms, is a system which allows natural, intuitive, and convenient translation from the original thought to the model.

- -

(‡) Teitelman's DWIM system for BBN LISP [53] (see Section 3.F) is intended to reduce the necessity for such precision by detecting and correcting simple errors (mismatches). We have not treated error detection, correction, or minimization in this treatise, although in Section 9.C5 we have attempted to indicate how our non-preëmptive methods can be used to soften the effect of errors.

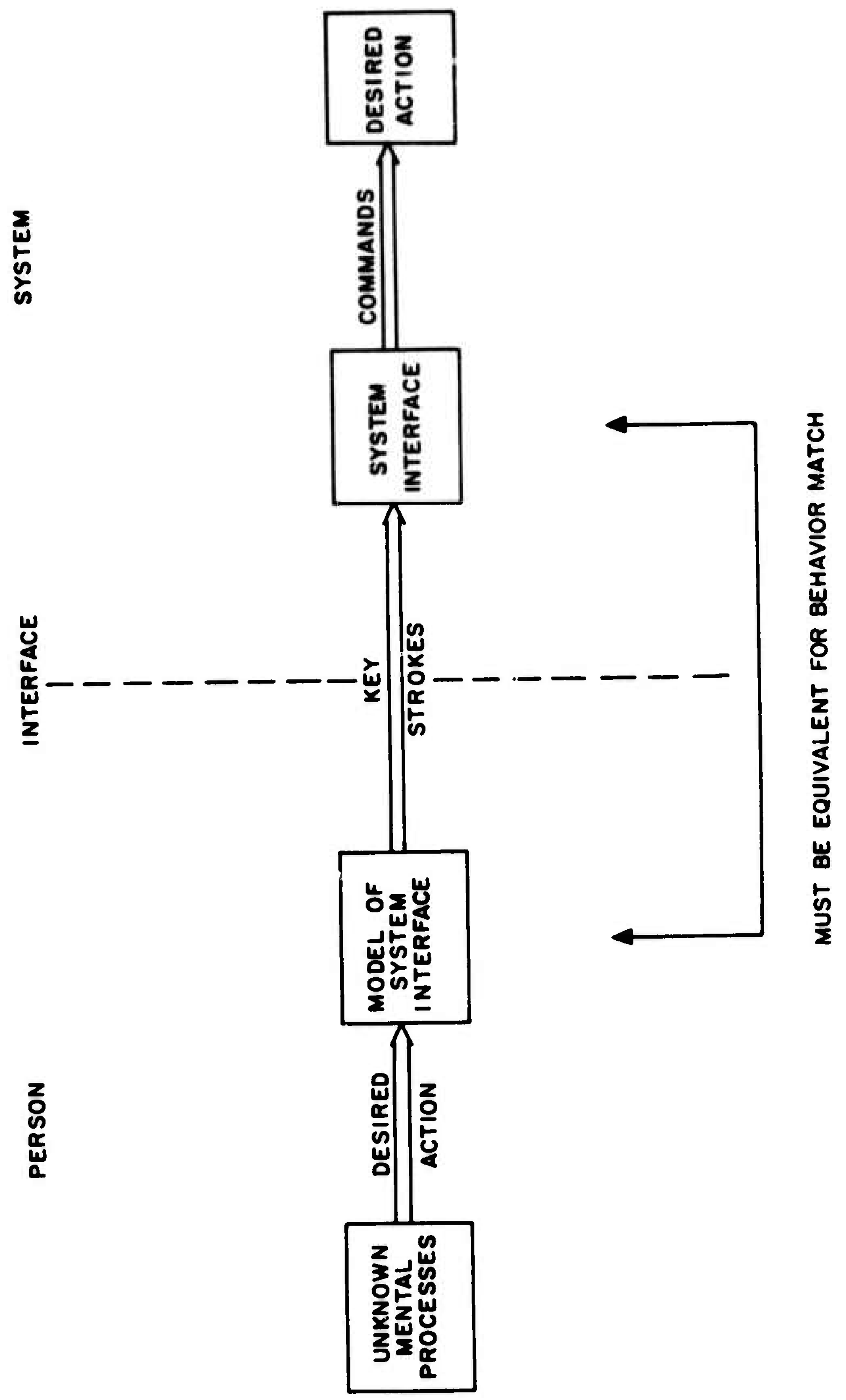


Figure 2-1. Thoughts to Action

CHAPTER 3
A SURVEY OF REPRESENTATIVE INTERACTIVE PROGRAMMING SYSTEMS

To the extent that designers of computer systems have considered behavior match issues, we believe that the designs reflect the designers' views of adequate user models: that the user could think quite naturally in the terms necessary for modelling the system's behavior. Just as we have suggested above, for example, that a person "is" a pseudo-parallel processor, the designer of one of the early systems described below might have said that a person "is" a finite-state automaton. We see a remarkable progression in complexity from early systems to today's IPS systems, reflecting perhaps an increased respect for the complexity of human processes. (*) In the discussion that follows, we present several different IS designs, each based on a different interface behavior model. Following the description of each model is a list of real systems which approximately fit into the category defined by the model.

3.A. BATCH COMPUTING SYSTEMS

We mention these systems only for completeness. The meager control languages provided for these systems are adequate to define the environment and resources necessary for a run, and to specify the order of application of programs in a multi-step job To be sure, systems exhibiting evidence of human engineering are welcome to batch users. In fact, we could profit by applying some of the lessons learned from IPS design to the batch regime. However, there is not much to be learned about the problem at hand from analysis of batch systems.

We include in this category systems which use terminals for so-called remote job entry (RJE), since they are not truly interactive systems.

....

(*) The structure of this section is largely the resull of a conversation with J. Mitchell.

3.B. EARLY INTERACTIVE SYSTEMS (FSA/IPS)

The terminal interface of some early time-shared computer systems (examples of which thrive today) provide an excellent example of what we call the FSA/IS model. Here the system is portrayed as a sort of Finite State Automaton (FSA), which enters a multitude of states, based on current input and previous states. These states typically fall into a much smaller set of classes (modes), as we shall explain.

(Based on the arguments in the introduction to this chapter, the implication of the FSA design is that the user, also, is fundamentally content cast in the role of a very clever FSA. He must maintain in his head a model of the current state, along with the meaning and legality of the commands he might issue while in each state. Given this human model, the FSA/IS system provides an excellent behavior match. The same sort of argument can be made for all of the systems which follow.)

This terminal interface model, though failing many of our behavior match requirements, has performed admirably, especially in light of the accompanying software systems (compilers, loaders, and the like), which are typically batch-oriented, and not suited for modification to highly interactive situations. Elements of this design exist in nearly every subsequent interactive system, though some of the shortcomings have been overcome.

The diagram of Figure 3-1 is a simplified state transition graph for the Digital Equipment Corp. TOPS-10 time-sharing system, written for the DecSystem-10 computer, a system we consider typical of the FSA/IS discipline.

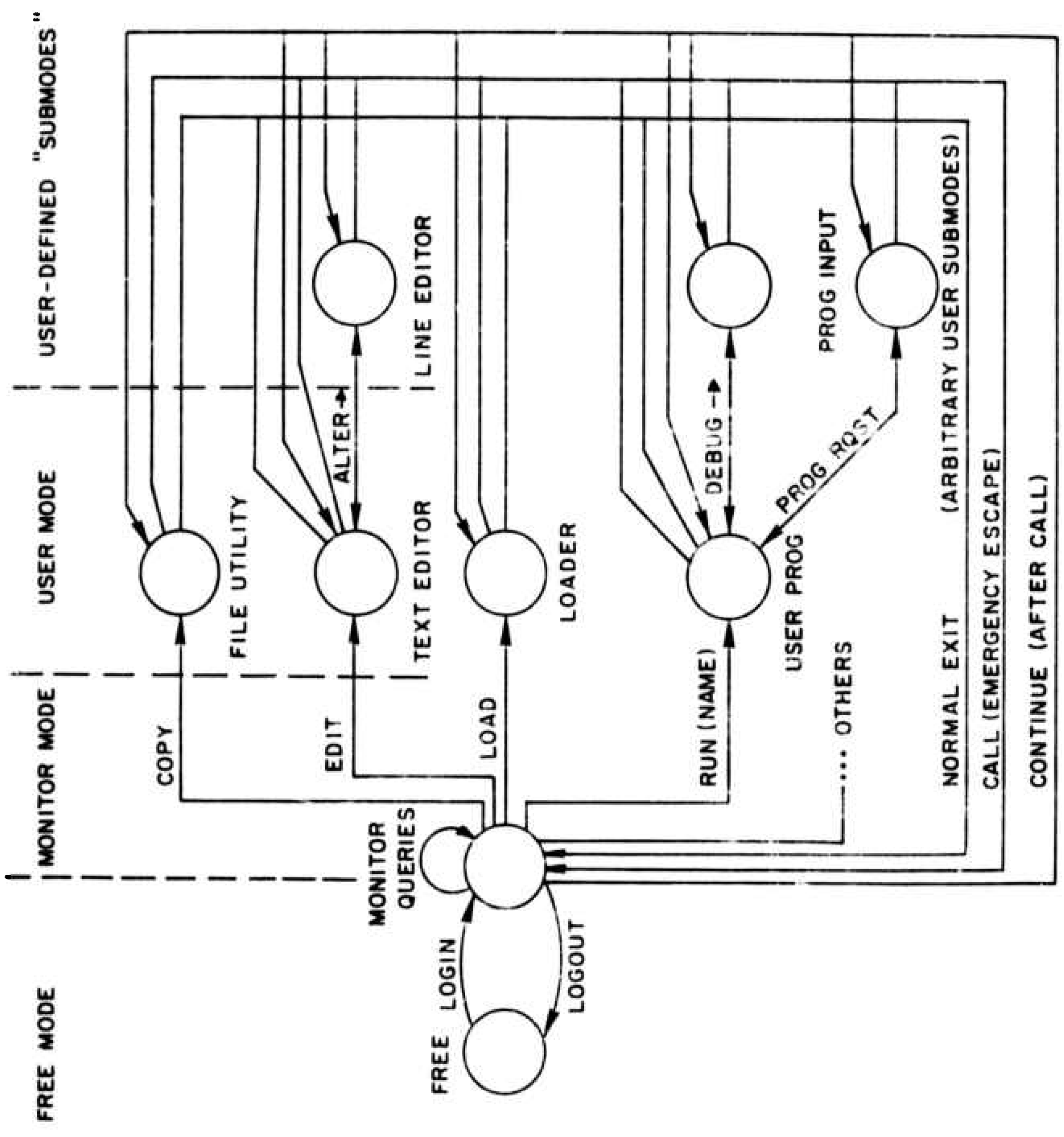


Figure 3-1 FSA/IS Behavior of DEC TOPS-10 Executive

To the reader already familiar with this common organization, the interpretation of this diagram should be particularly straightforward. The user approaches the terminal in **Free** mode (both system and user, according to our assumptions!) He (and the system) enters the basic System mode using a restricted, and unique, Login language interchange, then proceeds to work.

While in System, or Monitor mode, the user communicates with the system using a verb-argument syntax (e.g., "RUN X[20,35]" or "COMPILE PTRAN"), which is interpreted directly by the operating system. If this syntax appears elsewhere in the system (in other modes), it is due to mimicry, not to any global design. Some of the Monitor mode commands, notably those requesting simple status information ("What time is it?") perform their functions, then return immediately to Monitor mode to await additional commands. The more interesting commands, however, cause other programs to be mapped into main memory and run, entering one of a multitude of so-called User states (in User mode), whose input grammars depend entirely on the program implementations. From here on, the system makes little modal distinction. The user can, however, in his programs, define his own substates, specifying differently at different times what constitutes acceptable communication.

Control passes from User mode back to Monitor mode either by program request (only indirectly influenced by user input), or by use of the special interrupt character, CALL (or control-C), whose function is always to stop operation of the User-mode program and to return to Monitor mode.

This (crucial) CALL feature falls short of providing the non-preëmptive environment to which we subscribe, but its existence leads us to the following interesting observation: although the user of this system has no direct access to it, at some level of implementation– a very low one, in this instance– a non-preëmptive discipline is in effect. The system responds in a similar way to each character as it is typed, echos it on the output device (printer or display), analyzes it for special meaning (e.g., CALL), then either arranges for the return to System mode or dispatches the character to the process currently preëmpting the terminal. Thus, though control of it has not been granted directly to the user, the value of a non-preëmptive regime has long been implicitly recognized.

At this level, the non-preëmptive discipline reduces simply to an interrupt-driven, multiple process priority discipline. This example illuminates the intimate connection between non-preëmptive and multiple process organizations.

In general, no simple way exists in these systems to suspend one action temporarily, in order to perform some other (perhaps unrelated) action, then to return to the original task; mode changes are usually destructive in that sense. More generally, little, if any, information about previous states is retained by these systems – such memory must be provided by the user.

3.B1 Attribute Analysis

Let us now analyze systems of this character with respect to our Behavior match attributes. An attribute is classified as variable if it is typically absent, but could be included in a system without altering that system's basic category:

1) Multiple activities: nonexistent or cumbersome to use.

2) Non-preëmption: poor. As we have seen, the entire design embraces the concept of preëmption of the terminal by processes implementing different modes. One must in every instance type only what is expected at that point, or else a specific (e.g. exit or substate-entering) or general (e.g. CALL) "escape" character to change modes.

3) Response time: poor. The edit/compile/ run/debug cycles typical of these systems are long and sequential, often requiring manual intervention between steps to initiate the next. No fruitful work can be done during, for example, the compile phase.

4) Mode reduction: antithetical. In such systems there is a mode for every purpose.

5) Single Language: not provided. There is generally a different language for each mode.

6) Accessibility: variable. In a computer system we desire accessibility to such things as: the variables of the running environment (the data); the statements or functions of the language (the program); and, hopefully, the control structures of the system (the interpreter).

   The only global program and data variables in the TOPS-10 system are data and program files on secondary storage. Any other data are defined and controlled by the programs which run in User mode; the accessibility of these data is thus determined by these programs, varying with each instance. These operating systems do not limit the ability of their subsystems to provide good accessibility; most of the systems which we will discuss were implemented using the facilities of general-purpose FSA/IPS systems.

7) Context: variable, typically poor. Later we shall assert that a system cannot supply the continuous context information we advocate without display devices with rapid random-access capabilities. There is in principle no reason that such contextual displays could not be integrated into any IS. However economic considerations have legislated heavily against their use. Ironically, many batch systems have fairly good context displays for their operators [25].

8) Non-symbolic features: variable. The manipulative operations we envision could be provided in any IPS, regardless of category. We know of only scattered instances where any have been provided.

3.B2 Representative Systems

The command languages of most general purpose time sharing systems fit this category. In addition to the TOPS-10 system [10] used in this section, they include the pioneering CTSS system at MIT [9], The Stanford Computation Center time-sharing facility [25], as well as newer systems like TENEX [5], MULTICS [43], and ITS [17]. The latter three do possess facilities for controlling multiple processes, by explicit assignment of the user terminal to one process at a time. Nevertheless, for the most part they behave as FSA/IS systems.

3.C. EARLY DEDICATED-LANGUAGE SYSTEMS(FSA/IPS)

This class of programming system was developed for use where the needs of the user community did not warrant development of a general time-sharing system, or where the need for simplicity and comprehensive diagnostic information was paramount. Although, unlike the FSA/IS systems, these qualify as IPSs (using our requirement that an IPS be built around a single language), these systems are actually more restrictive in many ways.

The terminal state diagram for BASIC [29], which we consider representative of this system type, appears in Figure 3-2. Operation of the system alternates between the edit phase, in which programs are created, modified, fetched and stored to secondary storage, and the execution phase, in which the meaning of user inputs are defined by the user's program. The number of mode classes is not really reduced from our TOPS-10 example, but the number of User mode states is sharply reduced, restricting the user to the single language.

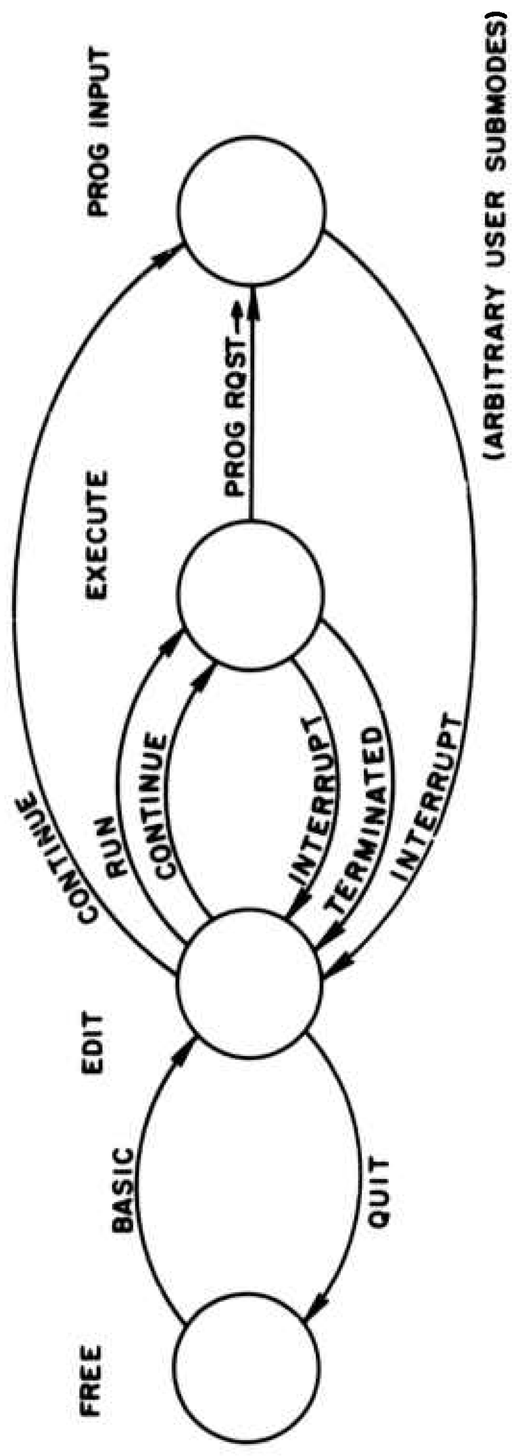


Figure 3-2. FSA/IPS Behavior of BASIC Terminal Interface

3.C1 Attribute Analysis

FSA/IPS systems have about the same degree of success at meeting the behavior match requirements that FSA/IS systems do. The one possible exception is the single language criterion. BASIC does not even really qualify as a single language system, though, but is simply a restricted (or dedicated) language system; there is no intersection between the syntax of the program editing commands and that of the statements which are edited.

3.C2 Representative Systems

BASIC and its derivatives are representative of this "compile and go" class.

3.D. REDUCED MODE SYSTEMS (FSA/IPS/RED)

These are the first truly interactive systems we have encountered. In these systems the user can switch rapidly from program modification to partial program execution to variable value query. They are also the first really single-language systems we have seen: statements which implement user algorithms resemble in syntax those for modifying program text and for controlling (starting, stopping, interrupting) execution of the algorithms. Also, in most cases, either type of statement is legal whether executed "directly" (typed in at the terminal, interpreted and obeyed immediately), or "indirectly" (as part of some previously created program).

Our archetypical system of the FSA/IPS/RED type is JOSS [7]. Figure 3-3 is an approximation to the console state transition diagram for JOSS. Chiefly due to the implementation of all functions as part of a single language, the segmentation of programs in that language into parts and subparts (steps) which can be executed separately, and the implementation of an interpreter for the language which can to perform these functions incrementally, the designers were able to reduce greatly the number of modes. In JOSS, there is the one predominant Command mode, the nearly irrelevant Free mode, and the mode entered to accept input to the user program, on program request.

A system of this sort could presumably support any programming language. However, most do not feature any but the simplest name scopes (static *or* dynamic), since the command routine operates only at the "top level" of the system, requiring suspension of user program execution (and perhaps loss of local context) before control returns to it. JOSS, for instance, has only a single naming level (all variables are global). Others allow simple local parameters to procedures. In other systen s, including some LISPs [49], it is possible to inhibit loss of local context after an error, or after an otherwise interrupted computation. Because the nested User structure to be exhibited in the next section does not exist in these systems, full interactive control is usually not possible in these suspended environments; typically, only variable query and "backtrace" operations are available.

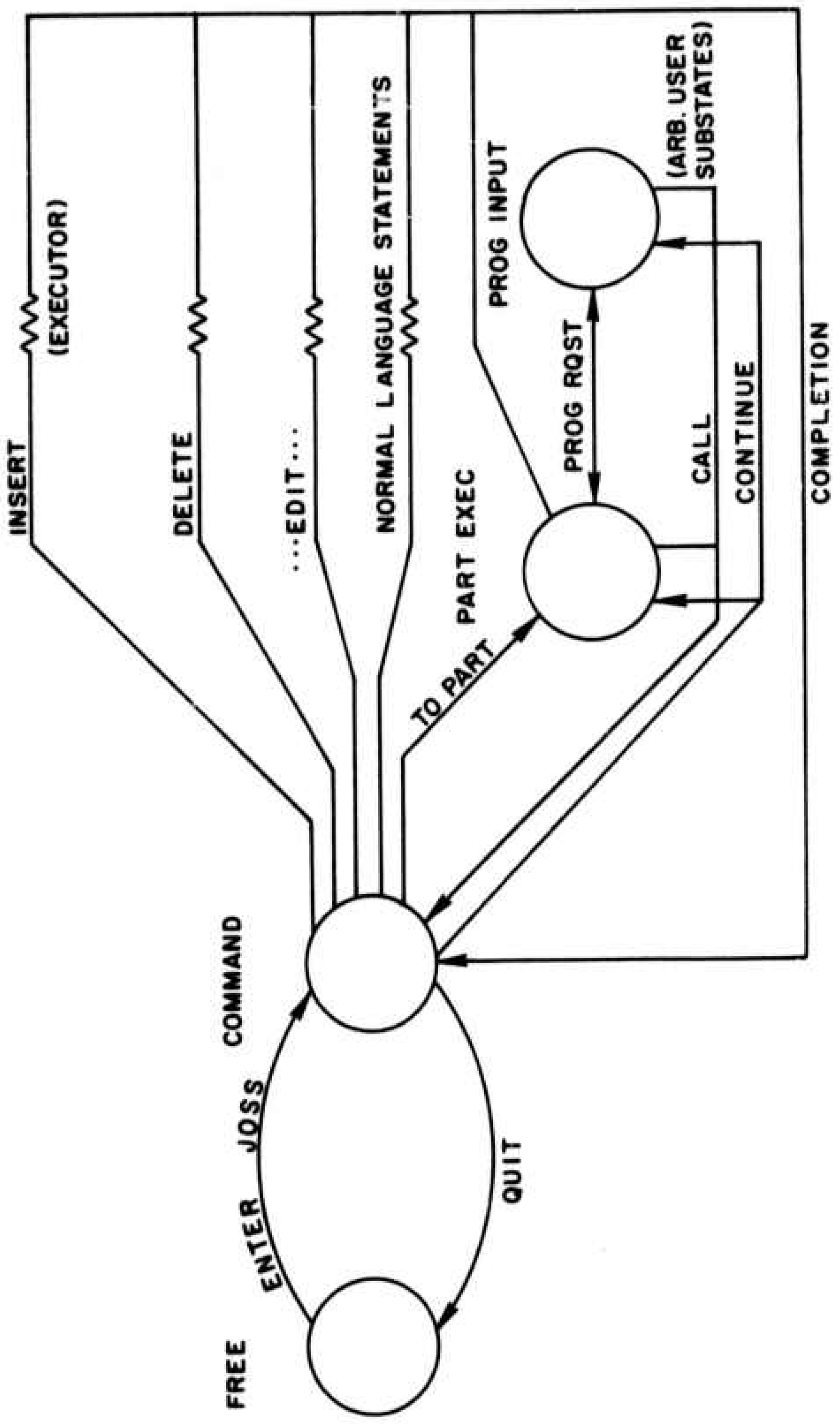


Figure 3-3. FSA/IPS/RED Behavior of JOSS Terminal Interface

3.D1 Attribute Analysis

1) Multiple activities: poor. The single program task may be interruptable, or even continuable, but only trivial operations may be performed in the interim without destroying the state of that task. Complete freedom does not necessarily exist to examine all active data using terminal commands.

2) Non-preëmption: not provided.

3) Response Time: fair. Unless the user's program is running, preventing the system from listening, commands are obeyed quickly (depending on system load, of course). Gaining control can sometimes be a destructive process, however.

4) Mode-reduction: good. Unless the terminal has been preëmpted for user input, nearly any statement or command is legal whenever the system is willing to listen.

5) Single language: good. All but user-defined commands are in the same language.

6) Accessibility: moderately good. In some systems one can examine the state of any data item, but only because the complexity of data declaration is sufficiently restricted. In others, one is denied complete freedom to examine all active data from the terminal.

7) Context: variable. These systems do not present data continuously (do not support displays), although they could. They therefore fall short of our context goals.

3.D2 Representative Systems

We have placed JOSS (and systems patterned after it: e.g., AID [11]), along with RUSH [1], PL/ACME [63], QUICKTRAN [13], and unaugmented (†) versions of some LISPs (e.g., [49]) in this category.

- -

(†) LISP is self-defining, allowing the user to write a command loop which, for the most part, upgrades the system to the next category.

3.E. NESTED USER SYSTEMS (DPDA/IPS)

The systems we have seen so far have restricted the complexity of the programming languages they could support. Major attributes of modern programming languages are the naming and data allocation facilities which allow multiple recursive or parallel instances of the data environments for procedures, and multiple use of names by scope-qualification. Most of these facilities have been sacrificed in the IPSs we have described, because otherwise they could not provide for the user convenient ways to "manipulate and roam around in the information space which is of interest to him when it is of interest to him." (‡) In our terms, they would provide inadequate accessibility.

The systems of the next category extend and modify the role of the user (or his representative system interface, if you wish), greatly extending his ability to interact with complex environments.

Our model system this time is LCC [45]. In LCC the user is modelled as a recursively-instantiable procedure "written" in the language supported by the IPS (see Figure 3-4). The system interface still interprets input as program statements, generally executing them consecutively, in FSA fashion. However, the means for accomplishing this are now more explicit: an activation record for a PART0 (or User) procedure exists on the stack, defining the environment of the user. Each statement submitted from the terminal is treated as if it were (had always been) the next statement in the User procedure. Such a system resembles at the user interface (or models the user as) a finite state automaton with access to a push-down stack for data and previous state information. Such a device is known in automata theory as a Deterministic Push-Down Automaton, or DPDA; thus our designation of this system type. LCC is quite representative of the DPDA/IPS.

The differences between DPDA systems and other FSA systems are not striking at the "top level"— while the keyboard input is driving the original outer-level User procedure instance. However multiple instances of User procedure, at different recursive levels, are permitted. The running program may instantiate a User procedure directly, by a procedure call; or an instance may be created synchronously (via a preset breakpoint), or asynchronously (e.g., an unexpected procedure call [47]) in response to a user-initiated "attention" signal. In any

- -

(‡) From [44].

case, only one User procedure may be active— responding to the keyboard— at any instant, and then only when that User instance is the most recently entered procedure. This automatically prevents any but the most deeply nested User procedure from being active.

Now it is possible to establish a precise interpretation for the meaning of names typed by the user: they are interpreted in the environment of the User procedure in control, just as names are interpreted in any other procedure. It is therefore possible to provide accessibility to variables in any environment, by arranging to instantiate a User procedure which can "see" that environment.

This arrangement still does not meet all our accessibility requirements. For instance, in any recursive language, for a *given* User procedure instance there can be variables hidden from view (using normal access methods) due to recursive instances of the same variable. In Algol-like languages, the problem is worse: each instance of the User procedure must be considered to be *declared* within the procedure from which it is called (or which it interrupts— it amounts to the same thing) in order to "see" the data for that procedure. Not only is this difficult to implement, but it also does not provide access to those active data not in the lexical scope of any User procedure instance.

LCC does not suffer from the latter (Algol-induced) malady, but shares the former with other systems. It solves them by providing rather clumsy (but complete) means for violating scope restrictions, through extended names or explicit scope specification, indicating environments of interest. We feel that some sort of scope-violation mechanism is inevitable for any IPS which provides both a powerful enough language and an accessible enough system.

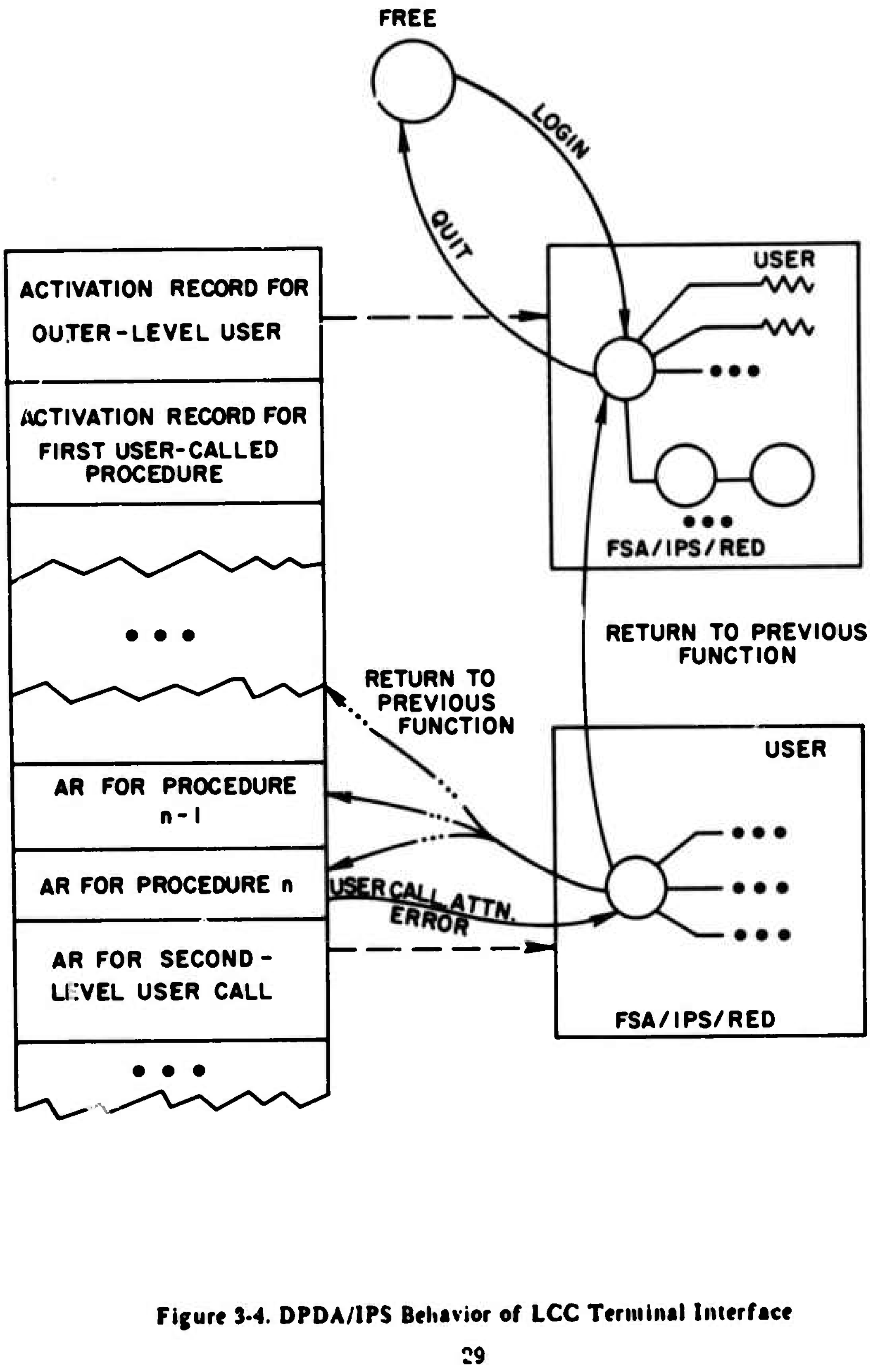

FREE
LOGIN
QUIT
ACTIVATION RECORD FOR OUTER-LEVEL USER
ACTIVATION RECORD FOR FIRST USER-CALLED PROCEDURE
AR FOR PROCEDURE n-1
AR FOR PROCEDURE n
AR FOR SECOND-LEVEL USER CALL
USER
FSA/IPS/RED
RETURN TO PREVIOUS FUNCTION
RETURN TO PREVIOUS FUNCTION
USER
USER CALL, ATTN, ERROR
FSA/IPS/RED

Figure 3-4. DPDA/IPS Behavior of LCC Terminal Interface

3.E1 Attribute Analysis

1) Multiple activities: fair. These systems, by allowing multiple instances of User procedure (or a similar construct), gain some of the control powers we advocate, at least allowing the user to switch his environment of interest without destroying previous information (losing his place). However the system still has too much of a hand in when and how this switch is made, which leads us to the following:

2) Non-Preëmption: poor. A breakpoint or explicit program call to the USER function preëmpts the terminal for the new instance and context. When the user gains control via "attention", he is the instrument of this preëmption. This facility lessens the preëmptive behavior, but does not eliminate it.

3) Response time: moderately good. When a User procedure is active, response is good by all our measures. During a lengthy operation (e.g., a user's problem program execution), a new User instance can be asynchronously instantiated, again providing good response time, at the expense of having to remember (with some system help) to return control to lower levels later.

4) Single Language, Modes: as before, good.

5) Accessibility: present, but impaired. In most of these systems, the complexity of the name and allocation structures has increased slightly beyond the ability of the user interface to accommodate it.

6) Context: variable.

3.E2 Representative Systems

LCC [45] and all LISPs, at least with appropriate user-provided functions, perform as DPDA systems. BBN Lisp [53] exhibits this organization and, as we shall see, surpasses it in some important ways. There are also elements of DPDA behavior in Kay's FLEX design [28], upon which we also intend to elaborate, for it too exhibits major behavior match improvements over the systems in this category. The current incarnation of the ECL System [58], under refinement at Harvard, seems to fall into this category. We shall discuss Mitchell's SLICE system [44] briefly, chiefly because of improvements in technique and human engineering attributes which we have not stressed. Other DPDA/IPS systems include APL [26] and CCS [50].

3.F. ADVANCED IPS SYSTEMS

In this section we will consider the salient behavior match features of recent IPS systems, or designs, which have provided much of the guidance and inspiration for this work.

3.F1 BBN LISP

This system [53] behaves mostly as a DPDA system, with several distinct modes in its FSA component; some of the additional modes provide function editing capabilities and special facilities within breakpoints. Of Particular interest to us are the contributions which Teitelman has made to BBN LISP. These facilities first appeared in his thesis, [54], and have since been presented and elaborated in [53], [55].

Teitelman shares with us the desire for a system whose behavior complements the user's, allowing him to work more efficiently and effectively. His chief emphases, however, treat user attributes which we have not addressed:

1) **Errors.** People make errors when they speak, write, or type. Simple typographical, logical or spelling errors do not usually interfere with the comprehension of messages if the recipients are also people. It is therefore irritating and diverting to be forced to correct such simple errors in order to be understood. Most IPSs are very unforgiving of errors.

2) **Repetition.** A common act is to develop, by trial and error, a method for accomplishing something, then to apply that method again when similar situations arise.

Teitelman's provision for the first attribute is the DWIM (for "Do What I Mean") facility. This constitutes a refinement of the User procedure/system interface: DWIM routines intervene before the User procedure is called. They examine the reason for calling User procedure, and try to handle the situation themselves (e.g., by correcting simple spelling errors, or simple parenthesis blunders.) In the most common configuration, DWIM simply notifies the user of its actions and returns to the caller with the error corrected. Only when DWIM fails to find a solution does it invoke BBN-LISP's User procedure analogue.

If the User can anticipate more complex errors or exceptional conditions, he can have his program handle them by advising selected functions to take specified temporary actions before, during, after, or in lieu of their normal operations.

It is quite often possible for the BBN LISP user to cancel the effect of an operation, even a complicated one, using the undo command. This feature is a powerful error-correcting tool in combination with the DWIM features.

If a user anticipates the need, he can arrange, in most IPSs, to repeat a complex sequence of operations: he can create a macro or function to do it, then call it repeatedly. However, if he has simply carried out this sequence of operations, he must then recreate them in order to repeat them. BBN LISP maintains a **History List** of recent terminal operations, typically the last thirty or so. One can redo one or more recent operations by referring to entries in this list. One can also save a sequence of History entries for permanent accessibility as a Lisp function. We have attempted to refine this facility in our system (see Section 6.A1, the UCP Scene).

3.F2 SLICE

The system described here is the one Mitchell uses in his thesis [44] to describe his IPS methods. His system, a derivative of LCC, shares with LCC the DPDA/FSA/RED classification, and would submit to essentially the same attribute analysis. Its novelty lies in its translation algorithms.

Mitchell demonstrates that there is a spectrum of possibilities between a purely interpretive and a purely compiled system. He discusses the merits of the two approaches in terms of the inherently conflicting qualities of flexibility and efficiency. Flexibility is the ability to modify program and data elements interactively, to inquire intelligently about program operation, and to intervene in the flow of control. Efficiency in this case is a measure of the speed of execution of the user's program.

Mitchell supports his view that flexibility decreases while efficiency increases as one traverses the spectrum from interpreted to compiled programs. He then describes an interpreter-based system which illuminates his contentions. Mitchell's system interprets the source program by compiling and immediately executing sections of it as they are encountered, retaining the compiled code segments as a fortunate side-effect. By reusing the compiled segments as long as they remain valid, he obtains a system which smoothly traverses the spectrum from flexibility to efficiency as an algorithm is perfected, and as the frequency of program modification decreases. The keys to his methods are the algorithms and data structures he developed to detect and correct segments made invalid by modifications to source statements and declarations.

We shall have more to say about Mitchell's findings in Section 8.E6, for we have borrowed heavily from them in our translation methods.

3.F3 ECL

ECL is the result of research begun by Wegbreit in his thesis [57] on extensible languages for IPSs. The current effort is a large project, directed by Wegbreit and Cheatham [58] at Harvard, dedicated to the creation of a software laboratory. An interpreter and equivalent compiler for the ECL language, EL1, will allow operation at both ends of the flexibility/efficiency spectrum. A major goal of ECL is application of sophisticated software aids to the development of very large, complex systems (for instance, an automatic programming experiment) [8], without sacrificing ultimate efficiency.

Most of the novel aspects of ECL lie in areas not directly treated in this work; efficient extensible language design is foremost among them. In our Behavior Match terms, as we mentioned, ECL is at present a DPDA/FSA system. We are unaware of plans for enhanced terminal facilities at this writing. However, we believe that our methods would apply very nicely in the ECL environment.

3.F4 FLEX

The FLEX mini-computer and extensible language system form the central subject of Kay's dissertation, The Reactive Engine [28]. This system (and its successors, for it is still in a state of evolution), until now existing only in experimental versions, gives one as much power to define and control his own language and programs as any now available, on machines of any size. Kay has combined theories of language, software, and machine design in a comprehensive proposal for an easily learned, personal, and very powerful system.

In the domain of our Behavior Match attributes, FLEX and its derivatives possess qualities which we have found missing in other systems. Kay's philosophies have strongly influenced our design.

FLEX is a display-oriented system, incorporating a graphics tablet and a special keyset for convenient manipulative inputs, along with a standard keyboard for symbolic input. The built-in, extensible FLEX language allows concurrent operation of multiple processes. The full-blown system, written in FLEX (*), makes copious use of this ability, using parallel components in the hardware to allow scanning, parsing, compiling, and execution of programs to proceed concurrently. In this way, though a structured text representation of a program is the only permanent (and displayable) representation of that program, acceptably efficient execution is maintained. The system provides powerful display techniques, for editing and observing the operation of programs, for displaying structured textual and graphical data, and for "echoing" the user's input of structured data.

In our classification system, FLEX is a DPDA/IPS/RED system, whose stack environment is extended to the stack configuration (similar to that used in the B6700 computer [47] or in Simula implementations [14]) needed for the operation of concurrent active processes.

3.F5 FLEX Attribute Analysis

1) Multiple activities: good. The system makes use of multiple processes, and the user has control of them, both in his programs, and directly at the terminal.

2) Minimal Modes: excellent, due to its single input language.

3) Single Language: excellent. All commands are expressible in the user-extensible language. A few "invisible" edit commands duplicate some FLEX functions, for convenience in editing. Like Lisp, FLEX is "homoiconic": the executable and external representations of programs are essentially the same.

4 Accessibility: good. All active data are accessible to the User procedure, and the user can activate a User procedure in arbitrary active processes.

5 Context: very good. The display facilities allow presentation of user programs in context, and observation of their operation in that context. The user is free to provide additional context-rich displays in his programs and subsystems.

- -

(*) We are being intentionally vague about the distinction between the hardware and software. The machine is microcoded, essentially implementing the nucleus language and the system kernel.

6 Non-symbolic operations: excellent. The combination of the keyset and tablet supply impressive manipulative tools which enhance editing and graphical operations. The short, easy editing commands, and the ability of the user to extend his language, supplement these tools.

7 Non-preëmption: almost provided. The recursive (and now concurrently recursive), nested USER concept is maintained in the breakpoint and terminal interrupt structure of FLEX. It is not made clear what happens if two processes attempt to break at once. The user may "ride piggyback" on the program evaluator (observe its interpretation of the execution of his program, step by step), in order to follow and control the flow of operations in his multiple processing environment.

Kay would not necessarily stress these points as the most important topics of his work. We would therefore be doing him a disservice to suggest that we have captured the "essence of FLEX" in this short report. **The Reactive Engine** is a comprehensive work, which has contributions to make to most areas of system design.

3.G. ATTRIBUTE SUMMARY

Table 3-1 is a summary of the attribute analyses for the basic system categories we have studied. FLEX is included in a separate column, because it excels in many attributes.

Table 3-1. Behavior Match Attribute Summary

Category / Attribute	FSA/IS	FSA/IPS	FSA/IPS/RED	DPDA/IPS	FLEX
Multiple Activities	?	-	-	x	✓
Single Language	-	x	✓	✓	✓
Non-Preëmption	-	-	-	-	x
Response time	-	-	-	x	✓
Minimal modes	-	-	x	x	✓
Maximum context	?	-	?	?	✓
Accessibility	?	-	✓	✓	✓
Non-symbolic ops.	?	-	?	?	✓

-	These systems do not support this behavior; their implementors may not agree that such behavior is desirable.
x	All or most of these systems partially support this behavior.
?	This attribute is generally absent from these systems, although nothing in their basic designs prevents its inclusion.
✓	These systems support this behavior.

CHAPTER 4
DESIGN OF COPILOT

In this chapter we shall use the criteria of Chapter 2 to help specify the design of our experimental IPS, COPILOT. Here, we shall match the human traits to the corresponding desired behavior of the system. We shall also introduce additional design decisions (choice of language, method of interpretation, etc.) with reasons for their choice, although these do not relate directly to the behavior match topics. Finally we shall present an overview of the COPILOT system, with emphasis on the ways in which it meets the design criteria.

Subsequent chapters will present the COPILOT system in more detail.

4.A. ACHIEVING THE BEHAVIOR MATCH

4.A1 Use of Multiple Processes

If one accepts our assumptions, people can monitor multiple simultaneous external activities, and can maintain, at the conscious interface, multiple pseudo-parallel "processes", or "coroutines" of their own, while pursuing a task. They want to be able to schedule their own actions independent of the order or frequency of external requests (non-preëmption), but they desire rapid response, at least by acknowledgement, to their own requests for services.

To satisfy these requirements, we must first include facilities, in the language and operating environment of our IPS, for the specification of multiple processes, allowing programs to instantiate, activate, suspend and terminate "simultaneous" operations. Thomas [56] defines a process as "an activity comprised of a time-ordered sequence of actions". Within a computer system, a process is usually represented by an algorithm, specifying the sequence to perform, a collection of data elements upon which that algorithm can operate, and a pair of indicators, or environment pointers, which together identify the current point of operation within the algorithm, and the current active values within the process data. By alternating among sets of environment pointer pairs, a single computer, or processor, can, in large part, simulate the concurrent operation of more than one process. This allows the creation of the multiple processing (or multiprogramming) environment upon which this work is predicated. We will describe the specific COPILOT implementation in Section 5.C4.

The use of multiple-process facilities must be extended to the operation of the IPS itself. This, as we will show, allows us to provide the terminal user the ability to control processes directly. More importantly, we will use the multiple process discipline to provide the decoupling effect needed for non-preëmptive control with good response time. Our process structure comprises a high-priority User process, operating a User loop (see Section 4.C2), to listen to the keyboard and respond to its commands, combined with a Post process to maintain a display of the status of all processes. This allows the user's problem, or target, programs to run in one or more target processes, undisturbed by terminal operations except where interaction is intended. Conversely, these target processes are not allowed to disturb (preëmpt) the User process, so they cannot bother the user save by supplying status information to the Post process. The User process replaces the recursively instantiated User procedure of previous systems.

We also hope to show that an IPS which uses multiple process structures properly can operate very efficiently, in its use of both time and space, particularly when the interactive facilities are not actively in use.

4.A2 Use of Displays

We have argued that, ideally, one's statements should not be dependent on context for their interpretation (mode-minimization), but that one finds it easier to interpret communications when they are surrounded by appropriate contextual (environmental) information.

Applied to IPS design, this need for adequate and current context, along with the need for rapid response, nearly eliminates the traditional hard-copy sequential-character computer terminal as a feasible terminal output device. To achieve our context match, we require a graphic display device, which we will henceforth term a display. The most common displays today are CRT-based point, vector, or raster-scan (TV) devices.

Current display devices do not contain sufficient area and resolution to present even the minimum information we require to operate the system. Unless and until displays are improved, we must provide a reasonable alternative. Among currently possible alternatives are:

1) to use multiple display screens.
2) to implement multiple virtual display screens. This is possible if the available display hardware and software permits rapid replacement of a screen's contents.
3) to provide a very flexible mapping of groups of lines to areas of the display screen, so that the user or program can select the most important text "windows" at any time.

We have chosen to design COPILOT in terms of multiple display screens. It would not be difficult to modify the design to operate in the virtual screen mode of item 2 above. The third method would require considerable redesign; its performance under the best implementation would, we believe, be unsatisfactory, since it would require the user to remember too much about the complex, time-varying screen organization.

One important attribute of a display is its speed, allowing it to make large amounts of information, and therefore adequate context, continuously visible. Perhaps as important is its two-dimensional, random-access characteristics. We must be able to select and change one section of the screen without affecting any other section. Using these facilities we can partition the screen(s) into Regions at fixed positions, each devoted to a specific purpose: the display of a portion of a program, of some program data, of system status, or of information generated by the user. We can use this positional constancy to our advantage in achieving several of our other specific goals:

1) In support of our non-preëmptive control, the user knows where to look for information generated by various runr [illegible]cesses, so he need not constantly focus his attention on the output activity of his [illegible].

2) These processes can make the user aware of important occurrences (e.g. breakpoints) without interfering with his current activities.

3) Due to these visual reminders and event notices, the user can increase the number of simultaneous activities which he can oversee without forgetting about them or losing track of their operation.

Our goal here is to give the user a window into his system which is wide enough and clear enough that there is nothing more he needs to see, and to give him tools for directly manipulating those things he can see. He should be able to perform most necessary control and modification functions by pointing and editing operations (again with random access) on this visual context.

4.A3 Single Language

We have asserted that people communicate with each other in a single language, with lexical extensions for special purposes. Therefore, to achieve our behavior match, we must provide our user with a single language with which to communicate with our IPS. We must give it enough power not only to perform the user's algorithms, but also to carry out all terminal operations: editing, program control, variable-monitoring, etc. The User process need only accept statements in that language in order to provide all system functions. Conversely, because all terminal commands are elements of our language, the user can write readable procedures whose execution he can substitute for sequences of terminal operations. If the user's recent commands are saved, he can even create these procedures from recent operations. This facility eliminates the need for a special "macro" provision at the terminal. (†)

Any additional representations for programs (compiled code or other internal structures) must be totally hidden from the user: we must at all times preserve for him the illusion that he is operating directly in the chosen language. We we will describe methods for maintaining "equivalent" parallel representations for programs, their data, and other information at several structural levels. We will maintain programs, for instance, as executable machine code, as parse trees, and in an intermediate "parse token" representation.

4.A4 Abbreviation

Our observations have suggested that people avoid repetitive circumlocution by developing formal concise notations or informal colloquialisms (jargon, slang), depending on the formality of the subject. It is usually possible to map formal notations unambiguously into sentences in the base language. There are also tasks which people do that are manipulative rather than symbolic in nature.

We have attempted to provide both abbreviation and manipulative control in our IPS design. The User process, while accepting complete base language statements (sentences) will also accept shortened, abbreviated commands, each of which can be algorithmically expanded into syntactically correct language forms. We have attempted to implement the most common

- -

(†) In the TVEDIT system for the PDP-1 [48], for instance, one can give a name to a string of command characters. He can subsequently issue a command, with that name as its argument, which will cause that sequence of commands to be executed.

simple commands as single keystrokes; in this way we hope to achieve a "manipulative" feeling for these operations in the mind of the experienced user.

Because these facilities are available, we do not need to worry too much about the length or ungainly structure of our basic system-control statements. Most of them turn out to be simple intrinsic ("built-in") procedures and functions, called with many parameters. The standard abbreviations which use them typically supply all or most of the parameters by referring to current visual context. The result is a simple, flexible, and well-defined command structure, as well as a reduced number of basic primitives.

These abbreviations provide a simple macro processor, which responds to user input, and creates syntactically correct output. We have devoted little effort to the design of this facet of COPILOT, except to attempt to make simple operations simple to evoke, and to partition the system so that these front-end recognition algorithms can be replaced or altered, hopefully even by the user, without affecting the base language facilities. A good deal of relevant research into macro processing has appeared in the literature, and could be useful in improving the appearance of the system. For instance, [34] and [62] suggest possible improvements. We do feel that the simple schemes described in the sequel will suffice to exhibit the power of the concept.

4.B. ADDITIONAL DESIGN DECISIONS

The developments of the previous section follow directly from the behavior match requirements. As we stated in Section 2.B, there is still room for a variety of systems within this framework. This section will discuss some of these parameters, presenting the particular selections we have made in the COPILOT implementation. To a large extent these choices reflect the environment in which this research was begun. The goal was to provide an IPS built around a local programming system (SAIL, [52]).

However, in each of the following cases, we seem to have chosen from the more difficult end of the spectrum of possibilities. This is not necessarily laudable, nor even wise. It is, however, fortunate in the context of this document, since, if our appraisal of the relative difficulties is true, we can show that our IPS methods are widely applicable.

4.B1 Compiler-Oriented

The predominant form of IPS is built around an interpreter. In such a system, a source program is first converted to some internal form, retaining much or all of the symbolic and structural information of the original. This program structure then drives a system routine, called an interpreter, whose function is to carry out the actions specified by the algorithm.

In a compiler-oriented system translation is from source program to machine code, in which the algorithm can be executed directly on a computer. Neither the source, nor any intermediate structures used during the translation, are needed for correct program execution after compilation is complete.

Arguments in favor of an interpretive IPS are:

1) The interpreter is an active agent throughout the execution process. It is therefore easy to include in the interpretation algorithm facilities for continuous monitoring of special conditions, dynamically set breakpoints, etc.

2) Semantic information about all program entities (variables, expressions, etc.) must be maintained throughout execution. Interpreters usually use this to advantage, maintaining data types and other attributes dynamically. This late binding reduces the number of attributes the user must declare, and increases the flexibility of the language.

3) Since this semantic information (and other data which is of interest to the user: names, etc.) is retained anyway, most systems provide sophisticated interactive features which put this information to good use. This kind of information is typically lost when a program is compiled.

As we stated in Section 3.F2, Mitchell's factored interpreter methods can achieve the speed of compiled, though not necessarily optimal, code in a basically interpretive system. With some loss of flexibility, we have adapted Mitchell's methods to a system which maintains all user programs in compiled form, compiling changes as they are made, rather than just before the changed sections are executed. This allows us to avoid periodic return of control to the interpreter to check for modified sections, which in turn enables us to approach execution speeds competitive with batch systems. This is an important attribute for very large systems, which often run for long periods before requiring any interactive operations. The disadvantage to compiling before execution is that we may recompile the same section of code many times without executing it. Under some circumstances this will significantly degrade performance. Our method also makes it more difficult for us to accept incomplete programs.

When they are not interacting with the user, Copilot code segments do not require the services, nor even the presence in memory, of the IPS routines or data; nor do they require the presence of the higher-level program structures (e.g., text strings or parse trees). With proper memory management, this allows debugged, non-interactive programs to approach the size efficiency of conventional batch environments, without sacrificing the interactive facilities when they are needed. This performance is achieved at the expense of additional time and space overhead in the IPS routines. In Section 8.D we will present these "selective efficiency" methods in some detail.

4.B2 Static Block Structure

Another important design parameter for any programming system concerns the meaning of a name in that system: its scope (lexical and dynamic range of validity); how its value is obtained; and when this binding of name to value occurs.

None of these issues has any direct bearing on our main topics of study. However the choice we make has a large effect on the behavior of the language, and therefore on the overall behavior of the system. It has an immense effect on the difficulty of implementing the language in an interactive environment.

We must consider this choice in the light of our previous decision to build a compiler-based system. Here a modification to the definition of a name can have far-reaching effects. These changes are particularly difficult to handle incrementally, if the code compiled to gain access to that name must also be changed; e.g., if the name is bound to its access algorithm at compile time.

Such is the case, for example, with the static block structure employed in Algol 60 [46], but not with the dynamic scope rules used to access variables in LISP 1.5, where all non-local names are bound to their values whenever they are referenced at run time. The problem is compounded in Algol 60 by the static lexical scope, which tends (in practice) to distribute the effects of changing a global variable's declaration over a wider range than do other methods.

True to form, we have chosen to use the Algol block structure, again picking the more difficult end of the spectrum of possibilities. Fortunately, Mitchell's incremental compilation methods are equipped to handle this structure, and we shall use them in our design. The static Algol block structure affects our ability to display program variables conveniently, as we shall see.

4.B3 Emphasis on Large Systems

The typical IPS is oriented towards aiding the development of the small (however complex) program or system. Typical users are the beginning student of programming, and the occasional user. They require that the system be easy to learn and use, that it be helpful, and that it be resilient to erroneous inputs. Efficiency is usually a secondary issue. When programs grow too large to survive economically in an interpretive environment, their creators must abandon these highly interactive and context-rich programming systems for more traditional batch-oriented methods. A few systems have survived the enlargement fairly well, among them most LISP systems. The LISP user sacrifices some of the flexibility and interactive facility of the interpreter by compiling most functions. In exchange, he achieves a significant improvement in speed and size. (The user may replace a compiled function by its interpretable equivalent in LISP, so that if he anticipates the need to interact with a function before calling it, he may not suffer at all. However, there is a danger that a function which must be interpreted may be executed frequently enough to dominate execution time).

In our experience, very large programs need comprehensive interactive methods most. Small programs, even very complex ones, can usually be debugged with relatively unsophisticated aids. In larger systems, troubles are often the result of "second or third order effects". These effects can appear, due perhaps to new kinds of inputs, in routines long thought perfected, whose details may have been forgotten. Such a situation typically develops only after a lengthy input sequence which would be expensive (or in real-time situations, impossible) to reproduce. The user needs the ability to apply a wide range of interactive aids to the problem, wherever it occurs.

Many of our COPILOT design decisions are independent of the size and complexity of the programs we expect to handle. Where they are not, however, we have chosen in favor of large systems. This is the chief reason for our emphasis on efficiency through compiled code. It is the reason we segment the system so that IPS features can "retract" when idle. It is even partially responsible for our choice of a static block structure, since this name structure sacrifices less efficiency for its power than do other schemes.

We do not claim to be alone in decrying the neglect of large systems in IPS designs. Remedying it is an important goal of the BBN block-compiled LISP features [53], the ECL system at Harvard [58], MPS and Smalltalk, being independently developed at the Xerox Palo Alto Research Center, and Lisp70 under development at Stanford. All of them are highly interactive systems, embodying many of the principles we support (see also Section 3.F).

4.B4 No Automatic Program Composition

Most language processors place no restrictions on the assignment of language elements to text lines, the indentation of lines, or the spacing between elements on a line. The composition, or physical appearance, of a program strongly affects its readability. Not only do people disagree with each other concerning program composition rules, but a programmer may also vary the format he chooses from one program area to another. We have therefore chosen to do no recomposition of user programs, but to retain the form in which they are submitted. This does not preclude the provision of composition tools (e.g., Prettyprint in BBN-LISP), as optional facilities.

4.C. AN OVERVIEW OF THE COPILOT SYSTEM

The final sections of this chapter serve as an introduction to the next chapter, which is a rather detailed presentation of our experimental IPS implementation, COPILOT (‡). COPILOT, as it appears on paper, possesses most of the traits we have advanced. The current PDP-10 implementation falls considerably short of that, but is complete enough to demonstrate the feasibility and utility of our recommendations. Section 9.B deals with the aspects of COPILOT which we consider incomplete.

Our overview consists of pictorial examples which should give the reader (and vicarious user) a "feel" for the use of COPILOT. We begin with a description of what he would see on his screens.

(‡) The name is derived from Teitelman's "PILOT" -- used with permission.

4.C1 The Environment

We will describe the system as it might appear after a significant amount of dialogue has taken place, taking us from the initial state to something more typical. The user faces one or more display screens– in our implementation we require at least two. Referring to figures 4-1 and 4-2, the available display area has been segmented into several Regions, each displaying a portion, or window, of a text Scene. (*) The configuration shown is a simple one. This user's entire target (applications) program requires but one process. It therefore contains at one time at most one active statement, which we will call the Instruction Point (IP). Our user has simplified the situation by selecting for display only those few Scenes required to understand the operation of his program, at the current IP and EP (or Environment Point, indicating the current "record of execution", or active data environment). We call the current time t1.

The Region marked RPROG, available in one form or another in every IPS, is a representation of a window of the user's program. The program is stored and displayed in exactly the same form in which the user (or some program) created it. The context cursor ("▸" character) indicates the exact location of the IP in the program, at time t1. The language is MISLE, which claims Algol 60 as a distant ancestor.

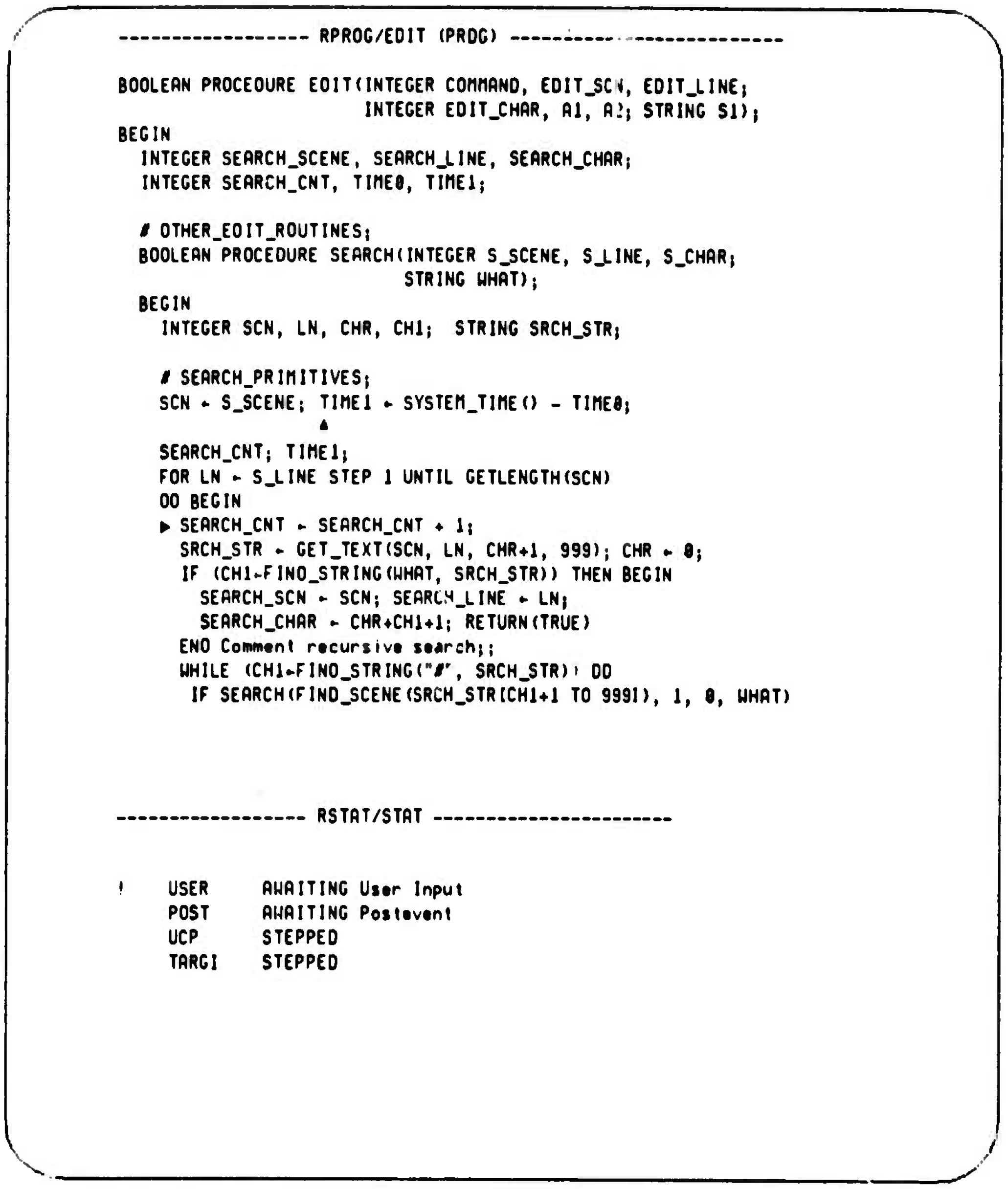
The RDATA Region is the visible representation of the instantaneous data environment, consisting of the names and values of selected variables at t1. The context cursor ("▸") here identifies the Environment point (EP), indicating the variables for the procedure most recently entered.

The RDYNA region reveals the dynamic state of the computation through a graphic representation of the process-stack configurations at time t1, while the RSTAT Region exhibits the current execution status of all processes (including in addition to the Target (applications) process the User and UCP processes which instantiate the basic IPS facilities).

These four Context Scene types nearly exhaust the COPILOT repertory, although unlimited additional user-defined Scene types are possible. A few secondary COPILOT Scene types are described in Section 5.B5.

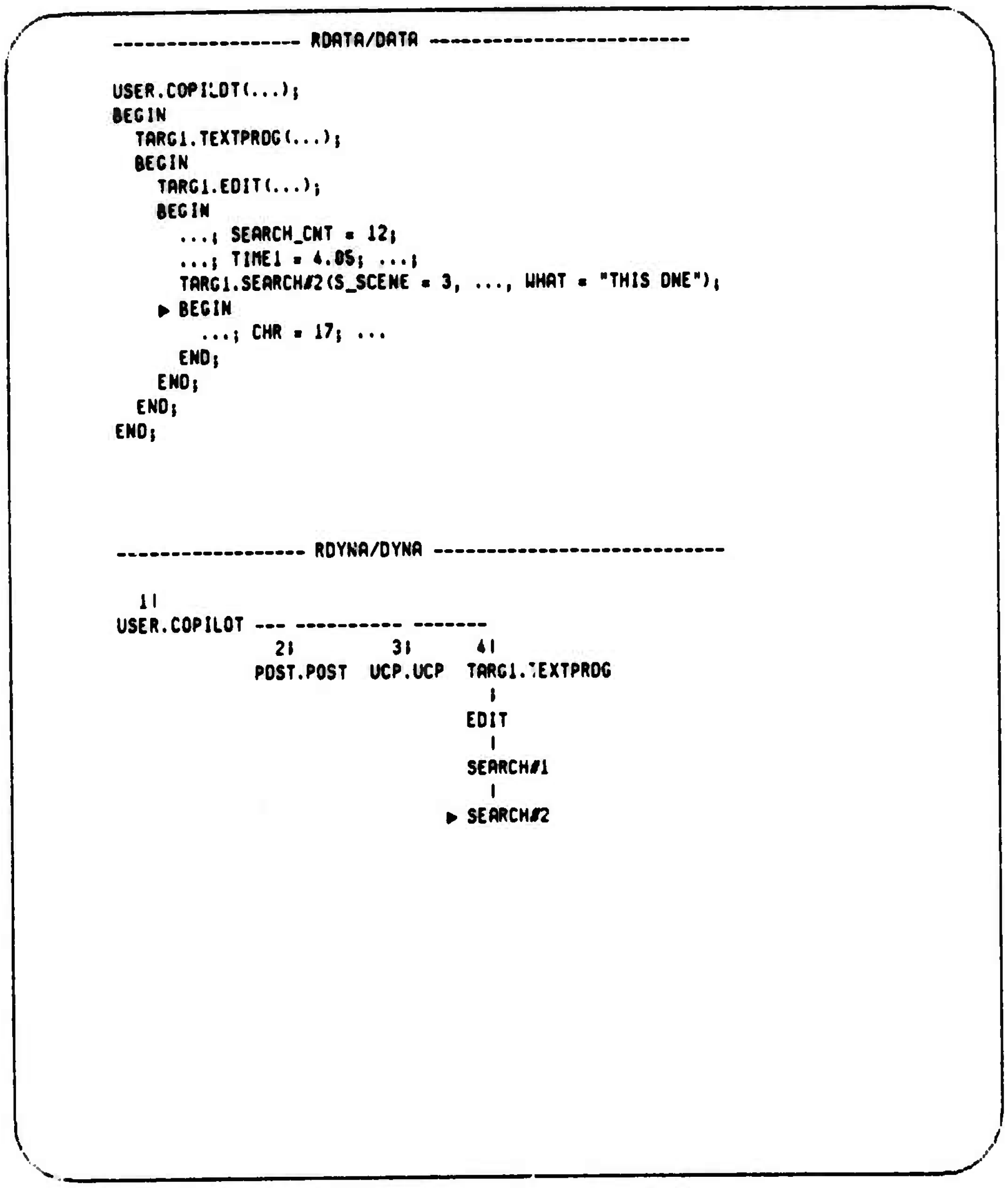
- -

(*) The labels at the top of each region name the entities represented there. They take the form <region>/<scene>(type), where the type entry is omitted if its name is the same as the scene.



```
------------------ RPROG/EDIT (PROG) ---------------------------

BOOLEAN PROCEDURE EDIT(INTEGER COMMAND, EDIT_SCN, EDIT_LINE;
                       INTEGER EDIT_CHAR, A1, A2; STRING S1);
BEGIN
  INTEGER SEARCH_SCENE, SEARCH_LINE, SEARCH_CHAR;
  INTEGER SEARCH_CNT, TIME0, TIME1;

  # OTHER_EDIT_ROUTINES;
  BOOLEAN PROCEDURE SEARCH(INTEGER S_SCENE, S_LINE, S_CHAR;
                           STRING WHAT);
  BEGIN
    INTEGER SCN, LN, CHR, CH1;  STRING SRCH_STR;

    # SEARCH_PRIMITIVES;
    SCN ← S_SCENE; TIME1 ← SYSTEM_TIME() - TIME0;
                   ▲
    SEARCH_CNT; TIME1;
    FOR LN ← S_LINE STEP 1 UNTIL GETLENGTH(SCN)
    DO BEGIN
    ▶ SEARCH_CNT ← SEARCH_CNT + 1;
      SRCH_STR ← GET_TEXT(SCN, LN, CHR+1, 999); CHR ← 0;
      IF (CH1←FIND_STRING(WHAT, SRCH_STR)) THEN BEGIN
        SEARCH_SCN ← SCN; SEARCH_LINE ← LN;
        SEARCH_CHAR ← CHR+CH1+1; RETURN(TRUE)
      END Comment recursive search;;
      WHILE (CH1←FIND_STRING("#", SRCH_STR)) DO
       IF SEARCH(FIND_SCENE(SRCH_STR[CH1+1 TO 999]), 1, 0, WHAT)



------------------ RSTAT/STAT -----------------------


!    USER     AWAITING User Input
     POST     AWAITING Postevent
     UCP      STEPPED
     TARGI    STEPPED
```

Figure 4-1. Typical COPILOT Scenes and Regions (screen 1)



```
------------------ RDATA/DATA -------------------------

USER.COPILOT(...);
BEGIN
  TARG1.TEXTPROG(...);
  BEGIN
    TARG1.EDIT(...);
    BEGIN
      ...; SEARCH_CNT = 12;
      ...; TIME1 = 4.05; ...;
      TARG1.SEARCH#2(S_SCENE = 3, ..., WHAT = "THIS ONE");
    ▶ BEGIN
        ...; CHR = 17; ...
      END;
    END;
  END;
END;



------------------ RDYNA/DYNA ----------------------------

  1|
USER.COPILOT --- ---------- -------
               2|        3|      4|
            POST.POST  UCP.UCP  TARG1.TEXTPROG
                                  |
                                EDIT
                                  |
                                SEARCH#1
                                  |
                              ▶ SEARCH#2
```

Figure 4-2. Typical COPILOT Scenes and Regions (screen 2)

Almost any modern computer terminal keyboard and operating system interface would suffice for a COPILOT-like system. Ours (see Figure 4-3) can communicate with the program one character at a time when desired, increasing the possibilities for abbreviation. These possibilities are further multiplied by the TOP, CONTROL, and META keys. These keys, like the alphabetic SHIFT, allow multiple-interpretation of each character. TOP selectes an alternate character, while the remaining two simply qualify the selected basic code. We will use "αA" for CONTROL-A, "βB" for META-B, and "•C" for CONTROL-META-C.

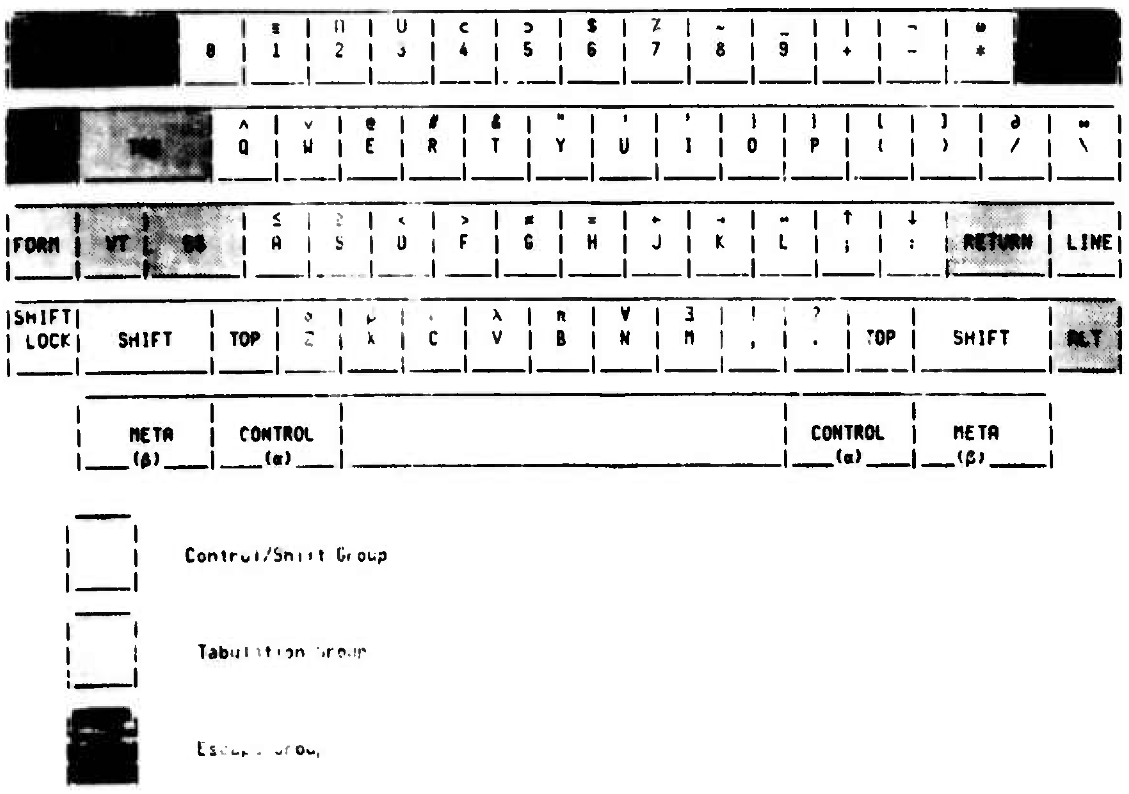


Figure 4-3. The Stanford AI Project Keyboard

4.C2 Basic Dialogue

The IPS must provide the routines for reading what the user types, and for invoking the facilities of the IPS in response. We have said that the nature of these interface routines establishes the behavior of the IPS, and thus the (interface) behavior the user must exhibit. We are now in a position to treat the interface behavior of our system in some detail.

We will call that routine which controls the operation of the user-IPS interface the basic control loop, or User loop. Its existence is at least implicit in all the IPSs reviewed in Chapter 3; usually it is quite explicit, forming the central control for the entire system. The basic User loops are remarkably similar from one system to the next. When the loop gains control (in a fashion to be described later), it performs approximately these functions:

1) Accepts one command from the keyboard.
2) Deciphers its meaning, and carries out its intent.
3) Reports the results, if necessary.
4) Returns to step 1.

An elegant example of this sort of algorithm is the top level of most LISP systems (e.g. [49]). This algorithm, itself expressed in LISP, can be approximately stated in the LISP M-expression language [40] as:

```
L: λ[];prog2[print[eval[read[],NIL]];L]
```

or, using the less pure PROG form

```
prog[];[L: print[eval[read[],NIL]; go[L]].
```

Although not all IPS implementations can express it quite this succinctly, they all have something like this Read-Eval-Print User loop operating at the command level. Though all are similar, there are important differences between these User loops. One is the nature of the commands supplied to the 'Read' function: in an IPS these commands are usually statements (S-expressions) in the single source language. The User loops of the various IPSs can be distinguished from each other by the ways in which they are able to gain control, the times when that is possible, and the meaning of statements for a given instance of the User procedure or process (the scope of interpretation). In general, they differ in the relationships between the basic control routines and the remainder of the system.

To a user familiar with any of these systems the User loop in COPILOT will present no immediate surprises. Commands in the form of MISLE statements are accepted sequentially from the keyboard, and *usually* are carried out in incoming order. Results of user commands, if they need to be reported, are revealed by changes in the text displayed in the appropriate Regions of his screen. As long as the operations to be performed are simple, commands and actions progress alternately, as in JOSS or BBN LISP. By describing situations designed to demonstrate the non-preëmptive aspects of COPILOT we shall soon shatter this illusion, but for the present we shall retain it.

COPILOT commands are available for editing program (and other) text, for examining program data, for controlling program operation, or simply for their effect as statements (to test program sections, or for "desk calculator" operations). Figures 4-4 and 4-5 are continuations of the picture sequence begun in Figure 4-1, showing the effects of COPILOT operations on the contents of the user's screens. Regions are sometimes shown in different positions from figure to figure in these examples, to minimize the information in each figure (in the actual system, the Regions would remain in fixed screen and line positions). A Region is shown only when there is a significant change in its data. Each figure represents the state of the Regions it shows *after* execution of the commands which accompany that figure.

The locus of user activity is indicated by the edit-cursor, a "▲" character beneath a selected character position in one Scene. Most of the editing primitives (EDIT_CHAR, INSERT_LINE, etc.) use the location of the edit cursor.

The entries in the COMMANDS column are the actual character strings the user types to perform the functions described in the examples. Entries in the EXPANDED column are the actual MISLE statements which he could type to get the same effect. Table 4-1 briefly describes the functions of the commands used in these examples. More complete descriptions of these commands and their expansions appear in Chapter 7.

Table 4-1. Commands Used in Chapter 4 Examples

COMMAND	MEANING
⊛n<cr>	Move the edit cursor ("▲") down n lines (n is a number, and <cr> means "carriage return").
⊛nαF<char>	Move the edit cursor to the nth occurrence of the character <char> following the current cursor position.
⊛:	Move the edit cursor to the first token of the statement which begins nearest the current cursor position.
<char>	Place <char> in the current edit cursor position. Replace any character which might already be there.
⊛nαD	Delete n characters.
⊛;	Move the control cursor (instruction or environment point indicator ("▶")) to the edit cursor position.
⊛,	Move the edit cursor to the control cursor position.
⊛B	Set a breakpoint (insert a BREAK statement, see below) at the statement nearest the edit cursor.
⊛P	Allow the process indicated by the DATA Region containing the EP control cursor ("▶") to proceed. This is usually used to resume a process after a break.
⊛X	Allow the process identified by the EP cursor to execute one statement, identified by the IP cursor.
⊛S	If the statement at IP contains substatements, allow the process to continue to its first substatement. Otherwise, this command is the same as ⊛X.
⊛&<string>	Execute the statement specified by <string>, in the environment specified by the EP context cursor ("▶").
⊛▶	Make visible the PROG, DATA, and DYNA Scenes corresponding to the most recently broken process.

●M<str><cr>	The string <str> is the name of new data which replaces the current data in the Region containing the edit cursor.
●n●R	Move the edit cursor to the last position it occupied in the Region n Regions away from the current one, where Regions are arranged in a reasonable circular order.

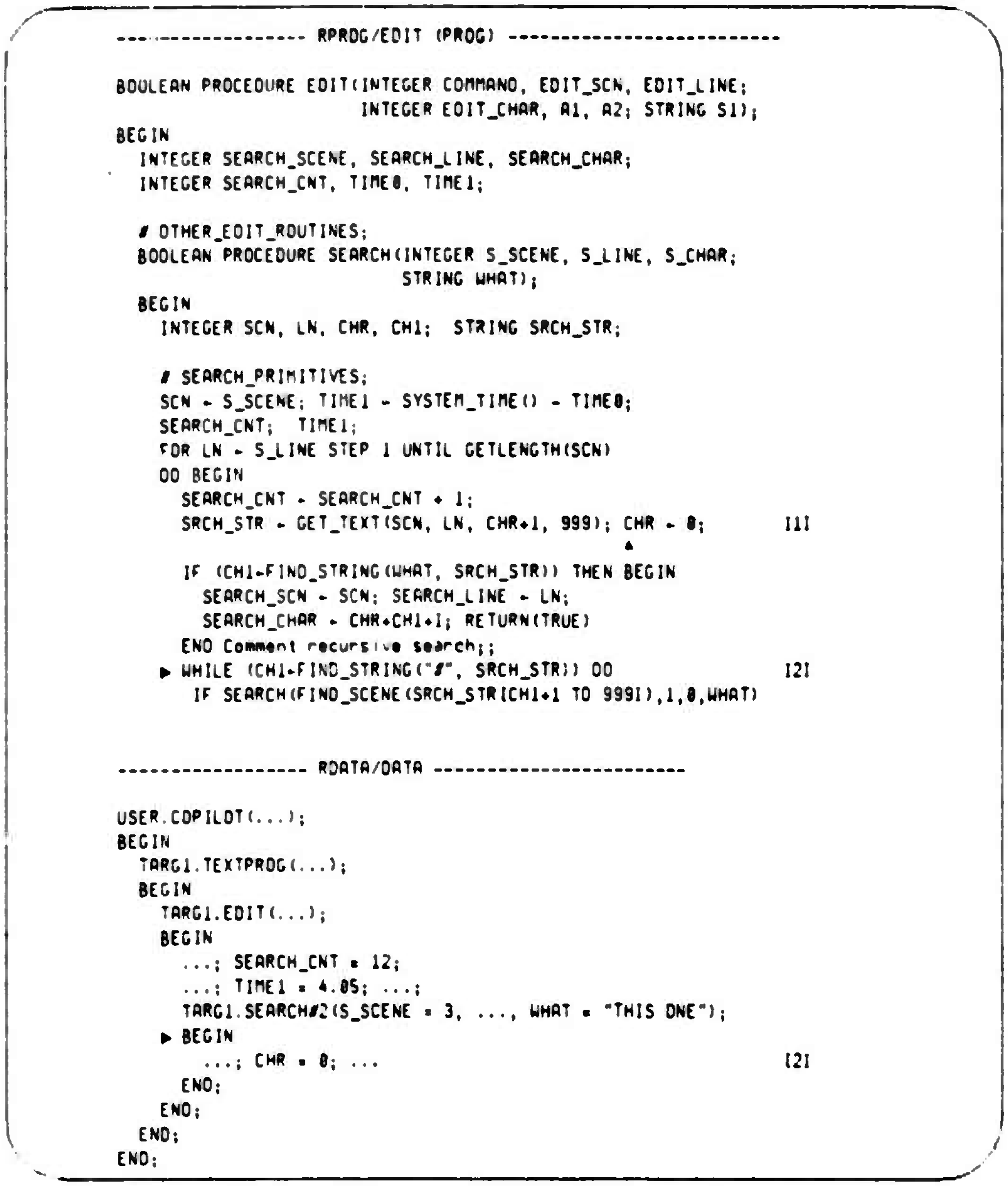
– A statement containing only an expression means that that expression's current value should be displayed in a DATA Region (e.g., "J;").

– "BREAK(proc)" will cause the process named proc to suspend when it encounters the BREAK statement.

– "{ s1; s2; ... sn } sm" , where s1, etc. are statements, is equivalent to "BEGIN s1; ... sn; sm END". See Section 7.D3, which describes these temporary statements.

After examining these figures, it should be clear why some form of abbreviation is desirable. A user should not be forced to submit a "mouthful" like "MOVE_CURSOR(...)" simply to reposition his edit-cursor, although the same string might be the best form (for precision and legibility) to include in a program ("macro") to position the cursor. Consequently, we have caused the command "<cr>" (carriage return) to perform the same action as the MOVE_CURSOR operation in Figure 4-4, by a mechanism explained in Section 6.B2. In fact, the form marked COMMAND in each of our examples is the preferred form of *direct* input to our User loop: the expanded forms are always available for inclusion in programs and for documentation.

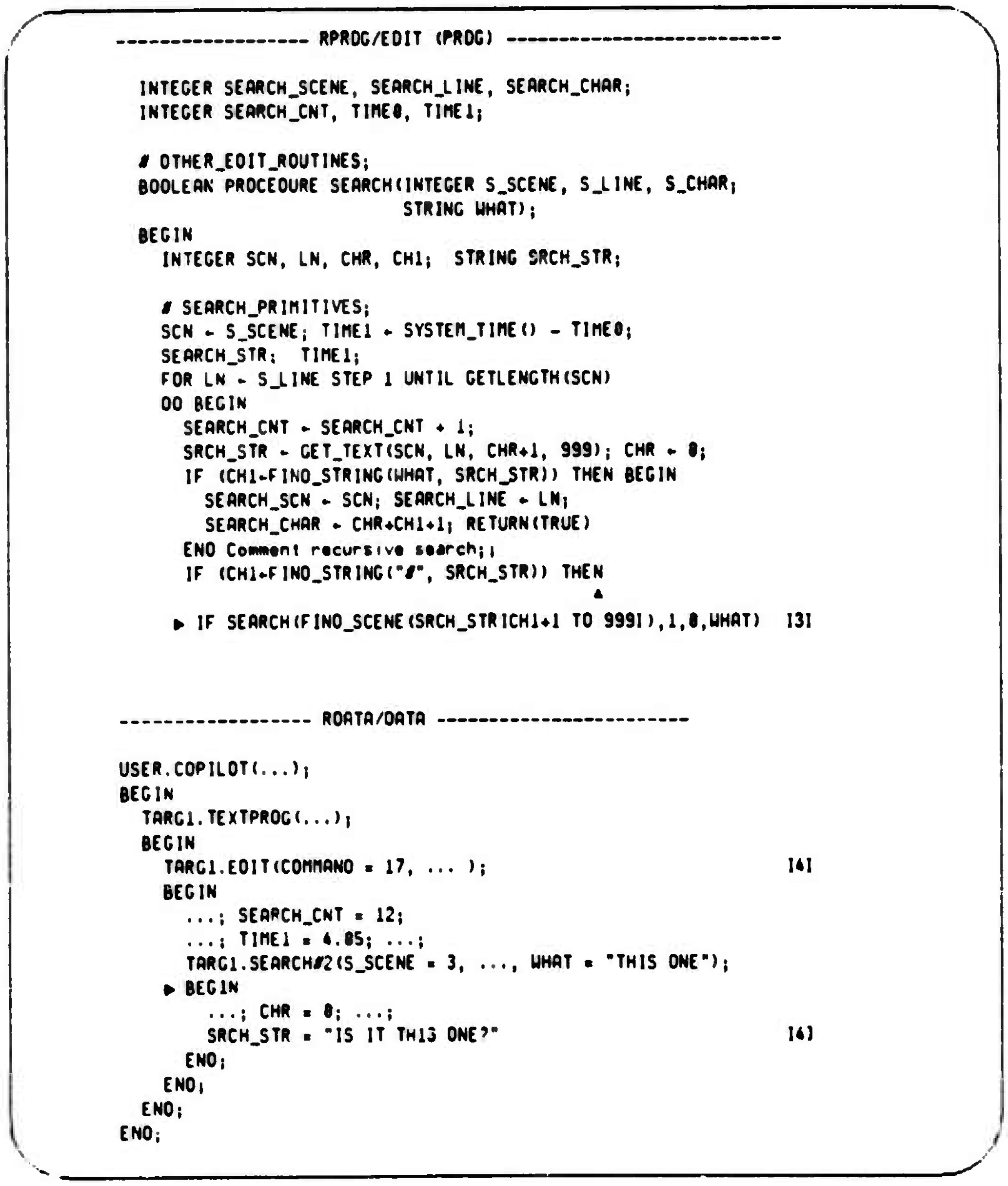
Notice that the data display statements of Figure 4-5 are executed for their effect on the program, operating in the program's environment. Others operate essentially in the environment of the system (the "interpreter"). We will show these relationships in detail in Section 7.C8. This distinction is a very important one, the subject of a great deal of study by Fisher [21] and others (for instance, Bobrow and Wegbreit in [6]).



```
------------------- RPROG/EDIT (PROG) -------------------------

BOOLEAN PROCEDURE EDIT(INTEGER COMMAND, EDIT_SCN, EDIT_LINE;
                       INTEGER EDIT_CHAR, A1, A2; STRING S1);
BEGIN
  INTEGER SEARCH_SCENE, SEARCH_LINE, SEARCH_CHAR;
  INTEGER SEARCH_CNT, TIME0, TIME1;

  # OTHER_EDIT_ROUTINES;
  BOOLEAN PROCEDURE SEARCH(INTEGER S_SCENE, S_LINE, S_CHAR;
                           STRING WHAT);
  BEGIN
    INTEGER SCN, LN, CHR, CH1;  STRING SRCH_STR;

    # SEARCH_PRIMITIVES;
    SCN ← S_SCENE; TIME1 ← SYSTEM_TIME() - TIME0;
    SEARCH_CNT;  TIME1;
    FOR LN ← S_LINE STEP 1 UNTIL GETLENGTH(SCN)
    DO BEGIN
      SEARCH_CNT ← SEARCH_CNT + 1;
      SRCH_STR ← GET_TEXT(SCN, LN, CHR+1, 999); CHR ← 0;       [1]
                                                ▲
      IF (CH1←FIND_STRING(WHAT, SRCH_STR)) THEN BEGIN
        SEARCH_SCN ← SCN; SEARCH_LINE ← LN;
        SEARCH_CHAR ← CHR+CH1+I; RETURN(TRUE)
      END Comment recursive search;;
    ▶ WHILE (CH1←FIND_STRING("#", SRCH_STR)) DO                [2]
       IF SEARCH(FIND_SCENE(SRCH_STR[CH1+1 TO 999]),1,0,WHAT)

------------------- RDATA/DATA ------------------------

USER.COPILOT(...);
BEGIN
  TARG1.TEXTPROG(...);
  BEGIN
    TARG1.EDIT(...);
    BEGIN
      ...; SEARCH_CNT = 12;
      ...; TIME1 = 4.05; ...;
      TARG1.SEARCH#2(S_SCENE = 3, ..., WHAT = "THIS ONE");
    ▶ BEGIN
        ...; CHR = 0; ...                                      [2]
      END;
    END;
  END;
END;
```

```
COMMAND          EXPANDED                            COMMENT

*4<cr>     MOVE_CURSOR(CRNT_REG, 4, 0, 0, 0);     [1] Move the edit-cursor (▲) down 4 lines,
oF;        FIND_STRING(CRNT_REG,";",1);               then out to the first "C" after a ";"
oFC        FIND_STRING(CRNT_REG,"C",1);

*;         SET_P(GET_PROCESS(EP),                 [2] Move the context cursor (▶), identifying
                 EDIT_STRUCT(CRNT_REG) );             the IP (Instruction Point) to the
*X         STEPP(GET_PROCESS(EP), "↓");              edit-cursor loc., then execute two
*X         STEPP(GET_PROCESS(EP), "↓");              stmts. The assignment to the
                                                      variable CHR has changed its value
                                                      from 17 in the previous diagram to
                                                      0 in this one.
```

Figure 4-4. Simple Editing and Execution Control (part 1)



```
------------------ RPROG/EDIT (PROG) -------------------------

  INTEGER SEARCH_SCENE, SEARCH_LINE, SEARCH_CHAR;
  INTEGER SEARCH_CNT, TIME0, TIME1;

  # OTHER_EDIT_ROUTINES;
  BOOLEAN PROCEDURE SEARCH(INTEGER S_SCENE, S_LINE, S_CHAR;
                           STRING WHAT);
  BEGIN
    INTEGER SCN, LN, CHR, CH1;  STRING SRCH_STR;

    # SEARCH_PRIMITIVES;
    SCN ← S_SCENE; TIME1 ← SYSTEM_TIME() - TIME0;
    SEARCH_STR;  TIME1;
    FOR LN ← S_LINE STEP 1 UNTIL GETLENGTH(SCN)
    DO BEGIN
      SEARCH_CNT ← SEARCH_CNT + 1;
      SRCH_STR ← GET_TEXT(SCN, LN, CHR+1, 999); CHR ← 0;
      IF (CH1←FIND_STRING(WHAT, SRCH_STR)) THEN BEGIN
        SEARCH_SCN ← SCN; SEARCH_LINE ← LN;
        SEARCH_CHAR ← CHR+CH1+1; RETURN(TRUE)
      END Comment recursive search;;
      IF (CH1←FIND_STRING("#", SRCH_STR)) THEN
                                               ▲
     ▶ IF SEARCH(FIND_SCENE(SRCH_STR[CH1+1 TO 999]),1,0,WHAT)   [3]



------------------ RDATA/DATA ------------------------

USER.COPILOT(...);
BEGIN
  TARG1.TEXTPROG(...);
  BEGIN
    TARG1.EDIT(COMMAND = 17, ... );                             [4]
    BEGIN
      ...; SEARCH_CNT = 12;
      ...; TIME1 = 4.05; ...;
      TARG1.SEARCH#2(S_SCENE = 3, ..., WHAT = "THIS ONE");
    ▶ BEGIN
        ...; CHR = 0; ...;
        SRCH_STR = "IS IT THIS ONE?"                            [4]
      END;
    END;
  END;
END;
```

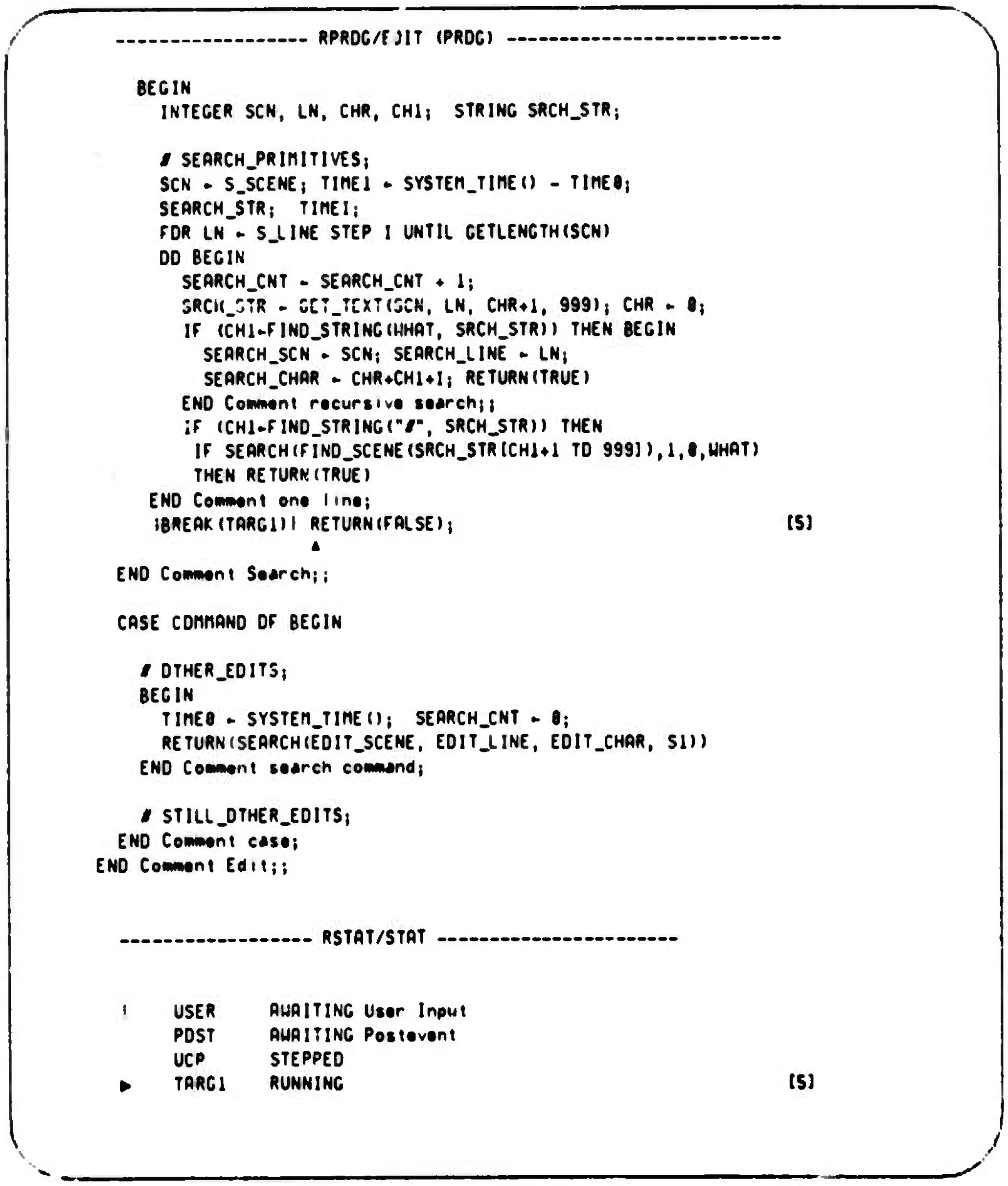
```
COMMAND        EXPANDED                              COMMENT

«,      SET_CURSOR(GET_REGION(IP),              [3] Now bring the edit cursor to the new
           GET_LINE(IP), GET_COLUMN(IP), -1);       context cursor (IP) position, change the
IF      EDIT_CHAR(CRNT_REG,"IF",0)                  "WHILE" to an "IF" (replace "WH" by "IF",
«3∘D    EDIT_CHAR(CRNT_REG,NULL,-3)                 then delete "ERE"), and the "DO" to "THEN".
«2∘FD   FIND_STRING(CRNT_REG,"D",2)                 Then "step in" to the statement at IP by
THEN    EDIT_CHAR(CRNT_REG,"THEN",0)                executing the (successful) test and suspending
«S      STEPP(GET_PROCESS(EP),"←")                  at the substatement.
«&SRCH_STR;<cr>                                 [4] Finally, execute data-display operations
        EVAL("SRCH_STR;",IP,EP)                     to inspect (and retain in view) some
«&COMMAND;<cr>                                      additional variables.
        EVAL("COMMAND;",IP,EP)
```

Figure 4-5. Simple Editing and Execution Control (part 2)

4.C3 A glimpse of Non-preëmption

The User-loop of COPILOT is continuously active. This means that, within second or two (a reasonable response interval) after accepting one command, it will be ready to accept (and act on) another. We have arranged to implement those operations which require longer intervals as separate, lower-priority processes, in order to maintain this response. Chief among these other processes are the user's target (applications) processes.

Figures 4-6 through 4-8 portray a sequence which we hope will not appear too contrived. Program-editing statements (expanded from the simple ●B command) first add a BREAK (breakpointing) statement temporarily. Then (Figure 4-6) the ●P (proceed) statement allows processing to continue from (IP, EP) in the Target process. The breakpoint has been planted to detect an unexpected condition, and the user knows that whether or not this condition develops, execution will take some time. He therefore (Figure 4-7) issues commands to change some of his Regions, selecting a new Scene for view in the PROG Region and cutting off most visual contact with the TARG1 process, which continues to operate, indicating its progress by occasional changes in the TIME1 and SEARCH_CNT variables. In this instance the new Scene (SUBST) is a piece of code which he has just begun to compose. Because the process(es) implementing the User loop algorithm operate at a high priority, his editing commands (Figure 4-8) receive service as they come in, "stealing cycles" from his running target, or applications, process. In short, he has been able to initiate an external operation, then to shift his locus of interest, while monitoring some aspects of the previous operation. He has issued a stream of interspersed editing, debugging, and program control operations. He has accomplished this, we contend, with no noticeable loss of continuity, from his standpoint. We have an IPS which satisfies our multiple-process, minimal mode, rich-context criteria.



```
------------------ RPROG/EDIT (PROG) -------------------------

  BEGIN
    INTEGER SCN, LN, CHR, CH1;  STRING SRCH_STR;

    # SEARCH_PRIMITIVES;
    SCN ← S_SCENE; TIME1 ← SYSTEM_TIME() - TIME0;
    SEARCH_STR;  TIME1;
    FOR LN ← S_LINE STEP 1 UNTIL GETLENGTH(SCN)
    DO BEGIN
      SEARCH_CNT ← SEARCH_CNT + 1;
      SRCH_STR ← GET_TEXT(SCN, LN, CHR+1, 999); CHR ← 0;
      IF (CH1←FIND_STRING(WHAT, SRCH_STR)) THEN BEGIN
        SEARCH_SCN ← SCN; SEARCH_LINE ← LN;
        SEARCH_CHAR ← CHR+CH1+1; RETURN(TRUE)
      END Comment recursive search;;
      IF (CH1←FIND_STRING("#", SRCH_STR)) THEN
       IF SEARCH(FIND_SCENE(SRCH_STR[CH1+1 TO 999]),1,0,WHAT)
       THEN RETURN(TRUE)
   END Comment one line;
   !BREAK(TARG1)! RETURN(FALSE);                        [S]
                  ▲
END Comment Search;;

CASE COMMAND OF BEGIN

  # OTHER_EDITS;
  BEGIN
    TIME0 ← SYSTEM_TIME();  SEARCH_CNT ← 0;
    RETURN(SEARCH(EDIT_SCENE, EDIT_LINE, EDIT_CHAR, S1))
  END Comment search command;

  # STILL_OTHER_EDITS;
 END Comment case;
END Comment Edit;;


------------------ RSTAT/STAT -----------------------


!    USER     AWAITING User Input
     POST     AWAITING Postevent
     UCP      STEPPED
▶    TARG1    RUNNING                                   [S]
```

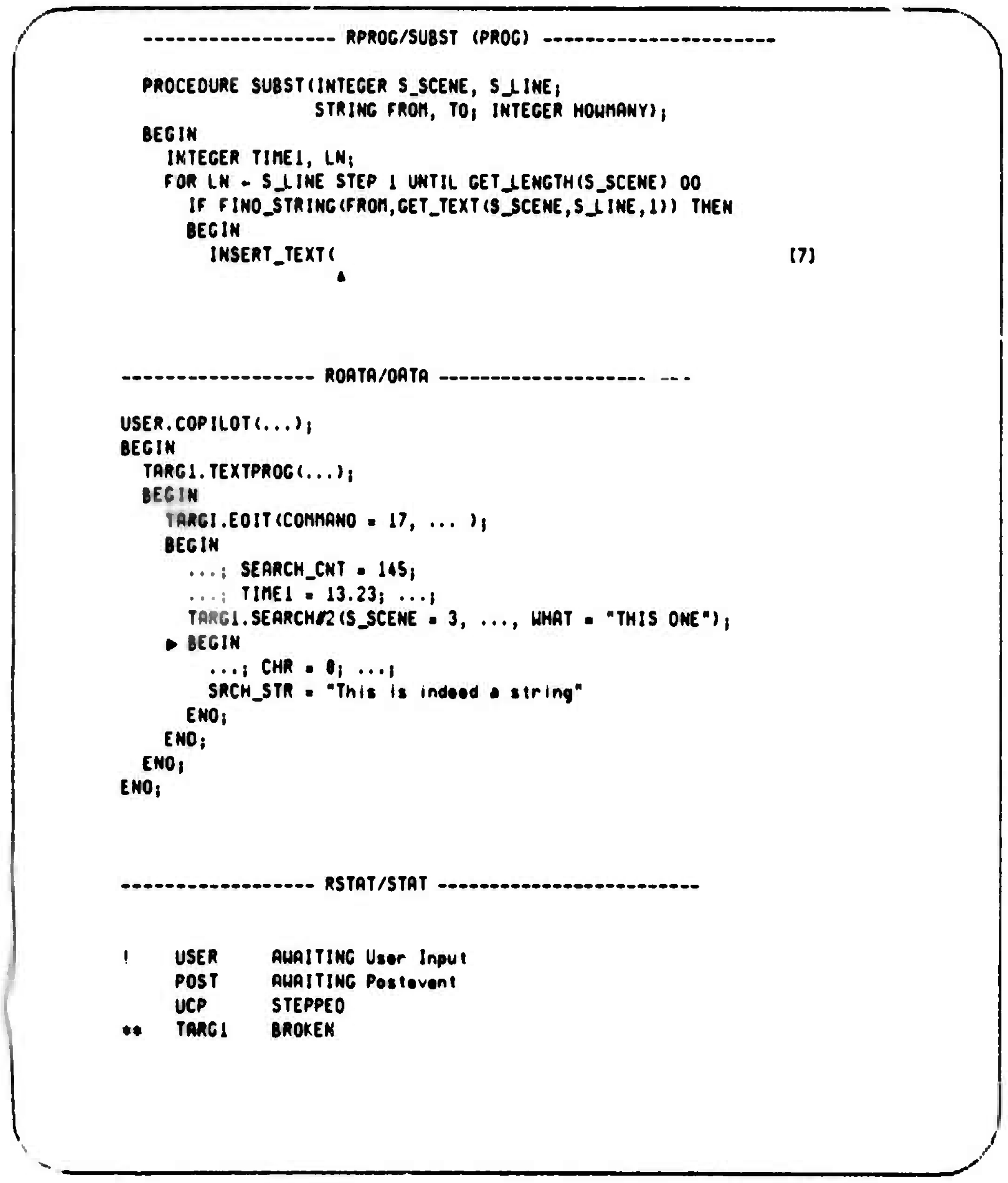
COMMAND	EXPANDED	COMMENT
⌘4<cr>	MOVE_CURSOR(CRNT_REG,4,0,0,0);	[S]Now "plant a breakpoint" (the temporary "BREAK(TARG1)") at a point which will only be reached if an error occurs, and let the process proceed.
⌘:	STRUCT_MOVE(CRNT_REG,":");	
⌘B	EDIT_CHAR(CRNT_REG,"!BREAK(TARG1)!",1);	
⌘P	ACTIVATE(GET_PROCESS(EP));	

Figure 4-6. Control of Multiple Processes (part 1)

```
      ------------------ RPROG/SUBST (PROG) ---------------------

      PROCEDURE SUBST(INTEGER S_SCENE, S_LINE, S_CHAR;
                      STRING FROM, TO; INTEGER HOWMANY);
      BEGIN
          INTEGER TIME1, LN;
          FOR LN ← S_LINE STEP                                   [6]
                             ▲




    ------------------ RDATA/DATA ------------------------

    USER.COPILOT(...);
    BEGIN
      TARG1.TEXTPROG(...);
      BEGIN
        TARG1.EDIT(COMMAND = 17, ... );
        BEGIN
          ...; SEARCH_CNT = 116;
          ...; TIME1 = 10.87; ...;
          TARG1.SEARCH#2(S_SCENE = 3, ..., WHAT = "THIS ONE");
          BEGIN
            ...; CHR = 0; ...;
            SRCH_STR = "A RANDOM SEARCH STRING"
          END;
        END;
      END;
    END;



  -------------------- RDYNA/CALSEQ (USER) ----------------        [6]

    Calling Sequences for Editing Primitives

      L ← GETLENGTH(SCENE)             returns number of lines in Scene
      S ← GET_TEXT(SCENE,LINE,         returns a selected substring, not
            STARTCHR, ENDCHR)          to exceed remaining length of line.
      B ← FIND_STRING("FOR", "IN")     TRUE if FOR in IN, FALSE oterwise.
      ...
```

```
COMMAND              EXPANDED                               COMMENT

*2*R           EDIT_REGION(NEXT_REGION(CRNT_REG,2),     [6]Move the edit cursor to the RDYNA
                                      -1, -1, -1);         Region, temporarily change its
*MCALSEQ<cr>   MAP_SCENE(CALSEQ,CRNT_REG,1,1,1);           Scene to one containing a
*-*2*R         EDIT_REGION(NEXT_REGION(CRNT_REGN,-2),      function-description dorument,
                                      -1, -1, -1);         then go back, switch the RPROG
*MSURST<cr>    MAP_SCENE(CRNT_REGN,SUBST);                 Region to a text Scene for a
 ...STRIN...   EDIT_CHAR(CRNT_REG,"...ST...",0);           routine under development, and
 ...BEGIN...   EDIT_CHAR(CRNT_REG,"...BEGIN..",0);         begin editing it.  RSTAT and RDYNA
    ...             ...                                    are still monitoring the activity
                                                           of the running process (TARG1).
```

Figure 4-7. Control of Multiple Processes (part 2)

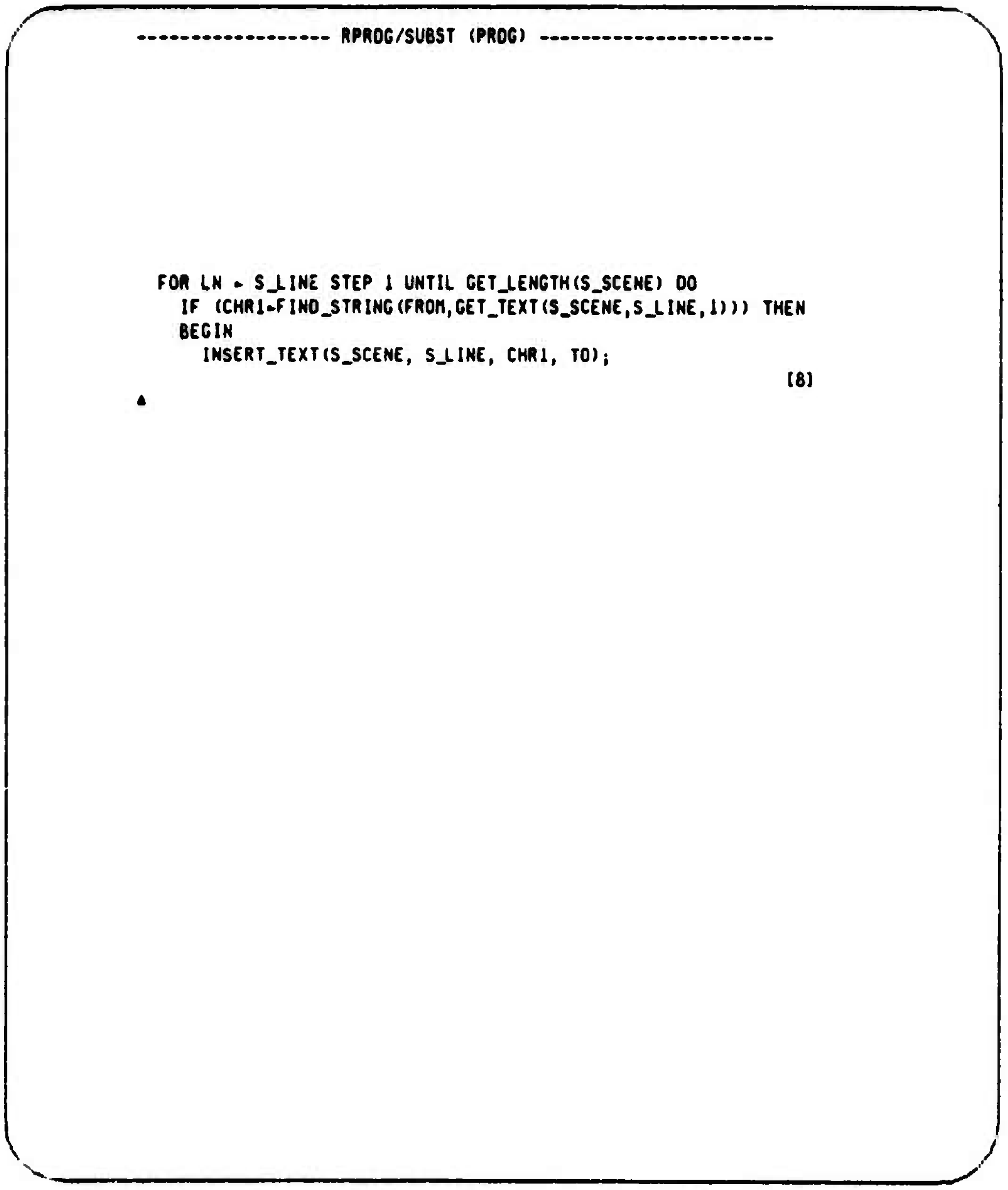


```
------------------ RPROG/SUBST (PROG) ---------------------

PROCEDURE SUBST(INTEGER S_SCENE, S_LINE;
               STRING FROM, TO; INTEGER HOWMANY);
BEGIN
  INTEGER TIME1, LN;
  FOR LN ← S_LINE STEP 1 UNTIL GET_LENGTH(S_SCENE) DO
    IF FIND_STRING(FROM,GET_TEXT(S_SCENE,S_LINE,1)) THEN
    BEGIN
      INSERT_TEXT(                                    [7]
                  ▲



------------------ RDATA/DATA ------------------- ---

USER.COPILOT(...);
BEGIN
  TARG1.TEXTPROG(...);
  BEGIN
    TARG1.EDIT(COMMAND = 17, ... );
    BEGIN
      ...; SEARCH_CNT = 145;
      ...; TIME1 = 13.23; ...;
      TARG1.SEARCH#2(S_SCENE = 3, ..., WHAT = "THIS ONE");
    ▶ BEGIN
        ...; CHR = 0; ...;
        SRCH_STR = "This is indeed a string"
      END;
    END;
  END;
END;



------------------ RSTAT/STAT -----------------------


!     USER     AWAITING User Input
      POST     AWAITING Postevent
      UCP      STEPPED
**    TARG1    BROKEN
```

COMMAND	EXPANDED	COMMENT
INTEGER TI..	EDIT_CHAR(CRNT_REG,"...IN..",0);	[7]Even though the TARG1 process has suspended at the BREAK statement, continue editing the SUBST Scene (an example of non-preemption).
FOR LN ..	EDIT_CHAR(CRNT_REG,"...FO..",0);	
BEGIN..	EDIT_CHAR(CRNT_REG,"...BEGIN..",0);	
...	...	

Figure 4-8. Control of Multiple Processes (part 3)

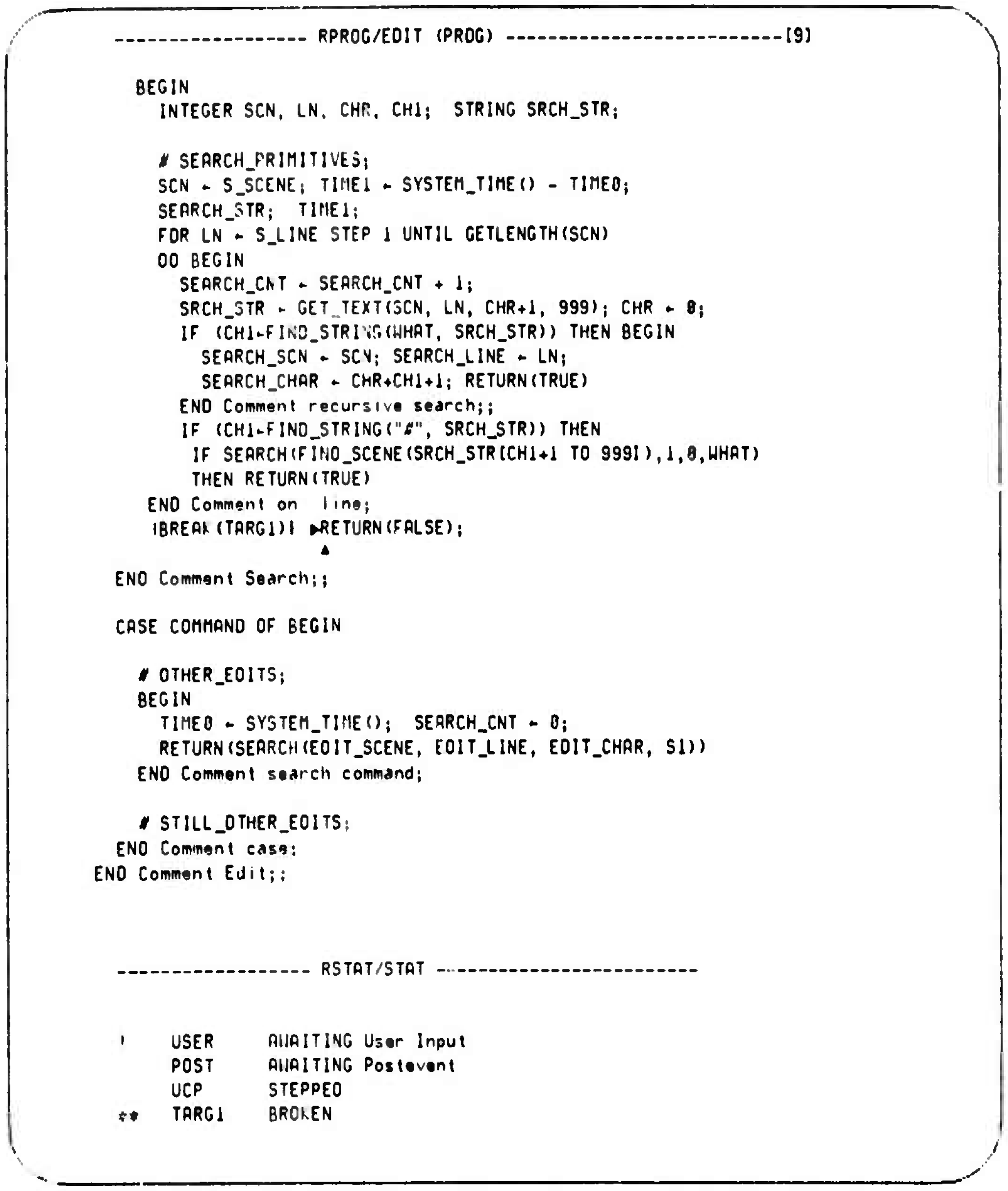
Figures 4-8 through 4-11 provide our last example, demonstrating non-preëmption. In Figure 4-8 the STAT Scene indicates suspension of the TARG1 process due to a BREAK statement, and flashes the asterisk (at one-second intervals) to attract attention. Our user, however, has devoted a good deal of thought to the construction of the line of code which he was inserting when the BREAK occurred. Fortunately, he is under no obligation to do anything about the broken Target process. He finishes his line, adds another (Figure 4-10), then (Figure 4-11) calls up the environment of the broken TARG1 process, and faces the bad news with a clear head.



```
------------------ RPROG/SUBST (PROG) ----------------------




   FOR LN ← S_LINE STEP 1 UNTIL GET_LENGTH(S_SCENE) DO
     IF (CHR1←FIND_STRING(FROM,GET_TEXT(S_SCENE,S_LINE,1))) THEN
     BEGIN
       INSERT_TEXT(S_SCENE, S_LINE, CHR1, TO);
                                                          [8]
▲
```

COMMAND	EXPANDED	COMMENT
..INSERT_T...	EDIT_CHAR(CRNT_REG,"...IN..",8);	[8]Reach a convenient place to stop editing SUBST before handling the breakpoint condition.

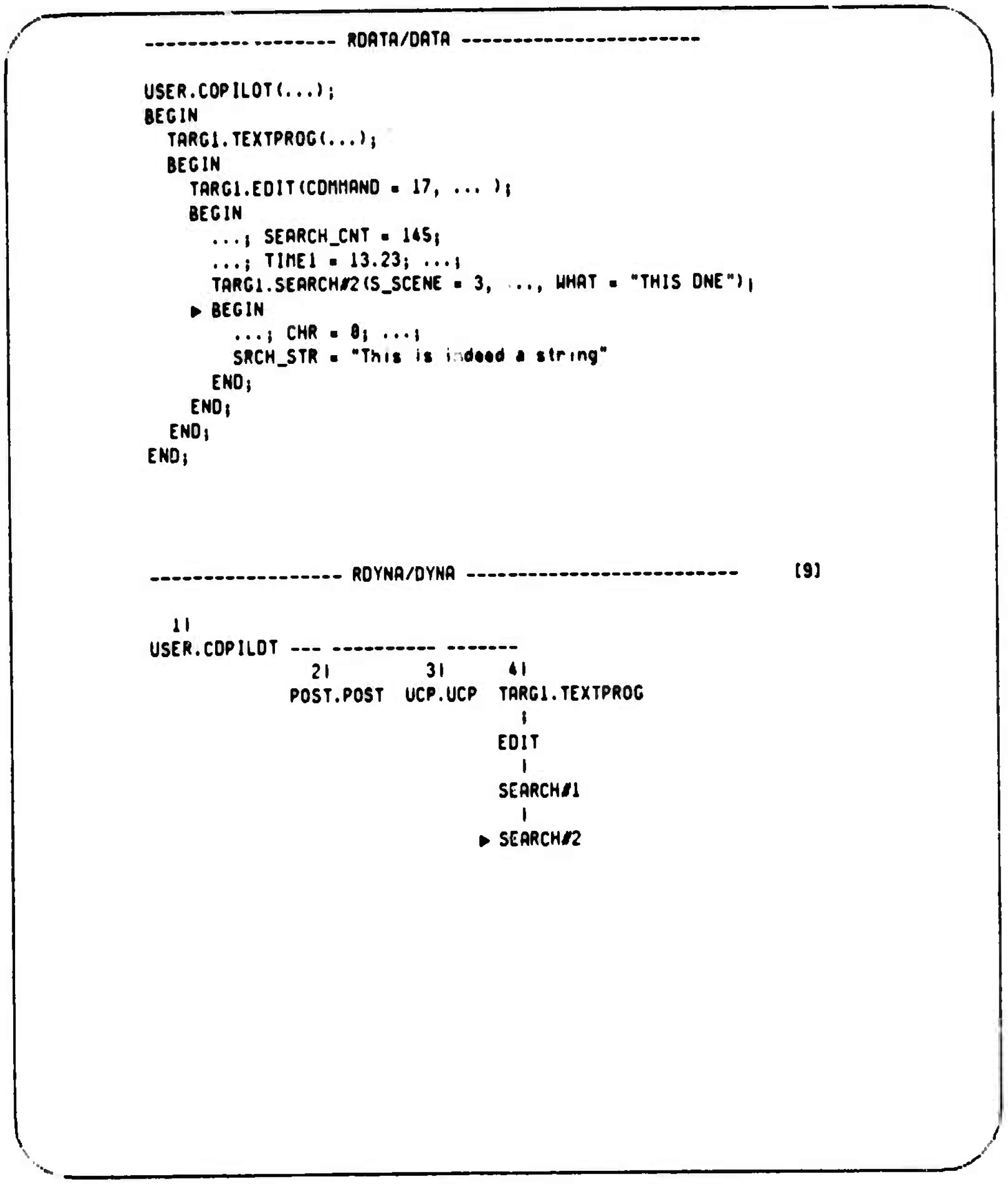
Figure 4-9. Non-Preëmptive Operation (part 1)



```
------------------ RPROG/EDIT (PROG) --------------------------[9]

  BEGIN
    INTEGER SCN, LN, CHR, CH1;  STRING SRCH_STR;

    # SEARCH_PRIMITIVES;
    SCN ← S_SCENE; TIME1 ← SYSTEM_TIME() - TIME0;
    SEARCH_STR;  TIME1;
    FOR LN ← S_LINE STEP 1 UNTIL GETLENGTH(SCN)
    DO BEGIN
      SEARCH_CNT ← SEARCH_CNT + 1;
      SRCH_STR ← GET_TEXT(SCN, LN, CHR+1, 999); CHR ← 0;
      IF (CH1←FIND_STRING(WHAT, SRCH_STR)) THEN BEGIN
        SEARCH_SCN ← SCN; SEARCH_LINE ← LN;
        SEARCH_CHAR ← CHR+CH1+1; RETURN(TRUE)
      END Comment recursive search;;
      IF (CH1←FIND_STRING("#", SRCH_STR)) THEN
       IF SEARCH(FIND_SCENE(SRCH_STR[CH1+1 TO 999]),1,0,WHAT)
       THEN RETURN(TRUE)
   END Comment on line;
   !BREAK(TARG1)! ▶RETURN(FALSE);
                  ▲
 END Comment Search;;

 CASE COMMAND OF BEGIN

   # OTHER_EDITS;
   BEGIN
     TIME0 ← SYSTEM_TIME();  SEARCH_CNT ← 0;
     RETURN(SEARCH(EDIT_SCENE, EDIT_LINE, EDIT_CHAR, S1))
   END Comment search command;

   # STILL_OTHER_EDITS;
 END Comment case;
END Comment Edit;;



------------------ RSTAT/STAT ------------------------


 !    USER     AWAITING User Input
      POST     AWAITING Postevent
      UCP      STEPPED
 **   TARG1    BROKEN
```

COMMAND	EXPANDED	COMMENT
«▶	TO_CONTEXT(-1);	[9] Finally, return RPROG and RDYNA Regions to the context of the process (TARG1) which broke, and prepare to fix it. See also the next figure.

Figure 4-10. Non-Preemptive Operation (part 2)



```
------------------- RDATA/DATA -----------------------

USER.COPILOT(...);
BEGIN
  TARG1.TEXTPROG(...);
  BEGIN
    TARG1.EDIT(COMMAND = 17, ... );
    BEGIN
      ...; SEARCH_CNT = 145;
      ...; TIME1 = 13.23; ...;
      TARG1.SEARCH#2(S_SCENE = 3, ..., WHAT = "THIS ONE");
    ▶ BEGIN
        ...; CHR = 0; ...;
        SRCH_STR = "This is indeed a string"
      END;
    END;
  END;
END;



------------------ RDYNA/DYNA -------------------------     [9]

  1|
USER.COPILOT --- ----------- -------
              2|         3|      4|
            POST.POST  UCP.UCP  TARG1.TEXTPROG
                                  |
                                 EDIT
                                  |
                                 SEARCH#1
                                  |
                               ▶ SEARCH#2
```

```
COMMAND                  EXPANDED           COMMENT
                                          Remainder of final state, after returning
                                          attention to the suspended process.
```

Figure 4-11. Non-Preemptive Operation (part 3)

4.D. ATTRIBUTE ANALYSIS OF COPILOT

We will apply the same behavior match analysis to COPILOT which we applied to other IPSs. We will indicate, for each attribute, those qualities of COPILOT which satisfy the requirements imposed by that attribute.

The User loop of COPILOT, in common with other systems, fits the reduced mode FSA model in its basic operation. In common with DPDA systems, the statements executed by this loop have different interpretations when applied to different program contexts. COPILOT can not be considered a DPDA system, however. We have replaced the nested user concept, which DPDA systems implement by creating instances of a User procedure in some operating environment, by a sort of "omniscient user" organization. The user is given the illusion that he is "above the plane of his program, looking down" (or some illusion to that effect). He can, by pointing, cause any active environment to be influenced by his actions. User "instances" no longer need follow any particular control discipline. (In reality, there is but one User instance, whose activities invoke appropriate activities in other processes.)

Let us now perform the detailed attribute analysis:

1) Multiple Activities. COPILOT allows the user complete control over the processes he creates. The system itself makes copious use of the multiple processing and event handling facilities of the language. We have described some of these system processes. Others operate behind the scenes; they will be described in Chapter 8.

2) Non-preëmption. Ironically, we have achieved non-preëmptive behavior by having one process, the User process, totally preëmpt the terminal. This process is, fortunately, designed as the mechanism for non-preëmptive control of the other processes. The terminal is always available for user commands.

3) Response time. The user may issue any meaningful command, and have it begun, immediately after the system has accepted the previous command (limited only by the time delay of the User loop, which is determined by system load, but should remain short). This is the combined result of the process structure, the User process design, and mode minimization.

4) Minimal modes. There are no global modes in COPILOT; no special command must be issued to begin editing a function, or to begin inspecting program variables. There is a different command, or statement, in the single input language, for each interactive operation in the system. This might require more different commands than systems which provide modes, but the increase is not too great. The number of commands is held in check by the use of the same text-oriented and structure-oriented editing operations on each kind of IPS data. Thus, editing the program (e.g., RPROG) Region corresponds to a "program edit mode", while editing a data (RDATA) or dynamic activation tree (RDYNA) region corresponds to some "debug mode" operations. Chapter 7 presents, in just 39 commands and special statements, a reasonably complete set of IPS facilities, whose power may be enhanced by direct execution of normal language statements.

5) Single language. Every action in COPILOT is expressible as a statement in the MISLE language. A statement, if correct and meaningful, will always mean the same thing, except for the environment-dependent bindings of names.

6) Accessibility. By referring to supplementary data structures, COPILOT facilities can transcend normal scope limits, gaining access in a controlled manner to names and values of any data in the "job".

7) Context. All program contexts: programs, data, and execution state, can be visually displayed, in a manner revealing their structure.

8) Non-symbolic operations. The common operations for editing and process control are very short, manipulative in nature. We could extend our expansion algorithms to accept non-symbolic input from devices such as a "mouse" or "graphics tablet", again creating MISLE statements for execution.

The chapters which follow present the COPILOT design in more detail– first the user level descriptions, then some implementation considerations. In the final chapter, we will discuss some of its shortcomings, and some possible extensions.

CHAPTER 5
THE COPILOT SYSTEM: A USER-LEVEL DESCRIPTION

In this chapter we wish to expand the introduction of Section 4.C, presenting the COPILOT experimental design in some detail. Our goal is not to write a user's manual, but to cover all the major aspects of the system, to give the reader a general understanding of its capabilities, and a feeling for its philosophy.

5.A. BASIC SYSTEM STRUCTURE TERMINOLOGY

Our discussion of COPILOT begins with the structures we have developed for the display of information. These structures, while they need not strongly affect such things as the programming language design– the control and data structures it supports– do determine how the user views his programs, and what role he can play in their operations.

We will show that the Scene types defined in COPILOT constitute an adequate external model for the Information Structure of most block-structured languages and that, when linked to the operant structures underlying them, these Scenes provide all necessary context for viewing and controlling program operation.

We begin the discussion with a definition of the COPILOT display terminology.

Table 5-1. Display Terminology

SCREEN	A physical display device, also known as a "display".
REGION	A contiguous, named group of lines on a Screen, assigned by user or program to a specific Screen location.
SCENE	A logically related, ordered set of text lines – a "page" from a user program, for instance. Each Scene also possesses a **Scene Type** to clarify its use. A Scene may be part of a program, of a data specification, or any other textually representable entity.
WINDOW	The contiguous set of lines from a Scene, visible in the Region to which the Scene is assigned (mapped).

5.A1 Screens

In Section 4.A2 we stated that we would support multiple screens. A **Screen**, or **Display**, is a device capable of presenting continuously several lines of text. The hardware and software supporting each display must allow programs to control completely the data displayed on the screen. Updating must be fast enough that no appreciable delay is encountered while changing part or all of the data on the Screen. In addition, it must be possible to show several distinct indicators, or cursors, without disturbing the data. In COPILOT, the Screens assigned to a user are assigned permanent numbers during installation– naming facilities at the screen level are not very important.

5.A2 Regions

Given enough Screens, a COPILOT user could devote one to each independent data Scene which he or the system has created. However, it is not usually possible to satisfy the voracious appetites of COPILOT processes for display area. Thus there is a need for facilities which will allocate sections of the available Screens to these disparate uses.

A **Region** is a named area on some Screen. Region names and ranges may be assigned by programs or by the user; the initial system configuration features a few Regions whose Scenes display the initial system context. The subsequent creation and mapping of Region to Screen is an infrequent operation. Typically, the user does it but once, at the beginning of a session, to establish an augmented configuration to suit his needs and resources.

A Region is usually named, created, and used for a specific kind of Scene; if one wishes to use the same Screen area for multiple purposes, he assigns multiple Regions to that area. In the current system, no two Regions whose areas overlap may have Scenes mapped into them (be visible) simultaneously. Such a facility would require a priority scheme to resolve conflicts.

We will treat Regions and their relationship to Scenes in Section 5.H, after a detailed consideration of Scenes and what we put into them.

5.A3 Scenes

We have used the term "Scene" loosely in the preceding paragraphs to describe the collections of lines displayed in a Region. In our formal definition, such a collection of lines is a "Window" of some Scene. If the Scene has fewer lines than its Region, enough empty lines will be inserted to fill the Window. The archetypical example is the Scene used for storing and displaying program text. Program Scenes resemble the user-defined "pages" which often segment program text files into logical groups. A program Scene might be just one page from the file, although we intend to suggest an organization of programs into Scenes which is more intuitively structured for interactive operation. We have avoided use of the term "page" to avoid confusion with the memory "pages" of some modern computing systems.

5.A4 Scene Types

Every Scene has the same format: a set of text lines. As we have suggested, though, Scenes are put to various uses. Some Scenes correspond to structures (such as compiled code) at other levels, or contain data which system processes need to read. The user may also define Scenes which require special treatment. We associate with each Scene a Scene type, a code identifying its uses.

Additional attributes for a Scene include: its name, a string optionally assigned to it when it is created, its length (the number of lines), and the current editing position within this Scene. The edit cursor ("▲" character) visibly indicates this point whenever the Scene is selected for terminal-controlled editing operations.

Other attributes could be used to place restrictions on the use of Scenes. These attributes would be similar to the "Capabilities" of [32] and would specify for each process whether, for instance, that process was permitted to read, modify, or (for Program Scenes) execute the Scene, who its owner was (for shared Scenes), etc.

5.B CONTEXT SCENES AS EXTERNAL INFORMATION STRUCTURES

Before we consider the Scene types which we have provided in COPILOT, we should say just what it is we want these Scenes to accomplish: to supply the user with that contextual information needed both to observe the instantaneous state of a computation in a coherent manner, and to predict and influence its future actions. We will refer collectively to these Scene types as **Context Scenes**.

5.B1 Information Structure Models

In [59], Wegner formalized the need for a way to describe program execution context with his Information Structure Models. He categorized programming languages by the data structures required to specify their **Information Structures** within a processor. These structures include algorithms, data, and their control mechanisms. A set of Information Structures, I, time-ordered "snapshots" of program and data configurations during a computation; an initial configuration I_0 from I; and a set of transformations (interpretation rules), F, taking configurations I to their successors– constitutes an **Information Structure Model** of the computation, in a given programming language and system. In the Context Scenes, we will be concerned with the external representation of elements from I. For most programming languages, Wegner shows that one can further factor the Information Structures of I into the following components:

1) The Program Component: a representation of the algorithm.
2) The Data Component: objects allocated and manipulated by the algorithm.
3) The Control Component: indicators of currently active program steps and data environments within each active process.

5.B2 The Contour Model

Johnston, [27], has developed an Information Structure Model, the Contour Model, for block-structured languages. This model has been shown adequate for representing the information structures of Algol60, Algol68 [47], and Oregano, [4], which was designed around it. The Contour Model appears to extend to the complex naming structures of PL/I and Simula, as well, although it does not support the dynamically inherited naming scopes of Lisp, LCC, and their ilk.

Figure 5-1 is an example of a "snapshot" from an Algol60 program, expressed in the Contour Model. The Program Component is called the **algorithm**, the Data Component the **record of activation**. In the latter the nested Contours define the lexically nested **access environment**, while the dynamic (control, e.g. caller and callee) nesting is shown by connecting arrows. The Control Component consists of one or more processors, each defining the locus of control of an independent process, each represented in the model by the IP (instruction point), and EP (environment point) arrows emanating from the "π" graphic which depicts the processor.

```
1 bl: BEGIN
      REAL  a, b, x;
2     PROCEDURE P(x,y);
         REAL x,y;
3     bp: BEGIN
         REAL c;
         ...
4        P(...,...)
5-6   END;
7     b2: BEGIN
         REAL b,c;
         ...
8        P(a,b)
         ...
9     END
10 END;
```

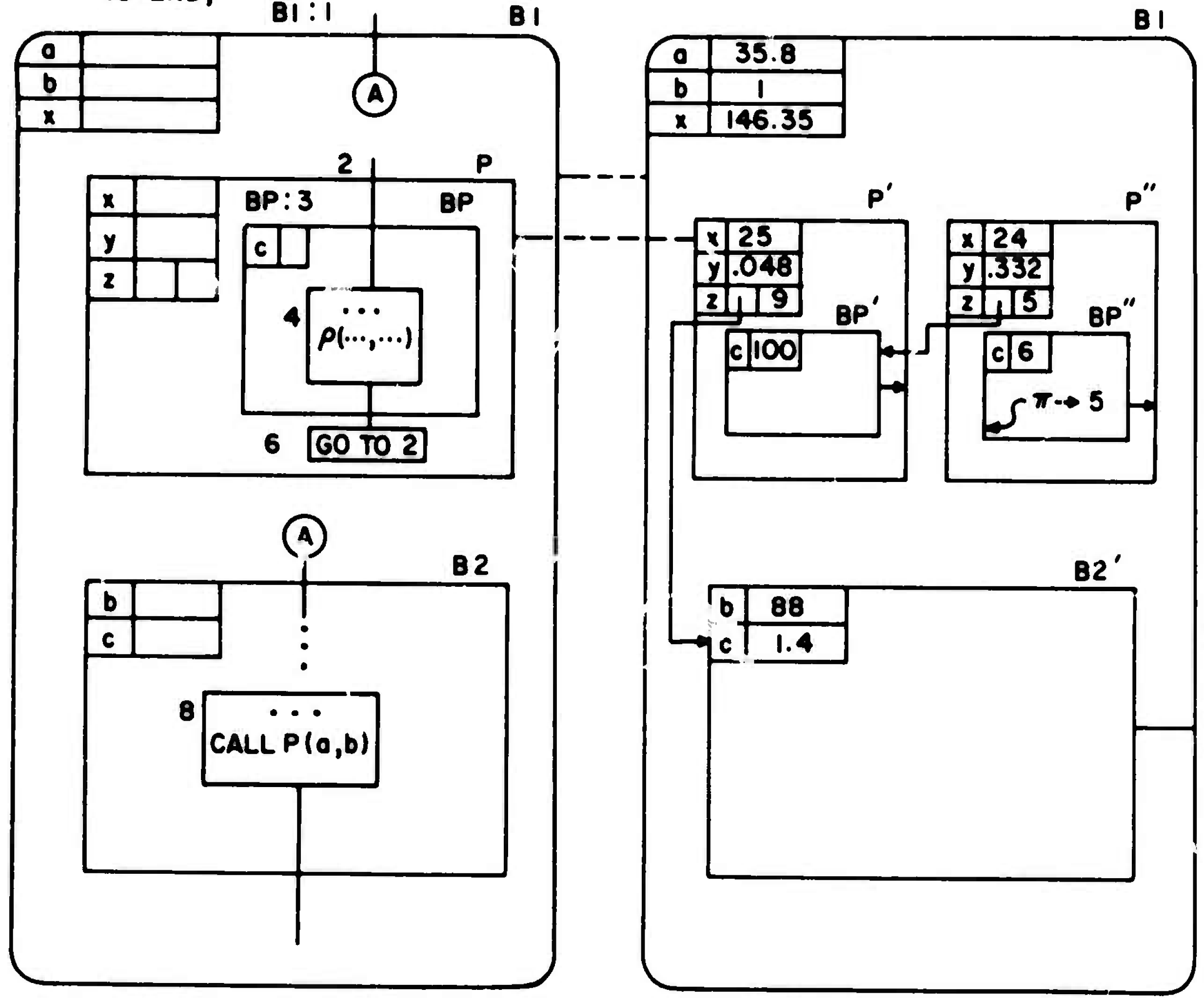


Figure 5-1. The Contour Model Representation for an Algorithm

5.B3 The COPILOT Context Scenes

By viewing a snapshot, I_i, in a Contour Model representation, and knowing how the interpreter, F, operates, one can predict the contents of snapshot I_{i+1}, to whatever level of detail one chooses. This is precisely the kind of condition we want to create with our Context Scenes. Although we have not used the Contour Model notation directly, we will show the (potential) functional equivalence between the Contour Model and our Context Scenes. This will demonstrate the adequacy of the Context Scenes as an external Information Structure representation, for MISLE and a variety of other languages. To handle Lisp-like structures would require additional development.

5.B4 The Snapshot Requirement

We are limited by current hardware in the amount of concurrence we can achieve. Because much of what we display (the name and value of a variable, for instance) must be converted from the internal forms required for efficient operation, and because of the expense of this conversion, it is impossible to record each change visibly as soon as it occurs. Text Scenes are made to agree with changes in the ultimate underlying structures, not instantaneously, but at frequent and adequate intervals, in a manner revealed in Section 6.C.

In order to preserve the "snapshot" quality of the Contour Model in our system, we will impose the following requirement: all visible context Scenes must be updated simultaneously, each time the display is changed. Therefore, at any instant, all visible system information is a correct representation of some subset of the system state at a single previous instant. Thus, the user sees is a single coherent "snapshot" of his system, not an album of individual pictures whose time relationship is unclear.

5.B5 COPILOT Context Scene Types

We can now present descriptions of the Context Scene types. In each we will follow approximately the same format:

a) Which component(s) of the Information Structure it exhibits.
b) Details of the information content of this type (syntax, semantics).
c) How the information is organized into Scenes.

There are only four different Scene types predefined in COPILOT: program, data, dynamic structure, and status Scenes. We will deal with each in turn.

5.C. PROGRAM SCENES – THE PROGRAM COMPONENT

We have designed a programming language, MISLE, in which the user both describes his algorithms and controls their operation, by manipulating their representations as program Scenes. Although these operators require substantial underlying structure, none is visible to the user: he sees only the text of his programs, stored in Scenes. We have chosen this standard textual representation over other alternatives (e.g., Johnston's representation of programs as flowcharts nested in Contour Templates) for a variety of reasons, among which are:

1) The notation is more compact.
2) The control structure is more obvious (with a slight loss in the clarity of the data structure).
3) Editing operations are easier.
4) The text format is more easily stored, transmitted and printed.

5.C1 The MISLE Language

MISLE is an easily-implemented subset of the language SAIL [52]. SAIL is derived from Algol60 [46], with some syntactic modifications to suit the designers. Extensions were originally made to this base to include a variable length character string facility, and to include a variant of the associative processing language LEAP [18]. More recently, in response to an increased need for sophisticated control and data structures in Artificial Intelligence research, a major revision was developed [19]. The addition most relevant to our needs is a comprehensive set of facilities providing multiple processes in the style of Algol 68 [61].

The current COPILOT implementation is written predominantly using SAIL; our preferred language would be a SAIL superset. However, we have yielded in this dissertation to the need for a language which is simple to implement, and to understand. Therefore, MISLE is a limited SAIL subset, adding to the basic Algol-like constructs just enough to support the IPS primitives which the user will need: process control primitives, text strings, etc.

5.C2 The Basic Features of MISLE

What follows is the syntax (and a brief semantic discussion) of the more or less standard SAIL-like aspects of MISLE. The reader who is already familiar with this sort of language would do well to skim this section and proceed with Section 5.C5, defining special additions to the language for interactive uses. Refer to Appendix A for a description of our syntax notation.

```
1  <program> ::= <block>
2  <block> ::= <head> <tail>
3  <head> ::= BEGIN <decl_list>
4  <tail> ::= <statement> { ; <statement> }* END

5  <decl_list> ::= <decl> {decl}*
6  <decl> ::= <type> <idlist> ;  | <pdec> |
                  <algol-like array declarations>
7  <idlist> ::= <id> { , <id> }*
8  <type> ::= <atype> | LABEL
9  <atype> ::= INTEGER | STRING
10  <pdec> ::= <untyped_pdec>  | <atype> <untyped_pdec>
11  <untyped_pdec> ::=
                 PROCEDURE <id> ( { <param_list> } ) ; <statement>
12  <param_list> ::= <param> { ; <param> }*
13  <param> ::= <atype> <id>

14  <statement> ::= <block>  |<compound_statement> |
                      <conditional>  | <assignment>  | <jump> |
                      <for>  | <while>  | <case>  | <pcall> |
                      <id> : <statement> | <process control statement>
15  <compound_statement> ::= BEGIN <tail>
16  <conditional> ::= IF <Boolean_expr>
                         THEN <statement> { ELSE <statement> }
17  <assignment> ::= <id> ← <expression>
18  <jump> ::= GO <id>
19  <for> ::= FOR <forhead> <statement>
20  <forhead> ::= <id> ← <arith_expression> STEP
                    <arith_expression> UNTIL <arith_expression>
```

```
21  <while> ::= WHILE <Boolean_expr> DO <statement>
22  <case> ::= CASE
                 <arith_expression> OF <compound_statement>
23  <pcall> ::= <id> ( { <expr_list> } )

24  <process control statement> ::= ACTIVATE ( <process_id> ) |
                      TERMINATE ( <process_id> ) |
                      SUSPEND ( <process_id> ) |
                      SET_PRIORITY ( <process_id> , <expression> )
25  <process_id> ::= <arith_expression>

26  <expr_list> ::= <expression> { , <expression> }*
27  <Boolean_expr> ::= <disjunct> { ∨ <disjunct> }*
28  <disjunct> ::= <relation> { ∧ <relation> }*
29  <relation> ::= <arith_expression>
                        { <relop> <arith_expression> }*
30  <expression> ::= <arith_expression>
                          { & <arith_expression> }*
31  <arith_expression> ::= { <pm> } <term> { <pm> <term> }*
32  <pm> ::= + | -
33  <term> ::= <primary> { <td> <primary> }*
34  <td> ::= * | / | MOD
35  <primary> ::= <id> { [ <arith_expression>
                              TO <arith_expression> ] } |
                      <pcall> | <constant> | ( <expression> ) |
                      <process_control_primary>
                       <algol-like array element specifications>

36  <process_control_primary> ::=
                       SPROUT (<pcall>,<father>,<stacksize>,<priority>) |
                       EV_TYPE () | CAUSE ( <evtype> , <value> ) |
                       EV_WAIT ( <evtype> ) | EV_GET (<evtype>) |
                       AR_EV_WAIT ( <evtypearray> ) | AR_EV_GET ...

37  <evtype> ::= <arith_expression>
38  <father> ::= <process_id>
```

39 <stacksize> ::= <arith_expression>

40 <priority> ::= <arith_expression>

41 <constant> ::= <string_constant> | <integer_constant>

42 <comment > ::= COMMENT <algol-like comment, ending in ";" >

5.C3 Semantics of Extensions

MISLE is for the most part a slightly modified subset of Algol60 with the SAIL String data type added. Its only data types are scalars and arrays of integer and string values, denoted by identifiers, constants, and expressions. Only explicit conversions (string to integer, integer to string) are provided. The operators +, -, *, /, and MOD are available for arithmetic operations; normal relationals are available for Booleans. Strings may be concatenated using the operator &. S[n FOR m] yields the m-character substring of S, beginning with the nth character. Parameters are passed to procedures by value only. Control facilities include (in addition to procedures), GO TO, IF, FOR, WHILE, and CASE (alternative selection) statements. A syntactic modification places both the naming and type descriptions of procedure parameters within the (parenthesized) parameter list, as in Algol W [3].

5.C4 Processes

The process-manipulation primitives of the unenhanced language allow creation, deletion, suspension and activation of processes (see [4] as a reference to the kind of "cactus stack" process structure we employ). We mean by "unenhanced" that these do not rely on the facilities of the IPS for their operation.

Processes are assigned execution priorities when they are created. Whenever a running process suspends, or specifically requests it, the system scheduler selects a new process to run, choosing the highest-priority process which is READY to run (see Section 5.F).

Events are interrupt and process-communication mechanisms. A process may cause an event of a chosen event type, and may specify a value to be associated with the event. When the scheduler next runs (the running process suspends), it will ready any processes which are waiting for an event of this type, returning the associated value as the result of the function which does the waiting.

For each occurrence of an external interrupt (I/O, timer, etc.) basic system routines simulate a very high priority process which causes an appropriate event and forces rescheduling as soon as possible. Processes handle interrupts by waiting for, or testing (polling) for, events of the corresponding type. This approach to interrupts, as opposed to more standard interrupt mechanisms like those in [47] ("unexpected" procedure calls), is supported by Wirth [64]. The result is a consistent, process-oriented method for handling all asynchronous activity.

Table 5-2 provides the meanings of the basic process-control primitives. In Section 7.C7 we will describe additional process control functions, intended for interactive use.

Table 5-2. COPILOT Process Control Primitives

Sprout(...)	Creates a new, suspended process, with given stack size and priority. An instance of the specified procedure is readied within the new process. Sprout returns a unique integer process identifier, or pid.
Activate(pid)	sets the state of the process pid to READY. It will be set RUNNING as soon as possible, based on its priority and the availability of resources.
Suspend(pid)	Sets the state of the process to SUSPENDED. It will not run again until some other process Activates it.
Terminate(pid)	Destroys the process pid, and any subprocesses.
Set_priority(...)	Changes the execution priority of a process.
Cause(...)	creates an event of given type and value, READIES any processes awaiting events of that type, and forces rescheduling.
Ev_wait(...)	yields the value of an event of given type. It causes the process calling it to wait (SUSPEND), if necessary, until such an event is available. The event is then forgotten by the system.
Ev_get(...)	never waits. It yields 0 if no such event has been caused (and still exists). Otherwise, it is the same as Ev_wait.
Ev_type()	creates a new event type.

Ar_ev_wait(...)	waits for one of a set of event types, specified in an array. The result is the type of event which was actually caused. Ar_ev_wait does not delete the event; hence, an Ev_get may subsequently be used to fetch the actual event.
Ar_ev_get(...)	never waits. It yields 0 if no such event exists. Otherwise, it is the same as Ar_ev_wait.

5.C5 Special Features

We have added the following additional constructs to the language in order to make some of the interactive facilities more convenient. The additions include variable-display (debugging) statements, breakpointing statements, and Scene linking constructs. The syntax follows:

```
1  <statement>  ::= <Scene link> ; <statement>
2  <declaration> ::= <Scene link> ; <declaration>
3  <Scene link> ::=  * <Scene id>
4  <Scene id> ::= <id>

5  <statement> ::= <show>
6  <show>  ::= <expression>

7  <statement> ::= <temporary statement> <statement> |
                   <statement> <temporary statement> |
                   <affect> <class>
8  <temporary statement> ::= '{  {<class> :} {<switch> !}
                    <statement> {; <statement>}* '}
9  <affect>    ::= TURN ON | TURN OFF | DELETE
10  <class>  ::= <id>
11  <switch> ::= ON | OFF

12  <statement> ::= BREAK ( <process_id> ) |
                   ARR_BREAK ( <process_id_array> )
```

Each of these additions depends heavily on aspects of the IPS which remain to be described. We will delay explanation of their semantics until the descriptions are complete.

For an example of a MISLE program, refer to the PROG Scene of Figure 4-1, or one of those which follow it.

5.C6 Program Scene Organization

Traditional program source text organization is straightforward: a deck of cards, a magnetic tape, or a disc file containing the lines of the program. In the latter case, perhaps the file is linearly segmented into logical pages, mostly for display purposes.

One notable exception is the file system for NLS [15], developed over the last decade at SRI's Augmentation Research Center. Very briefly, the purpose of this display-based system is to provide a complete interactive environment for the user, to dispense entirely with paper and pencil, yielding a corresponding increase (augmentation) in intellectual power. The NLS work has proved a major influence in this research. We hope to retain something of this power in COPILOT, while extending its domain to direct interaction with user algorithms.

Files (not only program files) are not organized in simple linear fashion in NLS. Instead, they are hierarchical, resembling outlines; the NLS user can choose to view only the level of detail which suits him: just the major topics, the major and first subtopics, or the entire structure. He can also place hidden or visible links at arbitrary points in his files, providing a path to related material in the same or other files. NLS makes it easy to follow these links, to save previous views, and generally to navigate fruitfully about a web of cross-references.

We cannot hope to do the NLS system justice in so short an introduction, nor have we space to describe other text-manipulation systems which support structured file organization. We can suggest in addition the references [60], [24], and [42].

MISLE programs, being block-structured, are inherently hierarchical We envision an implementation of COPILOT which would allow the user NLS-like control of the degree of detail (depth of nesting) of the displayed program. For instance, one could view only the top level statements of a block, with substatements merely indicated. Hansen used something like this in his thesis [24]. The BBN-Lisp editor, [53], because of the need to be concise, uses a similar structure-compression technique in its teletype-oriented system. Our system contains many hierarchical structures, and techniques like these would enhance any of them.

At present, however, our use of hierarchical design is explicit. Instead of fragmenting a program into consecutive linear Scenes, the user can include Scene link constructs to achieve a hierarchical segmentation. Figure 5-2 gives a simple example. The system views the program as if it were a procedure, expressed in one Scene, containing the data of Figure 5-2.c; it treats a Scene link as a sort of "macro" call. The user views it as a procedure containing a suppressed subprocedure (Figures 5-2.a and 5-2.b). The system provides complete facilities for "following the links", both forward and backward, when the user wishes more or less detail. When a Scene link occurs as the last line of a Scene, simulating linear connections, special treatment avoids unnecessary nesting.

Our personal experience (supported by Mills in [42]) is that it is useful to segment a program so that each Scene is fairly small, each representing a logical section of the program and of the control structure of the algorithm. The system will nevertheless support Scenes of arbitrary size.

```
Scene *St1;
PROCEDURE T1(INTEGER DUM);
BEGIN
   *SRNP ;
   INTEGER J,K; STRING S;
   FOR J←1 STEP 1 UNTIL 100 DO BEGIN
      K←J+3; K;
      WHILE K<J+10 DO OUTPUT(RNP(K))
   END
END
```

a) Containing Scene

```
Scene *Srnp;
STRING PROCEDURE RNP(INTEGER I);
  IF I=0 THEN RETURN("") ELSE
    RETURN(RNP(I/10)&PUTCH(I MOD 10+48));
```

b) Contained Scene

```
PROCEDURE T1(INTEGER DUM);
BEGIN
   STRING PROCEDURE RNP(INTEGER I);
     IF I=0 THEN RETURN("") ELSE
       RETURN(RNP(I/10)&PUTCH(I MOD 10+48));
   INTEGER J,K;
   FOR J←1 STEP 1 UNTIL 100 DO BEGIN
      K←J+3; K;
      WHILE K<J+10 DO OUTPUT(RNP(K))
   END
END
```

c) Apparent Program

Figure 5-2. PROG Scene Linkage

5.C7 The Instruction Point Portion of the Control Component

As we have indicated, we have distributed our representation of the Control Component among the Context Scenes. In Program Scenes we indicate the IP (for a selected process) by a special context cursor, represented by the "▶" character. This context cursor precedes the text for a statement in the selected process. Which of the active IPs is selected for display depends on an indicator in the DYNA Scene (see Section 5.E1). Any terminal commands which require *implicit* program location data obtain it from this selected IP.

The context cursor is the visible representation of the active statement within the selected process. No function used to retrieve program Scene data will ever yield a string containing the context cursor. See Chapter 7 for functions which yield its location.

5.D. DATA SCENES – THE STATIC DATA COMPONENT

Because algorithmic languages like MISLE were designed before we designed COPILOT, we had little trouble deciding a representation for the Program Component in the program Scenes. This is not true of the Data Component, where few attempts have been made to create formal external representations for the data environments (for any language).

Again, a logical candidate might be the Contour Model representation; again we have decided against using it directly. In addition to the reasons we gave in Section 5.C, we feel that use of Contours to display the Record of Execution would create Scenes of confusing complexity. We have instead developed a more linguistic method which we can prove equivalent in facilities to the Contour Model, thus adequate for Data Component representation.

Our solution requires two new constructs:

1) A data specification notation, or **Data Language** (†), intimately related to the MISLE language, for defining data values in their static (lexical) contexts (the static **Data Component**.)

2) A tree notation for exhibiting the dynamic (control) relationships of the Record of Execution.

- -

(†) Some object to this term because the "language" is not algorithmic (no verbs). It is a language formally, however. Read "specification" for "language" throughout, if you wish.

We begin with the Data Language.

5.D1 Data Language Syntax

```
1 <data layout>   ::= <data block>
2 <data block>    ::= BEGIN <data tail>
3 <data tail>     ::= <data spec> { ; <data spec> }* END
4 <data spec>     ::= <equation> | <data block> | <pcall spec>
5 <pcall spec>    ::= <pcall descr> ; <data block>
6 <equation>      ::= <id> = <constant> | ...
7 <pcall descr>   ::= <instance> ( { <equation list> } )
8 <instance>      ::= { <process name> . }
                          <procedure id> {* <nesting level> }
9 <equation list> ::= <equation> { , <equation> }*
```

5.D2 Semantics, Pragmatics

The Data Language is a parasitic language. The syntax hints at this in its resemblance to MISLE: the procedure and block structure productions are nearly identical; the equations of the Data Language correspond closely to MISLE declarations. We require that the dependence be even more pronounced, however. A <data layout> is meaningless without reference to a section of the MISLE program to which it is linked (we consider this linkage in more detail below). One <pcall spec> or <data block> may exist at any instant for each instance of a procedure or block activation.

There are two kinds of information in a <data layout>. The first, provided by **equations**, comprises the names and values of selected variables (and expressions) at some instant. The constant in an equation must agree in data type with the type of the linked variable whose name appears in the equation, and whose value it represents. We will say that an identifier is **marked** if it has been selected by the operations of Section 7.D1 for display in data Scenes.

The second is structural, provided by the block and procedure structure (whose interpretation is transparent), and by ellipses (...). The ellipsis is an optional device which informs the viewer that there are variables in the Contour whose values do not appear in the Scene.

The position of the ellipsis (or ellipses) in a <data block> or <equation list> corresponds to the position of the omitted names in the declaration list of the linked algorithm. Figure 4-2 contains a Data Layout for one of the states encountered by the program in the same figure during its execution.

5.D3 Data Scene Organization

The second COPILOT Context Scene type is the data Scene. Each data Scene contains one data layout which is linked (‡) to a procedure in some program Scene. In COPILOT, a single data Scene, DS, can contain text level representations for the data from at most one instance of some procedure, p, and from those forming its lexical ancestors. This means any older recursive instances of this same procedure, any instances of other procedures in the dynamic ancestry of p (and in other process branches) whose variables are not accessible to p, can have no representation in DS. It is possible, however, to form other data Scenes at the same time which do represent these hidden environments.

The user, or more commonly the system, can create a legal data Scene as follows:

1) Choose a procedure, $P=P_0$, from some Scene, and some instance of that procedure, $p=p_0$. Begin with an empty data Scene.

2) Record in a <pcall spec> the values of marked local variables and actual parameter values (with their formal names) from p, following the pattern established by **P**.

3) Obtain the immediate lexical parent, P', of P, and the corresponding instance, p', from the static environment of p. Quit if there is none.

4 Embed the lines of the <pcall spec> created in step 2 in a <pcall spec> formed by repeating steps 2 through 4, substituting P' for P, p' for p. An embedded <pcall spec> is inserted just after the other declarations in the <data block> which corresponds to its point of declaration.

The linkage of p_0 to P_0 defines completely the linkage of the data Scene to the program Scene.

We should emphasize that we have made many arbitrary decisions in this design. We

- -

(‡) This is the antecedent link of Johnston's model; its explicit existence is usually omitted in his examples, but would have to be present in any implementation.

considered several other algorithms for generating data Scenes. Some of these allowed multiple instances of the same procedure, thereby including much more dynamic context directly. Perhaps one of these methods (or one which did not occur to us) would be a superior one. Surely the designer of a COPILOT-like IPS for a different type of language should reconsider the issue. Our final choice is based mainly on a desire for clarity. The dynamic Scenes of the next section help cure many of the inadequacies of the data Scenes.

Section 5.H1 will depict data Scenes in action. There we shall show how these Scenes are created and used, emphasizing the most common situations.

5.D4 The Data Language as an Input Facility

Using Contour Model terminology, the Program Component of a Snapshot, I_i, of a computation (*)is externally represented in COPILOT by program Scenes. In some sense, these Scenes also form a complete external representation of the initial state, I_0, since the initial Record of Execution is empty; they cannot specify any subsequent Snapshot, I_j, $j \neq 0$. Thus, although the language can specify a computation via an algorithm, it cannot directly express intermediate states of that computation. R. Floyd has pointed out that it would be useful to have linguistic facilities for constructing these intermediate states (†). This would make it possible to:

1) Directly create a test environment for testing a routine in an incomplete program which does not yet include code for supplying that environment.

2) Directly modify an environment, perhaps to agree with a modified algorithm, perhaps preparatory to altering the instruction point (IP) of a process operating in that environment (in complicated cases this might be preferable to what the system could do automatically).

3) Save and restore intermediate computation states in human-readable form. (For small programs, this "core dump" technique would allow one to save computations over console sessions. In Section 8.B1 we will examine more efficient methods.)

4) View Snapshots of a computation in a reasonable form.

- -

(*) The collection of snapshots defining the total operation of one program "run".

(†) Personal communication, October 1972.

We have not seen this kind of facility in an IPS. Comment (4) above should reveal our approach to providing it. We already possess a linguistic facility, the Data Language, for displaying intermediate computation states. By selecting a data Scene for editing, then using standard text-editing operations to modify it, the user can even indicate changes he would like to make. To turn this into a full data-specification facility, it is only necessary to convince the system to convert these changes into corresponding changes in the actual underlying data structures. We have done this in the COPILOT design. Similar user changes will be shown useful in dynamic Scenes, as well (see Section 9.C1). This achieves a very pleasing symmetry within the Context Scenes: all constructs are useful for two-way communications between the user and the system.

The editing operations required to accomplish text changes are presented in Chapter 7, including special convenience commands particular to data Scenes.

5.D5 The Environment Point Portion of the Control Component

We use here a development paralleling that for program Scenes. There is an Environment Point (EP) in the Control Component for each active Process, defining its access environment. Information in the dynamic Scenes will indicate all the active Environment Points.

Again, the user (or one of his programs) may select a "distinguished" EP, which will be displayed as a context cursor ("▸") if the environment it defines appears in a visible data Scene. All terminal commands which require *implicit* environmental specification will obtain it from this cursor.

5.E. DYNAMIC SCENES – THE DYNAMIC DATA COMPONENT

Data Scenes can show any or every element of the Data Component, and the static (lexical) relationships between activations of <blocks> and <procedures>. They do not exhibit the dynamic connections (e.g., for procedure instance p, which procedure instance called it, or which created (sprouted) it; to which instance it will return). The purpose of the dynamic Scene is to provide this information.

We are tempted to suggest another "language" here, with its own related syntax; we have decided instead to develop a more graphical representation for the dynamic "cactus-stack" structure of MISLE programs. This dynamic structure tree does share constructs in common with Data Language elements, however, and this linkage is important to our powerful context-roaming operations (Section 5.H1).

There is but one dynamic Scene in a COPILOT environment, containing the single dynamic structure tree. Figure 4-2 is an example of the dynamic Scene. Its structure is quite simple: Each node (terminal or non-terminal) of the tree is an <instance>, as defined in the Data Language grammar in Section 5.D1. The root node ("USER.COPILOT*1") provides the base environment of the entire computation, or "job", including IPS facilities. Instances of active procedures in a process appear (in order of call) below each other in the same column. The root nodes of subordinate processes are placed in adjacent columns as shown, then connected by horizontal line segments to the processes which own them (‡). The terminal nodes of the dynamic tree define the set of active Environment Points.

5.E1 The Context Point

At any one time, there can be but one EP visible (as a context cursor) in a data Scene, and but one IP context cursor in a program Scene. In fact, given a computation in progress, and a particular EP, the corresponding IP is completely determined. Thus to select an (IP, EP) pair for display as context cursors, one need specify only the EP.

We accomplish this manual selection of execution environment using an additional indicator, which we will call the Context Point (CP). The CP is represented by a context cursor which selects an instance in the dynamic Scene. We have functions for moving the Context Point within the dynamic tree, and for generating data and program Scenes, with their context cursors, to exhibit the environments which the CP selects. We will describe these functions in Chapter 7.

5.E2 Adequacy of Scenes as External Information Structures

In Section 5.B3 we announced our intention to show a functional equivalence between the

- -

(‡) For simplicity, MISLE follows the retention rules implicit in Algol60, and explicit in Algol68: A process must be exterminated if its owner ceases to exist.

Context Scenes and Johnston's Contour Model. This is important, because it expands the power of our formulation to all the language types amenable to the Contour analysis.

Now that each of our Scene types has been developed, the demonstration of this equivalence is quite simple: one need only select all variables for display, then create enough data Scenes to contain each instance of each active procedure and block at least once. Then for each relationship or value revealed in a Contour Snapshot one can identify constructs from one or more Context Scenes which reveal the same relationship or value (a formal proof would simply enumerate these correspondences).

5.F. STAT SCENE – PROCESS STATUS

We have consistently omitted one important quantity from our Control Component descriptions: the execution state of each process. A user viewing a snapshot composed only of program, data, and dynamic Scenes could not predict from it the appearance of the next, since he does not know which processes are running, which suspended.

We have therefore added one last Context Scene type: the status Scene. Figure 4-5 contains an example of one. It indicates for each process the execution status of that process: RUNNING, READY, or SUSPENDED. In a single processor system there can be but one RUNNING process; those lacking only the processor to run them are instead termed READY. For most purposes the two states can be considered equivalent.

We have further distinguished suspended processes in the status Scene by including in their status the reason for their suspension. (A final state, terminal, is aften included in the set of process states (see for instance [14] or [4]). In MISLE programs, for simplicity, all structures connected with a process disappear on process termination. The entry therefore just disappears from the STAT Scene).

Table 5-3 is a list of the current STAT Scene state descriptors.

Table 5-3. Copilot Process Execution States

FLAG	STATE	REASON
▶	RUNNING	The processor is executing this process, either because it has the highest priority of any ready process, or because one of them, after all, has to run.
	READY	This process will run when the processor can be assigned to it.
	VIRGIN	This suspended process has been created, but has never been READY or RUNNING.
	SUSPENDED	This process was unconditionally suspended, either by its own volition or by some other process with the right to suspend it. Only another process can reactivate it.
	STEPPED	This process has unconditionally suspended itself due to completion of a "single step" command to execute but one complete statement (see Section 7.C7). The state is otherwise identical to SUSPENDED.
!	AWAITING x	The string x is a description of some condition whose occurrence will ready the process. The flag ("!") is present only if that condition is to be satisfied by user action (or a procedure running for the user). The flag blinks on and off until the user stops it.
*	BROKEN	This state is again equivalent to SUSPENDED, except that suspension occurred due to a Break Statement. The flag ("*") flashes until the user stops it (or causes the process to continue execution), in order to draw his attention to the breakpoint's occurrence.

5.G. USER SCENES

The Scene is the basic unit of classifiable allocation for the storage of data to be displayed by COPILOT. Only Scenes can be mapped into Regions for view. So if the user wishes to display and edit his own information, he will need Scene types in addition to those we have provided. Consequently we might provide primitives to do the following:

1) Create a new Scene type, assigning a name to a system-provided type identification.
2) Specify for a new Scene type a process which will activate (trigger) whenever selected events occur (Scene made visible, user changed line, etc.).
3) Create and name new Scenes of any type, and explicitly insert or delete information from them (unless they are protected from modification).
4) Delete Scenes.

This is an undeveloped area of COPILOT. The definition of user Scenes would follow the same sorts of derivations we have used for the context Scenes. The triggered process (above) could maintain user-defined structures corresponding to text Scenes, just as COPILOT routines do for context Scenes– we will describe these methods in Chapter 8. Graphic (non-textual) Scenes should not prove difficult.

5.H. REGIONS

Regions are named areas with fixed Screen locations. A Region contains the following fixed (*) attributes:

1) Its name, a unique global identifier.
2) Its location (x in columns, y in lines, Screen #) and extent (x in columns, y in lines) – thus its Window size.

When the user or the system Maps a Scene into a Region, the Region acquires the following dynamic attributes:

1) The mapped Scene's Name, and therefore indirectly the Scene data, type, length, capabilities, structures, edit and context cursors.
2) The index of the first visible line in the Region.
4) Other bookkeeping information.

- -

(*) Assigned by user or system at Region creation and allocation time -- can be changed by re-allocating.

This is all that is required to generate the display of an active Region.

The initial system configuration contains four Regions:

RSTAT,	with the STAT Scene (also named STAT) mapped into it, showing that only system processes exist, and none are running.
RDYNA,	with the DYNA Scene (named DYNA) mapped, showing that the only active procedures belong to system processes.
RPROG,	with no Scene mapped.
RDATA,	with no Scene mapped.

the user then proceeds to complicate this picture by fetching and interacting with his programs.

5.H1 Regions for Data Scenes – Special Problems and Provisions

Data Regions require special treatment, because more than one is required for all but the simplest tasks. In a fairly complex situation, for instance, there might be one or more data Regions monitoring the progress of running processes, which would cause occasional screen updates by executing data display statements. Another data Region, containing the context cursor, would display the data Scene for the possibly suspended process currently under the user's direct control.

We have discovered that these two uses– monitoring running processes, and manually investigating suspended ones– require data Regions with somewhat different behavior. We have therefore subdivided data Regions, providing **fixed** context and **variable** context Regions.

To monitor running processes, we need to guarantee that successive values of a variable (and *only* values of *that* variable) will be displayed in a single location of a screen. The alternative would be an impossibly "noisy", confusing situation. We therefore provide **fixed** context, or simply **fixed**, data Regions. A fixed Region is one which is constrained to the display of variables in the lexical range of but one program block, and from but one process. Whenever any instance of that block, or any of its lexical parents, executes a data display statement, a data Scene containing a Snapshot of that instance will appear in the Region. The structure of data appearing in that Pegion remains fixed, although the values, and even the procedure instances represented there, may change. The user should create a **fixed** Region for any program section whose behavior is of long-term interest.

For convenience, we have relaxed the Snapshot requirement of Section 5.B4 to permit the retention of a data Scene in a fixed Region after the corresponding procedure instance has disappeared. This Scene is replaced whenever a new instance of the same context is created; it is deleted whenever the process suspends, if no corresponding instance really exists. This facility prevents a fixed Region from flashing and flickering, as instances appear and disappear.

Our other application is the manual observation of data values. In this instance, a data Scene responding to variable query, or to "single-step" operations, will not be changing rapidly. Successive operations might require the creation of entirely different data Scenes. For convenience and conservation, we would like to be able to display all these Scenes, successively, within the same Region. We will call such a Region a **variable** context, or **variable**, data Region. Any data environment may be displayed in a variable Region.

One variable Region must be selected at all times as a **default** for the display of data which do not fit into any fixed Region. Initially, the RDATA Region is the only available Region for data Scenes. RDATA is a variable Region. Until more Regions are created, it provides all data display services. The user can create specific fixed Regions, and additional variable Regions, if he wishes. In particular, he can designate a new variable Region as the default, to handle otherwise unassigned display requests from running processes. The original RDATA Region may then react to his direct queries, without interference.

Due to our data Scene creation algorithms, variable Regions are susceptible to the annoying "flicker" properties which the fixed Regions avoid.

CHAPTER 6
THE CONTROL ALGORITHM

This chapter completes the user-level presentation of COPILOT. It has three major sections, roughly responsible for describing:

A) The Block and Process Structures of the COPILOT System

B) The COPILOT terminal control (USER) loop

C) Constraints on MISLE statements used for top-level control

6.A. SYSTEM STRUCTURE

Every COPILOT job, whatever its function, can be expressed in MISLE as:

```
procedure COPILOT;
begin
   •universal;      comment system intrinsics, basic process-control primitives, global data
                            structures;
   •targets;        comment a Scene which in turn contains links to the program Scenes
                            containing the user target, or applications, programs;
   begin
      •system;      comment all IPS data and primitives, display primitives, invisible to
                            Targets;
      •post;        comment a high-priority process to post state change information and
                            data display requests;
      •ucp;         comment the special User Control Process (see text);
      •assistants;  comment processes created by the user to perform "macro" actions for
                            him. They have access to IPS primitives and data.;
      sprout(post,post,...,high);
      sprout(ucp,ucp,...,higher);
      •user;        comment the terminal-response program, the active body of COPILOT;
   end
end comment the system;
```

Figure 6-1. Global COPILOT Structure

This program is contained within the system as a Scene named Copilot. When a user activates a COPILOT System for himself, the target and assistants Scenes are empty, and the UCP is initialized as shown below. During initialization, the equivalent of a Sprout(User,Copilot,...,highest priority) operation occurs, placing the Copilot procedure in the base of the environmental hierarchy: the process named User. This procedure, whose only active code is the basic control, or User loop, thus constitutes an entire COPILOT job. All user and system subprocesses are, as evidenced in the skeleton above, formed from routines local to Copilot. (This skeleton is a "real" one: it supports the actual COPILOT system, when it is fleshed out by expansion of the Scene links).

The target Scene contains links to the user's target, or applications, programs– the programs which he has written, and with which he wishes to interact. Target procedures, running in their own process or processes, have access only to the basic system routines and structures, and to the environments which they themselves create. They interact with system processes only indirectly (through the event mechanisms), when they suspend, terminate, or request modifications to data Scenes. This denial of lexical access to system environment is useful not only in the protection it affords, but also in the storage efficiency it can support (by allowing system routines and data to be "swapped out" of main storage while inactive– Section 8.D1).

Target procedures, lacking convenient access to the IPS's interactive environment and facilities, can not be written to operate in the user's stead, performing directly statements which define the meanings of terminal commands. Such an ability is desirable, both to facilitate execution of command sequences ("macros"), and to allow composition of more sophisticated sequences, embedding these IPS statements in conditional and iterative control statements.

The assistant Scene is designed to serve this purpose. Routines which are contained in the assistant Scene, or in Scenes accessible from it, do possess the necessary access to perform IPS operations. We will demonstrate in Section 9.A1 the means for invoking these sequences, and for maintaining system integrity when faulty routines are executed.

The sections which follow describe the operation of User, and its parasitic UCP, which define completely the input behavior of Copilot. The concluding section explains the role of the POST process in maintaining the Context Scenes– defining the output behavior.

6.A1 The UCP – User Control Process

The following represents the initial contents of the Scene named UCP:

```
procedure ucp;
begin
►end
 ▲
```

(The context and edit cursors are, as we have said, not part of the Scene data). The system implements but one instance of the UCP procedure, as shown above in Figure 6-1. The sole use of this process, also called UCP, is as a repository of user-submitted statements to be executed. Operation of the UCP is controlled exclusively by the User loop; descriptions of UCP functions begin in Section 6.B3.

6.A2 Crucial Primitives

We will introduce most of the IPS primitives, those statements invoked by the user at the terminal, in the next chapter. A few, however, are crucial to the operation of the USER loop itself. Brief descriptions of these routines follow.

stepp(process,"↓"). This function is a special modification to the activate function. Its effect is to activate (make READY) the selected process, having first conditioned that process to suspend itself on completing one MISLE statement – the one in the current environment (IP, EP) for this process. When the process suspends, it will cause a suspension event, for the Post process, containing the location of the suspension and the reason (STEPPED) for it. Stepp is usually executed by a process at the same or higher priority than the process it readies, so that the activated process will not run until the activator next suspends. The stepped statement may be simple, or it may be complex, containing substatements and procedure calls which arbitrarily extend its effects and duration. If Stepp is applied to a running process, its effect is normally to extend that process's execution by another statement, before suspending; but see Section 7.C7.

set_p(process, statement). The instruction point (IP) for this process is placed, if legal, at the indicated statement (an expression derived from functions like the next one).

get_struct(scene or region, line, space). This returns the unique statement identifier for the first statement beginning after the indicated point.

insert_line(scene or region, line, "string"). The string becomes a new line, in the selected Scene, just preceding the indicated one. This simple line-oriented function will suffice for insertions in the UCP. More versatile text modification functions may be found in Chapter 7.

delete_line(scene or region, line, count). The number of lines indicated by "count" disappear from the Scene.

6.B. THE USER LOOP

The ultimate interface behavior of COPILOT is determined by the program which listens to the terminal and responds to what it hears: the User loop. We shall first present a MISLE program for a basic User algorithm which (barely) implements a non-preëmptive terminal. We shall subsequently subject this algorithm to a series of refinements which enhance its power and efficiency.

The User process always operates at a scheduling priority higher than any other process's. This allows the User process (whose active agent is the User loop) to be set RUNNING immediately, whenever it becomes READY; the user's commands will have immediate effect. (See Section 8.E3 for our definition of "immediate"). Adjacent high-priority levels are reserved for the UCP, Post, and other special processes (see Section 6.A and Section 9.A1).

6.B1 Algorithm A – Basic

This program, and all subsequent refinements, would occupy the Copilot Scene in the system of Figure 6-1. The meanings of undefined procedures in these examples will be explained in the text following each example. The initial loop does not require the UCP.

```
1 user:
2 while true do begin
3         string statement; integer char, state;
4   read:  statement ← readaline;
5   insert: insert_line(user,8,statement);
6   compil: update_world;
7   doit:
8   cleanup:delete_line(user,8,1)
9 end;
```

Operation of this program is simple: it repeatedly accepts a one line statement from the terminal, inserts it into the text representation of User loop itself, translates (compiles) it, then "falls into" the just-presented statement, executing it in the environment of User. Before returning for more input, it deletes all representations of the statement.

In the function **Readaline** the User process suspends, allowing other processes to run, until the complete line (comprising but one statement) has been presented. When the line is complete, the User process supersedes any other running process and returns the resulting line as a text string. The **Insert** call puts this string into a new line between **Doit** and **Cleanup**. The function of **Update-world** is to perform any compilations necessary to make all program Scenes (including Copilot) executable as they are currently stated. We will defer any further compilation consideration until Chapter 8, where this and other implementation topics appear. The user sees only the source-language behavior; we shall at present assume that the system maintains all necessary structures to make this behavior correct.

The final delete statement returns the Copilot Scene to the state shown in the figure.

This program alone, coupled with the posting algorithm below, can support a nearly non-preëmptive IPS with adequate visual context. It does not, however, satisfy all our behavior match requirements, nor is it free from other shortcomings. Our objections are listed in Table 6-1.

Table 6-1. Shortcomings of User Loop Algorithm A

1) The Readaline function meets none of our abbreviation objectives: only complete statements are permitted. Commands requiring multiple lines are likewise not possible.

2) The User loop is self-modifying! This is unacceptable in Copilot, for all the usual reasons.

3) This algorithm maintains no record of the user's recent activity. Such a facility, although dispensable, is desirable, both as a reference for the user, and as a source of statements for future operations (see Section 9.A3).

4) If the statement at **Doit** requires a long (or infinite) time to execute, the non-preëmptive facility is lost: the user has no way to terminate its execution. The class of permissible statements must be severely restricted, probably to the original system-provided primitives.

5) In practice, this method proves too inefficient for the execution of frequent, simple operations (especially simple text-editing commands).

Remedying these objections is the goal of the refinements we have made to this algorithm. Let us first provide a mechanism allowing abbreviated and "manipulative" commands, in order to eliminate objection #1.

6.B2 Algorithm B – the Expand Routine

To achieve the abbreviation we desire, we replace the statement at Read by:

```
...
read:   char ← readachar;
expand: case getcom(char) of begin
          comment getcom provides a direct mapping of characters
                to commands, often many to one;
            f0( char );
            ...
            ...
            fi( char );
            ...
            ...
            fn( char )
        end
end;
...
```

This particular solution imposes a simple prefix grammar on our terminal "language"; another method with a comparable result would be equally acceptable. The f_i(char) statements may use the original char, as well as local state information, and perhaps even additional input characters (via Readachar). To do this, it may have to implement a sort of local FSA interpreter, in order to gather and correctly interpret the parameters, etc. In other words, although there are no global modes in COPILOT, the basic User loop recognition algorithm may establish local modes, corresponding to parser states, to interpret the syntax of user input. This will normally go unnoticed, but will result in the need for a continuously active facility which permits abortion of a partially completed command input, in order to begin a different one.

Executing any f_i statement assigns a string, comprising a complete MISLE statement, to the string named statement, which is used, as before, in the completion of the User loop. This facility, expanding commands to calls on the primitive IPS functions defined in Chapter 7, permits the terse commands exemplified in Section 4.C2.

This is but one recognition algorithm. Any method for generating statement strings from input character sequences could be substituted for it, to provide a custom-tailored user interface (see Section 9.C3).

6.B3 Algorithm C – Using the UCP

This small modification removes objections •2 and •3 from the list in Table 6-1 (self-modification and the lack of a history list), and alleviates •4 (lengthy or non-terminating input statements). In Algorithm C, we replace the Insert, Doit, and Delete statements by:

```
...
insert:  insert(statement, ucp, currentline);
          currentline ← currentline + 1;
compil:  ... as before ...
doit:     suspend(ucp); stepp(ucp);
cleanup: if desired then delete(ucp, 2, 1); comment optional;
...
```

This algorithm does not modify itself (objection •2). Instead, it adds its statements to the UCP text Scene, then causes them to be executed in the UCP process. Depending on the predicate **desired** (optional), algorithm C retains all or part of the user's input sequence, or **protocol**, in this Scene. By mapping the last Window of this Scene to a Region, the user can have a visible record of his recent activity. This Scene may be edited, with interesting results. We will pursue this subject further in Section 9.A3.

Objections •2 and •3 have been overcome by the introduction of the UCP Scene. However the most radical change in Algorithm C is the introduction of the UCP *process*. The Stepp call at **Doit** arranges to READY the UCP process. Its IP is set to the newly compiled statement. Its EP is the activation record for the UCP procedure within the UCP process; since the procedure has no parameters or local variables, this data environment is virtually identical to that of **Doit**. Thus this change in the algorithm cannot change the meanings of user statements.

The User process, because it has the highest priority, continues to run after the UCP activation statement at Doit. The User process does not suspend until control returns to **Read**; then the UCP, at a slightly lower priority, is guaranteed to run. The UCP process suspends again after executing the one statement. We have achieved the final decoupling needed for a non-preëmptive system since, by giving another command, the user can supersede execution of the previous one (assured by the explicit UCP suspension at Doit). This implicit abortion facility, though useful for terminating long or runaway commands, may not always be desirable. See Section 9.B2 for further consideration of the conflict between "type ahead" and command "abortion".

Although the decoupling achieved by executing user statements in the UCP process prevents any user-initiated operation from locking out (preëmpting) the terminal, it is not the preferred method for accomplishing lengthy functions. Instead, statements executed in the UCP should be restricted to those whose operations will complete in a time consistent with the response time of the system (a matter of one or two seconds at most). Anything which takes longer should be accomplished by activating a separate process to do it. The system provides this facility for standard kinds of operations (e.g., string search within text Scenes), and could make it easy for the user to use it for his own operations. The UCP's major functions are to collect a user input history and to eliminate modifications to User loop code. Normally, it will run in "lock-step" with the User process, behaving more as a subroutine than as a coroutine or parallel process.

6.B4 Algorithm D – Selective Interpretation

We can expect certain basic operations to occur quite frequently during the course of a session with COPILOT. Examples are cursor-moving operations, and process control functions such as Stepp. To perform these operations on current hardware, using the insert/compile/execute algorithms of this section, is quite expensive. For more complex operations, even fundamental ones, the inherent flexibility of these methods justify the cost.

As one possible remedy to the expense of basic operations, we can enclose the final steps of the User loop with the following conditional:

```
...
if length(statement) ≠ 0 then begin
    insert: ...;
    ...
    cleanup: ...;
    statement ← null
end;
...
```

Now any of the f_i cases at **Expand** can leave the string variable (statement) empty, and directly execute the statement which it would otherwise store there. Since the execution environment at f_i is effectively the same as the UCP environment, the effect is guaranteed the same.

A serious flaw in this modification is that by bypassing the Insert step we have eliminated the recording of some of the user protocol, thus re-introducing objection #3 (in Table 6-1), with an irritating mutation. We have deferred discussion of this anomaly to Section 9.B1. Fortunately, this recognition and expansion algorithm is easy to replace and modify, in a modular fashion (see Section 9.C3).

6.C. THE POST PROCESS

The input services of the User-UCP process pair join with the output services of the Post process to define the interface behavior of COPILOT. Post maintains and displays the context Scenes. Whenever it runs, it updates the contents of the dynamic, the static, and all data Scenes, assures that all program Scene cursors are correct (other processes maintain the program text), and displays the results for visible Scenes.

The Post process runs only in response to specific status changes in the running processes, or to specific requests by these processes. The mechanism in each case is the same: when a process makes such a request or changes its status, it causes a posting event, whose value contains a code describing the reason. This occurrence wakes the high-priority Post process, which then issues an updated snapshot.

The Post process operates in response to events caused by:

1) Process suspension. The process has BROKEN, STEPPED, SUSPENDED, or is AWAITING some external waking condition (an event occurrence). The reason for suspension, and the current process state, are supplied in the event notice's value.

2) Process activation. Snapshots are issued whenever a process becomes READY, and again when it begins RUNNING.

   (This choice assumes that processes change state infrequently with respect to the overhead for issuing a snapshot. We could choose to bypass snapshot issuance where process activation or suspension does not directly interact with the IPS facilities.)

3) Data display requests. The statements of Section 7.D1, by causing posting events, cause variables to be added to and removed from DATA Scenes. The Post process responds by adding or removing these variables, then performing a standard update.

We could have implemented posting through subroutines declared in Copilot's outer block. The scheduler routines and data-display statements would call them to report results. We have chosen the process/event mechanisms instead, as we have for other facilities, because this decoupling allows us to embed all system structures and display routines in a block inaccessible to target programs, affording them protection and name space independence from each other. In addition, in Section 8.D1, we will show that this structure, with appropriate segmentation, helps us achieve space efficiency.

6.C1 Display of Users' Scenes

The Post process only maintains context Scenes. However, it will update the display of all Scenes which currently have visible windows. this relieves the user of much of the effort of displaying his Scenes. He need only maintain the data in the Scene and indicate current cursor and window positions. He can have his programs issue a Post-only request for immediate visual response. In this way he can synchronize his data display with the Context snapshots. Facilities exist as well for directly updating a user-maintained Scene, for better efficiency.

CHAPTER 7
COPILOT TERMINAL PRIMITIVES

This "user's manual" chapter explains many of the terminal operations which are built in to COPILOT. It should also serve as a guide for the implementation of additional features.

The first section deals with the user-accessible structures for describing and manipulating system entities such as Scenes, Regions, and processes. It also defines terminology for these entities. The following section presents a small number of variables, global to the User and UCP routines, which are central to system operation.

Section 7.C is a description of the more important primitive system functions, and the terminal-level commands which use them. The last section defines the semantics of the special statements of Section 5.C5.

7.A. USER-ACCESSIBLE STRUCTURES

In the previous chapters we presented process-control statements which used integer values as process designators. We did this because the MISLE language lacks sophisticated data type facilities. We will extend the use of integer values as structure designators, to handle objects such as Scenes and Regions. We will also employ them as instruction point, environment point, and context point indicators (ip, ep, and cp). (†)

A structure designator is always generated by the system, on request. Structure designators are unique, like LISP atoms or LEAP Items. Associated with each structure is a structure type code defining what kind of entity it represents, as well as its actual value: Scene data, a process stack, etc.

Some structures (Scenes, Regions, processes) possess string-valued pnames, used to identify them in Scenes. Whenever such an entity is stored in a named variable, our convention is that the entity name and variable name should be the same.

- -

(†) In a language which provides structured data facilities, these entities lose their distinction.

Instruction, environment, and context points could be represented as <Scene, line, column> triples, where program, data, or dynamic Scenes, respectively, would be selected. For convenience, however, we have chosen to define structure designators for them, collectively called structptrs. One can derive from a structptr the <Scene, line, column> position it defines, as well as the process (if any) associated with it.

Some procedures need the ability to accept as arguments structures of different types. An example is an editing function, whose Scene argument could be supplied directly as a Scene structptr, or indirectly by specifying the Region to which the Scene is mapped. Another example is a structure-following procedure which can be applied to any structptr (see the successor functions of Section 7.C4). These functions can obtain the structure types of their parameters, and can perform appropriate conversions, using the access primitives of the next paragraph. (Scene types are subtypes of the structure type "Scene").

7.A1 Access Primitives

From a given structure it is often possible to derive related structures or values: the Scene, line, and column locations of an ip, the Region corresponding to a Scene, or the current line and column locations for the edit cursor ("▲") in a Scene (or Region, if mapped). The following table defines a set of access (conversion) primitives and the structure types they will accept. Legal types are marked "x" in the table; braces surround the entry ("[x]") if the legality of the function depends on the Scene type of its argument (e.g., an IP can only be obtained from a program Scene). Each function attempts to return some reasonable default when the requested value is meaningless (marked "-"), or does not exist.

Table 7-1. Structure Access (conversion) Primitives

Struct. Type	Scene	Region	ip	ep	cp	process
Function						
GET_SCENE	-	x	x	x	x	-
GET_REGION	x	-	x	x	x	-
GET_LINE	x	x	x	x	-	
GET_COLUMN	x	x	x	x	x	-
GET_IP	[x]	[x]	-	-	-	x
GET_EP	[x]	[x]	-	-	-	x
GET_CP	[x]	[x]	-	-	-	x
GET_PROCESS	-	-	x	x	x	-

7.B. GLOBAL STRUCTURE VARIABLES

The variables of Table 7-2 form the bases for access to all IPS structures. They are declared in the System block (in the *system Scene); they provide access to all Scenes and Regions, most processes, and some location structures (structptrs). (The primitives for creating Scenes and Regions cause declarations for the new objects to be inserted automatically into the System block.)

Table 7-2. Global IPS Structure Variables

RPROG, RDATA, ...	Structptrs of the initial Regions.
PROG, DATA, DYNA, ...	Structptrs of the initial Scenes.
CURRENT_REGION	Structptr of the Region, in which the edit cursor ("▲") is visible, and which therefore is affected by edit commands.
CP	Structptr of the current Context Point, seen in the RDYNA Region as a context cursor ("▶").
IP	The Instruction Point selected by CP.
EP	The Environment Point selected by CP.

7.C. THE COPILOT TERMINAL PRIMITIVES

The COPILOT design includes an intermediate mapping between the terminal commands and the corresponding lengthy primitives. For each command we have defined a **command procedure**, whose name is short and at least moderately mnemonic, which is defined in terms of one of the primitive functions (supplying the default arguments to it). Each command procedure accepts only one or two parameters, those which the user might provide in his terminal commands.

As an example, the command procedure expansion of the "<rept><cr>" command in Section 7.C2 is "DOWN(<rept>)"; its meaning is, as before,

"MOVE_CURSOR(CURRENT_REGION,<rept>,-999,0,0)" .

Although the intermediate command procedures make sequences of IPS statements easier to read and modify (in the UCP and in assistant procedures, for instance), the extra level of mapping does not aid their exposition. In the descriptions which follow, we will directly express the COPILOT commands in terms of the primitive functions.

The casual reader need not study the function descriptions in detail; he may scan the calling sequences and read the command descriptions to infer their general behavior.

7.C1 Notation

We have arranged the following pages in pairs: even-numbered (left-facing) pages contain the names, calling sequences, and descriptions of primitive functions. The odd-numbered (right-facing) pages describe the commands whose expansions use these functions. Some copies of this dissertation are printed in one side only. The reader may find it convenient to reverse the even pages in order to accomplish this correspondence.

On the function description pages, when more than one structure type is permitted as a parameter, the alternatives will appear as [scene|region]. This example will commonly be abbreviated [s|r].

The command descriptions employ the following conventions:

The left column, labelled "COMMANDS", lists the commands, with possible parameters, using the notation of Table 7-3. The middle column, "EXPANSIONS", defines for each command the MISLE statement, in terms of the specified parameters, which the User loop algorithm creates from that command, and which it will cause to be executed. The expanded statements in this presentation are all calls on primitive functions, using a "keyword parameter" form: PCALL(x=5, y="abc") means PCALL(5,"abc"), where the formals used in declaring PCALL were x and y, respectively. Whenever the procedure name is omitted from an expansion, the most recently mentioned procedure is intended; whenever a parameter is missing, the most recently mentioned parameter with the same keyword is intended.

Table 7-3. COPILOT Command Notation Conventions

α	The CONTROL key should modify the command character
β	The META key should be employed
●	Both CONTROL and META are required
<cr>	Carriage return
<lf>	Line Feed
<alt>	Alt mode -- a special "escape character"
<vt>	Vertical tabulation character
<sp>	Space, or Blank, character
<bs>	Backspace, or Delete, character
<rept>	A numerical repeat factor, composed of α<digits>

7.C2 FUNCTIONS, EXPLANATIONS

MOVE_CURSOR([scene|region],lines,spaces,windowline,limitflag)

This function moves the edit cursor for the selected Scene a specified distance relative to the current edit cursor position for this Scene. It also adjusts the position of the window on the Scene, if it is mapped.

[scene| region]	Must be a valid Scene, or the designator for a mapped Region. In the latter case, MOVE_CURSOR applies GET_SCENE to select a Scene.
lines	Number of text lines to move (positive is "down", negative is "up").
spaces	Number of columns to move (+/-).
windowline	If limitflag enables it, after determining the new cursor position, arranges the window such that line 1 of the window is 'windowline' lines away from the cursor line (+/-). Adjusts if necessary so that the cursor is in the window.
limitflag	
0:	Cursor may move beyond current window boundaries, adjust window to make the cursor visible, if mapped.
1:	Cursor may not move beyond current window boundaries.
2:	Cursor may move beyond current window boundaries, place window as nearly as possible to the position indicated by 'windowline'.
3:	Cursor may not move beyond current window boundaries. Update window after moving cursor.

SET_CURSOR([scene|region],line,space,windowline)

This function is equivalent to:

MOVE_CURSOR(region,-999...9,-999...9,0,2), then
MOVE_CURSOR(region,line,space,windowline,2).

In other words, SET_CURSOR sets cursor and window to "absolute" positions (relative to the beginning of the Scene).

FIND_STRING([scene|region],"srchstr",number)

This always uses GET_LINE(s|r) and GET_COLUMN(s|r) for its position. It searches from that position to the number'th occurrence of the search string, and does a new SET_CURSOR if it finds enough matches. Otherwise, the user is informed that the search failed, and the edit cursor is not moved.

Preceding page blank

COMMANDS	EXPANSIONS	COMMENTS
<rept><cr>	move_cursor (region=current_region, limitflag=0, spaces = -999, lines = <rept>)	Moves edit cursor <rept> lines vertically, horizontally to left margin. Cursor may move out of current window, requiring window adjustment.
<rept><vt>	lines = - <rept>	<rept><vt> performs ●-<rept><cr>.
<rept><lf>	spaces = 0 lines = <rept>	Moves edit cursor <rept> lines vertically, but not horizontally.
<rept><alt>	lines = <rept>	<rept><alt> performs ●-<rept><lf>.
<rept>α<sp>	lines = 0 spaces = <rept>	Moves cursor <rept> columns forward, horizontally.
<rept>α<bs>	spaces = -<rept>	Moves cursor <rept> column backward.
●T	limitflag=1 spaces= -999...9 lines = -999...9	Moves cursor to top left hand corner of screen (window).
●B	lines = 999...9	Moves cursor to bottom left corner.
●J	spaces = 0 lines = 0 windowline = line limitflag = 2 (or 3)	Moves current line to top of screen, adjusts window so that the line with the cursor is line one of the window.
●W	lines = 999...9, limitflag = 3, windowline= -999...9,	Moves the bottom line to the top of the Region (if possible), by adjusting the window.
●L	lines = -999...9, windowline = 999...9,	Moves the top line to the bottom of the Region, if possible.
<rept>●F<str><cr>	find_string (region = current_region, srchstr = "<str>", number = <rept>)	Sets the edit cursor to the location of the <rept>th copy of "str", starting at the current position.

7.C3 FUNCTIONS, EXPLANATIONS

NEXT_REGION (region, howmany)

There is some reasonable circular ordering among Regions, based on their Screen position. NEXT_REGION yields the Region structptr for the howmany'th region from the one specified.

EDIT_REGION(region, line, space, windowline)

This function selects the specified Region for (terminal) editing. It then performs a SET_CURSOR operation using the remaining parameters. If any parameter is -1 it is not changed from the setting it had the last time this Region was edited. (Region-switching is a sort of coroutine-switching operation.)

CHANGE_CHAR([scene|region],line,space,"char(s)",number)

CHANGE_CHAR can refer to its Scene directly, or indirectly through its mapped Region. Its function is to insert, replace, or delete characters from the Scene. The edit cursor is always placed beyond the affected string on termination of the command.

char		A 7-bit character.
number	=0:	replace current character(s) with 'char(s)'.
	>0:	insert 'char(s)' before current.
	<0:	delete |number| characters at current position.

scene, region, line, space as before.

EDIT_CHAR([scene|region], "char(s)", number) is:

```
CHANGE_CHAR([scene|region], GET_LINE([scene|region],
           GET_COLUMN([s|r]), "char(s)", number )
```

INSERT_LINE ([scene|region], line, string)
DELETE_LINE ([scene|region], line, count)

The specified string is inserted as a text line before the indicated line. (Or) count lines are deleted at the indicated line.

COMMANDS	EXPANSIONS	COMMENTS

<rept>●R	edit_region (
	line = -1
	space = -1
	windowline = -1
	region = next_region
	(current_region,<rept>)
		Selects for editing the howmany'th Region from the currently selected Region. Makes the edit cursor visible in that Region.

<arg>●R	region = <arg>	Selects the named region, as above.

<char>	change_char (
	line = get_line
	(current_region),
	space = get_column
	(current_region),
	region=current_region,
	number = 0,
	char = "<char>")
		<char> is a 7-bit, non-activating character. Replaces with it the character under the edit cursor.

β<char>	char = "<char>"	Inserts <char> at the edit cursor. Move other characters over.

<rept><bs>	space = cur.. - <rept>
	number = -<rept>
		Deletes <rept> characters to the left of the edit cursor.

<rept>αD	space = cur...	Deletes <rept> characters to the right of the edit cursor.

<rept>●<cr>	space = 999...9
	char = <●●●●...●>
	number = 1
		Inserts <rept> new lines after the current one. (● is <cr>).

<rept>●D	delete_line (
	region = current_region
	line = get_line(current_region)
	count = <rept>)
		Deletes <rept> lines.

7.C4 FUNCTIONS, EXPLANATIONS

structptr ← GET_STRUCT ([scene|region], line, space)

In a PROG Scene, finds the closest statement to the specified location, and returns its structptr. The effect is similar in a DATA Scene, returning the closest equation (next page).

structptr ← EDIT_STRUCT ([scene|region])

EDIT_STRUCT([s|r]) is defined as:

GET_STRUCT ([s|r], GET_LINE([s|r]), GET_SPACE([s|r]))

structptr ← NEXT_STRUCTURE (structptr1, "code")

code "↓" Given structptr1 (denoted by SC in the following examples), NEXT_STRUCTURE returns its successor (SN in these examples):

```
... BEGIN .... SC; SN ... END;
... BEGIN ... BEGIN .... SC END; SN ...
... IF ... THEN SC ELSE SN; ...
```

"↑" Returns the predecessor to structptr1. The definition is similar.

"→" Returns the first substructure of strucptr1, if it has any. Otherwise returns structptr1: SC is structptr1, SN the resultant substructure in the following:

```
... SC: BEGIN SN; ... END; ...
... SC: IF ... THEN SN ELSE ...
... SC: SN: I←3; ...
```

"←" Returns the "father" structure, SN, to the given structptr1, SC:

```
... SN: BEGIN .... SC; ... END;
... SN: IF ... THEN SC ELSE ...
```

"H" Returns a structptr to the block or compound statement containing the given structptr1.

COMMANDS	EXPANSIONS	COMMENTS
●↓	struct_move (region = current .region, code = "↓")	Moves edit cursor to the statement (or corresp. structure, for other Scene types), following the stmt. nearest current the cursor pos.
●↑	code = "↑"	Moves edit cursor to the statement preceding the nearest one.
●→	code = "→"	Moves cursor to first nested stmt.
●←	code = "←"	Moves cursor to father stmt.
●:	code = ":"	Moves edit cursor to the statement nearest its current position.
●H	code = "H"	Moves cursor to block head containing the nearest statement.

7.C5 FUNCTIONS, EXPLANATIONS

STRUCT_MOVE (region, "code")

STRUCT_MOVE is defined as:

```
BEGIN
    integer stmt;
    stmt←EDIT_STRUCT(region );
    if code ≠ ":" then
        stmt ← NEXT_STRUCTURE (stmt,code);
    SET_CURSOR (region, GET_LINE(stmt), GET_COLUMN(stmt), -1)
END;
```

APPLICATION OF STRUCT_MOVE TO OTHER SCENE TYPES:

DATA SCENES – Let EC be the equation nearest the edit cursor, EN the equation identified by that cursor after performing the command:

•↓	3.proc•2(...EC, EN,....) 3.proc•2(....EC); begin EN; ... EC: 3.proc•2(...); begin ... end; EN:...
•↑	(inverse of •↓)
•→	EC: 3.proc•2(....); begin EN;
•←	(inverse of •→)
•:	EN=EC.

DYNA SCENE – Each number is some instance node– "⊃" means "yields":

```
1
|                 •↓ at 1 ⊃ 2  •↓ at 2 ⊃ 3
2------           •↓ at 4 ⊃ 5  •↑ at 4 ⊃ 3
|     |           •↑ at 5 ⊃ 2  •→ at 2 ⊃ 5
3     5           •→ at 5 ⊃ 5  •← at 6 ⊃ 2
|     |           •← at 2 ⊃ 2  •← at 4 ⊃ 4
4     6           •H at 6 ⊃ 5 •H at 4 ⊃ 1
```

7.C6 FUNCTIONS, EXPLANATIONS

scene ← SCENE_LINK([scene|region], line, space)

This command follows Scene links ("* scene" constructs). Given a location within a Scene, it finds the nearest Scene link, if any, and returns a structptr to the Scene it identifies. If there are no Scene links, it returns a null structptr, which should be treated as an error or "no-operation".

MAP_SCENE (scene, region, first line, fspace, fwindow)

This makes the Scene visible within the Region, and sets the window and edit cursor positions as specified, using SET_CURSOR.

COMMANDS	EXPANSIONS	COMMENTS

•M	map_scene (
	scene = scene_link (
	region=current_region,
	line = cur...,
	space = cur...),
	region = current_region,
	first line = 1,
	fspace = 1,
	fwindow = 1)
		Follows the nearest •scene link.

<arg>•M	scene = <arg>	Maps the indicated Scene into the current Region (the one with an edit cursor).

7.C7 FUNCTIONS, EXPLANATIONS

SET_P (process, [ip|ep|cp])

Places the context cursor at IP if structptr is a statement, EP if it is an instance in a data Scene, or CP for a dynamic Scene.

STEPP (process, code)

code:

"↓"	Single-steps one statement.
"→"	The same as "↓", if the statement at IP has no substatements. Otherwise, executes to the first encountered substatement (see examples on next page).

Stepp activates the process, at its current IP and EP, first setting Synch variables to suspend after the desired execution. The "→" code suspends execution at the first encountered substatement of the one indicated by IP.

STEPPN (process, n)

This is a multiple-step command. If n=2, it executes the next two statements before suspending; if n=3, the next three, etc. When applied to a statement within n statements of the end of a loop statement, n is reduced to prevent executing beyond that scope.

STEPP(--,"↓"), when applied to a process which is already being stepped, has the effect of STEPPN(--,n), for n=2, 3, ...

(ACTIVATE (process), SUSPEND (process), SUSPALL())

These are the normal MISLE functions for activating, and suspending processes. SUSPALL suspends all but USER.

TO_CONTEXT (process)

This sets RDYNA, RSTAT, RPROG, and RDATA Regions to the Scenes describing the context point of the selected process. If -1 is the argument, it alternates among the suspended Target processes, beginning with the most recently broken one. This is the normal way to establish context after a BREAK.

Process	either -1 (some broken process), or a process id.

COMMANDS	EXPANSIONS	COMMENTS

●;	set_p (
	process = get_process (ep),
	edit_struct(current_region))
		Moves the context cursor, representing an ip, ep, or cp (depending on Scene type), to the stmt., equation, or procedure instance nearest the edit cursor.

●X	stepp (
	process = get_process(ep),
	code = "↓")
		Single-steps one stmt. in the p context cursor is visible (the current process).

<arg>●X	process = <arg>	Single-steps the selected process.
<arg>●S	code = "→"	"Steps in" to (executes to the first substmt. of) the current stmt. of the selected process.
●S	process = get_process(ep)	"Steps in" to the current process
●P	activate (process = get_process (ep))	Proceeds– readies the current process.
<arg>●P	process = <arg>	Readies the selected process.
{<arg>}●.	suspend (get_process (ep)), or <arg>	Stops (suspends) a process.
●:	suspall ()	Stops all processes.
●B	edit_char(region = current_region char(s) = "{break (get_process(ep))}", number = 1);	Sets a break point at the edit cursor pos. Will break only when the process encountering is the one which now has the context cursor.
<arg>●B	process = <arg>	instead of ..get_p... Sets a break point at the edit cursor position and specifies which process can trigger it.
●▶	to_context (process = -1)	Switches context Scenes to a representation of the environ-ment of some reasonable process (see previous page).
<arg>●▶	process = <arg>	Switches to the context of the chosen process.

7.C8 FUNCTIONS, EXPLANATIONS

EVAL("statement", ip, ep)

Effectively, the statement is inserted in the scene at ip. Then it is executed in the environment (therefore the process) of ep. When the process suspends (on eventual completion of that full step), the statement and all levels of representation are deleted.

COMMANDS	EXPANSIONS	COMMENTS
●!line<cr>	line	Executes the line as one statement.
●&line<cr>	eval("line", ip, ep)	Evaluates the line in the selected environment.

7.D. SEMANTICS OF SPECIAL STATEMENTS

In Section 5.C5, we presented the syntax for a set of MISLE constructs which are especially useful in an interactive environment. At that time we had not adequately presented the contexts in which they are useful. Here we will explain these special statements, by means of several examples.

7.D1 Variable Query (Data Display)

Example: J; K↑2+3;

A data display statement comprises a single expression. Executing one causes that expression's value to be displayed in a data Scene. The first statement, above, is representative of the most common use: the display of a named quantity. The variable J (in the scope of the current context cursor), is given the marked attribute (Section 5.D2), if it does not already possess it. this will cause an equation to be created for J, in any data Scenes which display instances of the block or procedure in which J is declared. Data display statements execute by causing posting events which awaken the Post process. When this process runs, it causes all visible data Scenes to be updated-- thus displaying J's current value, among others.

The second example above causes the selected expression to be displayed temporarily in the default data Region (see Section 5.H1). It is difficult to formulate a general algorithm for doing this satisfactorily. We will explore the problem further in Section 9.A2.

7.D2 Breakpoints

Example: BREAK(-1);

This statement always breaks. To do this, it simply suspends, after causing a POSTing event. The post process subsequently indicates in the STAT Scene that the process has BROKEN. The user can, when he chooses, turn his attention to the broken process, examine its causes, then take whatever action is appropriate.

Example: BREAK(TARG1);

This statement will break only when the process encountering it is the one designated by TARG1.

A last BREAK statement, ARR_BREAK, takes an array as its parameter, and will break if the running process is any of those specified in the array.

7.D3 Temporary Statements

Example:

```
{TEST_SRCH: ON! IF SEARCH_CNT MOD 50 ≠0 then BREAK(-1)} J←PT3(XMT);
```

A temporary statement of the form { ts1; ts2; ... tsn } s1 is functionally equivalent to BEGIN ts1; ts2; .... tsn; s1 END. Similarly, s1 {ts1; ...} behaves as BEGIN s1; ts1; ... END. We make the distinction for three reasons:

1) As a purely visual device. It is easier to see that the statements within the braces are temporary.

2) To aid in insertion and deletion. One need not find the end of the qualified statement (e.g., s1) in order to place an END there, or to remove it.

3) To allow the additional <class> and <switch> syntax.

A temporary statement containing the switch "ON" behaves as one without a switch at all: all its substatements are executed in order, as described above. However, if a temporary statement contains an "OFF", none of its substatements are executed. One may thus turn a temporary statement on and off by toggling this execution switch. Section 8.E8 presents an implementation for this feature which allows inactive (OFF) temporary statements to be left in a program, at no execution cost.

The class label need not be unique to one temporary statement. If a set of temporary statements exists, whose collective function is to monitor a particular situation, one may give them all the same class name. He may then use the TURN ON and TURN OFF statements to toggle all members of a class simultaneously. Class names are global labels, whose scope is the entire system.

The DELETE statement physically removes all statements labelled by a given class name from the Scenes they modify.

Temporary statements give us nearly all the power of Teitelman's ADVISE facilities for BBN Lisp, which allow a user to change temporarily the meaning of a function, whether compiled or interpreted, whether defined by the system or user. We cannot provide his selective advising facility in the current design. (When this is specified, a function is modified by its advice only when called from one of a selected list of functions.)

7.E. CONCLUSIONS

We have presented in this chapter only the essentials of COPILOT. We are convinced that this design provides the basis for many elegant capabilities which are not possible in a preëmptive system, or in one which presents less context. Some suggested extensions to COPILOT appear in Chapter 9. Others will require further research.

CHAPTER 8
IMPLEMENTATION CONSIDERATIONS

8 A. TIERS

We have intentionally couched all our descriptions in terms of the Text Scenes which the COPILOT user can see directly. We have demonstrated that we can provide a remarkably rich set of primitives for IPS control in these terms.

To provide the facilities described in the previous chapters, we require, in addition to the Text Scenes, the support of additional structures. We can see clear evidence of the kinds of structures required in the following:

1) We need the Text itself, for visual display and text operations.

2) We need to locate the **Tokens**, within a text line, which begin selected statements, as, for instance, in the EDIT_STRUCT(...) (●:) command. Some internal representation of program text as lists of Tokens would be useful, though not absolutely necessary.

3) We need access to the program structure, or **abstract syntax tree** [38], of the user's program, in order to perform operations like STRUCT_MOVE (●→, ●↓, etc.), and process control operations. Similarly, we need a structured representation of the names in the user's program (a **symbol table**), closely related to the program tree.

4) Because we have chosen a compiler-oriented system, each statement in each PROG Scene must have a corresponding **code segment** which, when run on the host machine, will perform the specified actions. Conversely, the data (**activation records**) on which these segments operate may be reflected in DATA Scenes at the text level.

We will call these levels of data representation **Tiers**. These same four kinds of Tiers (**text**, **token**, **tree**, and **code**) exist for most of the Context Scenes in COPILOT. We will treat each use in detail below. Each Tier is the most convenient representation of the facts it expresses for some class of system operations.

8.A1 Tier Equivalence

For each Tiered quantity in the system there is a source **Tier**, where new information is introduced. For programs, this is the text Tier, where new statements are added. For data, the code Tier (of activation records) usually supplies the needed information. The contents of each Tier (other than the source Tier) is the byproduct of some translation operation. For programs, these operations have familiar names:

NAME	TRANSLATION (Tier 1 to Tier 2)
Scanning	Text to Token
Parsing	Token to Tree
Compiling	Tree to Code

For data representations, we could speak of Uncompiling, Unparsing, and Unscanning, beginning with activation records in the Code Tier, yielding readable Data Language "programs".

In each case the intent is to create a representation which is in some sense equivalent to the original; that is, its meaning with respect to some set of attributes is invariant over the translation. (For compiling, this is the requirement of correctness. Most formal treatments of compiler correctness concentrate on proving this "equivalence" between the abstract syntax (Tree Tier) and the Code (Code Tier) [39].) In order for the translation to have any value, of course, there must be other attributes which are not invariant: some information will be lost, while other things will be added. Using our program example again, the scanning and parsing operations do not carry program format (spacing, etc.), into the Tree Tier, nor do they always preserve the order of expressions, or even the precise choice of keywords and operators. In addition, through these translations, explicit structural information about a program is added. Further compilation (to code) usually loses some of this structural information, and much symbolic data, while gaining efficient code for execution.

We will say that structures in two Tiers are weakly equivalent, or simply equivalent, if they satisfy (or presume to satisfy (‡)) specified correctness criteria for a selected set of attributes. We will say that two Tiers are strongly equivalent if either can be completely regenerated, given the other.

- -

(‡) We shall not offer any proofs.

There must be, for each class of multi-Tiered entities, and for each adjacent pair of Tiers, a translation rule (algorithm), operating in at least one direction, which will convert from one Tier to the other. Compilers, parsers, and scanners are elements of this set of translators.

8.A2 Inter-Tier Connections

The data of two equivalent Tiers need not be fully independent. Each may contain references to locations or entities in the other. It must be possible, for instance, to find the statement in the Code Tier corresponding to a given node in the program tree.

This division of IPS structures into Tiers and connections between Tiers allows us similarly to segment the universe of IPS system routines into those which deal with the relationships between "adjacent" Tiers, (compilers, etc., as well as routines like GET_STRUCT and GET_LINE), and those whose effects are confined to a single Tier (e.g., MOVE_CURSOR and STRUCT_MOVE).

We will find that it is useful in some Tiers to minimize the number of extra-Tier connections, while other Tiers will contain numerous connections to their neighbors. We will discuss the advantages and drawbacks to this imbalance in Section 8.D.

8.A3 Tier Fidelity

In his thesis [44], Mitchell states what he calls a **Visual Fidelity Principle**, which requires that "the user must be able to expect that the appearance (text) of a program is a reliable indication of the way that program acts (its semantics)." While this is predominantly a restatement of our Tier equivalence requirements, it carries some additional implications. Program Tiers are not always equivalent; there is a time after new text has been inserted in a program, but before it has been translated, when they are not. If we use the Visual Fidelity Principle as our guide, we require only that Tier equivalence between text and tree be restored before doing any structured editing, and that tree and code Tiers be updated before attempting execution of the modified algorithm. We can extend this notion of fidelity to other translations, specifying for each the conditions which require that necessary translations be made. For instance, code→tree→...→text translations, for data, dynamic, and status information, must occur whenever a posting event (Section 6.C) occurs; and our Snapshot requirement (Section 5.B4) means that *all* such translations must be done whenever any is done. (The Snapshot requirement states that the visible data must represent a subset of total system state at a single previous instant.)

We will define the specific conditions for each COPILOT translation in the following sections. These conditions may be different in other IPS systems, depending on the methods of translation and interpretation.

8.A4 Tiers in other Systems

The Tier concept is our attempt to normalize the naming conventions for the kinds of structures which have been developed for IPSs (and other language systems), including COPILOT. All of the systems we reviewed in Section 3, for instance, have constructs corresponding to the Text Tier; most possess representations corresponding to one or more of our other Tiers: JOSS maintains text only. Most LISP systems keep the trees (S-expressions) and, for compiled functions, the code. Mitchell's system has representations at each Tier levei. We are satisfied with the generality of the Tier levels we have chosen, since we have encountered no trouble in categorizing the structures of other systems in terms of these Tiers.

8.B. SCENE-TIER RELATIONSHIP

In the previous chapters we have developed two mechanisms for storing, naming, and manipulating the data structures in our IPS: Scenes, for managing the text that presents elements of the system to its user; and Tiers, for relating this text to its underlying structures. In this section we will consider the relationship between these mechanisms.

For each COPILOT program Scene in the text Tier there is a directly corresponding collection of token lists in the Token Tier, equivalent to it. Similarly, for each of these collections there exists an identifiable set of equivalent (*) instruction segments in the code Tier. It would be tempting to extend this observation, and to state that each Context Scene can be considered a multi-Tier structure, with disjoint equivalent representations in each Tier (Figure 8-1). This technique, however, immediately leads to trouble in the tree Tier. Since the information in a data Scene represents data generated from the algorithms of

- -

(*) Always in the weak sense

program Scenes, one must expect this relationship to be expressed at some level, through shared structures. The natural place for this sharing is the tree Tier. In COPILOT (Figure 8-2, and Figure 8-3), the tree structures which express data Scene information share symbol table nodes with the program trees; from these symbol nodes, block structure information from the program tree itself is available.

Another difficulty with the disjoint structure of Figure 8-1 is that many data Scenes may exist at once, for many simultaneous instances of the same procedure. These occurrences place a many to one relationship between some Text Scenes and some elements of their equivalent representations. Notice that not even data Scenes and their code Tier information need be in one to one correspondence, since the same information can appear in more than one data Scene.

Because of these arguments, we will relax our proposed Scene-Tier requirements, demanding only that:

1) An observer with access to all system data can derive from a quantity in one Tier all equivalent quantities (1-1 or 1-many) in all other Tiers.

2) Where necessary, direct or computable connections exist between Tiers to allow programs to derive the equivalent entities. Not all possible connections need be derivable.

8.B1 Permanent Scene Representation

For each type of Scene, one or more Tiers contain the most complete information about that Scene. From that Tier, all other representations can be generated. The source Tier (the one into which new information can be introduced) must be one such Tier.

We designate one of these Tiers as the Permanent Tier for each Scene type. We can then choose to maintain equivalent information in the other Tiers only when it is necessary. The permanent Tiers for each context Scene are:

SCENE TYPE	SOURCE	PERMANENT
PROG	Tree	Token
DATA	Code	Code
DYNA	Code	Code
STAT	Code	Code

Examples: data, status, and dynamic text Scenes are not needed at all for non-interactive system operation. Thus it is possible never to generate Token or Text level information for them at all, as long as the code and trees exist for regenerating them. In COPILOT, program Scenes need only exist in the text Tier when they are mapped to Regions, or when a text-oriented function needs to look at it. We maintain all programs permanently in the strongly equivalent Token Tier (see Section 8.C2).

While a user is logged in, COPILOT maintains his program representations for all Tiers. To save space, we could choose to delete Code and Tree information when the user leaves the system. This information would be regenerated when he next logged in, returning his system to the state it was in when he left. Notice that, although there are multiple representations for a given program, they all represent the same algorithm, maintaining the illusion that there is but one representation– text Scenes– within the system for a user's program.

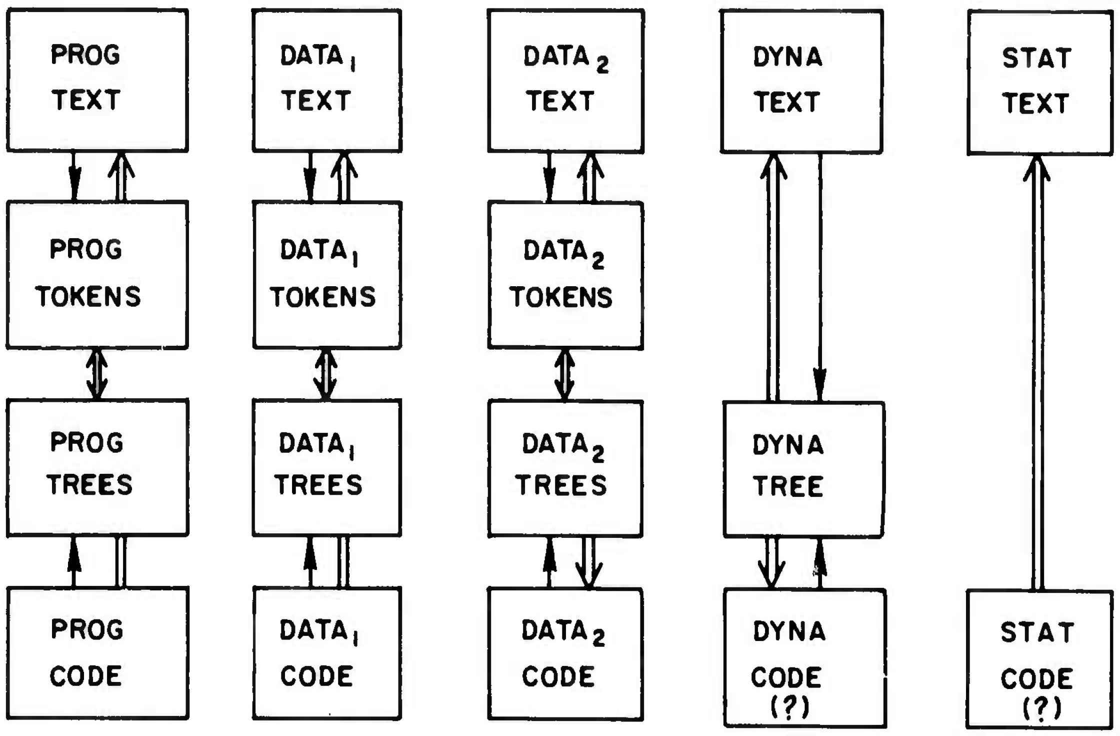


Figure 8-1. (Inadequate) View of Scene/Tier Structures

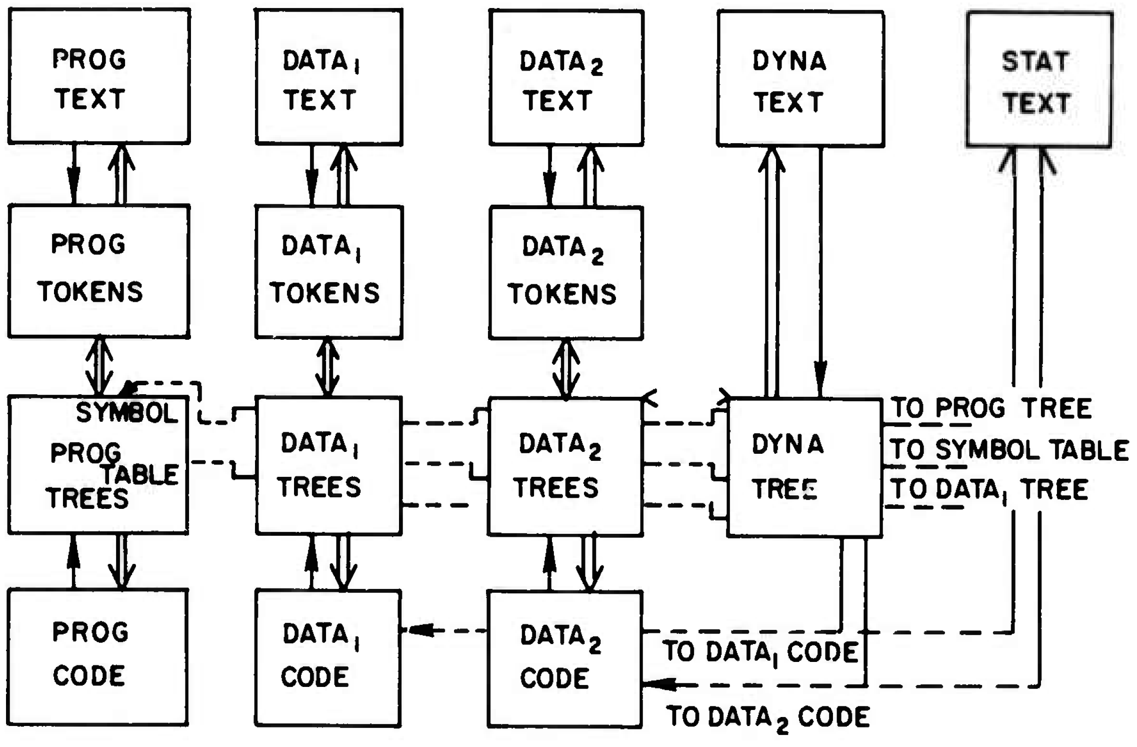


Figure 8-2. Interconnected COPILOT Scenes

8.C. COPILOT TIERS

This section briefly treats the COPILOT Tier structures, defining what each Tier is, and what it is used for.

The COPILOT system is coded in the SAIL language. The token and tree Tiers use the LEAP facilities of SAIL, creating the trees and lists which they require by making associations between items (see Appendix B for a very brief description of the LEAP associative facilities, and of their pictorial representations, used in this chapter.) Numerical and symbolic information in these Tiers are normal Algol-like structures. They are appended to the LEAP nodes as datums.

Specially coded machine language routines manage the allocation and maintenance of text and code Tier data, all in straightforward ways; type conversion routines exist to normalize inter-Tier references in these cases.

Figures 8-3 and 8-4 give an overall view of the COPILOT structures. Since each important aspect will be expanded in later diagrams, much detail is missing from these. They do, however, best demonstrate the sharing of Scene data in the Tree Tier, especially the symbol table entries. Notice also the relative density of references within and emanating from the central Tiers, compared to those of the outer two. Our reasons for this appear in Section 8.D. The process status Scene information (not shown) is quite simple in structure. It is composed only of its text Scenes and the corresponding status data in the code Tier.

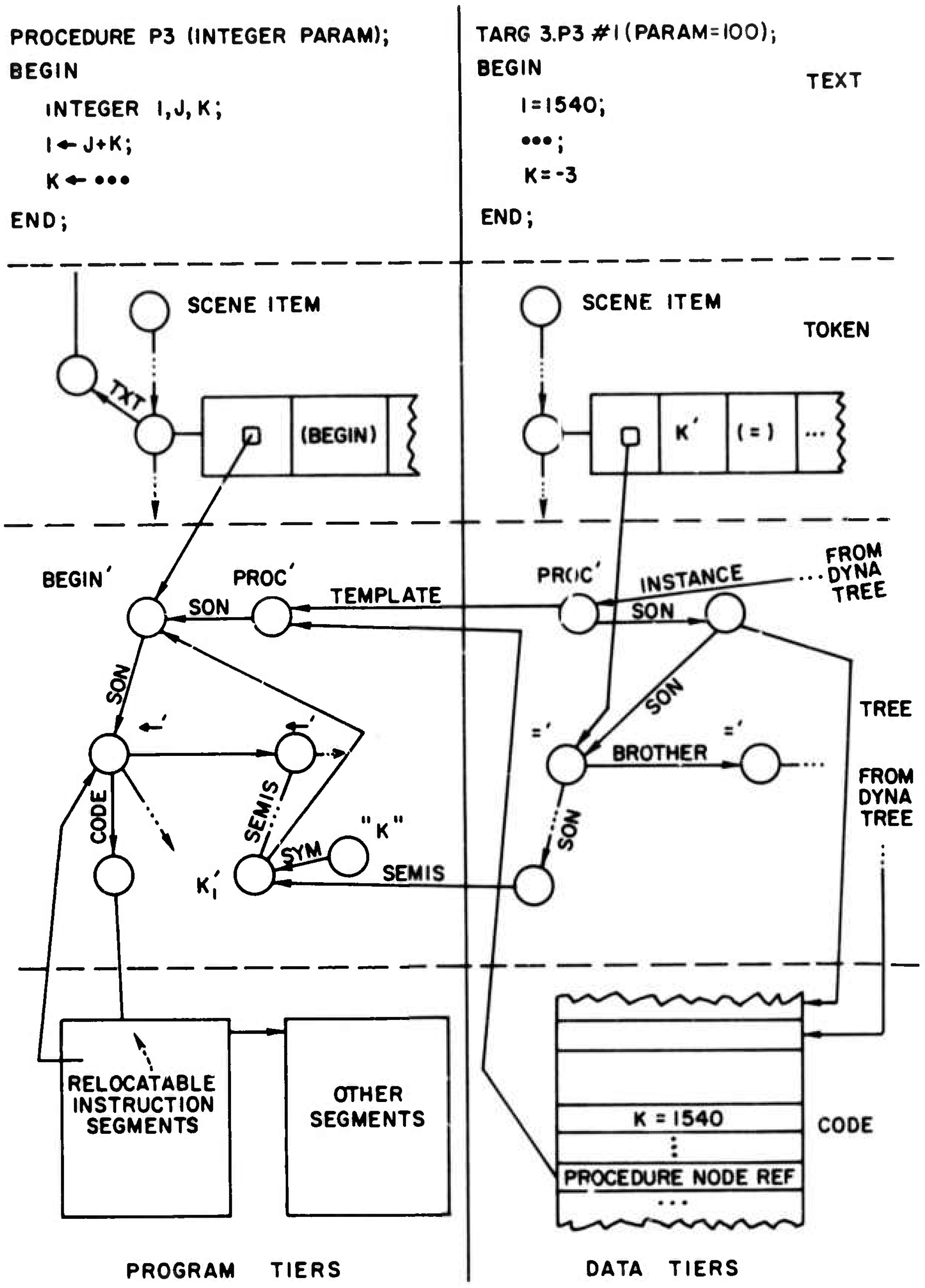


Figure 8-3. Overall View of COPILOT Tier Structures (part 1)

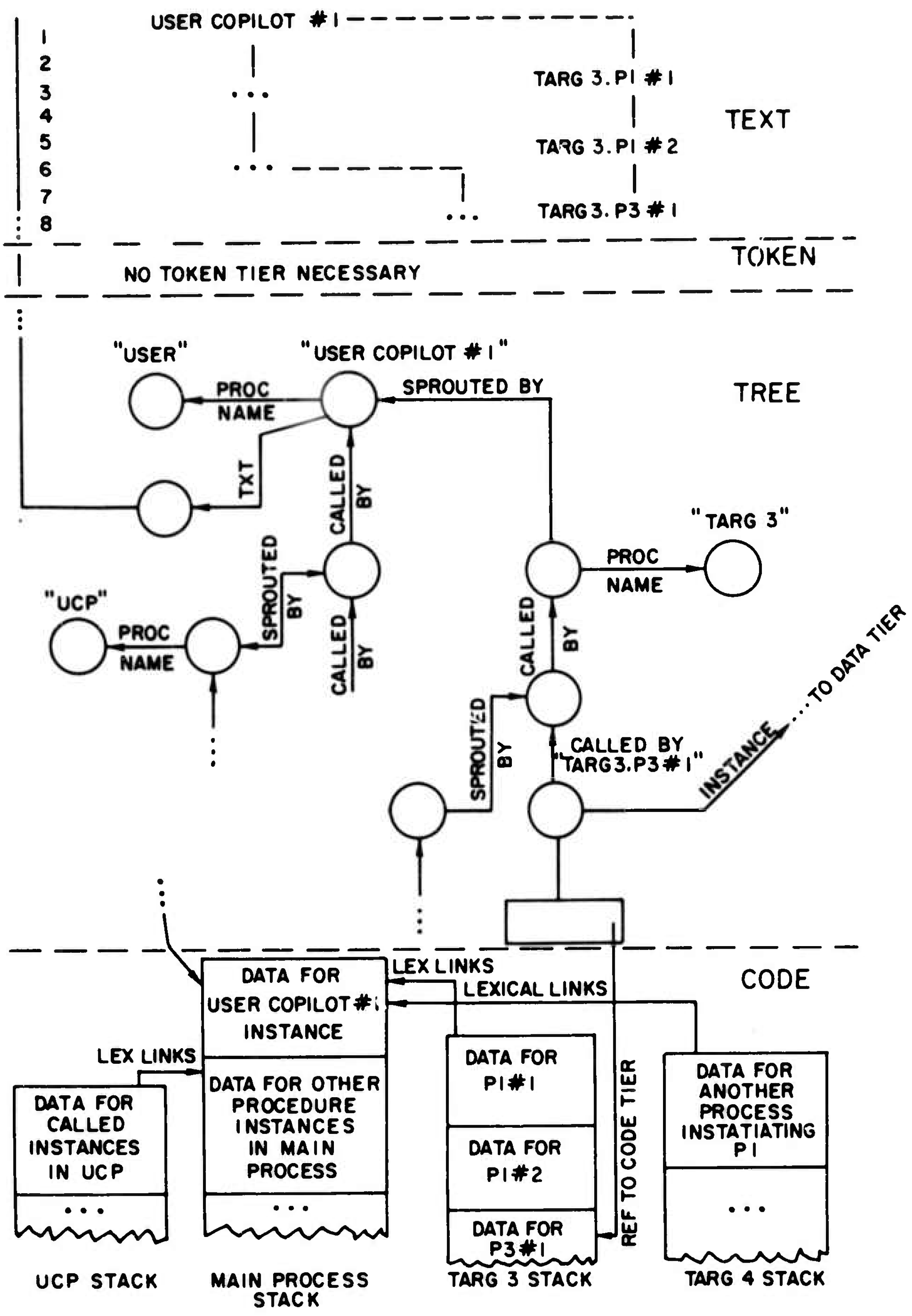


Figure 8-4. Overall View of COPILOT Tier Structures (part 2)

8.C1 Text Tier

We have exhaustively described this level. Its implementation is straightforward, providing for the storage, insertion, deletion, and replacement of lines of text. For convenience in implementing the user-level routines, these structures may be indexed by line and character number. Each line in the text Scene contains a reference to the LEAP item, in the token Tier, which represents that line.

8.C2 Token Tier

The output of a language processor's lexical scanner is a sequence of tokens, internal representations of the language symbols. In most languages, including MISLE, many program symbols are members of the relatively small set of terminal symbols, and the rest are identifiers and constants chosen from a relatively small number of declaration instances. Therefore, by proper encoding, an expression of the program in terms of these tokens may be smaller than its text representation, depending on the implementation and the user's identifier naming style. Its chief advantages, however, are the increased parsing speed when sections of the text must be recompiled, and the additional structure which can be maintained in token lists (see Section 8.A). For these reasons we have chosen the token Tier as the permanent Tier for programs. To do this, we must achieve strong equivalence by adding format information, chiefly to specify where spacing characters were present in the original. Figure 8-5 exhibits a section of the token Tier for the accompanying program. The Scene at this level is a two-way threaded chain of line items, each of whose datums is the token list and spacing information describing the line. Linked to each line item is an index into the text Scene for that line.

We have taken advantage of the discreet nature of token lists to insert connections to the tree Tier, so that statement nodes may be located (by GET_STRUCT(...), for instance). These frsmrk items (see Figure 8-5) are distinguishable from token entries. Their datums contain indices to aid (along with FIRST and LAST links) the inverse tree-to-token conversions.

We will introduce some additional Token structures in Section 8.E2, when we discuss the storage and parsing of program modifications.

```
FROM TOKEN --→ 1 BEGIN
FROM TOKEN --→ 2 INTEGER i, j;   (LEVEL  1)
               3  j←3, i←j•3+5;
               4  BEGIN
               5  INTEGER i, k;     (LEVEL 1a)
               6  k← j-5; i←k•fn(j)+7;
               7  IF i<j+1 THEN... ELSE...
               8  END;
FROM TOKEN--→ 9 j←i+2
FROM TOKEN--→ 10 END;
```

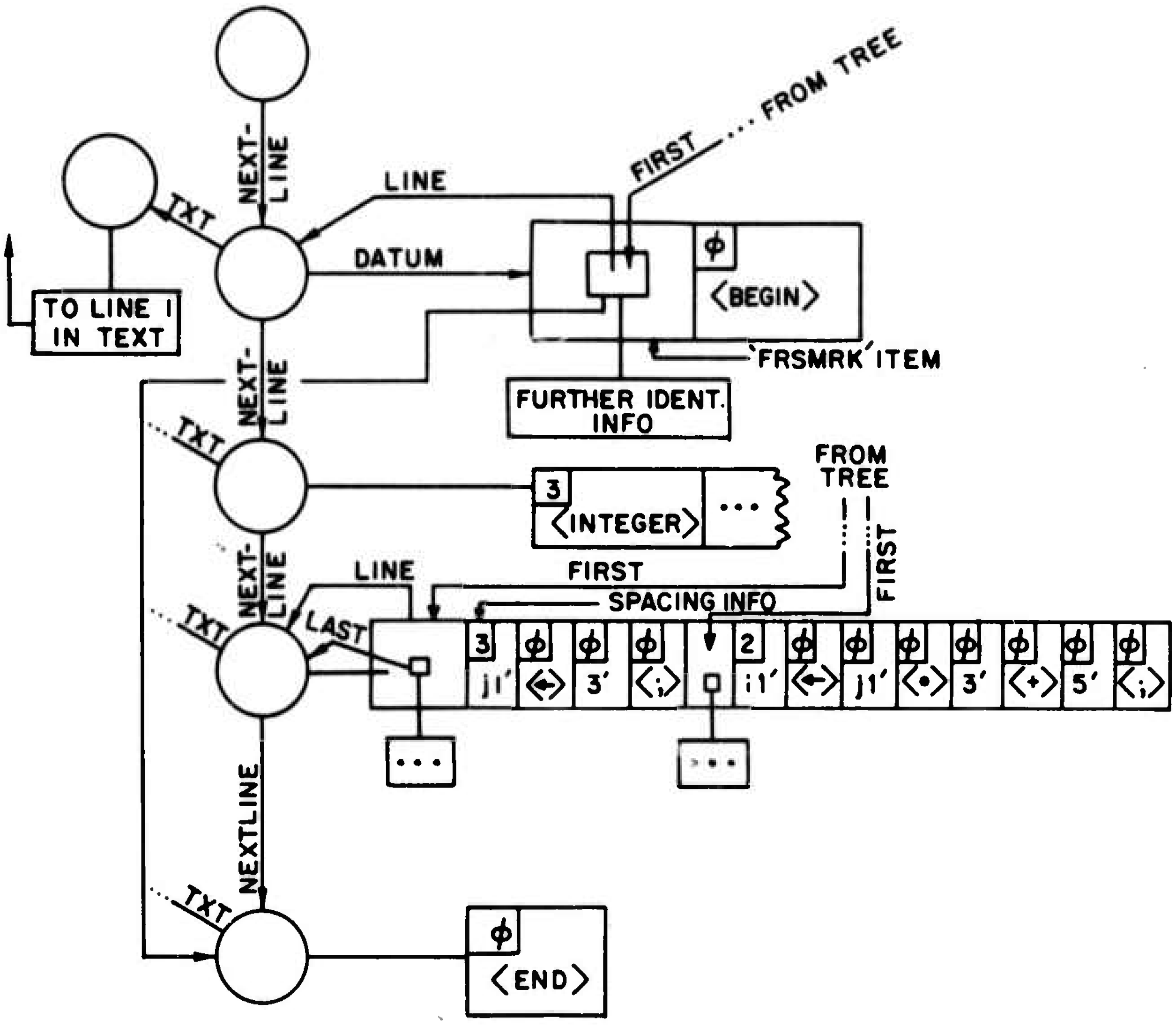


Figure 8-5. COPILOT Program Text and Token Tiers

8.C3 Tree Tier

This is the central data structure of the IPS. Program trees are the product of parsing operations ([36], [44]). Other tree Tier structures (data, status, dynamic) are derived from code-level information. The program tree of Figure 8-6 represents a fragment of the programs of the previous figures. It implements n-ary trees, where n is sometimes fixed ("IF <be> THEN <s> ELSE <s>", n=3), sometimes variable ("BEGIN <s>; ... <s> END", n=n). The trees are connected by leftmost-son, next-brother linkages [31]. The tree is pruned, after compilation, to include only the statement structure and lists of identifiers and function calls which appear in each statement. Although this limits the amount of resolution we can achieve in program control and in recompilation to statement units, it does not seem to us a great problem in view of the gain in compactness, especially for long, complex operations.

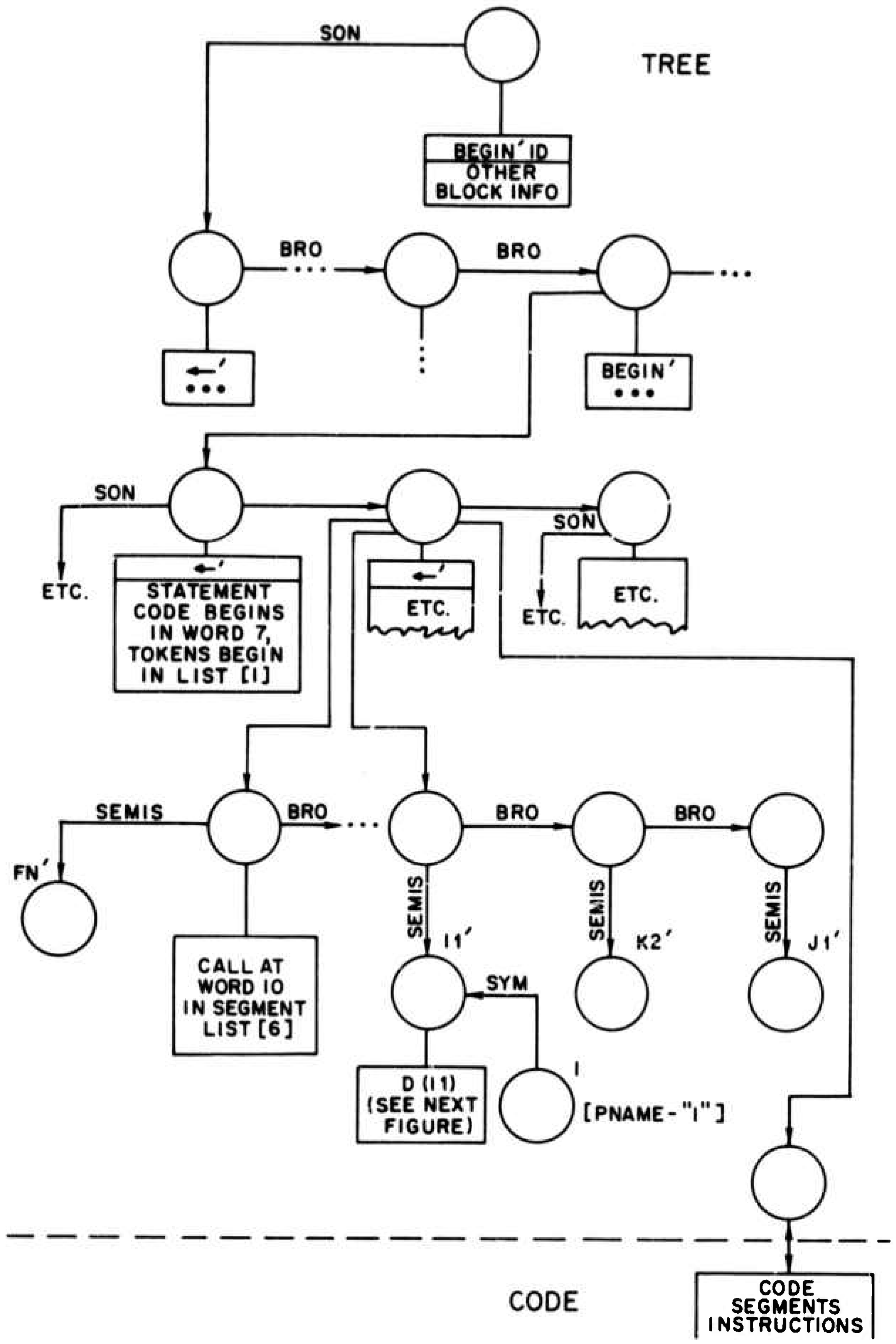


Figure 8-6. COPILOT Program Tree Tier

8.C4 The Symbol Table

We have placed the symbol table (†) in the tree Tier, because of its close ties to the program trees. Links from symbol entries to the block and procedure nodes in this tree define the scope (range of access) of the instances of a given name.

Figure 8-7 contains a program tree, pruned of all but block structure detail. The symbol entries are accessible in a variety of ways:

1) As terminal program tree nodes. The compiler follows the SEMIS connections to these entries to generate access or calling sequences for data and procedures.

2) By their point of declaration. Any local variable or formal parameter can be reached from the node for the block or procedure containing its declaration. The same links, followed backward, allow identification of the scope of a given entity.

3) Symbolically. There is a unique name item in COPILOT for each identifier name. Linked to it by SYM links are all the entities (symbol items) with that name. In most of the applications we have described, environmental information (in the form of block or procedure nodes) is then used to choose the correct entity for the current scope.

Symbol table quantities, though all their connections are in the tree Tier, are really multi-Tier entities. Identifiers in the token Tier lists are actually symbol items. Additionally, symbol items appear in generated code, to identify procedures on the stack, and to select variables for display.

(†) The use of "table" is historical, since our actual structures are hardly tabular.

```
BEGIN              (LEVEL 1)
  INTEGER i, j;
  BEGIN            (LEVEL 1a)
    INTEGER i, k;
    ...
  END;
  BEGIN            (LEVEL 1b)
    INTEGER i,m;
    ...
  END
END;
```

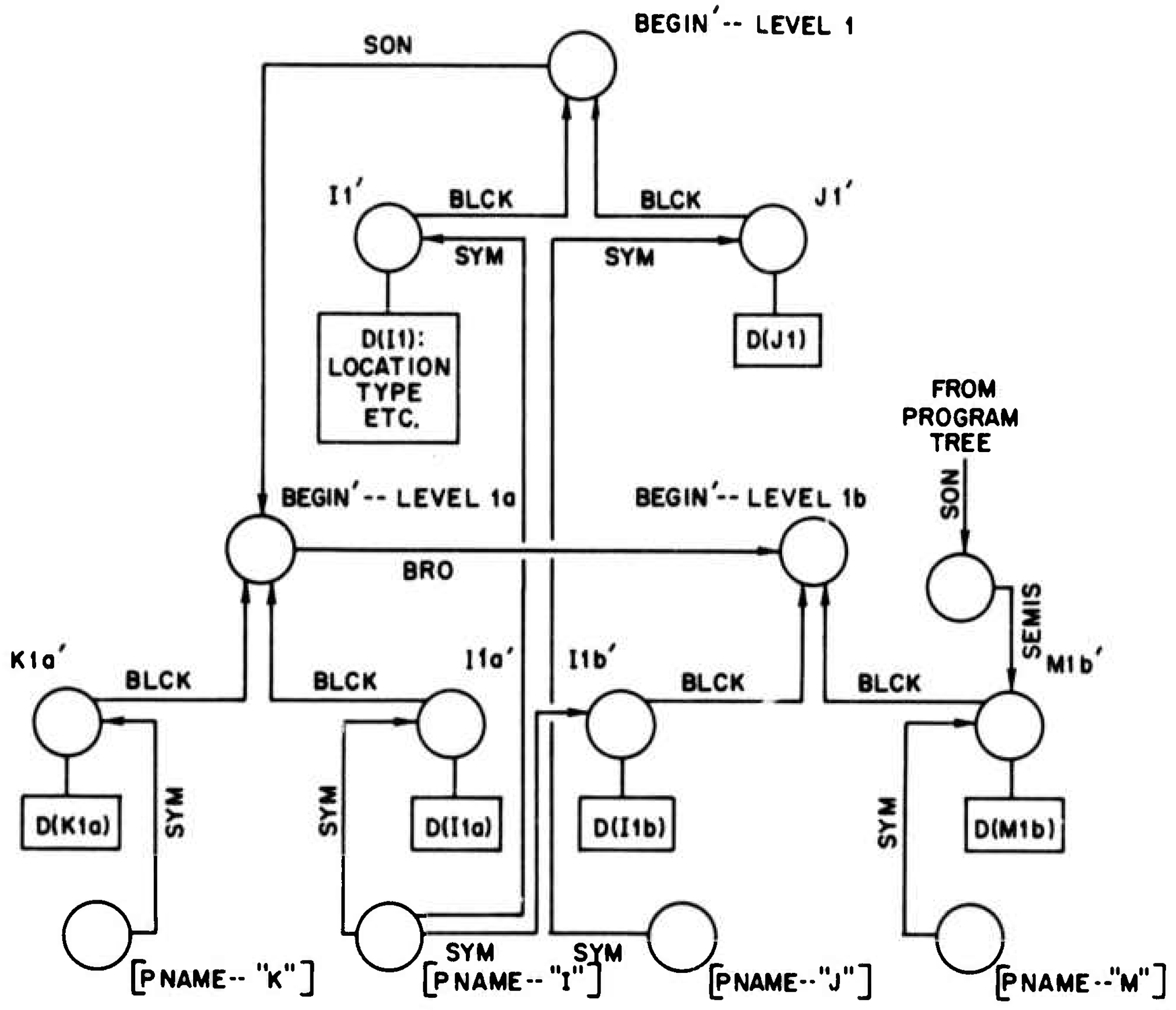


Figure 8-7. COPILOT Symbol Table Organization

8.C5 Other Trees

Figures 8-3 and 8-4 are examples of tree Tier structures underlying data and dynamic Scenes. There is one tree structure for the dynamic Scene, and one for each current data Scene. These trees are heavily connected to their "templates" in the program tree– algorithmic and symbolic information. This sharing of structure reduces the amount of tree Tier information which must be maintained for non-program Scenes.

In Chapter 7 we introduced entities called **structptrs**, produced by access primitives such as GET_IP and GET_EP, to provide compact representations for statements, data environments, Scenes, etc. In the COPILOT implementation, these structptrs are integer representations for the items forming tree nodes, token Tier entities, Scene and Region items. Extending MISLE to include the entire implementation language (SAIL) would eliminate this conversion, allowing structptrs to be directly represented as items.

8.C6 Code Tier

Since COPILOT is a compiler-based system, the most important (least dispensable) product of program translation is the set of machine instructions comprising the code Tier for programs. However, most aspects of code generation do not bear heavily on our IPS considerations. Consequently, we shall not discuss code generation techniques as such. (‡) We will be content to list the requirements and constraints which our generated code satisfies, in order to interface properly with the IPS and process facilities.

1. The code is organized as segments, built around the statement structure, which can be independently replaced. Major control points, labelled statements, procedures, and blocks always begin segments. Segments are limited in size, so that recompilation of still-correct statements in replaced segments will be acceptably infrequent. The compiler routines control the replacement, insertion and deletion of code, always in segment units.

- -

(‡) We might suggest Gries's book, **Compiler Construction for Digital Computers** [23], as an excellent reference for all aspects of compilation.

2. Code segments are relocatable. A segment can be moved in order to compact storage, or to accommodate the expansion of other storage blocks. We have chosen to make all but a small number of header instructions in each segment address-independent. The header words contain transfer instructions which link each segment to the segments which precede and follow it in the execution sequence, and to the segments which implement its substatements. The base address of a running segment is available in a machine register to allow relative transfers of control within the segment. Other registers provide data access. Transfers to other segments from within a segment are performed by transferring to instructions in the segment header. When a segment is moved, only the header instructions in those segments which link to it must change. We can locate these other segments by referring to the tree Tier structures, which contain complete segment location information.

3. To allow program modifications, we can delete and insert arbitrary code Segments. Given our relocation facilities, this is not hard. The tree Tier contains a complete description of the segment structure of the Code Tier. After a new segment or set of segments has been created, after header instructions have been inserted to link them together, and after the segments to be deleted have been identified (see Section 8.E5), it is then easy to modify the relocation routines to treat the new Segments as relocated versions of the old ones.

4. We insert synchronization instructions in the code, to denote points where process-rescheduling interrupts may take effect: so-called "clean points". These synchronization instructions also provide a mechanism for controlling Stepp, Break and data display operations.

8.C7 Synchronization

We have chosen to use the statement as our grain of resolution for synchronization. This is evident in the primitives of Chapter 7, where control is available down to the statement level. (*)

Within the data for each process, we allocate a variable, which we call a synch cell, for each code segment which can operate in that process. A synch cell, normally zero, may be set by

(*) We may also gain control at procedure calls within a statement.

system functions to request suspension of code running in the corresponding segment; the value placed in the cell indicates the reason for suspension, and also identifies the statement(s) within the segment for which the synchronization request is intended. This latter value is necessary because we sometimes compile several statements into one segment.

The initial instruction of code for each MISLE statement implements a synch test, which tests the corresponding synch cell for a non-zero value. If the test fails, execution of the body of the statement continues.

The second instruction of each statement is a routine call, or synch trap. This call is executed when the synch test succeeds. An argument to the call is a structptr to the tree Tier node corresponding to the trapping statement.

The synch routine, called by the synch trap instruction, is a small procedure in the global environment of all COPILOT processes. If the synch cell value indicates that the trapping statement should actually trap (is not simply a "segment-mate" of the intended statement), the synch routine collects: the current process structptr (from a global variable), a structptr to the tree Tier node which identifies the procedure which trapped (from the current activation record, see Figure 8-8), the statement node structptr provided in the call, and the value of the synch variable. It then causes an event, whose value contains the collected information. The event is either a keyboard event, if the process is becoming inactive to allow the User loop to run, or a posting event, if the deactivation is due to a Stepp, Break, Suspend, Terminate, or data display request. Having caused the synchronization event, the process may suspend, depending on the reason for the trap. The following paragraphs treat each trapping reason in more detail.

1) When the user types a character which the User process needs to react to, the resulting machine interrupt triggers a small procedure in the global environment. This interrupt procedure sets the synch cell for the next statement to be executed in the RUNNING process; the current-segment machine register which allows intra-segment control transfers also allows this routine to find the right segment. The interrupt procedure then releases the interrupt, allowing the program to run to the next synch test, which must trap. The synch routine causes a keyboard event, but does not suspend the trapping process, which therefore goes from RUNNING to READY, in deference to the higher-priority User process. The User process awakens at Readaline (see Section 6.B1), where it had been waiting for a keyboard event.

2) The Stepp function operates by setting the synch cells for all possible successors to the chosen statement (its immediate successor, as well as the successors of all its substatements, if they leave the range of the chosen statement). It then activates the selected process. When that process traps at one of the successor statements, the trap routine causes a posting event and suspends the stepped process, which will not run again until some other process restarts it.

3) A Break statement contains only the synch test and synch trap instructions. The synch cell for a segment containing a Break statement is always set, for the process selected by the argument to Break (all processes, if that argument is -1). Otherwise, a broken process behaves as a stepped one.

4) Whenever a RUNNING or READY process is suspended by a Suspend or Terminate call, the process-suspension primitive causes a posting event. The subsequent behavior is quite similar to that for Stepp. The only difference is the reason code in the synch cell.

We could eliminate the overhead of the synch test and trap operations by employing code-replacement techniques. We could temporarily replace the first instructions of a selected statement with a synch trap, then simulate their behavior when the process next ran. The trap instruction would be removed, and the originals replaced, when the trap condition no longer obtained. We are wary, however, of any technique which requires modification of the compiled code (†) for its operation, and have avoided it here. In Chapter 9 we will consider the extent to which specialized hardware can improve synchronization operations, eliminating the in-line instructions without code modification techniques.

Figure 8-8 demonstrates the general structure of the code Tier. Code Items, whose datums are code segments, form the interface between the program code and program tree Tiers. Additional information in the datum of each statement node locates that statement within its (first) Segment. This figure also sketches the storage organization for process data in the code Tier. Each process uses a stack array for storage of its activation records (frames) and temporary values. Each activation record contains a procedure node referent, and links to static and dynamic ancestors. This structure is dictated more by the requirements of the language than those of the IPS.

Although we have drawn them as the lowest Tiers of a multi-Tiered structure, in reality *all* the data in the system, implementing *all* Tiers, reside in, or are accessed through, references

(†) Except, of course, in response to changes in the source text.

in activation records of system processes within the code Tier. It is the special nature of this data, possessing references to information outside the normal lexical scope of the code possessing it, which allows us to circumvent control and environmental scope rules, in controlled fashion, to perform our complex IPS functions.

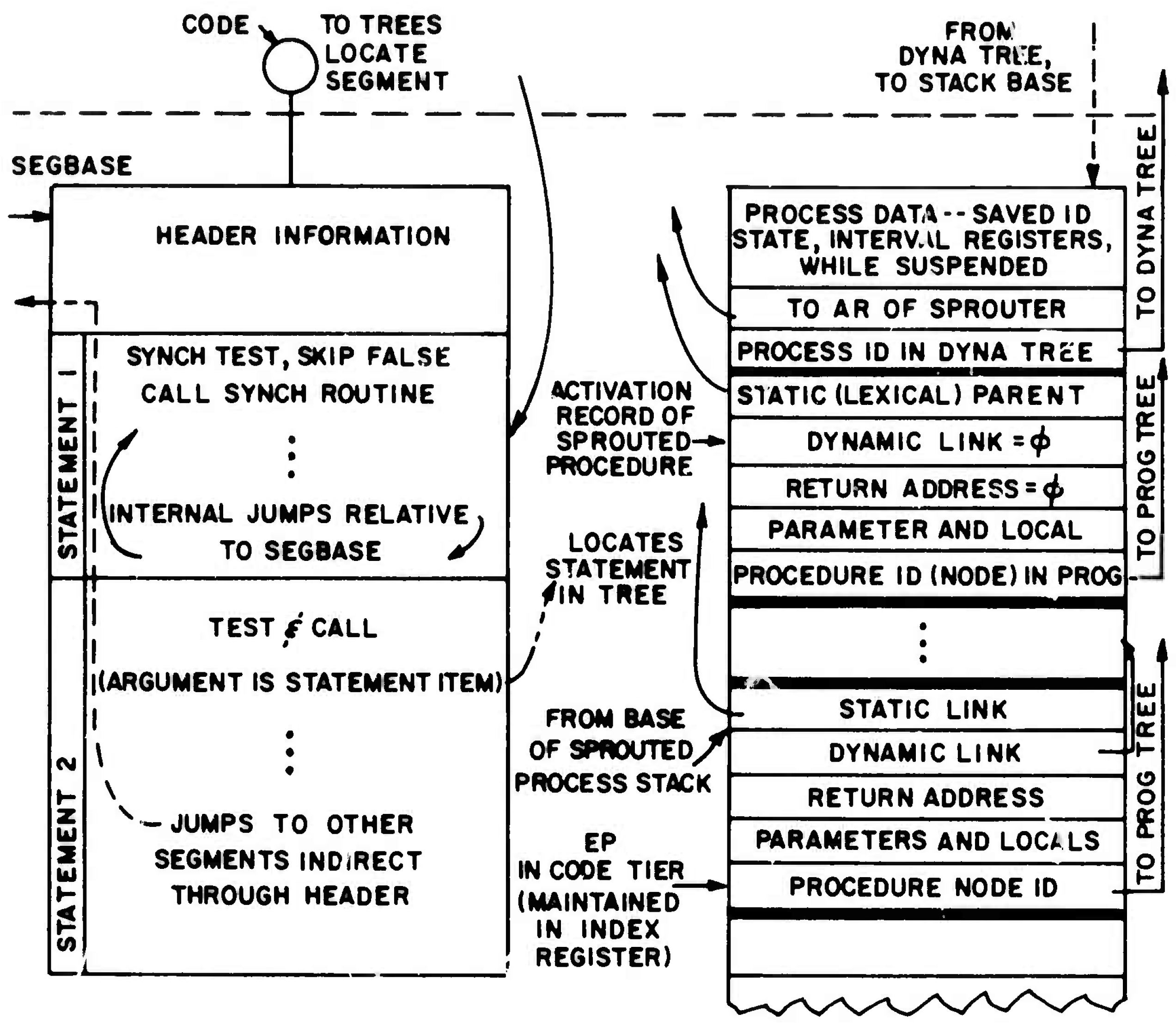

CODE
TO TREES LOCATE SEGMENT
FROM DYNA TREE, TO STACK BASE
SEGBASE
HEADER INFORMATION
STATEMENT 1
SYNCH TEST, SKIP FALSE
CALL SYNCH ROUTINE
INTERNAL JUMPS RELATIVE TO SEGBASE
STATEMENT 2
TEST & CALL
(ARGUMENT IS STATEMENT ITEM)
JUMPS TO OTHER SEGMENTS INDIRECT THROUGH HEADER
ACTIVATION RECORD OF SPROUTED PROCEDURE
LOCATES STATEMENT IN TREE
FROM BASE OF SPROUTED PROCESS STACK
EP IN CODE TIER (MAINTAINED IN INDEX REGISTER)
PROCESS DATA -- SAVED ID STATE, INTERVAL REGISTERS, WHILE SUSPENDED
TO AR OF SPROUTER
PROCESS ID IN DYNA TREE
STATIC (LEXICAL) PARENT
DYNAMIC LINK = φ
RETURN ADDRESS = φ
PARAMETER AND LOCAL
PROCEDURE ID (NODE) IN PROG
STATIC LINK
DYNAMIC LINK
RETURN ADDRESS
PARAMETERS AND LOCALS
PROCEDURE NODE ID
TO DYNA TREE
TO PROG TREE
TO PROG TREE

Figure 8-8. COPILOT Program and Data Code Tiers

8.D. SELECTIVE EFFICIENCY

When IPS facilities are not active, we would like our target processes to run nearly as fast, and occupy nearly as little space, as they would if :he interactive facilities did not exist. This requirement discourages extensive interaction between our compiled code and other Tiers, either to maintain them or to gain information from them. Our description of the COPILOT code Tier has reflected this paucity of code-to-tree connections: references in the code Tier are restricted to tree node structptrs, in synch trap calling sequences, and in the activation records for procedures; the compiled code makes no use of them except in the synch routine sequences described in the previous section.

We also have reason to minimize text Tier information:

1) All text Scenes are subject to the same set of text-oriented editing operations. An abundance of structural connections to the more specific underlying Tiers could interfere with the implementation of these commands.

2) Displaying text is often a costly operation. In most display systems, such simple activities as moving the Scene window, or inserting a line require the regeneration and transfer of large amounts of information. The complexity of the Text Scenes could adversely affect this cost.

The Token and Text Tiers, therefore, must provide the inter-Tier connections missing from the other two.

$$\begin{tikzcd}
\text{Text} \arrow[d, shift right] \\
\text{Token} \arrow[u, shift right] \arrow[d, leftrightarrow] \\
\text{Tree} \arrow[d, shift right] \\
\text{Code} \arrow[u, shift right]
\end{tikzcd}$$

Figure 8-9. Selective Connectivity

We pay for the selective efficiency we have gained in our outer Tiers with a corresponding loss in the inner ones, and in the operations which use and maintain these inner Tiers. Perhaps the greatest price is the increased difficulty of maintaining equivalence between the Tiers. As a running program modifies its environment (its data and control components), information in corresponding sections of higher Tiers becomes incorrect. When a snapshot is finally taken, updating these Tiers costs much more than constant maintenance would have cost. We present our maintenance methods below, in Section 8.E. (‡)

The interconnections between entities within the same Tier are also sparse for the outer Tiers. No more links are maintained in the code Tier than are needed to support the operation of the code. There is but one link per line in the text Tier. Any outer Tier operation which requires additional structure can find the corresponding tree node, follow appropriate tree Tier links to the desired structure, then return to the corresponding point in the original Tier. No power is lost; again, we sacrifice only time efficiency.

8.D1 Space Efficiency

Figure 8-4, exhibiting the "cactus stack" nature of MISLE processes, is a logical diagram of the structure of the computer memory while COPILOT is running. A contiguous data stack is allocated for each process, then linkages are created to establish the connections needed for normal references to lexically available names, in the stack of the sprouting process. Additional references (not shown) in the stacks for IPS processes provide the structured references to elements in *all* stacks which are needed for the Tier implementations we have presented. Program code segments possess a similar logical organization, although this is simplified because there is but one instance of the code for each procedure.

While a target process runs, only its code, its stack and those of processes in its lexical scope, the structures accessed through these stacks, and the global system routines need be present in memory until that process next suspends. We could accomplish this isolation in COPILOT by maintaining physical, as well as logical, separation of the code Tier segments from other elements of the system. Although the current implementation does not do this, we have designed all our structures with this separation in mind. Nowhere do we depend on physical proximity, of any pair of Tiers, or of the code segments for any pair of processes.

(‡) This analysis would almost certainly be different for an interpretive system.

Fortunately, the hardware and operating systems of many modern computers provide facilities for memory management, sharing, and protection which make it easy for us to implement this isolation. Figure 8-10 depicts the way in which the COPILOT implementation might be achieved in the MULTICS [43] system. Figure 8-11 is a possible solution for TENEX [5], which runs on a modified PDP-10. Both provide, through their memory management policies, the complete withdrawal of recently unused pages of information to inexpensive secondary storage, enhancing the Target process's performance. (Both figures assume a familiarity with the memory management facilities of these systems).

For systems whose memory structures are less sophisticated, the isolation properties can be simulated using either of these designs as a guide.

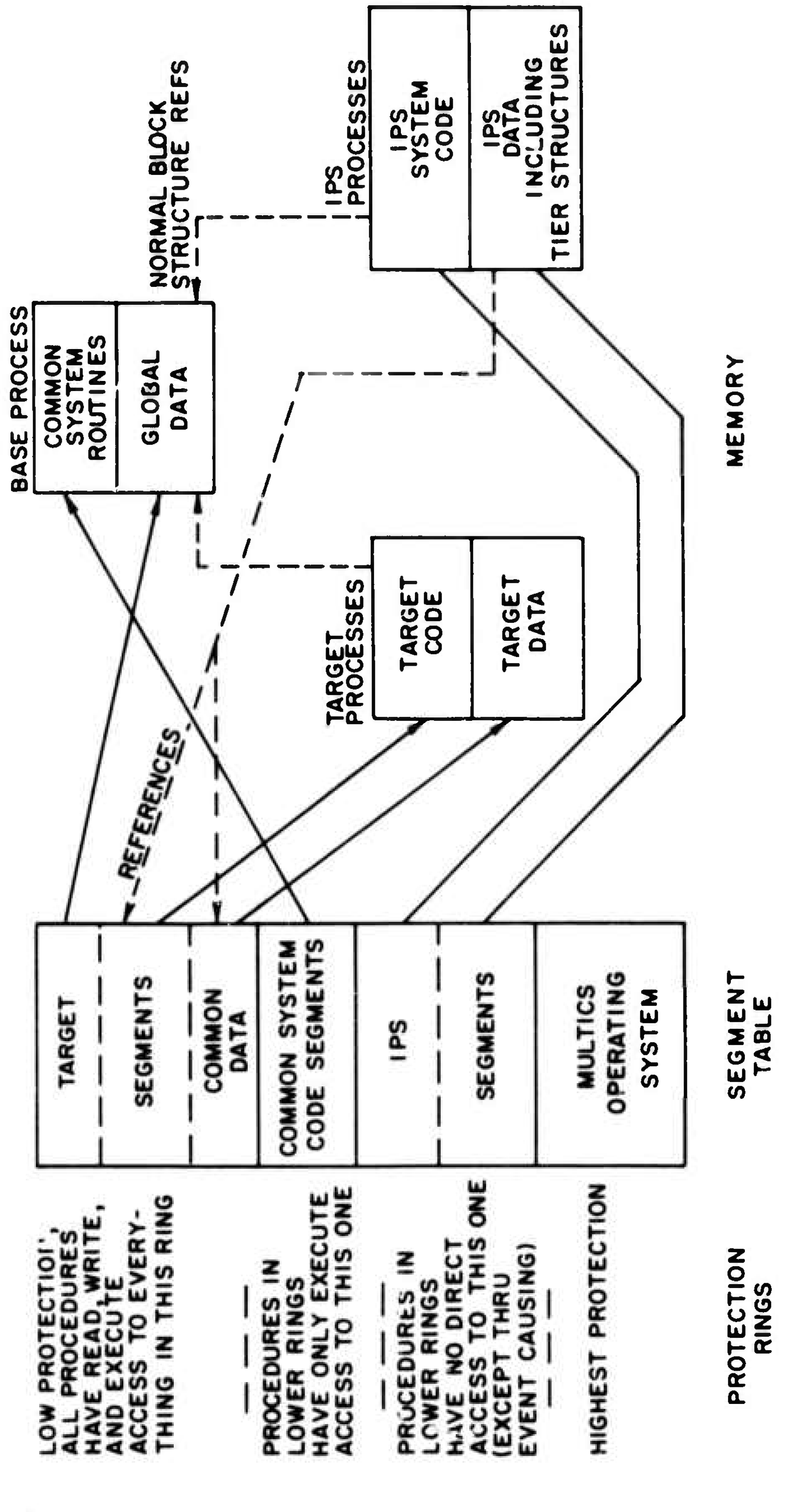

NORMAL BLOCK STRUCTURE REFS
IPS PROCESSES
IPS SYSTEM CODE
IPS DATA INCLUDING TIER STRUCTURES
BASE PROCESS
COMMON SYSTEM ROUTINES
GLOBAL DATA
TARGET PROCESSES
TARGET CODE
TARGET DATA
REFERENCES
MEMORY
TARGET
SEGMENTS
COMMON DATA
COMMON SYSTEM CODE SEGMENTS
IPS
SEGMENTS
MULTICS OPERATING SYSTEM
SEGMENT TABLE
LOW PROTECTION, ALL PROCEDURES HAVE READ, WRITE, AND EXECUTE ACCESS TO EVERYTHING IN THIS RING
PROCEDURES IN LOWER RINGS HAVE ONLY EXECUTE ACCESS TO THIS ONE
PROCEDURES IN LOWER RINGS HAVE NO DIRECT ACCESS TO THIS ONE (EXCEPT THRU EVENT CAUSING)
HIGHEST PROTECTION
PROTECTION RINGS

Figure 8-10. Proposed Memory Organization for COPILOT Implemented in MULTICS

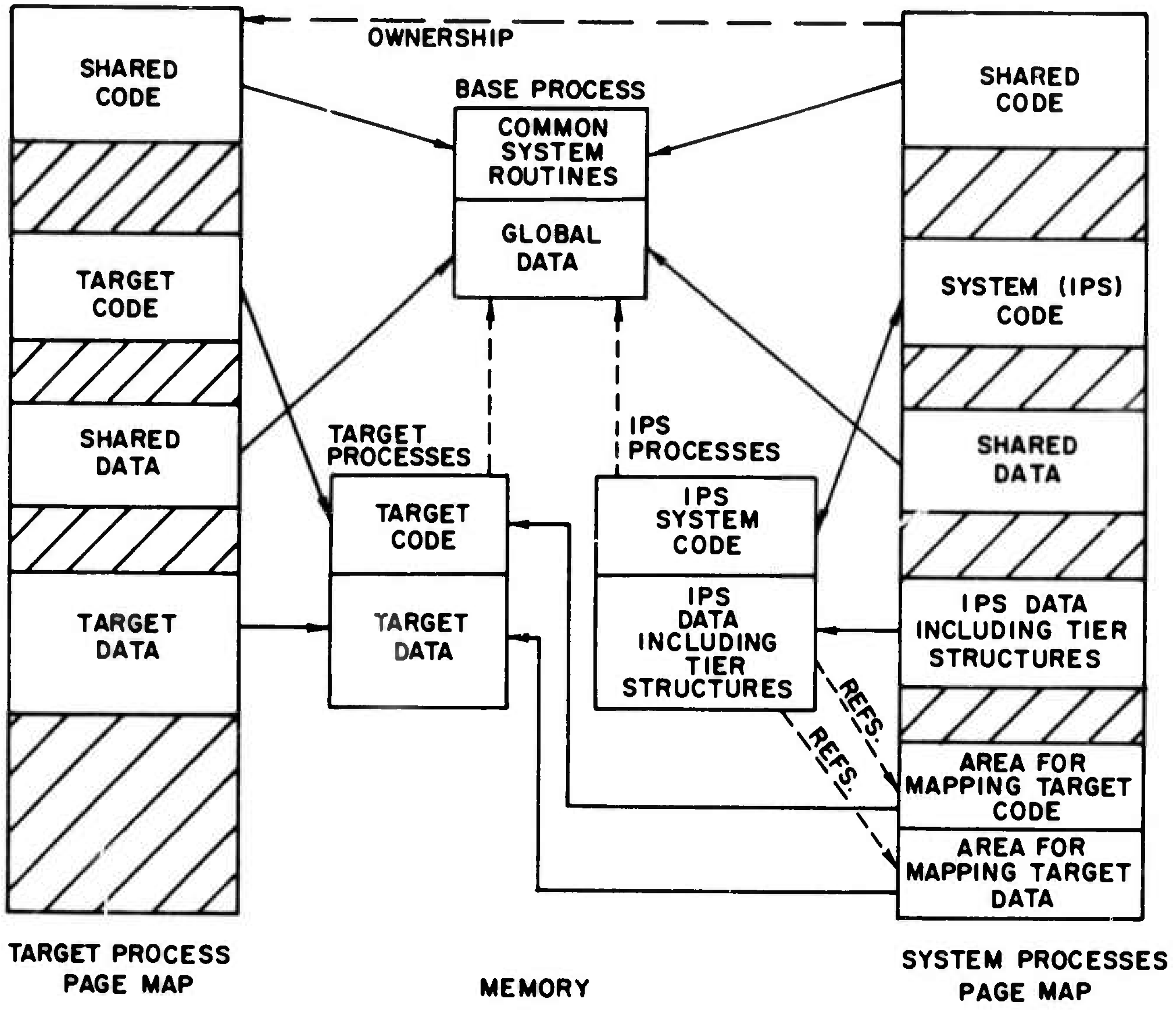


Figure 8-11. Proposed Memory Organization for COPILOT Implemented in TENEX

8.E. PARSING AND COMPILING

It is implicit in the COPILOT design that *incremental* changes to the text of programs must result in changes to the other Tiers. The translation, to be acceptably efficient, must minimize the replacement of information which is still correct; it must make corresponding *incremental* changes to the lower Tiers. The Visual Fidelity Principle (Section 8.A3) determines the maximum allowable delay between changes to text and the initiation of the corresponding translations. We could perform them more often, but do not, since by waiting we are often able to simplify the translation, and to make more changes at once.

We can not offer significant contributions to the incremental compilation area. We will, however, indicate the methods we have used in the COPILOT prototype, and hopefully reveal any insights we have gained in the process.

We consider Mitchell's thesis to be the most comprehensive on the subject of incremental compilation. For additional treatments see [35], [36], [37], [53], [50], [28], and [58].

8.E1 Parsing Methods

Lindstrom [35] defines an increment as a string of program elements (tokens) delimited by tokens from a distinguished increment set of terminal symbols (e.g., "Begin", "End", and ";"). He demonstrates that to limit parsing operations to the replacement of complete increments, rather than arbitrarily chosen strings, considerably reduces the complexity of an incremental parser.

We suspect that most parsing methods would survive the modification to incremental operation. Mitchell used a top-down approach, namely Tree-Meta [16]. Lindstrom, in a very promising approach to the subject, adapted the LR(k) algorithm of Knuth [30] for the purpose. We have chosen to use the variant of the Floyd-Evans production language parser (see references [22], [20], and [52]), which we developed for SAIL. Although we currently reparse and recompile only complete procedures, we believe that the flexibility of the production language technique would allow us to recreate a parse state which would accept less restrictive increments, and to merge the results into the old program trees. For the remainder of this section, we shall stipulate the existence of an adequate incremental parser and compiler, which can at least replace any sequence of complete statements, at the same block level.

8.E2 Detection of Increments

In order to identify which program increments need retranslation, we must keep track of text Scene changes as they occur. We must also relate these changes to the old program structure, for it is through study of the old structure that we can decide how to incorporate the new.

Figure 8-12 depicts an extension to our token Tier structures, which allows us to maintain the needed update records. By following OLDLINE links, or NEXTLINE links whenever OLDLINE links do not exist, we can recreate the original Scene. By following NEXTLINE links only, we obtain the current state of the Scene. There are no Token lists for new lines, since no token-scanning operations have yet been applied to them.

We will define a suspect procedure as one which will need to be processed by the parser and compiler before it is next executed, because it may contain invalid trees and/or invalid code. For each set of changed lines, we must mark as suspect the tree node for the procedure containing the lines. Because they may be invalidated by the change, we must also mark as suspect all subprocedures of a suspect procedure. We attach to each suspect procedure a set of references to the changed areas within its body. This algorithm guarantees that all procedures containing changes will be marked, and will therefore not escape the eventual attention of the parser.

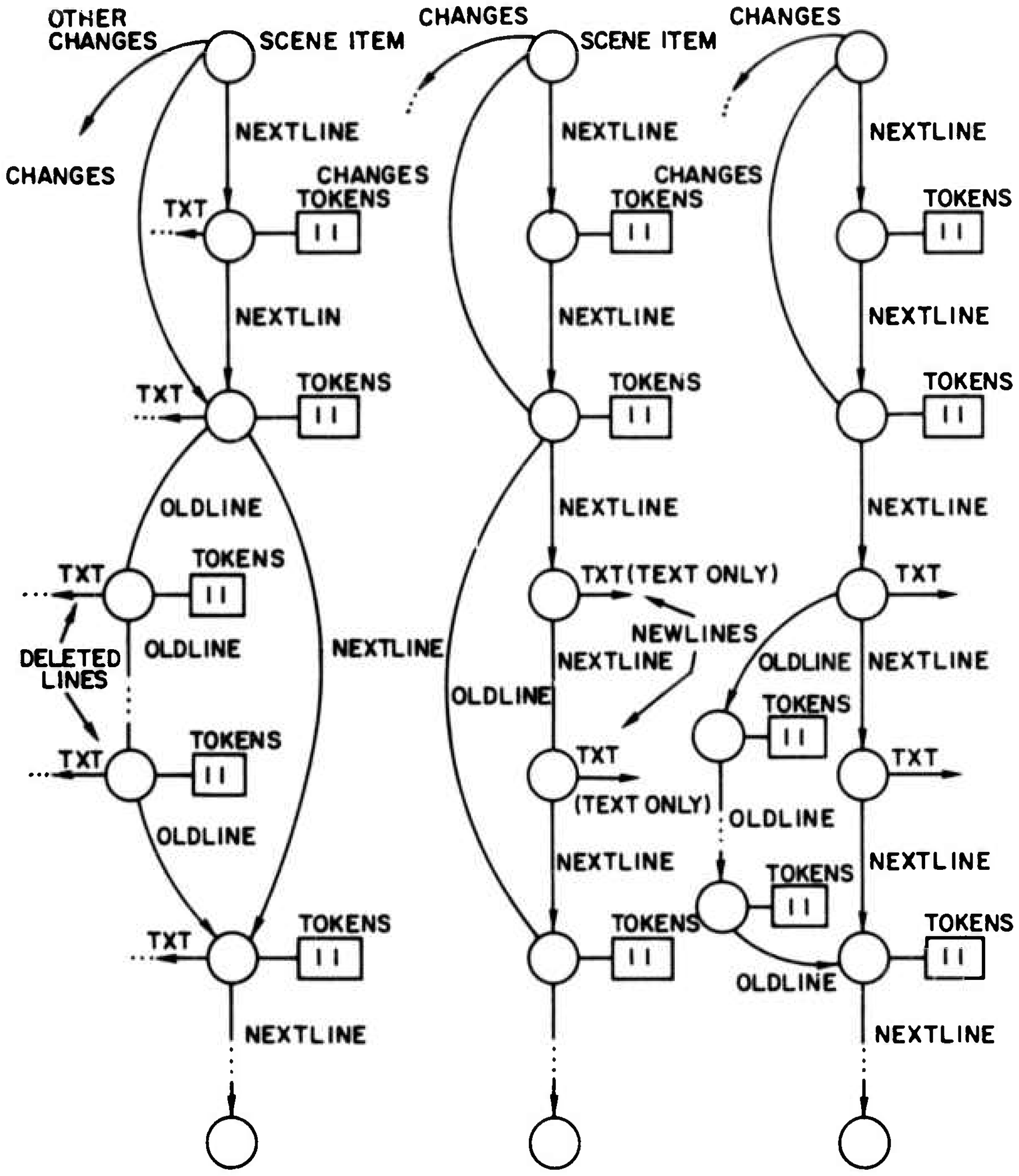


Figure 8-12. Additional Token Tier Structure to Record Source Changes

8.E3 Timing of Parse Events

There are possible advantages to parsing new Text Scene changes as they occur. Perhaps the most evident is that we could detect errors quickly, and notify the user of their nature. Continuous parsing would also allow us to prompt the user, continuously displaying the "menu" of legal successors to the last input (see, for instance, [24]). In addition, continuous parsing would make it easier to maintain tree equivalence.

There are also drawbacks to continuous parsing. The known methods for parsing incomplete program fragments either place undue restrictions on program composition (the relationship between line and statement boundaries, conventions concerning line numbers, etc.), or could not cope with the COPILOT compiled-code environment. We could also expect the continuous operation to be far less efficient, for not only must the routines for performing these translations be constantly active (incurring switching and "swapping" or "paging" overhead), but they must also maintain multiple parsing possibilities during the times when, because it is incomplete, the program is ambiguous. (*)

We would prefer to apply continuous parsing methods. Since the above problems are unsolved, however, we have not employed them. Instead, we delay parsing operations as long as possible, parsing only when not doing so would mean executing obsolete code. This allows the parser to expect that program changes are grammatically complete and correct when it parses them, or to be justified in seeking human aid if they are not.

Our methods for marking suspect procedures ensures that, if we operate the parser at the times specified in the following paragraphs, the system will never execute incorrect code, nor examine any incorrect data during structured editing operations.

The parser must be called:

1) Whenever a process activates (whether sprouted or resumed) if the procedure to be run is suspect. This includes any change in state to RUNNING, from SUSPENDED, STEPPED, ..., or READY.

2) Whenever a procedure is called, if it is suspect.

- -

(*) See, for instance, Lindstrom [35].

3) Whenever a structured editing operation is performed, if the procedure containing the indicated point is suspect (this covers operations which place the IP for a process into uncompiled regions).

Since the compilation operations can only occur when no other processes are RUNNING, case (1) above should be sufficient to guarantee that no incorrect code will be executed. However, by adding case (2), we do not have to compile all changes each time, but need only ensure that everything which can be reached without either suspending, or calling a procedure, is correct. Thus, for example, UCP changes, required for the User loop operation, can be compiled without attempting the translation of changes being made to other Scenes by these UCP statements.

Placing the compiler in the activation path between processes allows a simplification of the User algorithm. We may now remove the **Compil** step from line 3 of the program in Section 6.B3, and from the corresponding lines of subsequent examples, since the compiler will put things right during the **Stepp** activation on the following line.

8.E4 Process Structure

We have chosen to implement the parsing algorithm as an independent process, with access to all system structures. This gives it a particularly clean interface between the system and target processes. This process, the Parse process, is nested in the same system block which owns the Post and UCP processes. It runs at a priority between that of Post and UCP.

The Parse process, when not active, is suspended waiting for a parse event to occur. We can best consider what happens by considering the following cases:

1) The User process inserts a statement into the UCP Scene, then executes Stepp. Stepp, while activating the UCP at the new statement, causes a parse event. Because of the User process's priority, no further action occurs until it again suspends at **Readaline**. Then the Parse process activates, preparing the new statement for execution before it, in turn, suspends and allows the UCP process to run.

2) When a parse event is caused during a procedure call, or before a structured editing operation, the Parse process gains control immediately, due to its priority. Because of this event-driven operation, an instance of parser execution is invisible, except for a time-delay, to the normal control transfers between processes and procedures.

8.E5 The Parse Process

We will assume that, given a changed group of lines, our parser is capable of incrementally translating them (perhaps combining these changes with other nearby or related changes.) Here we will outline our procedure for applying this parser to a particular instance. The parameter to the parser is always the tree Tier node for a suspect procedure.

The parsing process:

1) Examines, starting with the given procedure, all static ancestors (father first), yielding the outermost suspect procedure. (†)

2) Determines, using the old tree and token lists, in conjunction with the modified lines, a range of tokens and text to reparse.

3) Performs the parsing operations in lexical order, so that declaration changes will occur before the statements affected by them are encountered. (‡)

4) Deletes old Token lists and linkages as they become useless. For each replaced or deleted tree node, the parser deletes the tree structures, subnode structures, and code Segments for it.

5) Invalidates (see [44]) those tree nodes whose code is now nonexistent or incorrect. The parser marks as invalid all new nodes, and all their lexical parents, terminating in each case with the outermost suspect procedure.

6) Handles changes to block structure or declarations. This demands special treatment, since the effects of these changes are distributed over a range of program statements which might otherwise remain correct. Mitchell's design offers clear solutions to the problems which arise from declaration changes. For each detected identifier deletion, insertion, or attribute change, we mark as invalid any program tree node which uses that identifier. We must also invalidate all the ancestors of an invalidated node, terminating at the block or procedure containing the innermost current declaration for the identifier. Although we use the associative facilities of SAIL to perform this search, its operation is analogous to searching Mitchell's dependency lists for the modified entities.

- -

(†) Since all subprocedures of a suspect procedure are also suspect, this determines the maximum range of current changes which could possibly impair correct operation of the code up to the next procedure call or process rescheduling operation.

(‡) This assumes that we require an identifier to be declared lexically ahead of its first use, even in a procedure nested within the same block. This is not a requirement of Algol 60. If we relax this restriclion, the parsing job becomes somewhat more complex.

The parser must sometimes decide to recompile statements whose code is still correct. For instance, since we only replace complete code segments, unchanged statements residing in the same segment with a modified statement must also be recompiled. Unfortunately, not enough information about a statement remains in the pruned tree to allow the compiler to generate new code (see Section 8.C3), so we must arrange to recreate the full tree by reparsing the token lists which specify it. This is not difficult, for we know both that the increment to be recompiled is syntactically correct, and that it already fits correctly into the surrounding structure.

8.E6 Compiling: When and How

We need not necessarily compile the incremental tree Tier changes as they are made, as long as the ultimate behavior satisfies the Visual Fidelity Principle. Previous systems have handled this in different ways. The simplest are those, like most LISPs, which can either interpret or compile their functions, accommodating both forms in the same program environment. In these systems, the user chooses the compilation time for each function; the only operational effect of compiling is to enhance speed and size characteristics.

Mitchell's system compiles code at the last possible moment, applying what he calls a **Tree Factored Interpreter** (TFI) to the tree structures (he parses changes immediately, on a line by line basis). A tree-structured interpreter, much like LISP's, is applied recursively to the program tree. Each program node inherits the code compiled for its subnodes, then has instructions of its own added to the inherited code. The code for a node is executed just after it is created. Thus interpretation is factored into a control component, which follows the tree structures, and an execution/interpretation component, which interprets the algorithm (by compiling and running machine code.)

In Mitchell's system, nodes are validated by compilation, and invalidated during parsing, using the methods described above (Section 8.E5). When the interpreter returns to a still valid node, it can execute the previously compiled code. This policy, recursively applied, means that only truly incorrect code need be replaced.

Mitchell's method requires IPS (interpreter) intervention at very frequent intervals, perhaps at every statement, even when executing correct code. We could perform last-minute compilation in COPILOT by using synchronization techniques, similar to those we have described (see Section 8.C7), to suspend in favor of compiler processes at the necessary intervals.

However, if each transfer of process control caused minimum code recompilation, we could expect an inordinate, probably unacceptable amount of process-switching to occur after even minor changes. We could partially avoid these problems by making some reasonable decisions each time about how much to compile.

At present, as we have seen, we restrict parsing events to times whose rarity would nullify any benefits of such selective and frequent compilation. In particular, we will seldom parse a change until just before the code it represents is scheduled for execution. We are therefore content to synchronize compilation with parsing events. After all changes have been parsed for the outermost suspect procedure (which will by the preceding constructions be invalid) we apply a TFI compilation algorithm, similar to Mitchell's, to the updated parse tree for that procedure, without executing the code segments we compile. (*)

A final compilation task is to insert the resultant code segments into the code tier, and to correctly link these segments to the surrounding code.

8.E7 Modifying Active Code

When the user (or any other agent) suspends operation of a process, then modifies the program in a way which affects code in any active procedure within that process, to maintain correct program behavior requires special treatment. The IP location might have to be repaired, for instance. If a procedure is changed so that it no longer calls some active procedure, or calls it from a different place, the return label needs to be modified. Modifications to declarations often require substantial changes to the data environment. Mitchell discusses this problem at length in his thesis. He presents an algorithm, called REVERT, which can restore a legal state whenever control transfers (by subroutine return, the only possible time) to a modified context.

We have not given this matter the same exhaustive analysis for COPILOT. In the

(*) Normally, Mitchell's TFI compiles only those nodes which it actually executes (for instance, it would compile only the selected alternative of an IF statement, leaving the other until it was selected.) He does provide modes, however, for compiling <u>all</u> nodes in such constructs, when desired. This is the algorithm we are using.

prototype, the COPILOT user occasionally has to help reëstablish a correct environment by direct editing of PROG, DATA, and DYNA Scenes. See Chapter 9 for thoughts on a more satisfactory facility.

8.E8 Compiling Temporary Statements

Temporary statements, when ON, are functionally indistinguishable from any other statements. When OFF, they are equivalent to null statements: they have no effect at all. Without the temporary statement facilities, the user could achieve most of the same effects by inserting conditional statements at selected program points. These statements would test variables used in place of our class identifiers, to determine whether or not to perform the operations.

We demonstrated in Section 7.D3 that the enhanced syntax for temporary statements constitutes a user convenience. It can benefit efficiency, as well. When a temporary statement is OFF, its code need not exist. The compiler can choose, while "in the vicinity", to delete any segments owned by inactive temporary statements. The expense of the recompilation required to turn such statements back ON is offset by the ability to leave potentially useful debugging or monitoring statements permanently in a program, without execution cost.

CHAPTER 9
SHORT SUBJECTS

In this concluding chapter, we wish to treat several topics:

1) Some facilities whose descriptions may be better understood in light of the implementation information of Chapter 8.
2) Unsolved problems, some with partial solutions. We have mentioned most of these in previous chapters.
3) Possible extensions to COPILOT, made possible by the basic design.

The topics to be discussed do not fall neatly into single categories of any of the attribute spectra we have discussed. They are therefore simply presented as separate discussions, with no significance attached to their order of appearance.

9.A. ADDITIONAL COPILOT SUBJECTS

9.A1 User Programs in the System Environment– Assistant Procedures

We have not mentioned this subject since Section 4.A3, when we briefly stated that the user could write assistant procedures to perform repetitive terminal operations in his stead, eliminating the need for a special "macro" facility.

We need neither additional structure nor additional commands to provide this ability. The system skeleton of Figure 6-1, in fact, has a provision for such programs. The •assistants entry in that example indicates where Scenes containing special user procedures can be placed; that Scene need not be called "assistants", nor is there a limit to one such Scene.

The global variables described in Chapter 7, which the terminal primitives use for their operations, form part of the environment of the assistant procedures. It is therefore possible, by construction, for an assistant procedure to do anything which the user can do in a single terminal operation. By combining several terminal primitives with normal language constructs– loops, conditionals, etc.– one can achieve much more complex actions.

Typically, the user will directly execute an assistant procedure, by typing, for example,

"!PROC(p1,..,pn)" The PROC call will execute in the UCP process environment. It should, like any UCP-executed statement, be written to complete quickly, or invoke another process if the operation might take longer than a few seconds.

The behavior of the system under sequential application of some primitives, particularly process-activation functions like Stepp, depends on the time intervals between successive calls, since an activated subprocess may or may not have suspended when called upon to do something else. Although this condition is present in the operation of the User-UCP processes, effecting the interpretation of user "type-ahead" (†), it is particularly troublesome in assistant procedures. We will discuss the problem further in Section 9.B2.

9.A2 Display of (unnamed) Expressions

We have heretofore considered the display, in DATA Scenes, of named entities only (e.g., variables). We would like to attach a meaning to the general data display statement which, in the syntax, allows us to select arbitrary expressions for display. Our current solution is to treat such expressions specially, adding the computed value to the current variable data Scene, for one snapshot only, using the name "<temp>" to identify it. Unless explicitly renewed, this entire entry disappears during generation of the next snapshot.

A better solution (but much more expensive) would require that we attach to the data tree node for an expression's equation a reference to that expression's node in the program tree. The expression would be re-evaluated, in the correct environment, during each snapshot update, until the proper environment no longer existed, or until the user explicitly deleted the equation from the Text Tier. In this case the text representation of the expression itself would be used to name it (e.g., "A+F(J)*4 - 50"), so that multiple simultaneous expressions could be maintained.

Another data display feature is provided as a convenience. The stepping (●X, ●S) operations are useless unless the user can see something of the results of executing a statement. He can, of course, explicitly select variables for display, but the necessity to do this can be irritating, particularly in those cases when what he probably wants is clear. A few situations are very clear: after stepping the execution of an assignment statement, one would like to know the

(†) The presentation of new commands faster than they are processed.

value of any affected variables; or when execution suspends, just prior to execution of a FOR loop's controlled statement, the value of the controlling variable is [illegible]arly always of interest.

The Post process spontaneously adds variables to the appropriate data Scene, whenever a process suspends after "stepping" a statement which changed but one variable. The variable is not, however, marked for continuous display. The effect is to display this variable during one snapshot only, unless data display statements have previously selected it.

We could extend this facility to more complex statements (e.g., complete blocks or compound statements), but we would, in each case, have to balance the added visual context this achieves against the danger of flooding the data Scene with too much information.

9.A3 Operations on the UCP Scene

Since the User and UCP Scenes are ordinary Scenes, they should submit to user modification through text-editing operations, particularly because such modifications could be quite useful, permitting the user to tailor his system. However, in practice, such operations could yield unpredictable results, some of which are detailed below. We must therefore place limits on what can be done, in order to protect the integrity of the system. We would also like to provide alternate facilities with equivalent power.

Any IPS which treats either the programs implementing the system or the history of user commands as user-accessible entries must tackle these same problems. Teitelman encountered some of them while implementing his BBN-Lisp facilities; he gives a lucid description of the results in [55]. We share with him the belief that many of these operations are useful enough that we should not prohibit them entirely. We have therefore introduced the following restrictions:

1) We will permit no direct changes to the programs implementing the User loop, the Post process, or the parser/compiler processes (but see Section 9.C3 for ways to obtain the same effect.) This preserves the integrity of the basic system operation.
2) We will permit no changes to the UCP Scene which alter the basic outer structure (that of a procedure whose body is a compound statement). In addition, we will allow no insertions of text into the UCP Scene below the current IP for that Scene, except by the User process.
3) Only the User process may control the activation of the UCP process, or the placement of its IP.

These latter requirements assure that the normal, sequential application of user commands will not be impeded by user modifications to the UCP.

Let us briefly consider what the user might want to achieve by direct operations on the UCP Scene. Perhaps most obvious, and most difficult to achieve in light of the above restrictions, is repetition of previous commands. A conceptually easy way to achieve this would be to map the UCP Scene to a visible Region, to point the edit cursor at a previous statement, and to single-step the execution of that statement. To re-execute a series of commands, one would surround a range of previous statements by BEGIN - END brackets, and step the execution of the resulting compound statement.

The problem with this technique is restriction (3) above: the IP-modification and STEPP operations implied by the above scenario are not allowed-- to allow them would destroy the integrity of our interactive control. The solution is quite simple: it costs little to create an additional process, which we might call UCP1, as another instance of the UCP procedure. As a separate process, UCP1 possesses an independent execution state (IP,EP); its operation will not interfere with the operation of the UCP process. Process control operations on UCP1 may be performed in a manner no different from the control of any other process. For convenience, we may devise explicit commands for the most common UCP1 transactions. An example would be a command whose effect is similar to Teitelman's redo operation (‡), repeating the action of a very recent statement.

The UCP Scene is a rich source of material for constructing assistant procedures, as well. Text-copying operations, which we have not shown, make this job easy. By embedding selected UCP statements within conditional and repetitive statements, a user can create quite sophisticated sequences. As an example, having constructed and executed a sequence of commands to test the performance of a new procedure, he could create an assistant procedure to perform the same sequence, for a range of parameter values, using the statements from the UCP to avoid reconstruction of the repeated text. One could, similarly, create another assistant procedure to perform a complex sequence of text editing operations, then apply it to a range of lines.

. . . .

(‡) see [53] or [55]

9.B PROBLEMS

9.B1 A UCP Scene Problem

If we use algorithm A, B, or C of Chapter 6, the UCP Scene will contain a complete record of recent terminal commands. However, algorithm D, which introduces the notion of selective interpretation, also introduces a potential problem. In algorithm D it is not always necessary to insert a statement string into the UCP in order to achieve that statement's effect; that statement may instead be executed directly. The UCP history will therefore be incomplete, rendering impossible automatic duplication of recent actions. A safe, although expensive, solution is to insert each expansion into the UCP Scene whether it is used or not. We could often increase efficiency by summarizing in the UCP Scene a sequence of actions (for instance, cursor-moving operations) by a smaller number of statements, appropriately parameterized. We do not have a more satisfactory solution to this problem.

9.B2 Type Ahead Problems

In Section 6.B3 we discovered a drawback to the decoupled control achieved in the User/UCP design: execution of one command will supersede that of a previous one if it is typed before that previous command completes operation. This behavior is necessary if we are to retain non-preëmptive control over errant UCP statements. However, it is not the only possible treatment of type-ahead. Our choices are:

1) To cause new statements to supersede old ones, as above.
2) To ignore statements completed while the UCP is active.
3) To queue new statements behind the executing ones, suspending the UCP only when none remain to run (the normal behavior of Stepp when applied to a process which is already stepping.)

We must immediately reject (2) as a solution, because it is completely unresponsive to the user's needs. When affairs are progressing normally, in fact, (3) is the proper course, performing all user commands in order. Finally, as we have stated, we need to be able to obtain the behavior of method (1).

No one has ever, to our knowledge, successfully resolved this conflict between the desire to be able to type ahead, and the desire to be able to abort previous operations. We can offer no complete solution here, but can at least offer a method which makes both the above acceptable methods explicitly possible. To do it, we have further modified the User algorithm, dividing the set of terminal commands into two classes. Each kind is expanded, inserted, and compiled as usual. When the UCP is inactive, both kinds behave identically. When it is active, however, there is a **type-ahead** situation. For elements of one class, we apply method (3), queueing the statements for eventual execution. We do it by bypassing the suspend statement of Algorithm C in Section 6.B3. Elements of the other we arrange to execute immediately, using method (1). We can now remove most of the commands of Chapter 7 from this latter, **immediate**, class and place them into the more orderly **queued** class.

Heretofore we have really needed only one command ("•!<statement string><cr>") to perform any terminal operation: all others could be defined in terms of this one. Let us place this command into the queued class. We now need a command which will execute a statement immediately: let us use "•?<statement string><cr>" for that one. To abort the current UCP operation, the user need type only "•?<cr>", which instantly terminates the current UCP statement in order to execute the null statement. To suspend all non-system processes immediately (a good idea in a crisis), one could type "•?SUSPALL()<cr>", or, ideally, a specially-designed **CALL** key which would expand to "SUSPALL()".

Having executed an immediate command, one could retry any or all of the interrupted statements using redo or something like it.

9.B3 Data Scene Flickering

We mentioned this problem before, in Section 5.H1. It does not arise when we are examining the state of a suspended process, either looking at previously selected values, or adding new ones via directly entered data display statements. However, consider a visible DATA Scene, D, which is monitoring a running process, P, where P's code contains several data display statements. It is possible that these statements are being executed often, generating a large number of snapshots in a short time. In this case, the value field for the equation of a displayed variable which is changing between each snapshot will become an unreadable blur (hopefully; otherwise the system is not fast enough). We are not concerned by this, though, for that blur is in itself useful information.

However, if these data display statements are scattered among several procedures, all defined within P and alternately called from within it, a far more serious "flicker" develops within D. It occurs because the variables for two disjoint procedures cannot appear simultaneously within D, even if both are simultaneously active. Even worse, if no action is taken to prevent it, the equations for each will alternately occupy the same positions within the Scene. If these transitions occur frequently enough, the result is not only chaotic, but uninterpretable.

If the user has enough display area, he can minimize this problem by creating several fixed Scenes (see Section 5.H1), thus distributing the equations to fixed positions. With the relaxation of the snapshot requirement which we described in that section, these Scenes should be fairly well behaved. However, we have no general solution to the problem, when it appears in the variable Scenes. Its effect could be reduced if the Post process were to apply heuristic guides to the placement of equations within variable Scenes.

9.B4 Data Monitoring

Data monitoring, or tracing, operations have always been a popular method of program debugging. For interactive systems, such a facility usually allows one to select a set of variables to monitor, specifying for each whether he is interested in every reference to it, or is only interested in store operations which change its value. The occurrence of such an event can cause the current value to be printed or displayed, can cause a "program break", or can invoke some user-specified action.

We may distinguish between facilities provided by translators (e.g., compilers), and those provided by the "virtual machine": the hardware and the IPS software. We have concentrated most of our efforts on the latter, assuming as well that we can control only the IPS software. The structure of the virtual machine determines the category into which continuous monitoring operations fall. If the hardware provides a way to interrupt when selected events occur to selected memory locations, or if the system is interpreter-based, monitoring may be handled as an IPS facility. Otherwise it is something which must be handled by the translator. In COPILOT, this would involve the recompilation of large amounts of code whenever the monitoring attribute changed for a given variable. We feel that, although we must accommodate explicit changes to identifier declarations, any other facility which requires for its operation such widespread replacement of program code is unacceptably inefficient, and should be avoided. The implementor willing to pay this price could do so, by treating monitoring as a declaration attribute, and by depending on the incremental compiler to find and replace the necessary references.

If our hardware possessed the ability to monitor individual variables, generating an interrupt or simulating a procedure call whenever one of them changed, our attitude would be much different. Monitoring would become an IPS facility, well within our domain. Our data display algorithm would respond readily to the needed modifications for displaying continously correct data values; and the process/event structures could provide more sophisticated monitoring operations, including the so-called "continuously evaluating expression" discussed by Kay [28] and Fisher [21].

The synch test and synch trap calls used to effect our process-control primitives are also translator-dependent, so our avoidance of such facilities is not completely consistent. In this case, since we always generate this synch code, massive changes need not be made to install and remove it on occasion. We have had to accept the additional overhead this method causes as unavoidable. Again, the addition of hardware memory facilities, which would generate the appropriate exception conditions when control passed to selected instructions, would virtually eliminate the synchronization overhead.

Several machines possess adequate memory monitoring facilities for a hardware implementation of these features.

9.B5 Restoration of Active Context

This is the problem of restoring the control and data environment of an active procedure, after its algorithm has been changed. We mention it here for completeness. We have already described the problem and our progress in this area in Section 8.E7: that solutions exist for similar systems, but we have not yet succeeded in applying them to COPILOT.

9.C. EXTENSIONS

9.C1 Environment Modification by DYNA Scene Editing

We have demonstrated the usefulness of structured pointing operations, applied to all the context Scenes, for selecting and communicating environmental information. We have similarly shown that one can modify this environment by suitable modifications to program and data Scenes. We would like to consider here what we could accomplish by allowing controlled modification to the dynamic Scene.

We would, as usual, limit the kinds of operations we would allow. Any changes which did not make sense would be repaired, perhaps by ignoring the changes. For this reason, the user would usually choose to "have the system make them", by calling specific primitives (e.g., Sprout), rather than use the general editing facilities, which would remain available for activities unanticipated by the designers.

We would provide a translator which would reflect, in the lower Tiers, controlled dynamic Scene changes of the following nature:

1) By deleting entries in the dynamic Scene one could "unwind the stack" of a process, perhaps returning the environment to an earlier state, or removing intermediate procedure instances (whatever that might mean).

2) By adding legal procedure instances, one could insert omitted procedure calls into the active environment, after correcting the omission in the code, or he could construct test environments (see also Section 5.D4). Default values would be assumed for variables in the new activation records, until explicitly overridden by additional user or program operations.

3) More importantly, by specifying that an entire process branch be copied, suitably renamed, and inserted into the dynamic tree, one could accomplish a sort of a posteriori process sprouting. Such a duplication could be useful when debugging, since it would implement what amounts to a checkpoint to which one could later return. One would run one of the duplicates for a while, then either terminate it and run the other (possibly modified) one, or terminate the second process if the first were successful. This is a facility similar to the one proposed by Lindstrom in [35].

4) Similarly (for symmetry) one could delete an entire process branch in the dyna tree, thus terminating the process. Directly terminating the process (using the Terminate primitive) would have the same effect.

9.C2 Scene Branching

In the COPILOT system as defined in chapters 5 through 7, there is but one copy, within each Tier, of the code for any given program segment; thus, the user need never perform redundant modifications, nor is there danger that changes will be left out of the "permanent" copy of the program text. (*) There is, however, a danger that he will make a change to the text which is difficult to reverse, especially during early development. Let us consider some of the kinds of things one would like to do during these early stages:

1) Try out proposed changes, without committing himself to them; try out several different versions of the same change.
2) Add new, independent program segments individually, eliminating any possibility of interference by other untested elements.
3) Merge several independent changes after each has been tested, resolving any conflicts between them.

We already possess the means for isolating a section of program for independent consideration: the nested program Scene. The user can accomplish something similar to the operations in the above list, using the existing facilities. By copying a Scene's data into a new Scene, making the necessary modifications, and replacing the *<scene> reference which includes it in the program, he can achieve the redundancy needed for all the above capabilities.

We could significantly increase the convenience and efficiency of these operations, however, if we were to extend the syntax for Scenes and Scene references to include something like Scene arrays, whose interpretation is shown in Figure 9-1. By editing SUPERSCN, or by executing a special command, the user could switch alternatives at will. The major benefits to this approach could be derived from proper implementation. For instance, all elements of one Scene array could share common token, tree, and code Tier representations (see 9-2 and 9-3) where possible, diverging only where they differed. The currently selected index (in the Scene link) would determine the accessible code segments for each divergent program increment. The data structures required for the other features would make the merger operations (item 3, above) quite simple. As a final example, because they would be inexpensive, one could retain several old "versions" of each Scene, for documentation or safety purposes.

(*) The token Tier is the permanent representation of his programs. It is retained when the user is not "logged in", eliminating the need for separate "source files".

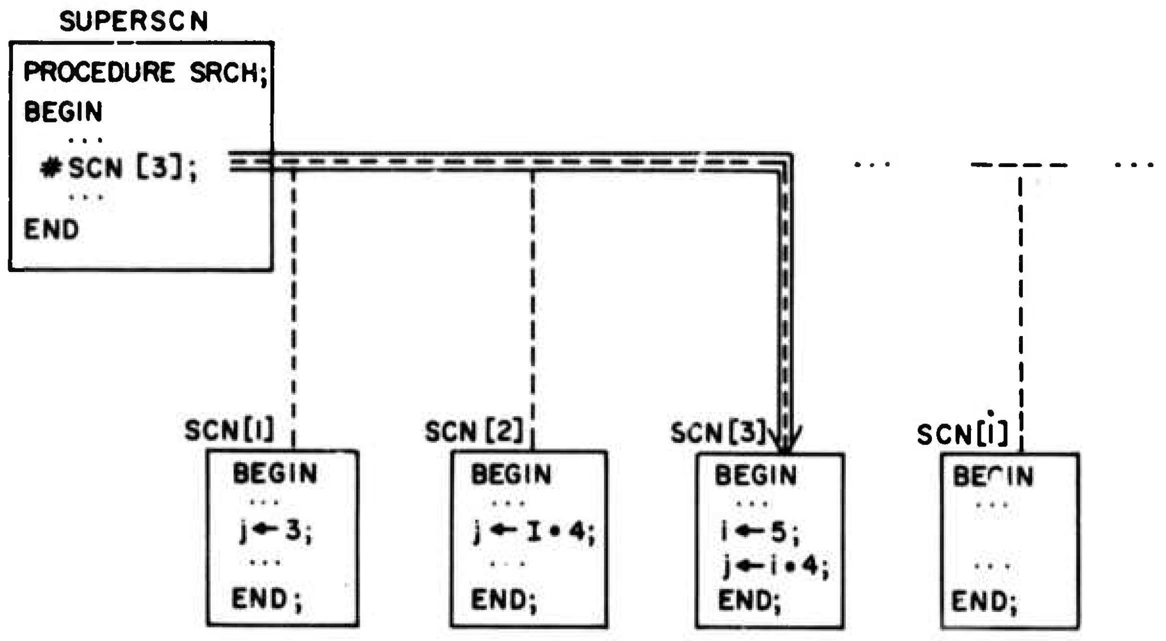


Figure 9-1. User's View of Scene Branching

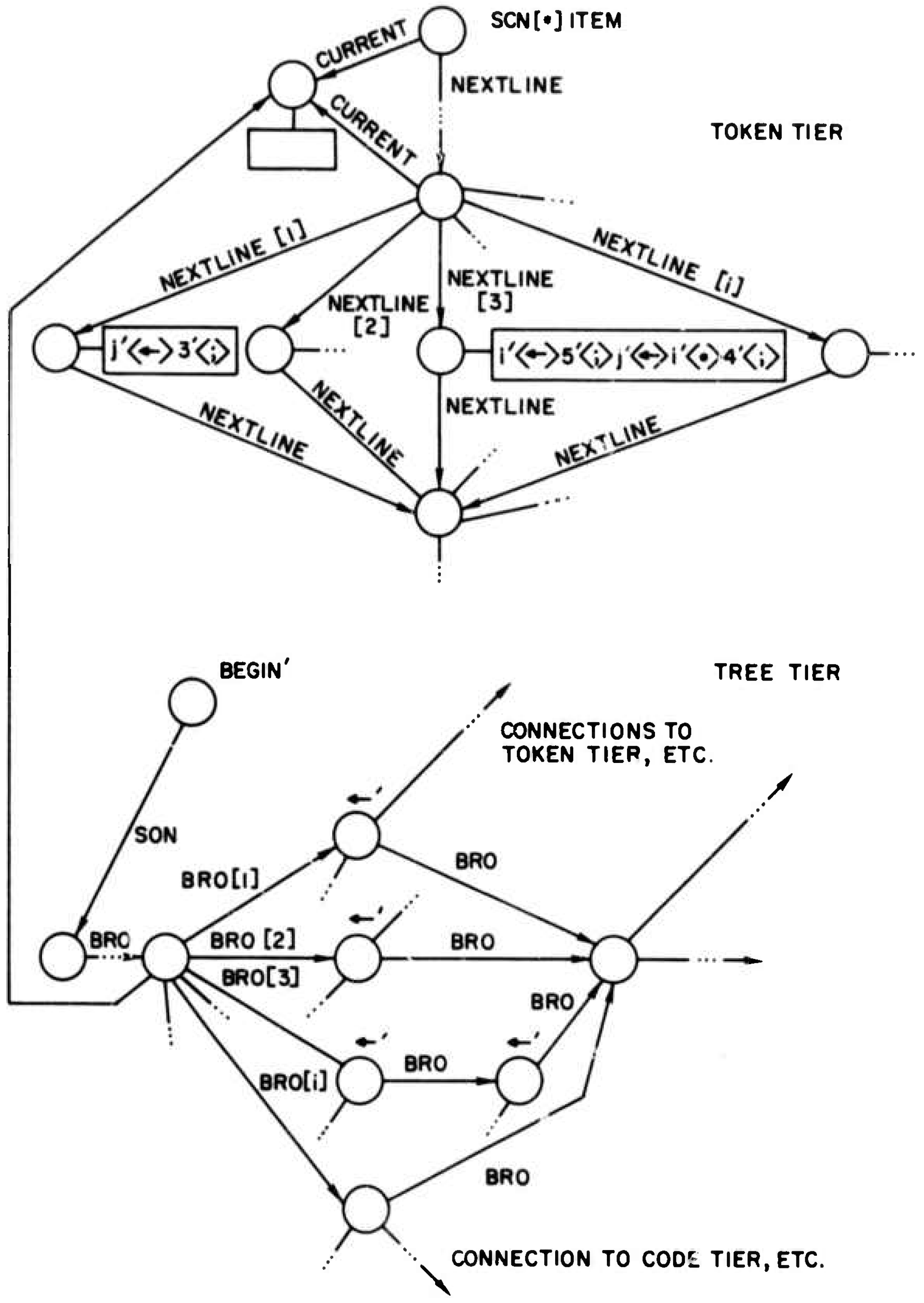


Figure 9-2. Efficient Scene Branching Implementation (Token, Tree)

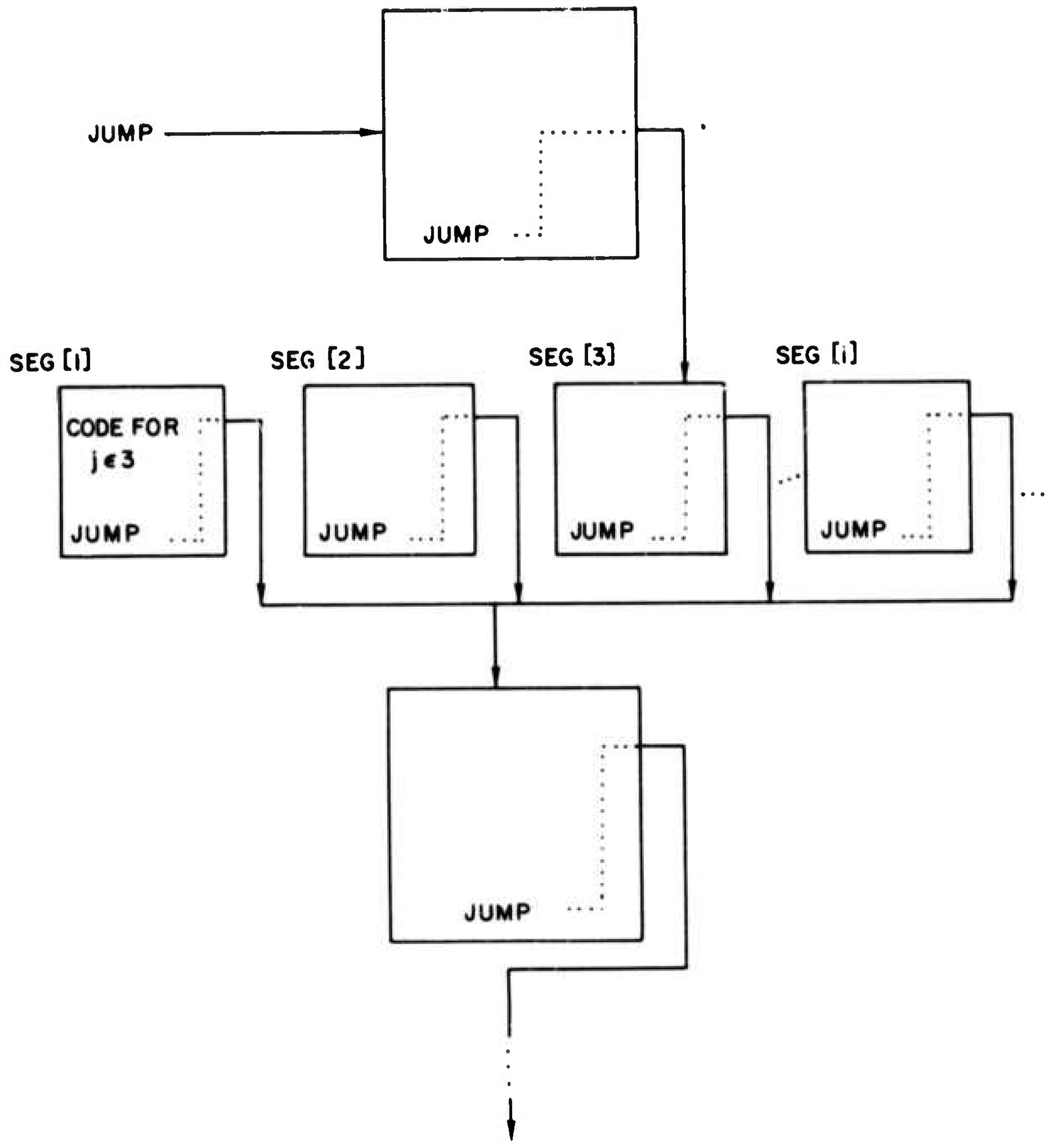


Figure 9-3. Efficient Scene Branching Implementation (Code)

9.C3 Modifying the User Loop

We have explicitly forbidden direct changes to the code implementing the User loop, and other critical system processes. We do not mean to prevent the user from designing his own; we simply want to ensure that the transition to a new algorithm is orderly and correct. We have already described an alternative for manipulations of the UCP, in Section 9.A3. The branching facility just described could be used to allow User loop modification. If the "•user" Scene link in Figure 6-1 were instead a link to an element of an array of User Scenes, the user could create a new element of this array, copy the old User algorithm to it, and make selected modifications. He would then call a special system primitive to switch from the old algorithm to the new, within the same User process, or, alternately, to create a new process and switch keyboard control to it.

We could also provide new primitives for customizing the User loop command structure by changing, adding, or deleting the expansion strings for selected command characters.

9.C4 Display of Structured Data

Current data Scenes can manage only scalar values. Thus, while it is possible to present single array elements in a data Scene, we cannot display an entire array, or selected rows and columns from an array. More complex structures (e.g., LEAP associations), are equally unmanageable in data Scenes.

We have already shown the benefits of a nested Scene structure for program Scenes. A similar approach could solve these data display problems. First, we would design a format for the particular kind of code Tier structure to be displayed. Then we would compose functions to create a text Scene, of a newly generated type, from the code Tier data for that structure— we might create intermediate Tiers as well.

Finally, we would add to the syntax for data language "programs" the productions:

```
<equation> ::= • <scene id>
<data comment> ::= <comment>
```

We would also extend the data display statement syntax to include structure statements such as "A", where A is a three dimensional array, "A[3,*,*]", or, in SAIL, "SON⊗?≡?". The first would display the entire array, A. The second would show just the rows and columns of "layer 3" of A. The third statement would present all those item (†) pairs related to each other by the SON attribute (father/son pairs).

To satisfy one of these requests, the system would create the appropriate structured Scene, map it to a selected Region, and insert a "•scene_id" entry, referring to this new Scene, into a selected data Scene. We would include with the entry a "data comment", bearing the original data display statement, to allow the user to identify the reference. Figure 9-4 is an example of this design for the partial array A[3,*,*]. Examples of possible display formats for SAIL associative structures abound in the figures of Chapter 8, for example, Figure 8-6.

We could extend this method to any of the basic, explicit structures of MISLE, of SAIL, or of virtually any programming language. There is, however, a limit to the comprehensiveness we could provide this way. A user, when developing data structures for a specific use, must use the provisions of the language to create them. The result need not resemble very closely the structure *as he visualizes it*. This is true even for extensible languages, such as ECL, Lisp70, or Algol68, in which the user tells the system a great deal about the structural hierarchies he creates— although we might expect to do a good deal better in these cases. In the past, as now, the burden for creating any custom-tailored external representation for structures has been on the user himself. In the present COPILOT system our text Scene primitives can offer some aid, but a method is still needed for specifying the external representation of user-defined structures. Balzer has done some work in this area (see [2]), as has Hansen (see [24]). Yonke, at Utah, is engaged in a promising study which could provide the needed facilities.

- -

(†) See Appendix B for a description of the LEAP associative features.

A[3,*,*]

	[1,*]	[2,*]	[3,*]	[4,*]
[1]	15	0	0	0
[2]	12	15	0	0
[3]	-4	9	15	0
[4]	-456	-137	-10	15

Figure 9-4. Possible Scene for Displaying Array Sections

9.C5 Error Messages

We could use the non-preëmptive nature of COPILOT to take the sting out of error messages: translation and execution-time errors detected by the system, or user-detected errors. To see how, we need to consider the nature of errors in a multiple-process environment. The effect of an error, in general, is to place conditions on the further activities of some process, but not necessarily to prohibit them entirely. As an example, the detection of a syntactic or semantic error during program translation need not, fortunately, prohibit further modification to the Scene text, although it might, for a serious problem, prevent execution of the resultant compiled code.

In many cases, then, we can replace the notion of "error" with that of "incompletion". A translation process can maintain, in an appropriate Scene, a list of things which must be done in order to remove all the constraints that have been placed on a situation. In our compiling example, the parser and compiler could maintain in an error Scene a list of the program Scene locations which contain incomplete or incorrect code. Underlying error Scene Tiers could, as usual, provide structure, linking the error entries to the errant locales in the program tree. This list, besides telling the user what problems remained, could help the translator to interpret the meaning of new changes in these locations. The important thing about this technique is its potential kindness: it is non-preëmptive, and it could provide substantial aid to the person attempting to rectify the situation.

9.C6 Text Scene Monitoring

We have described the control mechanisms for most of the translators which convert one COPILOT Tier to another. We have omitted the one which builds the OLDLINE and NEXTLINE structures of Figure 8-12 in Section 8.E2, when PROG Scenes are modified. The current method is ad hoc, and not very interesting. We make special tests in the Scene modification routines, for selected Scene types, and take special action when they are found. While designing translators between other Tiers, we have discovered the efficacy of building these translators as processes which monitor changes in their respective source Tiers. These processes awaken at convenient and adequate intervals to perform their specified translations. We subsequently developed the following generalization, which could handle the program Scene maintenance case above, as well as other useful translations, some of which we will consider.

In each case, the goal would be to provide a translation algorithm which would maintain the equivalence, as defined in Section 8.A1, of two or more structures, in order to satisfy a requirement such as the Visual Fidelity Principle of Section 8.A3. Each translator would be defined as a process with access to the data for its input Tier, and access to a suitable destination Tier. Its frequency of operation and translation volume would depend on the conditions for invoking it. Each translator would specify these conditions by providing two quantities as attributes of the input Scene type. They would provide an activation predicate, which would determine the conditions for invoking the translation, and an event type to cause whenever the predicate succeeded. The Scene modification primitives (e.g., "change_char(...)") would evaluate the predicate for a Scene just after modifying the Scene. This predicate could choose to activate its translator process:

a) On insertion, deletion, or replacement of a character in the Scene.
b) On insertion, deletion, or replacement of a line in the Scene.
c) On insertion of a character at the end of the Scene.
d) On insertion of a line at the end of the Scene.

The activation predicate could also contain other Boolean terms, testing such attributes as the name of the process doing the modification, and perhaps relevant Scene attributes: type, mapped status, etc.

The translation process would then wait (monitor) for an event of the type specified for the Scene, activating as soon after one occurred as its priority would allow (usually immediately). It would perform its actions, then suspend, awaiting another event. One process might handle more than one event type.

We will try in the following paragraphs to clarify this design with several examples.

The parse and compile processes form our first example, since they already operate this way. As a second example, we could formalize the ad hoc operations which implement the Token Tier change structures for PROG and DATA Scenes by a simple process causing, say, a Token event whenever a type (b) or (a) change were made to a program or data Scene. (‡)

- -

(‡) Combined with the compiler processes which lurk about the process activation interfaces, the resulting system would resemble Kay's FLEX system design. Here the monitoring has a random-access character, whereas Kay's processes operate linearly on their inputs.

9.C7 Program Communication

Scene monitoring can also aid user-program communications. We can categorize the kinds of demands for data which programs make of their users into two general classes. The first includes initial parameters, file names, limits, modes of operation and the like, which the user provides to tailor the program for a particular "run". The second is information actually processed by the program, e.g., commands and requests, statements to be translated, or data points to be considered.

Our interactive facilities have already eliminated the need for a third kind of user input to programs: status and variable value requests, and many other debugging operations. We think we have substantially reduced the need for the first kind (initialization), as well. After all, the purpose of most such parameter requests is to set internal program variables to the values provided, or perhaps to retain default values when the user's response so indicates. Typically the user, in testing his program, will give the same responses again and again, an operation which becomes something of a ritual after a time. We can eliminate this sort of request in COPILOT, since the user can set these internal variables using direct assignments or function calls, often allowing his selections to remain intact during multiple calls to the tested program segment. His program can post, in a visible Text Scene, the names and meanings of variables which it expects the user to set, or can simply create and present an appropriate data Scene as an indication of what things he may want to change. That process can further refuse to proceed until the user has provided satisfactory values for everything.

There will still, however, be occasions for more traditional input to programs (predominantly the second kind above). In this case, the general monitoring facilities of the previous section yield a very nice solution. The user could, for instance, engage in the following kinds of dialogue with his program:

1) Synchronous operation. Suppose that process **P** would like to ask a series of questions of the user, assuming that he will answer each question before being asked the next. **P** could create, or gain access to, a text Scene to use for communication, then arrange to be activated whenever a line were inserted, by any process other than **P** itself, as the last line of that Scene (a type (d) modification). Upon each activation, **P** could insert another question into that Scene (or into another Scene, if desired), and await user response. With a slight modification, **P** could allow the speedy user to answer questions before they were asked, interlacing the questions later to create a readable record. For some applications, the questions might be simple one-character prompts, indicating completion of previous processing. Although this last mode resembles the preëmptive User input loops of COPILOT's predecessors, neither this nor any of the following schemes are preëmptive, for the user could choose at any time to do something other than provide data, requested or otherwise, to **P**'s input/output Scene.

2) Command completion. Several recent operating systems, among them, the project Genie system for the XDS940 [33], and the TENEX operating system for the PDP-10 [5], provide a facility for minimizing user input, while providing a quite readable result. To do it, a program monitors each input character. On user request in some cases, automatically in others, as soon as the value of the current input implies but one legal successor, the system automatically "types" it, yielding complete commands, file names, etc. Some systems insert additional "noise words" to make the result even more readable. By using the methods of the previous example, but changing to a single character (type c) activation condition, we could easily implement this kind of dialogue.

3) General translation. The conditions labelled (a) and (b) in Section 9.C6 are the same ones used by our idealized COPILOT to maintain program Tier equivalence. By using them, the user could provide his own continuous incremental translation facilities, thereby supporting his own language, or perhaps something less comprehensive.

9.C8 A Final Modification to the User Loop

The User Loop algorithms we presented in Section 6 demonstrated the capabilities which we wanted our User loop to provide, but did so without forming an integral part of the Scene and Tier structures which lend consistency and flexibility to most of our system design. We have partially investigated the possibility of applying our Scene monitoring techniques to the design of an improved top level interface.

We would begin by writing a new User process, whose only function would be to insert typed characters into a linear terminal Scene. It could optionally perform simple editing operations, allowing for deletion and replacement of incorrectly typed characters, etc., depending on the Scene monitoring frequency (see below). This terminal Scene would serve merely as an input buffer, and operations upon it would be limited. We would implement it as a Scene, so that the normal COPILOT operations could be used to view it, react to what happened to it, and change it.

We could now create a process, **Expand**, to monitor changes in the terminal Scene, and to translate them either into direct action, or into complete statements in the UCP Scene for execution. By alternating between activation frequencies (c) and (d) of Section 9.C6, the Expand process could allow the User process the simple editing capabilities, mentioned above, whenever single-character reaction was unnecessary.

The ultimate behavior of these processes would not be too different from those of Chapter 6, but the overall organization would become clearer, and potentially more powerful. In fact, some useful extensions almost suggest themselves:

We need not limit to one the number of processes monitoring a Scene. We could, therefore, add a **Prompting** process, at the user's option, to help the novice or infrequent user with his commands. The prompter could complete commands, as described in Section 9.C7, and insert directives into the terminal Scene, as a guide to the user's responses, or to point out potential mistakes.

The monitor process structure would also make multiple-language systems possible: The expansion and compiling processes could be replaced in a modular fashion, so that any aspect-- the terminal "language", or the underlying base language-- could be changed, without altering basic system behavior. (We do not mean to imply that this task would be easy).

We feel that the monitoring technique dominating the preceding sections, though requiring additional research, would help achieve a desirable system unity.

9.D. SUMMARY

We have presented the COPILOT system design in order to investigate certain aspects of Interactive Programming Systems in a multiple processing environment. Our major approaches have been:

1) The application of multiple processing techniques to the IPS facilities themselves, leading to a non-preëmptive terminal operation, with convenient access to all relevant environments, and rapid response to user commands, independent of the activity of his target processes.

2) The use of (CRT) display devices, to increase the speed with which the system and user may communicate, and to allow information to be presented "in context", improving the user's ability both to comprehend complex environments and to specify points of interest within them.

3) The expression of all user algorithms and terminal commands in terms of a single programming language, providing a consistent, powerful user interface, and reducing the number of modes which determine the meaning of user input. Top-level abbreviation facilities allow the most common operations to become manipulative, reflex actions, rather than symbolic commands.

In Chapters 5 through 7 we described the COPILOT system, which employ these methods to meet the criteria of Chapter 2 for achieving a better behavior match.

In Chapter 8 we discussed important implementation considerations: the content and structure of information used to represent the system environment at different levels (Tiers), and the methods for maintaining the necessary relationships (or equivalence) between Tiers.

Finally, in these closing sections, we have attempted to indicate possible implications of this work, especially the potential for extension, using our methods as a basis.

APPENDIX A
SYNTAX CONVENTIONS

This appendix defines the modified BNF syntactic forms used to describe the MISLE language and the Data layout in Chapter 5. It assumes a general knowledge of BNF, as defined in [46], for instance.

Nonterminal symbols are expressed as lower case words surrounded by "<" and ">", e.g., "<statement>".

Terminal symbols include punctuation: single characters or "diphthongs" defining themselves; reserved words: BEGIN, END, ELSE, etc.; and the special nonterminal-like symbols <id>, <string_constant>, <constant>, and <integer_constant>.

The character "'" causes the following character to be interpreted literally, if it would otherwise have special meaning.

Each rule, or production, is a nonterminal, followed by the definer "::=", then by one or more alternatives, separated by the "|" character. An alternative is a list of terminal and nonterminal symbols, or is an option or a repeat alternative.

An option, of the form [<alternative> | <al...> | ... | <al...>] requires that one of the alternatives be chosen. The repeat alternative takes the form { <alternative> }* , and means that instances of the alternative may appear zero or more times:

<c> ::= A { , B }* is the same as <c> ::= A | <c> , B

Expressed in its own language, this syntactic specification is:

Terminals: '['] '{ '}* '::= NONTERM TERM
where NONTERM and TERM represent nonterminals, as defined above.

```
<production>  ::= NONTERM '::= <alternative> { , <alternative> }*

<alternative> ::= <element> { <element> }*

<element>     ::= TERM | NONTERM | <option> | <repeat>

<option>      ::= '[ <alternative> { '| <alternative> }* ']

<repeat>      ::= '{ <alternative> '}*
```

APPENDIX B
ASSOCIATIVE FACILITIES (LEAP) OF THE SAIL LANGUAGE

We have represented many of our COPILOT structures in terms of the LEAP associative facilities embedded in SAIL. The structural diagrams of chapters 8 and 9 were presented in a consistent pictorial style, representing these LEAP structures.

We will first briefly describe SAIL's associative facilities. Following that we will provide a correspondence between the SAIL structures and our pictorial representations.

The LEAP description has been extracted from [19], with the permission of the other authors:

> SAIL contains an associative data system called LEAP which is used for symbolic computations. LEAP is a combination of syntax and runtime subroutines for handling items, sets of items and associations.
>
> **Items**
>
> An Item is similar to a LISP atom. Items may be declared or obtained during execution from a pool of items by using the function NEW. Items may be stored in variables (Itemvars), be members of sets, be elements of lists, or be associated together to form triples (associations) within the associative store.
>
> **Triples**
>
> Triples are ordered three_tuples of items, and may themselves be considered items and occur in subsequent associations. They are added to the associative store by executing MAKE statements. For example:
>
> ```
>             MAKE use ⊗ plan1 ≡ task1;
> ```
>
> The three item components of an association are refered to as the "attribute", the "object", and the "value" respectively. Associations may be removed from the store by using ERASE statements such as:

```
ERASE use ⊗ plan1 ≡ ANY;
```

Datums

Each item other than those representing associations may have a **Datum** which is a scalar or array of any SAIL data-type. The data-type of a DATUM may be checked during execution. DATUMs are used much as variables are. For example:

```
DATUM(it) ← 5;
```

would cause the datum of the item "it" to be replaced with "5".

Sets and Lists

A Set is an unordered collection of distinct items. Items may be inserted into set variables by "PUT" statements and removed from set variables by "REMOVE" statements. Set expressions may also be assigned to set variables. Set expressions including set constants, set functions, set union, subtraction and intersection are provided.

Sets are deficient in some applications because they are unordered. To remedy this, SAIL contains a data-type called "list". A **List** is a (user)-ordered sequence of items. An item may appear more than once within a list. List operations include inserting and removing specific items from a list variable by indexed PUT and REMOVE statements. List variables may also be assigned list expressions, including list constants, list functions, concatenation, and sublists.

Foreach Statements

The standard way of searching the LEAP associative store is the **Foreach Statement**. A Foreach Statement specifies a "binding list" of itemvars to be assigned values (bindings), an "associative context" specifying how the data structure is to be searched to provide these bindings, and a statement to be repeated for each set of binding values. Consider the following example:

```
FOREACH gp,p,c | parent • c ≡ p ∧ parent • p ≡ gp DO
        MAKE grandparent • c ≡ gp;
```

In this example the binding-list consists of the itemvars "gp", "p", "c". The associative context consists of two "elements", "parent • c ≡ p", and "parent • p ≡ gp". The statement to be iterated is the MAKE statement.

Initially all three itemvars are "unbound". That is, they are considered to have no item value. Since "p" and "c" are unbound, the element "parent • c ≡ p" represents an associative search. The LEAP interpreter is instructed to look for triples containing "parent" as their attribute. On finding such a triple, the interpreter assigns the object and value components to "c" and "p" respectively. We continue to the next element "parent • p ≡ gp". In this element there is only one unbound itemvar, "gp". "p" is not unbound even though it is in the binding list because it was bound by a preceding element. A search is made for triples with "parent" as their attribute and the current binding for "p" as their object. If such a triple is found, its value component is bound to "gp" and the MAKE statement is executed. After execution of the MAKE statement, the LEAP interpreter will "back up" and attempt to find another binding for "gp" and then execute the MAKE statement again. When the interpreter fails to find another binding, it backs up to the preceding element and trys to find other bindings for "p" and "c". Finally when all triples matching the pattern of the first element have been tried, the execution of the FOREACH statement is complete.

Thus, with a FOREACH statement, one can provide answers to the following kinds of questions (SON, HARRY, and GEORGE are already bound items):

SON ● HARRY ≡ GEORGE	Does this relationship exist?
SON ● HARRY ≡ ?	Who is (are) the son(s) of Harry?
SON ● ? ≡ GEORGE	Who is (are) the father(s) of George?
? ● HARRY≡ GEORGE	What is (are) the relationships?
SON ● ? ≡ ?	What are the father/son relationships
? ● GEORGE ≡ ?	etc.
? ● ? ≡ HARRY	etc. (these aren't too interesting)
? ● ? ≡ ?	Dump associative memory (illegal in SAIL.)

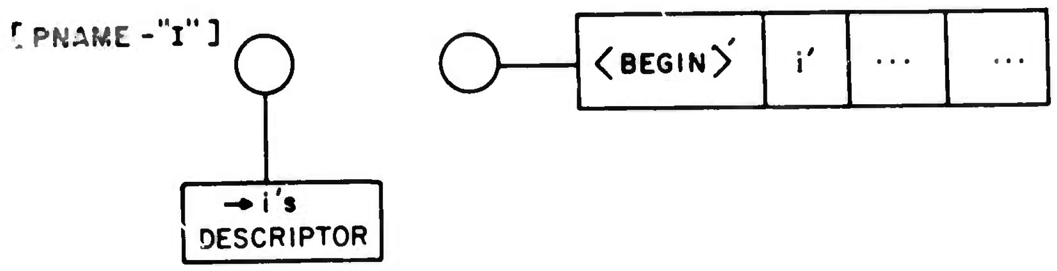
We suggested in Section 9.C4 that we might use the above question-mark form as a special syntax for display of associations.

Pnames

We can associate with each item a string value, which we call its Pname. There can be but one Pname for each item, and conversely. Efficient means are provided for finding one, given the other. We have used this Pname mechanism in COPILOT to implement the symbolic access to symbols.

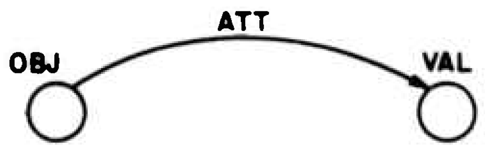
Pictorial Representation

In this dissertation an item is normally represented by a small circle, sometimes a small square. Its datum representation, if any, is appended to the item picture by a small unlabelled line segment. The datum is drawn in a convenient representation for its data type, meaning, etc. For example.



An item's pname, if relevant, appears near the item, enclosed in brackets, as [PNAME – "I"]. Any other names apparently labelling an item is unofficial, included in the diagrams for descriptive purposes.

The association "ATT⊗OBJ≡VAL" is drawn as an arc, labelled by the attribute ATT, connecting OBJ and VAL, as:



BIBLIOGRAPHY

[1] Rush: Terminal User's Manual. Allan-Babcock Company, Inc., 1966.

[2] Balzer, R.M., EXDAMS: Extendible Debugging and Monitoring System. Proc. 1969 Spring Joint Computer Conference, Vol. 34, pp. 567-580.

[3] Bauer, H., Becker, S., and Graham, S., ALGOL W Implementation. CS 98, Computer Science Dept., Stanford Univ., 1968.

[4] Berry, D.M., Introduction to Oregano. Proceedings of a Symposium on Data Structures in Programming Languages, Gainesville, Fla., February 1971.

[5] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S., TENEX, a Paged Time Sharing System for the PDP-10. Comm. ACM 15, 3 (March 1972), 135-143.

[6] --, and Wegbreit, B., A Model and Stack Implementation of Multiple Environments. BBN Report No. 2334, Cambridge, Mass., March 1972.

[7] Bryan, G.E., and Smith, J.W., Joss Language. Memorandum RM-5377-PR, The RAND Corporation, August 1967.

[8] Cheatham, T.E., and Wegbreit, B., A Laboratory for the Study of Automating Programming. Proc. AFIPS 1972 Spring Joint Computer Conference, Vol. 40, pp. 11-22.

[9] Corbato, F.J., CTSS Programmer's Guide, Project MAC, MIT, May 1965.

[10] Decsystem10 Users Handbook. The Digital Equipment Corporation, Maynard Mass., 1972.

[11] Algebraic Interpretive Dialogue Conversational Language Manual. The Digital Equipment Corp., DEC-10-AJCO-D, Maynard, Mass., 1970.

[12] Depres, R.F., A Command Structure for Interactive Programming. Project Genie Report No. P-17, Berkeley, Ca., March 1969.

[13] Dunn, T.M., and Morrissey, J.H., Remote Computing – An Experimental System. Proc. 1964 Spring Joint Computer Conference, Vol. 25, pp. 413-424.

[14] Dahl, O., Myhrhaug, B., and Nygaard, K., Common Base Language. Publication No. S-22, Norsk Regnesentral, Norwegian Computing Center, Oslo, Norway, October 1967.

[15] Engelbart, D.E., and English, W.K., A Research Center for Augmenting Human Intellect. Proc AFIPS 1968 Fall Joint Computer Conference, Vol. 33, part 1, pp. 395-410.

[16] ––, ––, and Rulifson, J.F., Development of a Multidisplay, Time-Shared Computer Facility and Computer-Augmented Management Research. Stanford Research Institute Report, April 1968.

[17] Eastlake, R., ITS 1.5 Reference Manual. Project MAC, Mass., Inst., of Tech., Cambridge, Mass., July 1969.

[18] Feldman, J.A., and Rovner, P.D., An Algol-Based Associative Language. Comm. ACM 12, 8 (Aug. 1969), 439-449.

[19] ––, Low, J.R., Swinehart, D.C., and Taylor, R.H., Recent Developments in SAIL – An ALGOL-Based Language for Artificial Intelligence. Proc. AFIPS 1972 Fall Joint Computer Conference, Vol. 41, pp. 1193-1202.

[20] ––, A Formal Semantics for Computer Oriented Languages (thesis). Carnegie Inst. of Tech., Pittsburgh, Pa., 1964.

[21] Fisher, D., Control Structures for Programming Languages (thesis). Carnegie-Mellon Univ., Dept. of Computer Science, May 1970.

[22] Floyd, R., A Descriptive Language for Symbol Manipulation. J. ACM 8, (1961) pp. 579-584.

[23] Gries, D., Compiler Construction for Digital Computers. John Wiley and Sons, Inc., New York, 1971.

[24] Hansen, W.J., Creation of Hierarchic Text with a Computer Display (thesis). Stanford Univ., Dept. of Computer Science, Palo Alto, Ca., May 1971.

[25] Hawker, E. (ed), USERS MANUAL. Computation Center, Stanford Univ., Stanford, Ca., 1971.

[26] Iverson, K.E., A Programming Language. John Wiley and Sons, Inc., New York, 1964.

[27] Johnston, J.B., The Contour Model of Block Structure Processes. Proceedings of a Symposium on Data Structures in Programming Languages, Gainesville, Fla., February 1971.

[28] Kay, A.C., The Reactive Engine (thesis). University of Utah, Dept. of Computer Science, Salt Lake City, Utah, August 1969.

[29] Kemeny, J.G., and Kurtz, T.E., BASIC – A Manual for BASIC, the Elementary Algebraic Language Designed for Use with the Dartmouth Time-Sharing System (third edition), Dartmouth College, Jan. 1966.

[30] Knuth, D.E., On the Translation of Languages From Left to Right. Information and Control, Vol. 8 (1965), 607-639.

[31] ––, The Art of Computer Programming, Volume 1; Fundamental Algorithms. Addison Wesley, New York, 1968, pp. 305-434.

[32] Lampson, B.W., Dynamic Protection Structures. Proc. 1969 Fall Joint Computer Conference, Vol. 35, pp. 27-38.

[33] ––, Time-Sharing System Reference Manual. Document #30.10.30, Dept. of Defense Contract SD-185, U.S. Printing Office, 1966.

[34] Leavenworth, B.M., Syntax Macros and Extended Translation. Comm. ACM 9, 11 (Nov. 1966), 790-792.

[35] Lindstrom, G., Variability in Programming Languages (thesis). Carnegie-Mellon Univ., Pittsburgh, Pa., July 1970.

[36] Lock, K., Structuring Programs for Multiprogram Time-Sharing On-Line Applications. Proc. AFIPS 1965 Spring Joint Computer Conference, Vol. 27.

[37] ——, Incremental Compilation (unpublished).

[38] McCarthy, J., Towards a Mathematical Science of Computation. Stanford University, 1962.

[39] ——, and Painter, J., Correctness of a Compiler for Arithmetic Expressions. Stanford Artificial Intelligence Memo Number 40, April 1966.

[40] ——, Abrahams, P., Edwards, D., Hart, T., and Levin, M., Lisp 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1962.

[41] Miller, R., Response Time in Man-computer Conversational Transactions. Proc. AFIPS 1968 Fall Joint Computer Conference, Vol. 33, pp. 267-278.

[42] Mills, H., Top Down Programming in Large Systems, **Debugging Techniques in Large Systems**, R. Rustin (ed). Prentice Hall, Englewood Cliffs, New Jersey, 1971.

[43] The Multiplexed Information and Computing Service: Programmers' Manual. Project MAC, Mass. Inst. of Tech., Cambridge, Mass., 1971.

[44] Mitchell, J.G., The Design and Construction of Flexible and Efficient Interactive Programming Systems (thesis). Carnegie Mellon Univ., Dept. of Computer Science, Pittsburgh, Pa., June 1970.

[45] ——, Newcomer, J., Perlis, A., Van Zoeren, H., and Wile, D., Conversational Programming – LCC. Carnegie-Mellon Univ., Dept. of Computer Science, Pittsburgh, Pa., June 1971.

[46] Naur, P. (Ed.), Revised report on the Algorithmic Language ALGOL 60. CACM 6, 1 (1963).

[47] Organick, E.I., and Cleary, J.G., A Data Structure Model of the B6700 Computer System. Proc. of a Symposium on Data Structures in Programming Languages, Gainesville, Fla., February 1971.

[48] Prebus, J., TVEDIT. Inst. for Math. Stud. in the Social Sciences (internal documentation), December, 1970.

[49] Quam, L.H., and Diffie, B.W., Lisp 1.6 Reference Manual. Stanford Artificial Intelligence Laboratory Operating Note 28.5 (Sept. 1970).

[50] Ryan, J.L, Crandall, R.L., and Medwedeff, M., A Conversational System for Incremental Compilation and Execution in a Time-Sharing Environment. Proc. AFIPS 1966 Fall Joint Computer Conference, Vol. 29, pp. 1-22.

[51] Simon, H., Reflections on Time Sharing From a User's Point of View. Carnegie Institute of Technology Research Review, 1967.

[52] Swinehart, D.C., and Sproull, R.F., SAIL. Stanford Artificial Intelligence Laboratory Operating Note 57.2, January 1971.

[53] Teitelman, W., Bobrow, D.G., Hartley, A.K., and Murphy, D.L., BBN-Lisp TENEX Reference Manual. Bolt Beranek and Newman Inc., Cambridge, Mass., July 1971.

[54] --, PILOT: A Step Toward Man-Computer Symbiosis (thesis). Report TR-32, MIT Project MAC, 1966.

[55] --, Automated Programmering – The Programmer's Assistant. Proc. AFIPS 1972 Fall Joint Computer Conference, Vol. 41, Part 2, pp. 917-922.

[56] Thomas, R. H., A Model for Process Representation and Synthesis (thesis). Report TR-87, MIT Project MAC, 1971.

[57] Wegbreit, B., Studies in Extensible Programming Languages (thesis). ESD-TR-70-297, Harvard University, Cambridge, Mass., May 1970.

[58] --, An Overview of the ECL Programming System. Proc. of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, Number 12 (December, 1971).

[59] Wegner, P., Data Structure Models for Programming Languages. Proc. of a Symposium on Data Structures in Programming Languages, Gainesville, Fla., February 1971.

[60] Van Dam, A., and Rice, D.E., On-Line Text Editing: A Survey. Acm Computing Surveys 3, 3 (Sept. 1971), 93-114.

[61] Van Wijngaarden, A.(ed), Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A., Report on the Algorithmic Language Algol 68. Numerische Mathematik 14:79-218 (1969).

[62] Waite, W.M., A Language-Independent Macro Processor. Comm. ACM 10, 7 (July 1967), 433-440.

[63] Wiederhold, V., PL/ACME. Stanford Univ. Computation Center ACME Facility, 1967.

[64] Wirth, N., On Multiprogramming, Machine Coding, and Computer Organization. Comm. ACM 12, 9 (Sept. 1969), 489-498).