THE DESIGN AND IMPLEMENTATION OF A
COOPERATIVE PROGRAM DEVELOPMENT ENVIRONMENT

James Elson Archer, Jr.

TR 81-468
August 1981

Department of Computer Science
Cornell University
Ithaca, New York
THE DESIGN AND IMPLEMENTATION

OF A

COOPERATIVE PROGRAM DEVELOPMENT ENVIRONMENT

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

James Elson Archer, Jr.

August 1981
BIBLIOGRAPHICAL SKETCH

James E. Archer, Jr. was born in Somerville, Massachusetts on October 12, 1947. He attended a variety of schools in a number of different cities before graduating from St. Mark's School of Texas in Dallas, Texas in 1965. As an undergraduate a M.I.T., he edited Tangent, the campus literary magazine, met and married Loren Sandiford, and received an S.B. in Management Computer Science. Graduate study in Management at M.I.T. was interrupted by an invitation to visit scenic Southeast Asia. The trip was unceremoniously cancelled when its sponsors discovered that the candidate had concealed a dislocated shoulder. As co-founder and president of Caprock Computing Systems in Lubbock, Texas, the author honed the ability to overengineer and underprice software that prepared him for graduate school.
ACKNOWLEDGEMENTS

Dick Conway is responsible for bringing me to Cornell, getting me involved in program development environments, and encouraging me to design and build COPE. He has served as my main supporter, co-conspirator, editor and critic throughout this research. Whatever is good about this work, he gets much of the credit; I take full credit for the remainder.

Andy Shore and Len Silver were key figures in the implementation and provided invaluable feedback on system goals and clarity. Without them, I would have had to choose between a working system and a speculative thesis.

Tim Teitelbaum set the standard for programming environments for novices and continued to provide expert criticism despite philosophical differences with the approach taken.

Alan Demers suggested improvements to the differential parsing section. Gary Levin and Dan Zlatin provided insights into formal semantics that helped in the formulation of the model for interleaved modification and execution. Dean Krafft participated in the early discussions that led to the design of the user file system. He and Merrick Furst provided general aid, comfort and support, as did the entire department, including Steve Fortune, Carl Hauser and too many others to name them all.

Fred Schneider, Dean Krafft and Loren Archer all conspired with Dick Conway to help convert this manuscript to English.

My parents gave me life, love, support and guidance — everything but the words to express my thanks. Finally, and most importantly, I would like to thank my wife for her contributions beyond counting. Her constant love, interest and enthusiasm made it all possible and enjoyable.
Foreword

COPE is a program development environment that provides a highly tolerant user interface specifically designed for novice users. The design and implementation of COPE has been the work of a group which includes the author, Richard Conway, Andrew Shore, and Leonard Silver. In presenting the results of the joint effort, attention has been given to balance the presentation of the author's contribution and the overall context of the system. The author has been involved in all of the development and design phases of the system. The initial system design and architecture were developed jointly with Richard Conway and the sections presenting design, architecture and user interface represent joint work. Later sections on particular issues (parsing, file system, etc.) are individual contributions.
# TABLE OF CONTENTS

Chapter 1 Introduction ........................................................................................................... 1
  1.1 Computer Assistance of Program Development ......................................................... 1
  1.2 Contemporary Programming Systems ......................................................................... 3
  1.3 Interactive Languages ............................................................................................... 5
  1.4 Recent Developments ............................................................................................... 7
    1.4.1 EMILY .................................................................................................................. 7
    1.4.2 PL/CT .................................................................................................................. 7
    1.4.3 CAPS .................................................................................................................. 8
    1.4.4 DLISP ................................................................................................................. 9
    1.4.5 COPILOT .......................................................................................................... 9
    1.4.6 UCSD Pascal ..................................................................................................... 10
    1.4.7 PDP-11 ........................................................................................................... 10
    1.4.8 The Cornell Program Synthesizer ................................................................... 11
  1.5 Research Hypothesis ............................................................................................... 12
  1.6 Thesis Organization ............................................................................................. 13

Chapter 2 COPE: Overview ................................................................................................ 14
  2.1 COPE: Ideas ............................................................................................................. 14
    2.1.1 Tolerance ......................................................................................................... 14
    2.1.2 Consistency ...................................................................................................... 15
    2.1.3 Simplicity ......................................................................................................... 16
  2.2 User View ................................................................................................................. 16
  2.3 Entry Paradigm ....................................................................................................... 17
  2.4 Mode-Free Operation ............................................................................................ 19
  2.5 Frugality .................................................................................................................. 21

Chapter 3 System Architecture .......................................................................................... 23
  3.1 Salient Characteristics ............................................................................................ 23
  3.2 The Command Interpreter ....................................................................................... 24
    3.2.1 Display Control ............................................................................................... 26
    3.2.2 Input vs. Display ............................................................................................ 26
  3.3 Execution Supervisor ............................................................................................. 27
  3.4 The Universal File System ...................................................................................... 29
    3.4.1 Interactive Files .............................................................................................. 30
  3.5 Controlling the Display Screen ............................................................................... 31
    3.5.1 Action Reversal .............................................................................................. 35
  3.6 Summary .................................................................................................................. 36
Chapter 4 User Interface ................................................................. 37
  4.1 Overview ................................................................. 37
  4.2 The Screen Formats ..................................................... 39
    4.2.1 Edit Screen ....................................................... 39
    4.2.2 Execution Screen ................................................ 41
  4.3 The PL/CS Language ..................................................... 43
  4.4 Immediate Statements .................................................. 44
  4.5 Commands ................................................................. 44
  4.6 The File System .......................................................... 46
    4.6.1 File Names ......................................................... 46
    4.6.2 File Types ......................................................... 47
    4.6.3 Standard Files .................................................... 48
    4.6.4 The .TEMP File ................................................... 51
    4.6.5 Multiple Input and Output Files ................................ 52
  4.7 Program Entry ............................................................ 52
    4.7.1 Entry of Immediate Statements ................................ 60
    4.7.2 Program Execution ................................................. 61
    4.7.3 Pauses in Execution .............................................. 63
    4.7.4 Testing Facilities ................................................ 66

Chapter 5 Differential Parsing .................................................... 67
  5.1 Overview ................................................................. 67
  5.2 Error Correction ........................................................ 68
  5.3 Incremental Parsing .................................................... 70
  5.4 COPI: Environment ..................................................... 71
  5.5 Hierarchical Specification .......................................... 72
    5.5.1 Separating the Parsers ......................................... 73
    5.5.2 Partitioning the Grammar ...................................... 74
  5.6 Correction and Specification ....................................... 76
  5.7 Top-Level I.I.(1) Parser Construction ................................ 77
    5.7.1 Creating the Parsing Tables ................................... 80
  5.8 Alternative Parser Choices ......................................... 82
  5.9 Insertion and Modification .......................................... 83
    5.9.1 Entry Discipline ................................................ 83
    5.9.2 Updating the State of the Parse ................................ 84

Chapter 6 Interleaved Execution and Modification ......................... 87
  6.1 Previous Work .......................................................... 88
  6.2 Overview ................................................................. 89
    6.2.1 Relation to Standard Models ................................... 89
    6.2.2 Program Text ..................................................... 90
  6.3 Modification Semantics ................................................. 91
    6.3.1 While Schemes ................................................... 92
    6.3.2 Flowchart Schemes ............................................... 95
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.3 Recursion Schemes</td>
<td>96</td>
</tr>
<tr>
<td>6.3.3.1 Resumption</td>
<td>97</td>
</tr>
<tr>
<td>6.3.4 Variable Declaration</td>
<td>98</td>
</tr>
<tr>
<td>6.3.4.1 Modification</td>
<td>99</td>
</tr>
<tr>
<td>6.3.5 Parameters</td>
<td>101</td>
</tr>
<tr>
<td>6.4 Implementation Note</td>
<td>102</td>
</tr>
<tr>
<td>6.5 Summary</td>
<td>104</td>
</tr>
<tr>
<td>Chapter 7 User File System</td>
<td>105</td>
</tr>
<tr>
<td>7.1 Levels of File Access</td>
<td>105</td>
</tr>
<tr>
<td>7.2 Unrestricted File Organization</td>
<td>107</td>
</tr>
<tr>
<td>7.2.1 Flexibility</td>
<td>107</td>
</tr>
<tr>
<td>7.2.2 Simplicity</td>
<td>108</td>
</tr>
<tr>
<td>7.2.3 Incrementality</td>
<td>109</td>
</tr>
<tr>
<td>7.3 File Structures</td>
<td>109</td>
</tr>
<tr>
<td>7.3.1 Basic File Structure</td>
<td>110</td>
</tr>
<tr>
<td>7.3.2 Data Area</td>
<td>111</td>
</tr>
<tr>
<td>7.3.3 Item Representation</td>
<td>112</td>
</tr>
<tr>
<td>7.3.4 Index Area</td>
<td>114</td>
</tr>
<tr>
<td>7.3.5 File Update</td>
<td>116</td>
</tr>
<tr>
<td>7.3.6 Implementation</td>
<td>117</td>
</tr>
<tr>
<td>7.4 Language Interface</td>
<td>118</td>
</tr>
<tr>
<td>7.4.1 Language Additions</td>
<td>119</td>
</tr>
<tr>
<td>7.4.2 Comparison with PL/I</td>
<td>122</td>
</tr>
<tr>
<td>7.5 Summary</td>
<td>124</td>
</tr>
<tr>
<td>Chapter 8 Summary, Conclusions, and Future Research</td>
<td>126</td>
</tr>
<tr>
<td>8.1 Summary</td>
<td>126</td>
</tr>
<tr>
<td>8.2 Conclusions</td>
<td>126</td>
</tr>
<tr>
<td>8.2.1 User Interface</td>
<td>127</td>
</tr>
<tr>
<td>8.2.2 Differential Parsing</td>
<td>127</td>
</tr>
<tr>
<td>8.2.3 Interleaved Execution and Modification</td>
<td>128</td>
</tr>
<tr>
<td>8.2.4 User File System</td>
<td>128</td>
</tr>
<tr>
<td>8.3 Future Research</td>
<td>129</td>
</tr>
<tr>
<td>8.3.1 Empirical Evaluation</td>
<td>129</td>
</tr>
<tr>
<td>8.3.2 Parsing</td>
<td>129</td>
</tr>
<tr>
<td>8.3.3 Program Display</td>
<td>130</td>
</tr>
<tr>
<td>8.3.4 Production Environments</td>
<td>130</td>
</tr>
<tr>
<td>8.4 General Conclusions</td>
<td>131</td>
</tr>
<tr>
<td>Appendix A COPI::PL/CS Language Summary</td>
<td>132</td>
</tr>
<tr>
<td>A.1 Procedures</td>
<td>132</td>
</tr>
<tr>
<td>A.2 Declarations</td>
<td>132</td>
</tr>
<tr>
<td>A.3 Statements</td>
<td>133</td>
</tr>
</tbody>
</table>
A.4 Comments ................................................................. 135
Appendix B COPE Command Summary .................................. 136
   B.1 Notation .................................................................. 136
   B.2 Commands ................................................................ 136
       B.2.1 Command Summary .......................................... 140
Appendix C File System Extension .......................................... 141
   C.1 Program Statements .................................................. 141
   C.2 Builtin ..................................................................... 142
   C.3 Summary of File Interactions ..................................... 143
Appendix D Unrestricted File Organization ............................... 145
   D.1 Access Method ......................................................... 145
   D.2 FSE Implementation ................................................ 146
CHAPTER 1

Introduction

Developing correct computer programs is a complicated and challenging activity requiring a variety of skills and disciplines. A principal concerns of computer science throughout its history has been the study of the logical components of the programming process and the development of the theory and practice of programming. A natural outgrowth of this research has been a continual and substantial increase in the employment of the computer to aid in the program development process.

Historically, advances in the theory and practice of computer-aided programming have been concentrated within the various phases of the programming process. The result has been systems composed of large collections of programs, each program addressing a particular need and providing a particular set of facilities. In such systems, the programmer must supply all the expertise necessary to integrate the various pieces. Recent research has focussed on systems, called "program development environments", which in principle take advantage of the best of each of the individual programming-aid disciplines to provide a complete service. This thesis reports the results of research that has resulted in the design and implementation of COPE, a particularly effective and innovative development environment for novice programmers. COPE is characterized by an extremely simple user interface, complete consistency of representation, and treatment and an unusual tolerance of incomplete or incorrect user entries or actions.

1.1. Computer Assistance of Program Development

At the very beginning of electronic computing, programming was performed directly in machine language with only the most primitive communication available between the machine and the programmer. The basic needs of the programmer came quickly to the fore as
became obvious that getting programs to work properly was going to be a continuing problem, mitigated only to a limited extent by diligence and attention to detail.

Early improvements featured increased access to the basic machine components. The memory dump provided a snapshot of memory at a particular point in the execution of the program. The ability to control the extent of execution (using breakpoints and single-stepping) provided valuable information on the progress of program execution. Symbolic assemblers assumed much of the responsibility for program bookkeeping.

The advent of higher-level languages greatly increased the expressive power of the programmer. Their development initiated the formal study of programming languages and the many issues associated with it. They also introduced considerable complexity into the programming environment. Since the program text and its executable form no longer corresponded in a transparent way, the programmer was forced to deal with the program on two levels, object and source.

The increased speed of machines made it necessary to provide automatic facilities for sequencing the execution of the various programs to be run on a particular machine. People were simply too slow and machines too expensive to allow programmers to continue to physically control the hardware during development sessions. A considerable diversity of operating system facilities were required to provide automatic equivalents for user control and support a variety of user needs.

Each of the generic types of advancement — debuggers, language translators, operating systems — has marked an increased role for the computer in the program development process. Each advance in the ability of the machine to aid in the process has brought new facilities for the programmer; unfortunately, each advance also generally increased the user knowledge required to exploit the facility. The aggregate of the "command languages" for editors, file systems, operating systems, etc. has also substantially increased.
1.2. Contemporary Programming Systems

Contemporary programming systems are a distinct improvement over those generally available even a short time ago. Time-sharing systems have become common. The state of the art for providing effective time-shared access has improved considerably, partially as a result of advances in software technology and partially through the lower cost and higher performance of hardware components. The advent of micro-computers, in particular, has made the user interface of the best time-sharing systems available in single-user systems.

Although interactive systems differ in many ways, it is possible to characterize the salient features of good contemporary systems.

Interesting contemporary systems are interactive. The hardware interface of choice is a CRT display terminal with communication bandwidth to redraw the screen in a few seconds. The software consists of an editor, language processors, and assorted utility programs linked together by a command language for selecting among these various facilities.

A substantial amount of the user's interaction is with the system editor. The same editor typically serves each of a variety of users for all purposes, but may contain specific features to facilitate the creation of commonly-used file types. A good editor is able to take advantage of the display to present any specified section of the program (or whatever is being edited) and provide direct "line-editing" operations that change the line and its image on the screen at the same time. The editor command language provides the various functions required to read and change files. Although the command language for the editor is usually simple, it is extensive and seldom bears any relation to other commands the user must learn.

Having prepared a program, the user submits it to the appropriate language processor to be converted to executable form, assuming that no errors are found. To help with testing, an execution monitor, or debugger, provides facilities for controlling program execution and examining the program's state. Another distinct command language controls these debugging facilities.
Effective management of program components — source files, object files, data, etc. — requires the user to become familiar with the utilities for managing files in the system. This requires familiarity with yet another set of programs, their command conventions and their operating procedures.

At least four command languages, each unrelated to the user programming language used, are required: operating system, file system, editor, and debugger. Some functions can be performed in more than one of these settings, although the function would be invoked using different command sequences in different command languages. Perversely, command names (or abbreviations) are often found to be common to two or more command languages but to mean very different things. An experienced user learns to distinguish the reasonable context for each function, though it is still common to attempt commands from one command language in another context.

For the novice or casual user, context distinctions are considerably more difficult. Not only must the novice understand when each command can be used, he must remember which commands make permanent changes to the program and which only change a derived version of the program. Since the “program” exists in more than one form, it is possible to create inconsistent versions (e.g. source has been updated, but the old object still exists) or unintentionally temporary changes (e.g. changes in the debugger that last only for the particular session).

There are many reasons for this separation of function and profusion of languages; user convenience is not one of them. Rather, the benefits accrue to the system implementor. By having a set of interchangeable parts, there are fewer programs to write and maintain. Since each program only deals with a small part of the task, each one can suboptimize the system purpose to its own advantage. In addition, since the interface between components of the system is minimal, there is less work involved in making the pieces fit together.
The separation of the various programming tasks along implementation lines creates systems that are unnatural to the user. As a result, one of the first things that the new user must learn to distinguish the various states and command sets for the system. Often, this is a more challenging task than writing the first program. Learning to program is hard enough without adding such problems.

As a further aggravation, the burden of communication is with the user, not with the system. Not only does each subsystem have its own commands and syntax, each insists that entries be complete and correct.

Interactive systems typically detect, but do not repair, errors made by the user in entering and executing programs. When an error is detected, the user is invited to try again, usually with little or no assistance. This authoritarian insistence that the user learn the correct form simplifies the implementation and protects the user from having minor errors interpreted as major mistakes. Concomitantly, the implementor feels justified in ignoring possible catastrophes ordered by the user, requiring only that the command is adequately specified. The assumption seems to be that automatic error repair is only useful when no other recourse is available, as happens in batch systems. System implementors thus have been trading an attractive feature of previous systems (correction) for a more attractive capability (user interaction) when both features could have been used together to good effect.

The problems described above notwithstanding, interactive systems work and people are able to acquire the skills to make even the most complicated interfaces work. Whatever the drawbacks, most of these systems provide an appreciable improvement in programming power and convenience in comparison with even the best of the batch systems.

1.3. Interactive Languages

The environments available for naturally interactive languages have historically provided much better facilities than have those for structured languages. Whatever their other benefits or drawbacks, the popularity of APL, BASIC and LISP has been increased by the attractiveness
of their environments.

The most important characteristic of the environments provided for these languages has been the integration of system functions into a single user interface. The user is able to enter programs, to execute them, and also to execute immediate statements — all within the same environment, using the same command language. This makes it possible for the user to access system components directly, at his own convenience, not the system's.

A typical BASIC system places program lines that begin with a number into the current program in statement-number order. Similar statements, (without the leading statement number), can be executed immediately. In this way, the facilities available to the user directly are precisely those available for inclusion in programs. BASIC also provides the most rudimentary form of language-cognizant editor. Since the meaning of each statement is determinable on a statement-by-statement basis, and since statements are available to be executed immediately, BASIC can provide immediate syntax checking. Further, the line-number ordering of the programming provides a simple, if limited, basis for editing the program.

LISP and APL are both applicative languages, making the distinction between immediate execution and stored programs somewhat less clear than in procedural languages. In both systems, the user manipulates a workspace (the APL term) that consists of the values of previous computations and a set of functions that have been created to deal with data. "Running" a program consists executing a statement or function that performs the desired computation; this is exactly the same as executing a function immediately. It is possible to save the entire state of the computation for future use. In this way, the duration of a computational session is defined by user convenience, not by the length of the execution of a single program.

Many of the desirable characteristics of these systems are closely linked to the languages they support. The problem is to provide a comparably attractive environment for structured languages.
1.4. Recent Developments

Recent work on development environments has provided several components and some complete systems that promise to improve the state of the art. A brief review of the research that has been most influential in the development of programming environments is appropriate at this point. Research specific to the narrower technical topics emphasized in later chapters is discussed in the appropriate chapter.

1.4.1. EMILY

Hansen's EMILY, an entry system for PL/I, was one of the first language-cognizant editors [28]. EMILY forces the user to develop programs top-down, without errors by selecting alternatives from menus. The menus limit user selections to items that are legal additions to the current parse. Only new items — variable declarations, constants, comments — can be entered directly. The implementation required considerable attention to the details of the interaction, since the user was forced to use this menu mechanism to get every token of the program. EMILY provided only editor functions, but influenced editors and command disciplines in later environments.

1.4.2. PL/CT

PL/CT [17] is an interactive version of PL/C that integrated the error repair facilities of PL/C with a comprehensive, source-language runtime environment. When execution is paused (because of errors or user request), the user can execute a subset of the PL/C statements immediately, making it possible to change the values of variables, skip statements, etc. Only simple statements (assignment, PUT, GOTO, etc.) can be used immediately, and the use of expressions is severely limited.

Portions of the source listing can be printed to find a context for further debugging actions. The program text is created and modified using the standard system editor between PL/CT sessions. The edited file is presented to the parser and subjected to the standard PL/C
error-repair process. Error corrections made by the PL/CT parser are not made in the source version and are reiterated at the beginning of each session until fixed by the user. The result is perhaps better than having no automatic repair, but batch repair of the complete file is obviously less helpful to the user than an immediate response. PL/CT was an interim system, and strongly indicated the desirability of a language-cognizant, error-tolerant, interactive editor.

1.4.3. CAPS

The first integrated development system for procedural languages was the Computer Aided Programming System (CAPS) system developed by Wilcox [64]. CAPS is a table-driven, language-cognizant environment for subsets of COBOL, FORTRAN, and PL/I, that was implemented to run on the PLATO CAI system.

The CAPS editor parses the program as the user enter it, taking great pains to detect and report errors well. When errors are detected, the user can review a series of proposed correction possibilities based on the language grammar. These correction possibilities include both terminal and non-terminal symbols. No attempt is made to correct the error for the user, but a consistently high level of information and assistance is available at all times.

CAPS also provides a novel mechanism for the analysis of execution errors. When an execution error is encountered, the system attempts to execute the program in reverse until it finds the place in the program where the inappropriate value was assigned. At this point, a "common misconception table" provides information that may be useful to the user as a possible aid in detecting why the error occurred. In this way, CAPS attempts not only to help find the error, but also to help find out how to go about finding errors.

CAPS was a very good first effort. It demonstrated the value of a basic combination of features that has been adopted by later systems: a language-cognizant editor and an attractive execution support system combined in a single environment. Unfortunately, the implementation was prohibitively expensive for practical use.
1.4.4. DLISP

Teitelman's DLISP [57] provides two distinct facilities that are of particular interest outside of the LISP base on which the system is built: windows and DWIM.

This was the first system to demonstrate the potential of displaying more than one object in logical windows on the screen. This ability to view multiple objects simultaneously was shown to make a significant difference in the utility of the programming environment. A high-resolution all-points-addressable display and pointing device are used to provide the user with considerable control over the contents of the display and the various phases of the computation. While the special terminal hardware facilities available to DLISP certainly add to the attractiveness of the package, the basic notion of multiple windows into the different aspects of the same computation is crucial. The size and flexibility of the display only defer the problem of choosing effective ways of displaying the "important" (to the user) parts of programs and data that are too large to fit on a fixed-sized screen.

The Do What I Mean (DWIM) facility attempts to remove the onus of perfection from the user. As used in DLISP, this freedom from perfection mostly pertains to the interpretation of commands and the correction of spelling, all based on provided or "deduced" characteristics of the particular user. The AI orientation of the particular facility is closely tied to the LISP base for the system, but the notion of having the system provide the best service possible, rather than making the user correct the mistakes and try again, is important.

1.4.5. COPILOT

COPILOT [52] is a system developed to support interactive programming in SAIL. [53], an ALGOL-based language with LISP-like features. COPILOT provides an attractive environment in which it is possible to perform the various tasks necessary to develop programs (or use them) within a single-language system. Particular emphasis is placed on the convenient control of multiple processes. By making it easily possible to switch between processes without disturbing them or losing control of their execution, COPILOT provides a way of handling
complicated interactive tasks with complete user control.

COPilot provides a powerful facility for the expert user. The control mechanisms and flexibility are well-considered, as evidenced by the detailed comparison with previous interactive systems in [52]. Considerable attention is given to matching system characteristics with the expected behavior of the target user population. The assumption of a sophisticated user makes the system a powerful one, but the resulting system makes more demands on the user than could be expected to be met by even a moderately experienced programmer. Even so, the COPilot goals and designs merit careful consideration by any system interface designer.

1.4.6. UCSID Pascal

UCSID Pascal [9] is probably the first system to provide an integrated user environment for a stand-alone microprocessor. The UCSD system provides an editor/compiler/execution system for a dialect of Pascal. The editor is only moderately language-cognizant, making some attempt to provide appropriate Pascal indentation, but relying on the compiler for any real checking. The principal differences between the UCSD system and a conventional single-user system (aside from supporting Pascal) lies in its uniform tree-oriented command-selection system. The user selects the next function from a menu at the top of the display screen. Functions have subfunctions, etc. The system is structured around separate components: editor, compiler, file system, etc., but the uniform menu discipline provides a form of command integration. Even so, navigation to perform sequences of functions in different subtrees can require more ingenuity than should be expected of the novice.

1.4.7. PDEIL

The PDEIL system [42] is an outgrowth of LISPEDIT [3]. It is designed to provide an environment for the development of programs in a subset of PL/I. PDEIL has an automatic system for controlling the contents of the screen. Since only part of a program can fit on the
screen, the problem is to decide what should be visible. The simplest solution, to show the immediate neighborhood of the current line in the program, shows very little of the global context of the system. PDEIL uses a function based on the reference activity of the program to determine the “best” set of lines to display. The system also increases the available context by displaying multiple statements on a single line in some sections of the program. The choices made are a distinct advance over local-context-only display, although it can be difficult to force PDEIL to display only local context when that is desired.

4.8. The Cornell Program Synthesizer

The Cornell Program Synthesizer [58] is the first system to provide a complete program development environment for a stand-alone microprocessor. Most of the facilities provided in JAPS have equivalents in the Synthesizer, but in a form and on hardware that provides essentially instantaneous response to most requests, and highly cost-effective performance.

Statement entry starts with the programmer locating the cursor at an insertion point in the program and selecting a “template” for the statement type to be inserted. The statement (and any matching constructs, such as END statements) immediately appear on the screen and the cursor is moved to the first “unexpanded nonterminal” position, where the user is supposed to fill in the expression contents (the condition of an IF, etc.). This approach is a compromise between the selection-only entry in EMILY and more traditional text entry of the program. The user is not forced to expand or enter the contents for each statement. If the entry sequence is altered to avoid entering the contents of a statement, an appropriate syntactic placeholder or prompt remains in that position of the statement to remind the user of its absence.

The Synthesizer forces the programmer to understand the tree representation underlying the entry discipline. All text manipulation is performed on subtrees, and cursor movement through the text of the program follows this tree rather than the apparent text. The user cannot directly enter any statement for which there exists an entry template. For example, the user
must remember that an IF or PUT statement is created by a template command, but an assignment or CALL statement is entered directly.

Both EMILY and the Synthesizer are effective at enforcing a hierarchical view of the program and eliminating structuring errors. It is impossible to enter an inappropriately structured program while using either system. The Synthesizer allows recovery from expression errors by direct editing of the offending text. Neither system allows the programmer to bypass the prescribed entry sequence to enter the program text directly. Our experience is that this insistence on the template form of entry, coupled with a lack of error correction, can be confusing to a novice user.

The Synthesizer provides a comprehensive execution support package based on a visual trace of the program text on the display screen and individual widows devoted to the display of program characteristics. This provides a particularly attractive way of displaying the state of the computation to the user. Execution errors cause the system to stop and show the cursor at the statement where the error occurred. The user can then modify the program and restart execution from the beginning of the program. No attempt is made to resume execution of programs that have been modified.

The command interface for the Synthesizer is relatively uniform. Each command is preceded by a designated command recognition character, and each command has the same effect whenever it is allowed. In several situations, the Synthesizer accepts only a small subset of the available commands. In particular, when the system needs to “turn the page” on output, the next character, regardless of which one it is, only signals the resumption of the output statement. The system is similarly insistent when it is expecting input from the terminal.

1.5. Research Hypothesis

COPI is an experimental system intended to explore the limits of an extreme approach to program development environments. The goal is to provide an aggressively simple, tolerant and consistent environment for the development of correct programs in an appropriate modern
style. The system attempts to provide an integrated facility that allows the user to develop programs incrementally and naturally, without constraints or complexities introduced by implementation restrictions or facility enhancements for production systems and sophisticated users.

1.6. Thesis Organization

The remainder of this thesis is devoted to presentation of the research and development involved in COPE. Chapter 2 presents basic precepts of the COPE system and how these precepts have guided the system design. Chapter 3 presents the architecture for the system and shows how each of the functional components of the system work together to achieve the end goal. Chapter 4 is a specification for the user interface of the initial implementation; this provides a concrete demonstration of how the system is assembled. Specific implementation decisions are discussed in the context of the family architecture. The success of the system depends materially on error correction and a tolerant interface. Chapter 5 presents the parsing techniques used to accomplish this interface. Chapter 6 discusses a model of execution and modification of programs used to make decisions about the legality of sequences of user commands. Chapter 7 presents the design and implementation of the file system that is integrated into COPE. This file system provides a simple, but powerful, facility for the novice user. Chapter 8 summarizes the most important characteristics of the system and presents directions for future research in related areas.
CHAPTER 2

COPE Overview

2.1. COPE Ideas

COPE is an attempt to explore one extreme of the program development system spectrum. The system is designed to provide a simple, tolerant and consistent environment for novice programmers. The purpose of this chapter is to explain the goals of the system and relate them to the design of the system and its user interface.

2.1.1. Tolerance

It is clearly unreasonable to expect that the user will make no errors, so some facility must be provided to deal with less than perfect input. Simply throwing errors back at the user simplifies implementation, but provides little service and less encouragement. If an error is trivial, it is an annoyance to have to fix it when the machine could have done so just as well. If the error is not trivial, simply insisting on correction is unlikely to be of much help. In either case, a reasonable correction, properly presented, is at least as useful as an error message.

Standard error correction only deals with syntax errors. Erroneous entries may be syntactically correct, but semantically disastrous. Traditional systems, while rejecting even the slightest syntax error, have been willing to do anything that was syntactically well-formed. For the user to be comfortable in learning and using the features of the system, it must be possible to simply cancel the effect of previous actions, whether syntactically correct or not.

The provision of an interface that provides active and consistent correction, and easy recovery should lead to a change in user entry patterns. The prime user concern is the program to be created. If the system corrects errors and fills in obvious omissions, the expectation is that the user can develop his own entry style. To help support this development
f individual entry styles, COPE is willing to correct whatever the user provides. In effect, this
relaxes the constraints on the user and obliges the system to create (at least syntactically) correct programs.

Tolerance of errors should not be confused with license. Because it is as helpful as
possible with the details of entry and syntax, COPE assumes that it can afford to be entirely
authoritarian in its view of larger issues. Regardless of the form of the user’s input, the partially
completed program is guaranteed to be well-structured and appropriately formatted. This
provides a considerable facility and incentive for developing programs in a top-down manner.
Although there is considerable freedom in how this facility is used, there is no way to subvert
its goals. The automatic repair facility allows the system to require (and supply) many syntactic
details that would be unreasonably onerous for the user. For example, all loops are labelled,
which is helpful in identifying END statements (especially when the entire loop doesn’t fit on a
single screen). The user can choose a label, but does not have to. In the absence of a user-
 supplied label, the system generates one — which can subsequently be altered, if desired, by
the user.

1.2. Consistency

The key requirement for integrating the program development process into a single,
coherent system is to provide a consistent user interface. The same function should be done
the same way, regardless of the details of the current system state.

A sophisticated user must be able to distinguish the modes, binding times and causal
relationships between the entities of multi-level computational processes. Learning how to
cope with this extraneous complexity is inordinately important to the novice’s success in
learning to program. Learning to program is already challenging enough without additional
complication.

COPE provides a consistent set of mechanisms for dealing with all phases of the
programming process without any apparent modes or levels. Any system operation can be
performed at any time. Program structures and system commands are interpreted uniformly regardless of the current state of the user session. All system objects can be manipulated using the same commands.

2.1.3. Simplicity

All design decisions in COPE are intended to make the system simple to use. We believe that error correction reduces the frustration caused by syntax errors. The disciplined creation of control structures reduces semantic errors. The consistency of command usage significantly reduces the number of different situations that the user must distinguish.

In addition, the language and the command set have been kept as small as possible. The PL/CS language was already small. A small set of language extensions have been made to provide allow file processing and display-screen control. The command set has been restricted to the "minimal" set of operations required to allow convenient use.

2.2. User View

The COPE system permits two types of activities: file modification and program execution. Every user entry is one or the other. Each of the system commands either initiates execution, modifies the contents of a file, or provides position/character control for an entry. There are no commands to control ancillary system functions (e.g. listing the names of user files, removing files, etc.), except orderly system shutdown. All user requests either modify files directly or invoke language statements. The user need only be concerned with controlling and monitoring the various program and data files which are (or can be) visible on the display screen.

The file system assumes a central role in the COPE system view — all user entries go into the file system. Program segments that are executed immediately are entered into a special stack-structured file, .TEMP, using the same mechanisms and techniques as other program entries. The only thing "special" about such an entry is that it is executed before control is
returned to the user. The user gets the desired immediate execution, is guaranteed that the action of the statement is the same as for the segment entered into the program, and can retrieve the segment from the temporary file for placement in a permanent file, all without special advance planning. The contents of this temporary file also provide an historical record of recent user actions.

The file nature of user entries is extended to include interactive terminal input. The user program reads data using the standard input (GET) statement. The program pauses for input. At this point, the user can issue any command in the COPE repertoire. The standard response, however, is to enter the requested data. By doing so, the user is editing the data into his input file to be read by the program when it resumes execution. The editing, retention, and consistent-entry characteristics of the file interpretation are maintained, but no change in user behavior is required.

The consistent use of the file system to represent system objects is carried over into such disparate facilities as providing help (.HELP) and allowing the user to list the file system table of contents (.TC). Even the runtime stack and check histories are kept in files. Some file objects are protect against direct user modification to prevent confusion and possible destructive side-effects. This unity of mechanism and concept simplifies the user's view and should greatly simplify learning to use the system.

2.3. Entry Paradigm

Each user-entry is eventually placed in either a data or a program file. Which file (and its type) is determined by the command used and the content of the entry. For either type of file, the user-entry is edited into the file and the image on the screen displays the file contents in a formatted, readable manner. Data entries are placed directly into the appropriate data file without modification; entries for program files require considerably more work.

Each program entry is processed by the Procedure Syntax Editor (PSE) for inclusion into the current user procedure. To achieve COPE goals, the PSE must compensate for incomplete
and incorrect entries wherever they occur.

Each entry is parsed (using techniques explained in Chapter 5) so that any errors are repaired and so that omitted tokens can be supplied by the system. The system supplies everything that is implied by the user’s entry. The result is a structurally complete, corrected, internal representation of the user entry. The program segment that appears on the display is re-created from this internal form in a consistent format, with appropriate indentation and spacing.

Since COPE undertakes to provide radical correction of the user’s entries, great care is taken to display the original entry and to highlight the corrected version in the user program. The user can always determine exactly what was added to the program in response to the most recent entry. With only minimal effort, it is possible to ascertain what correction was provided and determine if it is consistent with the intent of the entry.

Since no correction mechanism can be successful for every possible entry, COPE provides a command that allows the user to “undo” the effect of each of the preceding entries (in reverse order). In doing so, the corresponding entry is returned to the user-entry area so that the user can easily make modifications and resubmit it. Regardless of the consequences of a system-provided repair, the user is only one key stroke from replacing this with his own correction. This mechanism is also extremely useful for reversing the effects of mistakes unrelated to syntax correction (e.g. inadvertently changing the wrong statement).

The combination of a complete and consistent correction mechanism, effective and timely display of the correction, and a convenient recovery method should reduce the stress of program entry. The user can retain complete control over the content of the program with less concern for the details. The ability to produce correct, well-formatted programs without worrying about the minutiae of their entry should allow the novice user to focus on the important issues of program design and analysis.
4. Mode-Free Operation

COPE is intentionally extreme in its attempts to provide a "mode-free" user interface. The goal is to make it possible for the user to know how the system interprets an entry in any circumstances, without having to distinguish the details of the current system state.

At the highest level, a traditional system with many separate components (editor, compiler, etc.) supports at least one mode of operation for each component. For example, similar functions in the editor and the debugger in any such system typically require different commands, and almost certainly have different scope and persistence of action. Further, accessing the functions of one subsystem from within another, requires that some sequence of command be issued that explicitly moves the interaction to the new command processor. This high-level separation is rooted in implementation histories, but it also reflects the reasonable assumption of persistence of purpose and action: the most likely thing for the user to want to do next is something highly related to whatever he just did. Continuity of action of a predictable sort serves as the justification for modes; their use makes it easy to continue one type of operation at the cost of making it harder to change to another. A successful mode-free system must make both equally easy, not equally difficult.

Even within a particular function, subordinate modes are frequently used in traditional systems to distinguish user commands from text to be entered. COPE avoids this problem by taking commands separate keys that the user is incapable of incorporating into programs. (An "escape" mechanism could easily be provided to allow their inclusion, but not without introducing complications unwarranted for the target audience.)

COPE provides convenience of operation without the straightjacket of modes by making each user entry a separate command line. Each command specifies changes to the system's files. The effect of each entry is determined by the choice of command. A segment of text allowed by EXECUTE is always interpreted as a request to execute the current entry immediately. The same text followed by FILE will always be inserted at the current position in
the appropriate file. The ENTER command simply repeats the last EXECUTE or FILE command, providing "persistence", and continuing the current interaction pattern. Since each command is a single key, no additional work is required regardless of which of the three terminators is chosen.

In effect, each entry that the user makes becomes a transaction to be performed by COPE. The system is structured so that the effects of each of these transactions are stored explicitly. This saved state information serves the same purpose as the implicit state of conventional systems in making it possible to reasonably interpret the user request. As long as the user persists in an interactive pattern, COPE seems as natural as a conventional system. The important difference is that unplanned switches to different contexts are just as easy.

Another, more subtle, form of mode creeps into user interfaces in the guise of forced interactions. A forced interaction is incorporated in a circumstance where the system "knows" what the user should do next. Common instances are interactive error negotiation, pauses for page turn, and requests for data entry. Whatever the user enters is interpreted as an answer to the system's request for action. Depending on the system, the user must acknowledge the request with a desired response before proceeding, or the system will interpret the next entry as the response, ignoring or misinterpreting all or part of what the user has entered. This behavior is particularly aggravating when the user doesn't care about the error noted, doesn't know how to fix it, or wants to do something else to help decide how to fix the problem.

COPE goes to great lengths to avoid making the user respond to requests. Any error that is detected (and cannot be fixed) is reported to the user in terms of the action taken. The user can "undo" the action or ignore it, solely at his own discretion. At any time when a command can be entered, any of the system commands can be entered. As a logical extension of this capability, the user is able to stop the system when it is executing a program simply by beginning a new entry. The system pauses to accept the new command and carry out the indicated function.
2.5. Frugality

For novice users, a prime goal is to provide a system that is easily comprehended. Both the language and command set for COPE are intentionally small.

The language supported by the initial implementation is PL/CS. PL/CS is a small, well-understood subset of PL/I that supports program development in an appropriate modern style (a summary of COPE-PL/CS is presented in Appendix A). Although PL/CS is an attractive vehicle for COPE, a similar subset of Pascal or Ada could work just as well. The history of systems supporting PL/C and PL/CS at Cornell provides both a built-in expertise and audience for the current COPE implementation. The audience is particularly important, since it will make it possible to compare the efficacy of COPE, batch PL/CS, and the Cornell Program Synthesizer in a way that would not have been possible if a different language had been chosen for COPE.

A larger language subset could have been supported, but not without introducing additional concepts that the student would have to learn to cope with. The PL/CS language is sufficient for a first course in programming, and the additional features that could be added would only begin to compromise the focus on novice users. Further, additional features have a way of introducing new complications and keywords for the unsuspecting user to stumble over. A small set of language constructs also greatly increase the chances that error repair will be successful by reducing the number of possible alternative corrections.

The COPE command set (described in Appendix B) is also small. Each of the 23 commands serves a specific file management or cursor motion purpose. Since each command is a reserved key (or two-key sequence), there is an obvious premium placed on limiting their number. Whatever set of commands are chosen, they must be remembered by the user. A large set of commands provides small marginal utility at a cost of remembering a larger group of actions. Further, a sequence of well-understood operations is better training for the programming process than memorization of a long list.
As a result of its frugality, COPE should be easy for the novice to learn, but will occasionally be missing features the more sophisticated user might expect. These additional features would be easy to add; more effort was required to exclude them from the design than would have been required to include them in the implementation.
CHAPTER 3

System Architecture

The challenge in putting together an architecture for the COPE system is to provide a structure that will easily support the goals of the system without adding complexity to the individual functions. The primary requirement to accomplish this is the successful separation of data representation from assumed purpose or function. By making the functions that deal with the representation and modification of system data structures independent of the global command control flow, it is possible to provide the desired characteristics.

3.1. Salient Characteristics

Decisions about the state of the system and the states of the various processing routines have significant on the system architecture.

1) The system is entirely "transaction-oriented". Each processing routine provides functions that update, interpret and represent the objects of the system on a transaction-by-transaction basis. Only the module responsible for user interaction is required to deal with any interactive characteristics of the system.

2) All system objects are represented by file structures in a commonly accessed file system. Only knowledgeable service routines access the internal file formats; all other routines deal only with the implementing functions and applicable text representations of them.

3) All processing is done independent of the context in which the user issues commands. The same routine provides system service and user service; the two are indistinguishable.
(4) No processing routine, except the communication controller, ever does anything to affect the contents of the display screen. All display functions are the side-effects of transactions being processed.

We believe that this architecture is unusually flexible, and at the same time is exceptionally clean and simple. It has facilitated construction of the system and should simplify its maintenance.

3.2. The Command Interpreter

A user views a session with COPE as a sequence of "commands", each followed by the appropriate system response. The user must understand the difference between commands and statements, and the difference between storing a statement and executing it immediately.

Statements are the objects of ultimate concern to the user. They are constructions in the host programming language. The system facilitates the development, storage, and execution of statements.

Commands, on the other hand, are the imperatives by which the user controls the actions of the system — that is, actions to construct, to save, and to execute statements.

The distinction between statements and commands is emphasized by having entirely different forms for them. The number of commands has been kept very small, and each command is assigned to a special key. Statements are sequences of characters generated on the normal alpha-numeric portion of the keyboard.

The distinction between immediate and stored statements is entirely separate from the statement-command dichotomy.

The user interface is perhaps simpler than that of most other interactive systems by reason of the elimination of modes or levels. The set of commands has not been partitioned: all commands are accessible at all times.
Many of COPE's commands allow (or require) an argument, which is a string of characters. A user action consists of the entry of a string of characters, and then the entry of a command that specifies how that string is to be treated. For example, the SAVE command causes the line it is associated with to be stored; the EXECUTE command causes it to be executed.

The top level control loop for COPE is shown in Figure 3.1. The "perform" section of this loop selects one of a set of routines, one for each user command. A given command routine may have an "editing phase" or an "execution phase" or both. For example, immediate execution is accomplished by editing the given statements(s) into a file — the editing phase — and then calling on the execution supervisor to execute that file — the execution phase. By design, the execution supervisor always executes from a file; it is entirely unaware of any distinction between "immediate" and "stored program" execution.

\[
\text{DO UNTIL (command \# QUIT);} \\
\text{Get user entry ([text-line], command);} \\
\text{Perform action specified by command;} \\
\text{END;} \\
\]

Figure 3.1 — Top Level Control Loop

Readers who are experienced users of interactive systems will have noticed that the isolation of command functions into special function keys precludes the development of higher level functions composed of these operations, at least without resort to some objectionable escape sequence. This is by design. For all but the most complicated purposes, the native programming language, PL/CS, provides the functions of an "exec" language. The novice user is poorly prepared to handle multiple languages and can quickly learn to use the programming language or editing facilities to provide the required functionality.
3.2.1. Display Control

While the command set is not partitioned by modes, the display does change format to reflect the type of the last command given. Two different formats, *screens*, are presented to the user depending on the function the system is (or has been) performing. The "edit screen" is displayed when the command is explicitly altering some file, and the "execution screen" is displayed when the command causes execution of some statement. The shift between these different displays is completely automatic and unobtrusive. Each command determines what will be displayed, rather than having the current display determine which commands are currently accessible.

The shift from one screen appearance to another in no way restricts the user access to information. When the user command dictates that the previous screen format be used, the system automatically redraws the complete contents required for the new format. The appearance of the screen is governed by the contents of the underlying file objects, not a history of modification operations specifically directed to the display. Access to the display screen is guarded in such a way that any screen configuration, including any combination or arrangement of windows, can be supported.

The choice of fixed screen formats makes the transitions automatic and logical for the novice user. The underlying window management structure is sufficient to support much more complicated strategies. This flexibility has not been fully exploited in our initial version because to do so would require the addition of commands or language constructs to guide the selection and control of windows. Further, the limited display space on available terminals does not invite attempts to find more complicated forms of display.

3.2.2. Input vs. Display

The error-repair capability of COPE is entirely localized to the module that edits files whose type is "procedure". No other module must know that the text-line submitted by the user may not be what is actually inserted in the file. All file processing, regardless of file type,
shares this characteristic. Each type has an editor that is responsible for interpreting the user’s intent, parsing the input, and modifying the internal form of the file. An entirely separate program (one that presumably agrees on the format and semantics of the file representation) performs output formatting. Display lines are formatted at the request of the module that is responsible for controlling the contents of the display.

3.3. Execution Supervisor

The Execution Supervisor, described in detail in [5], is called by many of the command routines. When appropriate, it is called as a second phase of command execution after a file has been edited. Parameters to the call specify the particular procedure to be executed, and the environment in which to execute it. The procedure, the environment, and the input are all given in separate files; the output resulting from execution is placed in other files. As execution proceeds, the environment file is updated to reflect the current state of execution.

The basic structure of the Execution Supervisor is shown in Figure 3.2. Five types of situation terminate the execution interval:

1. Normal program completion.
2. Any input entered at the terminal.
3. Unable to write to file because window is full (waiting for page turn).
4. Input file contains insufficient data; waiting for terminal input.
5. Error encountered in execution.

Operation of the Execution Supervisor is synchronous in the sense that it can never be ‘surprised’ in the middle of a statement. Any user input sets an interrupt flag, which is interrogated only after execution of the current statement is completed. The last three of these situations do involve aborting execution in the middle of a statement — for page-turn, data-supply, and execution error, respectively. For data-supply and execution error, the statement causing the interruption is executed again from the beginning when execution is resumed.
Only page-turn interruptions are resumed from the exact point of interruption.

\[
\text{DO UNTIL (execution is terminated);} \\
\quad \text{Fetch next statement from procedure file;} \\
\quad \text{Execute instruction (modifying appropriate files);} \\
\quad \text{END;}
\]

Figure 3.2 — Execution Supervisor Control Loop

An interesting characteristic of the Execution Supervisor is that it is unaware of the distinction between “restart from the beginning” and “resume from the point of interruption”. This gives the system maximum flexibility in the execution of immediate statements, which are executed in the environment of the interrupted procedure, and in the alteration of a procedure while its execution is interrupted. Of course, some alterations make resumption impossible, but these are treated like any other run-time error during execution. (The interesting problems that stem from the interleaving of editing and execution are discussed in Chapter 6.)

It is worth repeating that any of the system commands can be executed at any point where execution stops, regardless of the cause of the pause. For example, while it may seem most natural to simply continue after a page-turn pause, nothing in the system expects or requires that this be the case. The system is just as willing to perform an orderly shutdown as to continue or to perform any other specific command. This is a departure from the common practice of using the next character typed as a page-turn release, throwing it away, and proceeding with execution regardless of the character entered.

Users will commonly want to examine and modify the state of the computation during pauses caused by errors. The effectiveness of a particular repair can be tested by simply re-executing the modified instruction(s). The ability to examine and modify the state of the computation during any pause also increases the user’s opportunities to take advantage of the available facilities to analyze and correct the cause of an interruption. In particular, restarting
GET operations from the beginning makes it possible to correct common mistakes in input format and value correspondence. Since both the input data and GET statement operands can be modified as necessary, it is possible to make sure that these two lists correspond without restarting the execution.

3.4. The Universal File System

An important characteristic of the COPE architecture is the construction of the system around a comprehensive file system. The same file system is used by the system for internal purposes, by the user to save programs, by the user to supply input data, by the user for auxiliary file operations (PUT FILE and GET FILE), and by the system to store the results of execution.

The file system is also fundamental to the architecture in that every module operates with file objects as its only input and output. For example, the Execution Supervisor is called with parameters that specify a procedure file, a environment file, and one or more input files. The Execution Supervisor modifies the environment file, and adds execution output to one or more serial files. In addition, the information destined for the execution screen (tracing and checking information) is also directed to separate serial files. What portion of this information is displayed on the screen, and in what arrangement, is the function of the Screen Control module, which uses these files as input. The Execution Supervisor has no direct involvement in screen display.

The result is a highly interactive system, but one in which the consequences of user interaction are carefully isolated. None of the main processing modules is complicated by the intricacies of user interaction; all operate strictly on a file-to-file basis.

The only exception to this structure arises from actions of indefinite duration, for example "execute"), that are terminated only by the beginning of the next action, i.e. by user "interrupt". To accomplish this, the main action control performs the execution phase of certain actions as a sequence of short executions. After each short execution interval, the "user
"input" file is queried to see if the user has typed something.

Transformations are frequently required to use the information in a file. For example, the internal representation of a program in a COPE file is a compromise between the form needed for textual display to the user, the parse-tree form required for constructing programs, and the form needed for efficient interpretive execution. Consequently, there exists a collection of utility routines to translate between the internal form of each type of file and its corresponding display representation. In aggregate, these utility routines represent a substantial fraction of the implementing code. Logically, these routines give the system's files a uniform appearance and access pattern, making it possible to deal with them uniformly.

Even the table-of-contents of the file system is itself a file in the system, and is displayed and edited much like any other file. It is a file of a special type, with translation utilities to display it in intelligible form. This design avoids having additional special commands to manage the file system.

The price of this uniformity of view is undoubtedly relatively slow execution. As yet we do not know how large this price might be, but do not regard machine time as the critical resource in the program development process. We believe that a substantial increase in machine time can be justified by even a small increase in the programmer's effectiveness.

3.4.1. Interactive Files

The COPE treatment of files, commands, and execution provides an effective management facility for interactive input. It is common in an interactive environment for the user to enter either too much or too little data at any particular time. The system must be able to deal with such occurrences uniformly, with the consequences of the system action predictable by the user. COPE treats the input that the user provides in response to data-supply pauses as insertions into the file named .DATA. As such, these insertions can be made at any time the system is paused (not only when it is expecting input). In addition, the contents of this "interactive" input are available for editing and can be automatically used as the data
stream for the program the next time that it is executed. This latter feature greatly facilitates testing the interactive program on a consistent set of data. Further, since the file can be edited at any time, input/value mismatches that are a common occurrence for novice programmers can be dealt with as they occur by either modifying the program to expect a different variable list, or modifying the data, or both.

Remember that the appearance of the display is governed by the contents of the various files as they are represented by decoding programs. This provides some problem for the normal display of interactive sessions: the input appears in one file and the output in another. In conventional systems, this is dealt with naturally by the interposition of echoed input and output directed to the terminal. This desirable form of output can be recreated by a file that contains a record of the combined input and output file actions. This not only solves the problem of keeping the display correct in the form that has been chosen, it clarifies the relationship between program input statements and data values where input and display become asynchronous. As with other displays in the system, the monitoring of this representation depends on which of the files are to be tracked on the terminal display.

3.5. Controlling the Display Screen

It remains to be shown how the system creates the appearance (and hence the reality) of interaction. As explained above, each processing unit is designed to process individual transactions that cause file contents to be read, interpreted and modified. The burden of making this transaction-oriented process seem both interactive and cooperative is born largely by the Communication Control (CC), which is responsible for the contents of the user’s display terminal. To effectively provide an interactive facility, the contents of the user’s display must be kept current with the files that are represented.

Most user actions modify one or more files. The editing actions obviously modify a file under the user’s explicit control. However, most file modifications are made implicitly and automatically by the Execution Supervisor while executing statements. The effect on the
.DATA and .OUTPUT files is obvious, but the Execution Supervisor also modifies files that contain trace, check and environment information.

Each of the system’s file types has both an internal and an external representation. The external representation is designed to communicate to the user the current state of the file and to allow the user to make changes in a logical way. For programs, the external representation is the programming language source; for other file types, appropriate representations are chosen for user communication.

Whenever a file is modified, its internal representation is changed and CC is informed of the change. If the user is monitoring a particular file on the screen, CC must modify the contents of the screen to reflect the file change. If the user makes a direct editing modification to the representation on the screen, CC must pass the text string which is the modified user representation into the system as the argument of a command, along with specific operational instructions arising in the user’s choice of command. The final form of the user’s modification is then reflected on the screen when the processing routine receiving the command modifies the affected file. The file monitoring mechanism makes the system appear to respond directly to the user’s action.

CC deals entirely with the external form of the files and the processing routines deal with the internal form. Figure 3.3 depicts the conceptual stages of a file transformation and the routines performing the transformations.

Consider the file interactions involved in changing a line of the user’s program. The program is stored in a file in a form that facilitates interpretive execution, but is sufficient for displaying the program. To display the program on the screen, CC must have access to enough program text to display the appropriate segment. This is called the “display text segment”, and is created by “decoding” (translating from internal to external form) each of the relevant program lines. Depending on a number of factors (including recent usage history and available memory), CC may already have text representations of some of the program text beyond the
lines required for the current display. In the extreme case, the entire program might exist as text (in addition to the internal representation). The "virtual text file" is what the user thinks the file looks like, and is equivalent to the "represented text segment" when the entire program is represented as text.

To perform a direct, screen-edit modification of the program, the user moves the cursor to the appropriate line and inserts or deletes characters to change the line to the desired form. When the user completes the change and indicates that the change should be entered (or executed or inserted elsewhere in the program), CC merely appends the appropriate command character to the line as modified and passes it on to the Command Supervisor, just as if the user had entered the entire line as an explicit command. The Command Supervisor routes the request to the editor for the appropriate file type. The editor converts the "new" line into internal form, incrementally parsing the new text, converting it to internal form, and updating the procedure file. As part of the file update, the file system notifies CC which line(s) have
been modified. CC then requests text versions of the lines for display on the screen. The parsed, formatted and corrected segment then appears on the screen. As part of the history of the modification, sufficient information is kept to allow the user to back out of the change with a single “undo” key. The original text entry is then available for modification.

By following this uniform access path, it is possible to allow any system command to be entered at any time. The automatic update notification from the file system to the CC greatly simplifies the processing routines. Processing routines only need to read and write files and can ignore the form and content of the display screen.

If this system is to appear natural to the user, the CC must provide the user the ability to move around on the screen and make changes at will. These immediate modifications to the screen are written to a file that is repeatedly read by the Command Supervisor. The Command Supervisor is responsible for correctly routing transactions to processing routines. When additional information is required from the user, a processing routine must return to the Command Supervisor and wait for the next transaction. Note that the removal of command context (in conjunction with the ability to stop sessions in the middle) from the processing routines requires that the files used to represent system objects be complete and consistent whenever the processing routine is not in control.

CC assembles input command lines and displays output window lines. The characteristics of the physical terminal are known only to CC, which is solely responsible for handling differences between the logical and physical terminal devices. The implementation of all other system modules is insensitive to changes in terminal characteristics.

The logical terminal is a simple input device and a powerful output device. Logical terminal input lines are considered to be character strings of indefinite length, each representing a complete user input. All line editing features are handled by CC, invisible to the user in the same way that they would be on an intelligent terminal. The logical output screen consists of a (potentially large) set of file windows. CC maintains a formatted screen consisting of those
windows that "should" be displayed at any particular time. The selection of windows is determined by the current system state, user-selected parameters, and available screen space. The logical screen consists of lines of indefinite length made up of a wide variety of characters in multiple fonts. CC is responsible for displaying these characters in the best way for each terminal type.

The isolation of all terminal communication in the CC makes it possible for the other modules to operate in transaction-processing mode, with almost no provision for the interactive nature of the system. This not only simplifies the system implementation, it makes it possible for the system to easily adapt to available hardware. The bandwidth of the terminal connection is important for a system that keeps a full screen-image up-to-date. For slower terminals, it is possible for CC to show parts of the current window set only as they are needed, filling the screen as "needed" instead of automatically. For intelligent terminals, some or most of CC can be moved to the terminal. In the extreme case, the text of all currently accurate window segments could be retained in the terminal, requiring only requests for new or changed line contents to redraw the screen.

3.5.1. Action Reversal

A key objective of the system is to provide a consistent mechanism for reversing commands and their effects. This capability is provided by transaction-oriented processing capabilities provided in the file system. Since each command constitutes an independent transaction, which modifies system objects as files, action reversal is equivalent to restoring system files to their prior contents. By incorporating the logging and transaction completion functions in the file system, it is possible to isolate the additional complexity of the general undo/redo facility. The major overhead in providing this facility is transaction completion notices and the file storage necessary to keep the transaction logs. For almost all of the operations that the user performs directly, this incurs only modest additional overhead.
However, during execution, the overhead of logging each instruction execution could easily require an order of magnitude more processing time than the operations themselves. This problem is solved by limiting the number of recoverable states. The unit of reverse execution is length of an execution interval, not a single statement. In this way, it is possible to limit the amount of overhead incurred by the extent of total state change. This is significantly more efficient than true reverse execution and provides similar functionality in an interactive system.

3.6. Summary

The COPE architecture is elegantly simple. The configuration of the system to operate as a series of transaction processors greatly simplifies the design of each of these modules, without sacrificing generality or utility. By preventing the current system state from becoming tied up in the call history of a tree of procedure calls, it is possible to provide the uniform access to command structures — a system goal. In addition, the file-contents oriented window displays provide an effective means of maintaining a meaningful representation of the system state independent of the particular program in control. The uniform access to files in readable representations provides both an attractive method of specifying changes and a way of enforcing consistent representation of objects regardless of the context in which they occur. The likely penalties for this organization are an increase in the control information passed to the transaction programs (which normally would have been implicit in the execution history) and an increase in overhead caused by the file-saving implementation of objects that are normally stored only in transient form, e.g. the runtime environment.
CHAPTER 4

User Interface

COPE has a novel architecture, an unusual approach to parsing and incremental translation, and an innovative file system, but these are simply steps involved in the implementation of a system to present a unique user interface. The interface described below is the reason for COPE's existence.

Our intent is to explore an extreme position with regard to the simplicity, consistency and tolerance that can be provided in a user interface. The interface is unusually frugal, consistent, and has no modes. It allows the user great flexibility in choosing styles in program entry, and is unusually tolerant of user frailties.

4.1. Overview

A user session with COPE consists of a sequence of distinct actions. Each action consists of a command given by the user, followed by an appropriate response by the system. The response performs the task specified by the user command, or explains why the command is unsuitable. When the response to a command is completed, the system returns control to the user for entry of the next command. Since the response to the EXECUTE command may require a long time, the user can terminate the response just by beginning the entry of the next command.

This process continues until the QUIT command is given, which terminates the session. The next session automatically begins in exactly the state that existed when the previous session was terminated.

There are only twenty-three different commands, including cursor motion (LEFT, RIGHT, LEFT END, WORD TAB, etc.) and editing commands (ERASE, CLEAR,
Many commands allow a string of characters to be given as an argument; one command specifically requires such an argument. The argument precedes the command key; in effect, the command key specifies what is to be done with the characters that have just been entered.

There is a clear distinction between commands and "statements". Statements are the underlying objects of the system. They can be executed or they can be saved — presumably for execution later. Commands are the imperatives that control the actions of the system; that is, control the saving and execution of statements. Commands take effect immediately — there is no provision by which a command can be saved in a file. All commands are entered with special keys, rather than by keywords or English-like sentences.

Every command is always available to the user. There are no "modes" or "levels" in the system that partition the set of commands. There are no special commands whose sole function is to change mode.

COPF: automatically formats (indents) the lines of a program to display its logical control structure. Systematic indentation makes a program easier to read and understand, but also indentation in COPF is important in the interpretation of various commands. Commands deal with program "units", not lines or statements. For simple statements such as assignment, GET, PUT, etc., the unit corresponds to a single statement. But for complex statements such as loops and conditionals, the unit consists of the initiating statement and all lines indented from that statement. For example, consider the segment in Figure 4.1. At the highest level shown in this segment there are three units. The three lines of the loop are all part of the same unit. If, for example, the REPLACE command were given when the edit-pointer is pointing to the "I = X(1)" line, that single line would be replaced (by whatever is in the entry area of the screen). But if the edit-pointer were pointing at the "I:1: DO" line when the REPLACE command is given, then all three lines of that unit would be replaced. At the next lower level, there is a single unit inside the loop. If the edit-pointer were pointing at this "X(I) =" line,
\begin{verbatim}
T = X(1);
L1: DO I = 1 TO N-1 BY 1;
    X(I) = X(I+1);
END L1:
X(N) = T;
\end{verbatim}

Figure 4.1 – Program Indentation Example

then that single line would be replaced.

The user has the option of compressing the display of any individual unit — presumably to reduce screen space for sections of the program that are not of immediate interest. The CONDENSE command causes all the lines of a unit to be strung out on a single line. The normal indented, multi-line display is restored by EXPAND.

4.2. The Screen Formats

There are two screen formats: the execution screen, which is displayed while the system is executing statements, and while it is paused in execution; and the edit screen, which is displayed while the user is explicitly altering or examining the contents of some file. One or the other of these screens is displayed on the terminal screen at all times.

4.2.1. Edit Screen

The format of the edit screen is shown in Figure 4.2. The title area identifies the file being changed and its file-type.

The entry area echoes the user's keyboard input. The entry cursor moves horizontally in the entry area to where the next character will be inserted. It is controlled by commands such as LEFT, RIGHT, LEFT END, RIGHT END, WORD TAB, and STEM TAB. The editing commands, such as ERASE, CLEAR, affect this area. The FETCH command copies a unit to this area.
<table>
<thead>
<tr>
<th>entry area</th>
</tr>
</thead>
<tbody>
<tr>
<td>previous command area</td>
</tr>
<tr>
<td>message area</td>
</tr>
<tr>
<td>text area</td>
</tr>
</tbody>
</table>

**Figure 4.2 — Edit Screen Format**

The previous command area displays the last command entered, with its entry argument. (The command displayed here may be undone by the UNDO command.)

The message area displays messages from the system: error messages, prompting messages, and confirmation messages. The text area displays a segment of the file being modified. At the left margin of the text area is the *edit pointer* (edit-ptr) identifying one line of the text area for certain editing commands. The text is automatically scrolled to keep the edit-ptr well-positioned in the text area. Scrolling is by half-pages: when the edit-ptr is moved off the bottom of the area the contents are scrolled up one-half page so the edit-ptr is in the middle of the area. The edit-ptr can be moved by the UP, DOWN, BACK PAGE, and FORWARD PAGE commands. Immediately to the left of the edit-ptr is the *change indicator*, where a "*" appears on each line modified by the previous command.

The normal format for display in the text area is canonically indented. Alternatively, the user can CONDENSE a section so that all the lines that would normally be indented from a
given line are concatenated to that line. However, for display in the text-area all statements after the first on any line are replaced by an ellipsis ("..."). If the length of a line exceeds the width of the text area its display is simply truncated. (It is not continued on the next line.) To see the left part of a long line, FETCH the line to the entry area where it can be scrolled left and right.

4.2.2. Execution Screen

The execution screen is formatted as shown in Figure 4.3. The entry, previous command, and message areas are the same as on the edit screen.

The stack area displays the top of the execution environment stack — the procedures currently being executed. The rightmost procedure on this list is called the "current procedure". For example, the stack might show

<table>
<thead>
<tr>
<th>entry area</th>
<th>stack area</th>
</tr>
</thead>
<tbody>
<tr>
<td>previous command area</td>
<td>message area</td>
</tr>
<tr>
<td>trace area</td>
<td>check area</td>
</tr>
<tr>
<td>output area</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3 — Execution Screen Format
P1 -> P2 -> P3

which would mean that P1 is being executed as a main procedure; P2 was called from P1; P3 was called from P2 and is currently being executed. If the procedure stack exceeds the capacity of the stack display area, only the top of the stack is displayed.

The trace area shows a somewhat abbreviated version of the text of the current procedure. Only executable lines (and comments) are shown: PROC, DCL, END are omitted. Lines are truncated at the right margin of the trace area; no continuations are shown. Statements are shown only to the indentation level established by the TRACE statement in the current procedure. Individual units are displayed in either normal (indented) or condensed (single line) form, depending on previous actions with CONDENSE and EXPAND commands.

At the left margin of the trace area is the execution pointer (exec-ptr), indicating the statement currently being executed. When execution is paused, the exec-ptr marks the first statement that will be executed when execution is resumed. The exec-ptr can be moved using the UP and DOWN commands, which are similar in effect to an "immediate GOTO". If TRACE(0) is executed in a called procedure, tracing of that procedure ceases (or may never begin). The trace area then shows the calling procedure with the exec-ptr on the calling statement (unless, of course, the calling procedure is also TRACE(0)).

The check area records each assignment of value to a variable. The form is:

\[
\begin{align*}
\text{X} & = 20.7 \\
\text{STR} & = \text{'new value'}
\end{align*}
\]

Both names and values are truncated as necessary to fit in the check area. If a variable receiving a new value is already listed in the check area, the value is changed. If the variable is not already listed, it is added to the bottom of the list if there is room. If the check area is already full, the newly assigned variable replaces the variable in the area to which a value was least recently assigned.
The variables participating in this checking can be controlled with the NOCHECK statement. Checking is subordinate to tracing in the sense that TRACE(0) suppresses checking as well as tracing. The procedure being checked is always the same as the procedure being traced.

The output area displays the output generated by execution of the PUT statements without a FILE phrase. When the output area is filled and the program needs to write another line, execution is paused for "page turn". In this situation, the EXECUTE command (with null entry) clears the output area and resumes execution. The output area can also be cleared under program control by the PAGE option in a PUT statement. The size of the output area can be modified by using the PAGE option. The space on the display not claimed by the output area is automatically included in the trace/check area.

4.3. The PL/CS Language

The first version of COPE supports program development in a slightly modified dialect of the PL/CS language. PL/CS is a carefully disciplined subset of PL/I. "Disciplined" in this context means that the usage of the constructs included in PL/CS is more restricted than for the same construct in PL/I. For example, in PL/CS functions have no side-effects whatever, and GOTO statements only permit forward references. In effect, these restrictions enforce usage that is generally considered good programming practice. The usage required in PL/CS is allowable in PL/I, so the language remains upward compatible with PL/I. (PL/CS includes an ASSERT statement and a READONLY attribute that are not part of PL/I. A program that uses either feature will not be upward compatible.)

PL/CS is a very small subset — easily learned and used. It is comparable to BASIC in this regard. It was intended for introductory instruction but, like BASIC, can be used for significant programming tasks.

There are two previous implementations of PL/CS. The first was a highly diagnostic batch processing system for 370-compatible machines [13]. The second was the Cornell
Program Synthesizer, an interactive development system comparable to COPE [58].

The modifications of batch PL/CS that have been made for COPE are the following: enhancement of the file processing capability, elimination of ON ENDFILE, which is unnecessary with the file processing enhancements, and the addition of statements to facilitate interactive execution.

The statements of COPE-PL/CS are described in Appendix A. The COPE File System Extension to PL/CS is described in Chapter 7, and a complete description of the statements is given in Appendix C. Except as noted, the syntax and semantics of statements are those of PL/I.

4.4. Immediate Statements

All COPE-PL/CS statements can be inserted in a procedure-type file and subsequently executed in "stored-program mode". All statement types except DECLARE, null, and PROCEDURE can also be entered for immediate execution.

Immediate statements are executed in the environment that exists at the time they are entered — that is, as if they were (temporarily) inserted in the program being executed, at the point where that program is paused. (Immediate statements are saved in a special file named .TEMP, displacing the previous contents of .TEMP.)

An exception to the usual PL/I environment rules occurs after the normal completion of execution of a program. In COPE, the environment of the main procedure continues to exist until the execution of some other program begins. This allows immediate statements to be executed as if they were in the main procedure. This can be useful for post-mortem purposes, and is the interactive equivalent of the post-mortem dump in PL/C [16].

4.5. Commands

There are twenty-three commands. A complete description of each command and its function is presented in Appendix B. A brief list of the command names is shown in Figure
4.4. Each command initiates a system action — the system automatically switches to the appropriate form of display screen for the command entered.

Every command is entered using one of the special keys on the keyboard (or in some cases, by a combination of keys). Commands cannot be entered by their keyword names. That is, the EXECUTE command can only be entered with the special EXECUTE key, and not by typing the word "execute" on the alphabetic portion of the keyboard. The command names are shown uppercase BOLDFACE in this description to emphasize their association with special keys.

Many of the commands allow an argument (the REPLACE command requires an argument). An argument is a string of characters typed on the normal alpha-numeric portion of the keyboard. The argument is always entered before the command. The content of the entry area at the time the command is given is taken as the argument of the command (if one is allowed or required). The system response to such commands includes clearing the entry area in preparation for receipt of the next command. Commands that do not allow an argument (for example, UP, ERASE, EXPAND, etc.) do not use or affect the entry area. The content of the entry area remains intact as an argument for the next command.

| [c] EXECUTE | [c] FETCH | [i] [f] MOVE | QUIT       |
| [c] FILE   | c REPLACE | [f] COPY     |            |
| [c] ENTER  | UNDO      |             |            |

| UP          | BACK PAGE | WORD TAB    | CONDENSE   |
| DOWN        | FORWARD PAGE | STMNT TAB  | EXPAND     |
| LEFT        | LEFT END   | CLEAR      |            |
| RIGHT       | RIGHT END  | ERASE      |            |

Figure 4.4 — Command List
4.6. The File System

The same file system is used in COPE to store procedures, data, and the results of execution. The same commands are used to create, display and change any of these files. Chapter 7 includes more detailed explanations of the operations provided by the file system, with particular emphasis on how data files are represented.

4.6.1. File Names

File names in COPE are normal identifiers (letters and digits, starting with a letter, lowercase letters are converted to uppercase automatically) except that a period is added as a prefix. File names are global and this distinctive prefix prevents ambiguity between file names and other identifiers used in procedures.

In some contexts, such as the FILE phrase of GET and PUT statements where an identifier is obviously a file name, COPE will forgive omission of the period prefix. But in general, the distinction is important and the user should supply the prefix period for all file names. For example, consider the entry

```
p   FILE
```

If P is interpreted as a file name, this means "switch to edit screen for file .P, creating a new file .P if such a file does not already exist". On the other hand, if P is not a file name, this means "insert a new line in the current file, based on the entry 'p'". This entry will be repaired to

```
PUT LIST( P );
```

and if a declaration for P does not already exist, such a declaration is generated. In short, the two commands

```
.p   FILE
    p   FILE
```

are both always meaningful and valid, but each has an entirely different meaning. Obviously, specification of the period prefix is crucial and the COPE editor cannot supply or discard it in this context.
Essentially the same situation exists with the EXECUTE command. Consider the command

```
.p EXECUTE
```

Assuming .P is a procedure file, this command means "start execution of a new program with procedure P as the main procedure". This implies clearing the previous environment, clearing the .OUTPUT file, resetting the pointer in the .DATA file, etc. On the other hand, the command

```
p EXECUTE
```

is an immediate execution command. If P is a variable, this is repaired to

```
PUT LIST( P );
```

and executed immediately. If P is a procedure, the entry is repaired to

```
CALL P;
```

and executed immediately. In neither case is the current environment changed. So, again it is critical whether or not the period prefix is given.

### 4.6.2. File Types

From the user's point of view there are only two types of file in COPE: **procedure** files and **text** files. Every procedure file contains one PL/CS procedure. Entries to procedure files are controlled by a "procedure syntax editor" (PSE) that ensures that a syntactically and structurally correct procedure is constructed.

Text files are used for everything except procedures. Entry into text files is essentially undisciplined — in particular, it is not controlled by the PSE.

A file becomes a procedure file by having the keyword PROC (or PROCEDURE) appear in the first entry after it is created. If PROC appears, the file type is committed, and the PSE is invoked for all entries to the file. If not, the file becomes a text file, and the entries are unconstrained.
One can store PL/CS statements in a text file, but then they are simply raw text and are neither checked nor formatted by the PSE. They could subsequently be copied into a procedure file. For example, suppose there is such a text file named .T containing a segment of a procedure. One could create a new procedure file named .R, and copy .T into .R by the commands in Figure 4.5. The COPY command streams the text from .T through the PSE, so it is checked, repaired, and formatted as it enters .R just as if it was arriving from the terminal. This can be useful for dealing with small segments of a procedure, but it is not recommended as the normal mode of program development.

```
.r FILE
proc ENTER
.t COPY
```

Figure 4.5 — Creating a Procedure File from a Text File

This ability to store a segment of a procedure in a text file is used for immediate statements in COPE. Every entry made for immediate execution is also automatically saved in file .TEMP as a text file. As such it can be re-executed, changed, streamed into a procedure file, etc.

4.6.3. Standard Files

COPE uses five standard files, presented in Figure 4.6. All these files (except .TEMP) are of type text. All can be displayed just like any other text file. All (except .TC) can be edited directly by the user.
The standard input file, .DATA — corresponding to SYSIN in batch PL/CS, is a normal ext file and is treated in exactly the same manner as any other data file. In particular, two characteristics of .DATA are of particular interest:

1. When the execution of a new program is begun, the "read pointer" of .DATA is automatically set back to the beginning of the file. Note, however, that the contents of the file are *not* automatically cleared by starting a new program execution. (.DATA can be cleared before starting executing a new program by first executing an immediate statement:

   `DELETE FILE(.DATA); EXECUTE.`

   Thus, reading from the standard input does not presuppose a fresh start from terminal input.

2. When the contents of .DATA are insufficient to supply the GET statement being executed, execution pauses for "data input". In this case the ENTER command appends entries to the end of the data file referenced by the GET. This can be done repeatedly prior to resuming execution.

If execution was in the middle of the list of a GET statement when the contents of .DATA was exhausted, the entire GET statement is re-executed (and the read-pointer in .DATA is restored) when execution resumes. (These privileges apply to all input files and not just to the .DATA file.)
The standard output file, .OUTPUT — corresponding to SYSPRINT in batch PL/CS, is a normal text file to which lines are automatically appended by the execution of PUT statements. .OUTPUT can be displayed and edited like any other text file. It enjoys three special privileges:

1) Lines are automatically added to .OUTPUT by the execution of PUT statements without FILE phrases.

2) The contents of .OUTPUT are automatically cleared each time the execution of a new program is begun.

3) The contents of .OUTPUT are automatically displayed in the output area of the execution screen. The size of this display area can be controlled by the PAGE phrase of PUT statements; the output area can also be cleared by the PAGE phrase. The PAGE phrase has no effect on the information appended to the .OUTPUT file — it affects only the automatic display of that file.

The FILE and COPY commands can be used to save the contents of .OUTPUT prior to beginning another execution.

The .HELP file is the starting point for a set of user instructions for COPE. A user seeking help enters

```
.help  *  FILE
```

and the beginning of the .HELP file is displayed. This gives directions leading to other files with more detailed instructions. (.HELP is a normal text file and can be edited, copied, cleared, etc.)

.TC is the table of contents of the file system. The contents of .TC are managed automatically by the file system. New entries are made as files are created; entries are deleted as files are cleared (empty files are not listed). .TC differs from other files only in that it is protected from direct change by the user. It is still displayed by the normal commands.
4.6.4. The .TEMP File

The .TEMP file serves several purposes in COPE:

(1) It is the default file for the MOVE and COPY commands. When no file is specified in either of these commands, they apply to .TEMP.

(2) It is the “recovery file” for the MOVE command. This command clears the target file before moving objects into it, but only after the contents are automatically copied into .TEMP. This makes recovery from an accidental MOVE a simple operation with either the UNDO or COPY command.

(3) It is used to automatically save every “immediate statement”.

The .TEMP file is unusual in several respects:

(1) .TEMP is a stack of files rather than a single file. Whenever a file is copied into .TEMP, or an immediate statement is saved in .TEMP, this does not destroy the previous contents of .TEMP, but only adds a new level to the top of the stack. Files are “pushed” onto .TEMP to the limit of space available in the system. Files can be “popped” off the .TEMP stack by the DELETE statement. Each execution of a DELETE statement (with FILE(.TEMP) and without a RECORD phrase) deletes the top level from the .TEMP stack (the items at the bottom of the stack are automatically cleared if the stack becomes too large).

(2) Each level of .TEMP has the same type (procedure or text) as the file that was copied into it.

Probably the most common use of .TEMP is as the default file for the MOVE and COPY commands. For example, to move a segment consisting of 3 units from one position to another, one would use a sequence of commands such as that given in Figure 4.7. In between the MOVE and the COPY in this sequence, the segment could be altered by editing the .TEMP file.
4.6.5. Multiple Input and Output Files

The PL/CS language has been extended in COPE to allow multiple input files, multiple output files, and convenient access to particular items in these files. The additions to the language for this purpose are simple compared to their PL/I counterparts, and are not compatible with PL/I.

The additions, which are described in detail in Chapter 7 and Appendix C, consist of:

1. the FILE phrase in the PUT statement,
2. the FILE and NEXT phrases in the GET statement,
3. the DELETE statement, and
4. six functions and pseudo-variables: REC, REMAIN, KEY, FIND, COUNT and ITEM.

These elements provide a powerful, yet simple facility to manage files in either sequential or direct access mode.

4.7. Program Entry

A PL/CS program consists of one or more procedures. Each procedure in COPE is a separate file. Program development is the process of creating these procedure files, entering the declarations and statements that constitute the body of each procedure, and subsequently modifying the body until it performs the required task.
A new file is created by giving the FILE command with the new file-name as argument. For example, to create a new file named .P

```
.p   FILE
```

The edit screen is displayed, with the title indicating that a new file .P is being edited. The text area is blank.

 The file .P becomes a procedure file if the first insertion includes the keyword PROC (or PROCEDURE). The type thus established is a permanent characteristic of the .P file. For example, if the command is

```
proc   ENTER
```

the text area of the edit screen will appear as in Figure 4.8. Note the following:

1. The procedure name is the same as the file name, but without the identifying prefix period.

2. The result is given in upper-case letters, although the input was given in lower-case. All input except string values and comments is automatically translated to upper-case for entry into a procedure file.

3. The edit-ptr is positioned on the PROC line, indicating that this is the current insertion point in the file. The next insertion entry will follow this line.

4. Both lines are marked with an asterisk, indicating that they were changed by the previous command.

Note also that the FILE command could have been used instead of ENTER. When the edit screen is being displayed ENTER is equivalent to FILE.
COPE allows the user a great deal of flexibility in the manner in which a procedure is developed. At one extreme, the user can first write his procedure out completely on paper and then copy the text, line by line, onto the terminal. Although this is the traditional way of entering a program, it does not exploit the capability of a structured development system like COPE.

At the other extreme, the user can construct his procedure interactively, in a strict "top-down" development, by entering only various distinctive keywords and allowing the system to supply whatever is implied. For example, if the next entry in P is

```
while ENTER
```

the response shown in the text area is as shown in Figure 4.9. The word "cond" in the response is called a prompt. Prompts are always displayed in lower-case letters. A prompt indicates that the construction is incomplete, and that an element of the specified kind must be supplied — although not necessarily immediately. The user can supply the prompted element at once, or can proceed to develop another part of the program first and then return to

```
P: PROC;
  WI: DO WHILE cond;
    END WI;
  END P;
```

Figure 4.9 — Addition of "while" to Empty Procedure
complete this aspect of the procedure. A partially complete procedure still containing prompts can be executed but each prompt is a barrier to execution. Every time a prompt is encountered execution pauses and cannot proceed until the prompt is replaced by a suitable element.

In the example above, the user has, in effect, prompted the system to construct a loop by entering the keyword "while". The system constructed all that was implied by this keyword, and in turn prompted the user for any elements that are necessary to complete the construction.

At this point the user has many choices. He can supply the required element immediately, or he can go on to some other aspect of the development and fill this in later. Assuming he chooses to fill in the condition at once, and if his next entry is acceptable as a condition, it will replace the condition prompt on the edit.ptr line. For example, the entry:

\[
\text{a < b ENTER}
\]

results in the display in Figure 4.10. However, if the entry was not acceptable as a condition, it would automatically be considered as the entry of the next statement and would be inserted in the body of the loop. For example, suppose instead of "a < b" the following entry had been given:

\[
\text{call ENTER}
\]

The result would be as shown in Figure 4.11.

Because PL/I uses the same symbol for equality and assignment, certain entries could be taken as either a condition or an assignment statement. For example, in the situation above, if

---

```
P: PROC;
-*> W1: DO WHILE (A < B);
   END W1;
END P;
```

Figure 4.10 — Figure 4.9 with Condition Filled In
instead of "call" the following entry was made:

\[ a = b \quad \text{ENTER} \]

the system could not be sure whether a condition or a statement was being entered. In COPE
this ambiguity is arbitrarily resolved in favor of a statement, so the result would be as shown in
Figure 4.12. To overcome this bias toward the assignment statement, a condition can be
entered in parentheses. (All conditions in PL/CS are parenthesized, although in most contexts
these parentheses are automatically supplied by the system and need not be entered by the
user.) That is, the entry would be given as:

\[ (a = b) \quad \text{ENTER} \]

Note that when the condition involved an inequality rather than equality there was no
ambiguity and the parentheses were not necessary in the entry.
Another possible surprise would occur if the user re-entered one of the keywords rather than just the prompted element. For example, suppose the following entry were given when the screen appears as in Figure 4.9:

```
while a = b  ENTER
```

The result would be as shown in Figure 4.13. The re-entry of the keyword "while" has triggered the construction of a second loop. (If this was not intended the result can be undone by giving the UNDO command.)

COPE allows any entry mode between the extremes of full text entry and structured prompted generation. Each user can develop his own preferred mode of development, and vary his pattern as circumstances suggest. But note that regardless of the entry mode being used COPE ensures that at every point the program is *syntactically correct*, although perhaps incomplete with residual prompts.

Note that the system has *generated* labels on the loops, rather than prompt the user to supply them (loop labels are required in PL/CS). The labels chosen are short, and serve to identify the matching ENDS. (If W1 or W2 had already been used by the user, the next available integer suffix would have been chosen.) The user can choose his own loop names by giving the name with the entry that creates the loop. For example, the entry

```
col_loop until  ENTER
```

---

```
P: PROC;
  W1: DO WHILE cond;
  W2: WHILE ( A = B );
  END W2;
  END W1;
  END P;
```

Figure 4.13 — Figure 4.9 with "while a = b" Inserted
would create an UNTIL loop named COL_LOOP.

If the user does not supply a label initially, he can later change a label generated by the system. This is done by positioning the edit-ptr on the line to be changed (with UP or DOWN commands) and giving the FETCHI command. This copies the edit-ptr line into the entry area. There it can be modified in any way, and returned to the procedure with the REPLACE command. If the label on the DO line is changed, the label suffix on the matching END is automatically changed.

The labels generated for different constructions are shown in Figure 4.14. Labels on all loops and SELECTs would unquestionably be a nuisance if the user had to supply them. But they are particularly helpful in a system such as this that depends on a small screen where only a small fragment of the program is visible. Even the automatic indenting is not adequate for nested structures when only part of the nest is visible, so the labeling of ENDS is quite helpful. It also makes it possible to require a loop-name on every LEAVE statement, which makes its intent much clearer.

There are many contexts in which COPE employs this strategy of generating a specific element, rather than prompting the user to supply the element. The user can readily change the generated element if it is inappropriate. Figure 4.15 provides some examples of what is constructed from particular user entries. Each entry is assumed to be followed by the ENTER command.

<table>
<thead>
<tr>
<th>Wi</th>
<th>for WHILE loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ui</td>
<td>for UNTIL loops</td>
</tr>
<tr>
<td>Li</td>
<td>for indexed loops</td>
</tr>
<tr>
<td>Si</td>
<td>for SELECT statements</td>
</tr>
</tbody>
</table>

Figure 4.14 — Default Label Names
<table>
<thead>
<tr>
<th>user entry</th>
<th>COPE repair</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) do i = 1</td>
<td>L1: DO I = 1 TO expr BY 1; END L1;</td>
</tr>
<tr>
<td>(b) shift do i = 1 BY 2</td>
<td>SHIFT: DO I = 1 TO expr BY 2; END SHIFT;</td>
</tr>
<tr>
<td>(c) dcl x</td>
<td>DCL ( X ) FLOAT;</td>
</tr>
<tr>
<td>(d) x fixed</td>
<td>DCL ( X ) FIXED;</td>
</tr>
<tr>
<td>(e) s 10 char</td>
<td>DCL ( S(1:10) ) CHAR(80) VAR;</td>
</tr>
</tbody>
</table>

Figure 4.15 — Examples of COPE repair

In 4.15(a), COPE provides a label and a default increment of 1; both may be changed. To avoid having to change the generated value, a specific value can be supplied initially as in 4.15(b).

A declaration without specified type attribute yields the default type as in 4.15(c). The FLOAT attribute can subsequently be changed, but if another type is intended it would have been easier to specify the correct type initially as in 4.15(d). The entry in 4.15(c) illustrates the provision of the default string length of 80 (screen width) is generated in the absence of initial length specification.

Each of these generated elements is intended to simplify program entry by eliminating a prompt-response cycle in cases where there is a plausible default. However, in each case the user can avoid having the system generate elements simply by specifying his choice with the initial entry.

Any line in a procedure can be changed by moving the line into the entry area with the FETCHI command, making the changes and then returning the line to the procedure with the REPLACE command. The WORD TAB command is useful in positioning the cursor in the entry area to make such changes.
format), the STMT TAB command moves the cursor from one statement to the next. If the changes involve replacing a prompt with a specific element, recall that the entire prompt is treated as a single character by the LEFT, RIGHT, and CLEAR commands. When the changed line is replaced in the procedure, it is completely rechecked by the PSE just as if it were a brand new entry.

The examples above illustrate that the ENDS in a procedure are all generated automatically by entries such as PROC, WHILE or SELECT. The user need never enter an END explicitly, and in fact, cannot do so. However, the entry

```plaintext
end    ENTER
```

has special significance and is very useful. It causes no change in the procedure, but simply: *repositions the edit-ptr* to the END line of the containing unit. This provides a convenient way of escaping from the body of a loop, so that the next insertion follows the loop, rather than be contained in it.

For example, consider the sequence of entries in Figure 4.16(a). This treatment of end entries allows a complete PL/CS program to be entered directly, in spite of the fact that END lines are generated automatically by the PSE. This allows a program to be written out in advance, in the traditional manner, and then entered as continuous text without requiring any changes to accommodate COPE. More importantly, it allows text files to be given as input to the PSE. This permits communication with other development systems, and also allows programs to write files that are program text.

4.7.1. Entry of Immediate Statements

Very little need be said about the entry of immediate statements, since almost all of the section above applies directly. The distinction is simply that the EXECUTE command is given instead of FILE. The entry is still corrected and expanded in precisely the same way, but the result is executed immediately rather than inserted in some file.
(a) user enters:

```plaintext
to ENTER
when ENTER
end; end ENTER
get p; p ENTER
```

(b) produces:

```plaintext
L1: DO index = expr TO expr BY 1;
S1: SELECT;
  WHEN cond ;
  OTHERWISE;
  END S1;
END L1;
GET LIST( P );
->*
PUT LIST( P );
```

Figure 4.16 - Entry Sequence Using END

Note that the execution takes place in the environment of the procedure whose execution is paused, and not in the procedure being edited if those happen to be different procedures.

In fact, an entry for immediate execution is also saved in the special file named .TEMP. This means that the user can subsequently examine the entry (by editing .TEMP), change it, and re-execute it. It is important to understand that the storage of such an entry in .TEMP does not make .TEMP a procedure file. Consequently, the execution of .TEMP is treated as an immediate execution in the current environment, not as the execution of a new procedure in its own environment.

4.7.2. Program Execution

COPE is very flexible in its control of the execution of procedures, but can produce some surprises if the user is not aware of the rules for establishing execution environments. The most important point is the difference between starting execution of a new program, resuming execution of the previous program, and executing an immediate "call" of a procedure.
.proc-file-name EXECUTE

Begins execution of a new program. The procedure in the specified file is treated as
the main procedure of a new program. The previous execution environment is
cleared.

EXECUTE (with a null entry)

Resumes execution at the position indicated by the exec.ptr. That pointer may have
been repositioned, and/or changes in the environment may have been made by
commands since execution was interrupted, but subject to these changes execution
is resumed.

entry EXECUTE (entry neither null nor proc-file-name)

Causes immediate execution of the entry in the current environment.

When the entry is a simple statement (i.e. assignment, PUT, etc.) this is obvious; it becomes
complicated if the immediate statement calls another procedure — particularly if execution of
that procedure is interrupted. For example, the entry

call p EXECUTE

would be executed as an immediate statement, executing procedure P just as if the statement
CALL P; existed in the current procedure at the point where the current procedure was
interrupted.

The error-repair facilities of the PSF are significant in this situation. For example, if P is
the name of a variable in the current environment, then the entry

p EXECUTE

is interpreted as

PUT LIST( P );

If P is not the name of a variable, but is the name of a function procedure, the repair is the
same, and the value returned by the function is displayed. If P is not a variable, but is the
name of a procedure that is not a function, the entry is interpreted as
CALL P;

The greatest potential confusion is a command of the form

file-name EXECUTE

where the file specified is not a procedure. In this case the text of that non-procedure file is treated as an entry for immediate execution — that is, without clearing the environment and beginning a new program.

This means that fragments of procedure text can be stored in files, and then used for immediate execution just as if they were entered at the terminal. In fact, text entered at the terminal for immediate execution is also automatically stored in the file named .TEMP. As such it can be re-executed without manual re-entry by the command

.temp EXECUTE

The point is that the result of a command such as

file-name EXECUTE

is very different (in its change of environment) depending on whether or not the file specified is of procedure type.

Note that although the examples above have all shown the EXECUTE command, whenever the execution screen is display the ENTER command is equivalent to EXECUTE (except when execution is paused for data entry).

4.7.3. Pauses in Execution

The execution of a procedure can be interrupted by any of the following events:

(1) the execution of a PAUSE statement

(2) the execution of a GET statement with inadequate data in the input file

(3) the execution of a PUT statement when there is no line available in the output area of the screen
(4) a run-time error has occurred during execution.

The system can also be in a similar state because

(5) the execution task has been completed.

In each of these five situations the execution screen is displayed and the system is waiting for a user command. In each case an explanatory message is given in the message area.

Execution can also be interrupted by entering a command on the keyboard. In this case, after interrupting, the system immediately begins to carry out the new command. There is no apparent pause, and no message explaining the nature of the pause.

Users with experience on other interactive systems should note the absence of a "break" or "interrupt" command. Such a command would be redundant since every COPE command is capable of interrupting execution. (This feature must be disabled if real-time interactive usage of COPE programs is desired, video games for instance, but is not a matter of concern for the novice programmer.)

During a pause in execution the user can:

(1) Edit any file. When the file containing one of the procedures currently being executed is changed, the ability to resume execution may be impaired (an appropriate warning is given). Except when one of the program's input files is augmented, the display will automatically change to an edit screen for the file being edited. The standard .DATA and .OUTPUT files can be edited with the normal editing commands, but these do not allow repositioning of the read/write pointers in these files. When the file of the current procedure is edited, the edit-ptr is initially positioned to coincide with the exec-ptr. However, the converse is not done - on resuming execution the exec-ptr has not been moved by movement of the edit-ptr.

(2) Execute immediate statements. Almost any PL/CS statement can be executed "immediately" in the current execution environment. This can alter the values of
variables, move the exec-ptr, produce output, read input, call other procedures. The most common immediate statements are PUT and assignment. Note that both are subject to normal scope rules: they can only access variables in the current environment. The COPE repair facilities are particularly convenient for immediate PUT statements since a variable name alone is automatically "repaired" to a PUT statement. For example, the entry

\[ x \text{ EXECUTE} \]

is equivalent to

\[ \text{PUT LIST( } X \text{ );} \]

(3) Perform other commands.

CONDENSE | EXPAND changes the format of display.

UP | DOWN moves the exec-ptr. This is, in effect, a generalized GOTO. It allows backward as well as forward jumps, and allows jumps to arbitrary statements rather than just labelled nulls.

BACKPAGE | FWDPAGE allows examination of other sections of the procedure. These commands automatically switch the display to the edit screen. This does not move the exec-ptr; when execution resumes the execution screen is restored with the trace area restored to the locale of the exec-ptr.

UNDO undoes the effect of the previous command.

QUIT to terminate the session.

After the completion of execution of a program these same actions can be taken, except of course, that execution cannot be resumed. (Consequently moving the exec-ptr is pointless.)

There are a few situations in which execution can be interrupted in the middle of a statement, rather than between statements: pause for page turn, pause for data input, and execution error. When paused for page turn, execution resumes exactly where it was interrupted, completing execution of the PUT statement that overflowed the output area of the
execution screen. (To avoid completing the PUT statement move the exec-ptr ahead before resuming.) In every other case execution resumes by repeating from the beginning the execution of the statement that was partially executed.

4.7.4. Testing Facilities

Unfortunately most programs contain errors as initially written, in spite of careful development by the user, and enthusiastic error-repair by COPE. Consequently, thorough testing is still required before a program can be convincingly claimed to be correct.

COPE has many helpful facilities for testing. Moreover, since a development system is primarily concerned with new programs, the default state for these facilities is enabled. That is, the testing facilities are automatically active until explicitly disabled. Thus neither effort nor knowledge on the part of the user is required to exercise these diagnostic facilities.

Specifically, these default facilities are the following:

1. Procedures are traced to 2 levels of indentation.

2. All variables are checked.

3. The system is slowed (at least one-half second between screen updates) so the progress of execution can be visually tracked.

During testing, the user can alter these actions using immediate TRACE, NOCHECK and SLOW statements. When testing of a procedure is complete, and the user is satisfied as to its correctness, these facilities can be disabled by including TRACE, NOCHECK and SLOW statements in the body of the procedure.

Note that the TRACE, NOCHECK and SLOW statements are completely local — they affect only the treatment of the procedure in which they appear (either stored or immediate). This facilitates testing on a procedure-by-procedure basis.

Note also that incomplete procedures can be executed for testing. Execution will proceed until a missing element is encountered.
CHAPTER 5

Differential Parsing

5.1. Overview

COPE provides a very tolerant programming environment by repairing errors and supporting simple, language-cognizant program maintenance. In particular, COPE helps to construct the program correctly rather than simply check the correctness of the finished product. Previous systems have demonstrated this type of facility [28, 64, 58]. COPE moves a step farther by providing a comparable facility without requiring the user to learn a substantial language to control construction, or to be subjugated to a rigid construction protocol. Differential parsing is the key to the implementation of this facility.

Providing an effective program entry interface requires facilities not normally associated with parsing. Differential parsing (DP) is an attempt to unify the mechanisms required to provide effective error repair, correct program construction and program editing operations. In this sense, DP is the underlying discipline for all operations performed by the Procedure Syntax Editor (PSE). To support the spectrum of functions provided by the PSE, two separate disciplines of parsing theory must be considered: error correction and incremental parsing. Although the two disciplines interact in determining the characteristics of DP, the concerns of the two areas can be largely segregated for the explanation of the principles involved.

To place DP in context with previous work in error-correction and incremental parsing, each of these areas is discussed briefly with special emphasis on the goals and assumptions of each and their applicability to interactive development. This is followed by a theoretical explanation of the basis for DP and an elaboration of the steps involved in developing the parser from a grammar specification. Finally, the practical issues involved in using the parser to make an effective program entry environment are discussed.
5.2. Error Correction

Syntactic error detection or correction has long been an active area of parsing research. Ideally, a parser is expected to report all errors present (and none that are not present). A variety of approaches have been taken to error correction, each with certain advantages and disadvantages.

The most general error correction methods are based on “minimal distance” considerations. Presented with a candidate program, these methods find the program that is “closest” to the candidate and is a member of the language specified by the grammar [39, 55]. Two serious problems arise with these methods: (a) they are slow, $O(n^3)$, and (b) the formal minimal distance criterion may not correspond to the programmer’s intuitive “closeness” criterion (“equivalent” alternatives may not even resemble each other). Since there are adequate linear-time parsing methods, the time required by these methods is considered prohibitive. Further, the additional complexity is incurred independent of the presence of errors in the candidate program.

Among the more successful error recovery techniques in practice have been implementation-specific routines that incorporate the implementor’s knowledge of the language and the projected user audience to correct a set of “expected” errors [16, 26]. The specific techniques used vary with the particular language and implementation, but can provide extremely effective repair for common errors and syntactic shortcomings of the target language. They can also fail spectacularly on unusual errors. Perhaps more importantly, these techniques can effectively augment most automatically generated correction mechanism. COPE tries to take maximum advantage of both automatically generated and handwritten error recovery.

Most recent results in error correction have focussed on grammar-based correction mechanisms that do not degrade parsing performance on correct programs [37, 38]. Most recently, the focus has been on correction mechanisms that are themselves linear. Thus, though error correction may increase parsing time, it does not force parsing into a higher complexity
The Graham-Rhodes method [27] for error recovery in bottom-up parsers centers on the use of "phrase-level" recovery to locate an appropriately local correction to make the program correct. Phrase-level recovery, introduced by Leinuis [37], involves successive reduction of the parsing stack in the neighborhood of an error by the substitution on the stack of a legal non-terminal at the site of the error. This technique requires that common components of the language will be parsed similarly regardless of where they appear in the program. It takes advantage of the contents of the stack to allow locality of recovery to depend on the context required to form the handle rather than some fixed portion of the input string. In the Graham-Rhodes method, this is called the "condensation phase". It is followed by a "correction phase" in which the production of the grammar with the minimally-distant right-hand side replaces the erroneous stack contents. The method provides linear-time correction, a relatively accurate location for the cause of the error and a flexible scheme (using the distance weights for candidate right-hand sides) for controlling the choice of corrections. The method suffers from occasional loss of context in which there exist no right-hand sides that resemble the input remaining to be parsed.

Pai and Kieburtz [44] propose a top-down correction method that is able to regain context in a wide variety of circumstances. Fischer, Milton and Quiring [22] propose a related method, called "insertion-only correction", which involves the insertion of symbols to "construct" a correct program from the user input. Both methods are based on LL(1) parsers.

The Pai-Kieburtz method separates error recovery into two phases: local and global. Local recovery depends on bounded lookahead to detect simply-determined single-symbol errors. Global context recovery depends heavily on the fiducial (trustworthy) symbols of the programming language to provide correction candidates when local recovery fails. A significant portion of their effort is dedicated to the careful description of the desirable characteristics of fiducial symbols to be used in this process. The global context recovery algorithm is
guaranteed to generate a "correct" context, even if it requires restarting the parse with the
original goal symbol. The combined method compares favorably with other methods reported,
largely on the the strength of the global recovery facility. DP makes extensive use of keywords
(and selected punctuation) to establish global context for parsing the program. In contrast to
the Pui-Kieburtz method, DP uses structurally significant keywords to control the parse rather
than to recover context.

Insertion-only correction is an attractive way of achieving correct programs solely by
adding to the user's input. Insertion-only correction calculates the cost of insertions to repair
errors. In an interactive environment, cost functions to determine the appropriate insertion can
easily be extended to provide a threshold value beyond which insertion is abandoned. In
response to an error, appropriate terminals are inserted into the parse stack, and at the front of
the input, to simulate the presence of the least expensive text that leads to a correct program.
A drawback of the method is that it is unable to take advantage of the fact that all I.L.(1)
grammars are strong, because to do so would allow the parse to continue beyond the site of an
error in a way that makes correction infeasible. As a result, parsing tables can become larger.
DP uses similar characteristics of I.L.(1) grammars, though in a different presentation, to do
insertion-based correction within the framework of a strong I.L.(1) parser. In addition, by
limiting the size of the grammar over which the technique is applied (by separating parsing into
levels), it is possible to keep the dimensions of the I.L.(1) tables small.

5.3. Incremental Parsing

Incremental parsing is an attempt to limit the amount of work required to reparse a
program. The complexity of the reparsing is a function of the amount of text modified. The
Ghezzi-Mandrioli [25] method relies on the correct prefix (both right-to-left and left-to-right)
property of L.R(1) ∩ R.I.(k) languages to generate a fast reparse around a local change. The
Wegman [61] technique for reparsing modified programs relies on saving the state of the parse
at nodes of the parse tree. In this way it is necessary to reparse text only where it has been
changed and at positions in the parse tree when nonterminals no longer match. Both of these methods assume that a complete parse tree will be maintained at the time of the modifications. COPE stores programs in an intermediate form similar to a parse tree, but more efficiently executable.

However, incremental parsing normally includes an assumption about program changes that COPE does not share: that the parse of the program should be independent of the history of changes that lead to it. That is, modifications to the program deal directly with the program text and the new parse must be the same as the one that would have resulted if the resulting text had been entered that way originally. The broad interpretation that COPE gives to newly added program text makes this interpretation both too narrow and inconsistent with its treatment of initial entries. Furthermore, the most complex cases for incremental parsing are precisely those changes that are most likely to confuse novice users and least likely to result a resulting correct program. In particular, the DP use of structural terminators as sign posts in the user entry process makes it impossible to justify rearranging the entire program structure for a single (perhaps ill-advised) keyword insertion.

5.4. COPE Environment

The COPE entry discipline is unique, and it imposes a unique outlook on the parsing methods to be used. The factors in determining this outlook and the effect that each has had on the construction of the parser are presented in Figure 5.1.

Our primary objective is to be able to support interactive program development with a system that is flexible and tolerant as well as effective. The program entry method envisioned guarantees that the structure of the program is complete and correct at all times. For example, there are never missing END statements which would need to be entered later to make the program complete. On the other hand, the expressions that form the content of a particular statement are allowed to be absent (though such incomplete statements cannot be executed). In this way, it is possible to support top-down program development in which the complete
(1) The user will be taking active part in the entry process.

(2) The user is not expected to enter syntactically correct programs. The combination of extensive error correction and immediate response changes the purpose of the parser. We expect that each user will develop an individualized shorthand for program entry based on experience with COPE corrections. This implies that a reliance on the details of syntax will be ineffective in guiding the parse. The construction of the differential parser from a grammatical specification of the language thus involves the extension of the language recognized to a compatible superset.

(3) The system should support the entry of incomplete statements. While it is possible to provide significant assistance in the entry of program structure, it is difficult to synthesize useful contents (expressions, assignments, etc.).

(4) The basic mode of program entry is assumed to be incremental insertion. However, presentation of a complete procedure text should be allowable. Each entry, regardless of size, should result in a program that is correct and executable to the maximum degree possible. This goal implies that syntactic structures involving more than one statement must be completed by the parser.

(5) Program entry should be independent of parser insertions. The user must be able to enter the program without making special provisions for items that have been inserted to complete its structure. This implies that structural brackets (e.g. a matching END) cannot be inserted where they are entered and must be searched for as sign-pots during parsing. Having the parser provide structural brackets overcomes a major obstacle to truly interactive use of structured languages. Successful interactive languages (e.g. BASIC and APL) typically do not have any real structure to enter.

(6) Parsing must be grammar-based. Language-specific corrections may be added, but the main error-repair algorithm must be based on the grammar of the particular target language.

Figure 5.1 — COPE Parsing Assumptions

structure is laid out before the detailed contents of the statements is known.

5.5. Hierarchical Specification

The key to DP is the separation of the language to be parsed into appropriate sublanguages, each of which can be effectively repaired on its own terms. This separation is effected by partitioning the source language alphabet in such a way to differentiate structure from content.

A natural way to present the partition is as a hierarchical specification. The top-level specification, called the “structural syntax”, deals with the statements of the language. In
particular, the structural syntax emphasizes keywords that guide the construction of programs. A lower-level specification, the "expression syntax," describes the content of statements. A separate parser is constructed for each of the specifications. The method does not require that there are exactly two levels of specification, but two are sufficient for exposition.

The impetus behind the separate specifications is to make it easier to provide repair appropriate to each level of the language. The selection of the particular parsers to be used at each level is a somewhat independent issue. First, we will describe the criteria for creating a useful partition of the language to be parsed and the implications of the partition on the parsing problem. The structural parser used in COPE is described in some detail. It is an improvement on existing techniques that takes advantage of the partition that has been effected.

5.5.1. Separating the Parsers

Separating parsers into more than one level is not new. Korenják [36] proposes a method for splitting LR parsers to reduce their size. Lexical analysis has long been treated as a separate level of the parser. DP generalizes this separation by constructing the expression parser to act as a high-level lexical scanner for the structural parser.

Lexical analysis can be represented by a finite-state transducer that accepts characters as input and creates a string of tokens from the alphabet of the parser as output. The expression parser accepts the tokens from the scanner and creates a string of tokens (from a different alphabet) for the top-level parser. It converts a series of tokens made of up operators, keywords, constants, identifiers, etc. into a series of tokens consisting of keywords, significant punctuation and expressions. Such a separation is possible for a variety of practical programming languages. It is clearly not possible to parse full PML, which does not reserve its keywords, using this method.
5.5.2. Partitioning the Grammar

Define a grammar \( G \) as \((N, \Sigma, P, S)\), the nonterminals, terminals, productions and starting symbol, respectively. A hierarchical specification for \( L(G) \) consists of two grammars \( G_S \) and \( G_E \) that can be used to generate \( L(G) \). Let \( N_E \) be the set of nonterminals for \( G_E \), etc. Intuitively, \( G_S \) represents \( L(G) \) in terms of \( N_S \) and the "expressions" represented by \( L(G_E) \).

\( G_S \) and \( G_E \) are derived from a partition of \( \Sigma \). Typically, this means that keywords are in \( \Sigma_S \) and everything else is in \( \Sigma_E \). Partition \( N \) (and correspondingly \( P \)) so that:

\[
\begin{align*}
A & \in N_S \text{ and } A \rightarrow \alpha \implies \alpha \in (N_S \cup N_E \cup \Sigma_S)^* \\
B & \in N_E \text{ and } B \rightarrow \beta \implies \beta \in (N_E \cup \Sigma_E)^*
\end{align*}
\]

For some grammars and choices of terminal alphabets, this partition is not unique; this presents no problem. There is no guarantee that such a partition exists for any particular \( G \) and alphabets \( \Sigma_S \) and \( \Sigma_E \). (Any \( G \) can be partitioned by using \( \Sigma_E = \{\} \), but this is hardly useful.) In practice, a useful partition has been found for PL/CS and similar subsets of Ada and Pascal, with few modifications to the original grammars.

Productions in \( P_S \) now contain references to nonterminals in \( N_E \) that are not resolved in \( G_S \). Two solutions are available:

1. The \( G_S \) parser can pass \( G_E \) a goal symbol for its parse. This suggests a top-down parse of the expressions to meet the goal symbol.

2. The \( G_E \) parser can parse the available expression text bottom-up, providing a terminal to \( G_S \). \( \Sigma_S \) must be augmented by any member of \( N_E \) that appears in a production in \( P_S \). Exactly this subset of \( N_E \) are the acceptable set of expressions for the \( G_E \) parser.

A top-down parser for an attribute grammar could also provide structural information for the parse of lower-level terminals in the latter form. An important advantage of the bottom-up
approach is the detection of multiple expressions within the same contiguous set of $\Sigma_E^*$. To continue the lexical analysis analogy, the $G_E$ parser becomes a push-down transducer that accepts input from $\Sigma^*$ and produces a string from $\Sigma_S^+ (\Sigma_E^* \Sigma_S^+)^*$, assuming distinguished endmarkers in $\Sigma_S$. The definition of $G_E$ now involves a set of goal symbols consisting of exactly the set of former nonterminals from $N$ that became terminals in $\Sigma_S$.

Consider a parse tree for some string in $L(G)$. The partition of $G$ into $G_S$ and $G_E$ also partitions the parse tree. The portion of the tree corresponding to $G_S$ consists of the root and all its descendants up to and including nonterminal nodes all of whose descendants are in $\Sigma_E \cup N_E$. The portion corresponding to $G_E$ is the forest of subtrees beginning at each of these nonterminals. From a parsing point of view, each pair of adjacent terminal symbols from distinct alphabets marks the transition from leaves of the tree for $G_S$ and leaves for a subtree for $G_E$. Note, however, that there is no guarantee that any particular string of contiguous symbols in $\Sigma_E$ belong to the same subtree.

The structure of commonly used programming languages is such that adjacent expressions never occur without intervening keywords or punctuation. On the other hand, the correction/repair process will not be able to depend on this characteristic, because keywords that do intervene between expressions are likely candidates for omission by the user. Wherever adjacent expressions do occur, the original grammar must have been sufficient to detect the end of one expression and the beginning of the other. For this reason, it is desirable to retain each of the separate goal symbols corresponding to the communications with $G_S$, thereby retaining the recognition power of the original grammar. It is clearly possible, using nondeterministic methods, to develop a partition without retaining this information, but this increases the complexity of the parsing algorithms, without providing any additional power. It is counterproductive to allow $G_E$ to become difficult to parse.

We argue that the hierarchical arrangement of parsers produces a parser that recognizes any legal program. No claim is made that the resulting parser can detect all possible errors; it
may also recognize incorrect programs as if they were correct (a common "problem" with Pascal parsers discussed in [32]). Since our goal is to create a parser that generates a legal parse from whatever input is provided, we ignore this problem. For correct programs, each contiguous string from $\Sigma_E$ is guaranteed to be legal in $G_E$. Construct the $G_E$ parser to use terminals from $\Sigma_S$ as endmarkers for expressions. In its role as a push-down transducer, each expression is output as a terminal of $G_S$ and any terminal not in $\Sigma_E$ is passed along untouched.

5.6. Correction and Specification

The separation of the grammatical specification into structural and expression levels makes it possible to give human-readable grammars. Figure 5.2 is an example of a portion of an structural grammar for the imperative statements for PL/CS. Optional phrases are enclosed by "[]": phrases repeatable zero or more times are enclosed by "{}". Nonterminals are shown in lower case. Tokens from the expression grammar are delimited by "<>".

This grammar has the following attractive characteristics: it is easy to read; an equivalent LI(1) grammar (and parser) can be easily generated for it (ignoring the ambiguous ELSE, which can be shown to yield the desired result); it captures the basic structure of the language components. For the current purpose, its most important characteristics is that each of the

\begin{verbatim}
proc -> PROC; stmt-list END;
stmt-list -> {stmt}
stmt -> <assign>; | if-stmt | select-stmt | do-stmt
if-stmt -> IF <expr> THEN <stmt> | IF <expr> ELSE <stmt> |
select-stmt -> SELECT; {WHEN <expr>; <stmt> } OTHERWISE <stmt> END;
do-stmt -> DO [ iteration ] ; stmt-list END;
iteration -> WHILE <expr> | UNTIL <expr> 
            | <assign> TO <expr> | [BY <expr>]
\end{verbatim}

Figure 5.2 – Sample Grammar: Imperative Statements of PL/CS
keywords appears exactly once in the grammar and consequently can be used to force the insertion of the construct in which it appears. The unique reference to keywords makes it possible to fill in the empty LL(1) table entries in a way that provides useful repairs.

5.7. Top-Level LL(1) Parser Construction

The current COPI top-level parser is derived from an LL(1) parser for the structural syntax. The top-level parser need not be LL(1), but the emphasis on structure makes a top-down parser a natural choice. Furthermore, both the original insertion-only correction [22] and our modification make use of the predictive nature of I.L. parsing. As presented, the original algorithm relies on a weak LL(1) parser to make sure that the stack contains sufficient information to make the error-search provably successful. The LL(1) parser itself is not modified.

The COPE top-level parser performs an almost equivalent insertion-based correction by modifying the LL(1) parser tables to incorporate correction actions into a strong LL(1) parser. This approach generates very good correction from a considerably smaller set of parsing tables.

An I.L.(1) parsing table consists of a row for each terminal or nonterminal that can appear on the parsing stack and a column for each terminal which can appear as the input lookahead. The separation of the parser into two levels reduces both dimensions considerably in comparison to tables for the entire language. The entries in an LL(1) parsing table are very simple. For any pair of symbols, it is possible to push a production onto the parsing stack, pop a symbol from the top of the stack (clearing the input lookahead), accept the string, or indicate an error. The productive entries in the table are relatively sparse for most grammars, with error entries predominating. (This is typically true even ignoring the fact that all entries which correspond to distinct pairs of terminal symbols are errors.) This corresponds to the fact that very few of the possible combinations of symbols are legal in the parse of real programming languages. By filling in these “unused” error entries with correction actions based on the grammar, it is possible to encode correction directly into the parser while still guaranteeing that
the parse of correct input is unaffected.

The parsing stack in an LL(1) parser is increased whenever the input lookahead requires a new production. In insertion-only parsing, correction is performed by inserting new input symbols in front of the current lookahead. When an error is encountered, the parsing stack is searched for an appropriate set of insertions to make the augmented program correct. The COPE top-level parser uses the "uniquely implied" criterion to push new productions in circumstances in which it is "known" that the matching terminal does not exist on the stack. As a result, the standard parsing action when faced with two distinct terminals is to "ignore" the one on the stack, forgiving the terminal's absence from the input. This is equivalent to inserting the terminal at the front of the input.

In order to construct the parser tables, a few functions are required (in addition to the standard FIRST and FOLLOW [2]). The function WITHIN defined in [44] determines which production should be pushed when a (nonterminal, terminal) pair would otherwise generate an error. Intuitively, WITHIN(Λ, t) is true when there is some derivation from the nonterminal Λ which includes the terminal t. A new function, RIGIT, encodes the information that would otherwise be found by searching the stack. RIGIT(t) is true if the terminal t has been "predicted" by the current state of the parse.

The predicate WITHIN determines which production is pushed in response to a terminal symbol. Define WITHIN: Σ_Σ ∪ Σ_Σ × Σ_Σ → {true, false} according to [44] so that WITHIN(Λ, t) is true if Λ can derive a string containing t; false otherwise. Note that if Λ is a terminal, WITHIN(Λ, t) implies Λ = t. WITHIN(Λ, t) is a statically determined by G_Σ. When a terminal t is WITHIN the current top of the stack, it is possible to push productions which will lead to t being recognized.

The appropriate correction for "out of place" depends on whether the terminal has been predicted by the current state of the parse. In insertion-only correction, this function is carried out by searching back through the stack for the minimum cost element for which
$\text{WITHIN}(\text{element}, t)$ is true. The function $\text{RIGHT}$ performs the same function without searching the stack for a large class of corrections. Whenever a production is stacked, each of the terminals in the production is “predicted” by increasing a counter; when the parse moves past the terminal (or completes the production), the count is decremented. Define $\text{RIGHT}$:

$$\Sigma \rightarrow \{\text{true, false}\}$$

so that $\text{RIGHT}(t)$ is true if $t$ appears to the right of the current position in any production that has not been completed. Table entries will be defined below whose parsing action depends on $\text{RIGHT}$.

Consider what “should” happen for any pair of symbols $(S, t)$ where $S$ is the top of the stack and $t$ is the input lookahead terminal. If $(S, t)$ has a defined function in a normal LL(1) parser, use that. If $\text{RIGHT}(t)$ at any point, the appropriate action is to remove $S$ from the stack, eventually causing $t$ to match the top of the stack. Otherwise, the action depends on $\text{WITHIN}(S, t)$ in the manner shown in Figure 5.3. By defining appropriate operations which use the current value of $\text{RIGHT}$, it is possible to describe table construction solely on the basis of the grammar presented.

---

**f $\text{WITHIN}(S, t)$ is true:**

By definition of $\text{WITHIN}$, $S$ derives $t$ in some finite number of steps. Push the first production derived from $S$ which includes or derives $t$. If there is not a unique derivation of $t$ from $S$, one can be chosen on the basis of any desired criterion. It is also possible that $t$ is sufficiently unimportant that it should be cleared rather than replacing $S$.

**f $\text{WITHIN}(S, t)$ is false:**

Either $S$ should be popped from the stack or $t$ should be cleared from the input; which should happen depends on the nature of $t$. If $t$ is important (appears uniquely, for instance), $S$ should be popped.

---

**Figure 5.3 — Effect of $\text{WITHIN}$ on DP Parsing Actions**

---

The COPE parser uses three new parsing actions and modifies the meaning of a fourth, $\text{top}$. These actions are defined in Figure 5.4.
(1) pop. The default action where both \( S \) and \( t \) are terminals. If \( S=t \), clear \( t \) from the input lookahead; regardless, pop \( S \) from the parsing stack.

(2) ifpush(\( \alpha \), i). If \( \text{RIGHT}(t) \), pop \( S \); otherwise perform a push(\( \alpha \), i).

(3) emit. For non-terminal \( S \), pop \( S \) from the stack and perform actions necessary to indicate that it was not recognized.

(4) ifemit. If \( \text{RIGHT}(t) \), emit; otherwise clear \( t \) from the input lookahead.

The "standard" operations are:

(5) push(\( \alpha \), i). For the \( i \)th production, \( A \rightarrow \alpha \), replace the top of the stack with the representation of \( \alpha \) and add \( i \) to the output.

(6) accept. Successful completion of the parse.

(7) error. An error has been encountered which will not be repaired, ignore the current input symbol.

Figure 5.4 — DP Parsing Actions

5.7.1. Creating the Parsing Tables

The table \( M \) produced by the following modification of the normal LI.(1) parser construction differs only in that:

(1) error entries have been replaced with new or modified actions, and

(2) pop has an expanded meaning which makes (terminal, terminal) entries in the table unnecessary.

Neither change alters the parse of correct programs. The input grammar is assumed to be an LI.(1) form of the extended grammars presented above. The algorithm for creating the parsing tables is presented in Figure 5.5. The description in (4) is not intended as an algorithm; there are more efficient ways of performing the closure. Entries filled in with error in (5) correspond to language constructs that are not insert-correctable LI.(1). An example from PL/CS would be a declaration in the middle of a group of statement. (COPE accepts the declaration and inserts it at the end of the previous declarations.)

The construction for \( M \) works quite well except for non-unique terminals. For example, the terminal for "expression" appears in such a variety of contexts that it provides very little
Input. Given an LL(1) grammar $G = (N, \Sigma, P, S)$.
Output. Generate $M$, a correcting parsing table for $G$.
Method. Assume that $\$ is the bottom of stack marker. $M$ is defined on $(N \cup \Sigma \cup \{\$\}) \times (\Sigma \cup \{\lambda\})$ as follows:

1) If $A \rightarrow \alpha$ is the $i$th production in $P$, then $M(A,a) = \text{push}(\alpha,i)$ for all $a$ in $\text{FIRST}(\alpha)$, $a \neq \lambda$. If $\lambda$ is also in $\text{FIRST}(\alpha)$, then $M(A, b) = \text{push}(\alpha,i)$ for all $b$ in $\text{FOLLOW}(A)$.

2) $M(a,b) = \text{pop}$ for all $a$ and $b$ in $\Sigma$.

3) $M(\$, \lambda) = \text{accept}$.

4) For each $t$ in $\Sigma$, where $M(A,t)$ is not already defined above:
   For each $\Lambda$ in $N$ where $A \rightarrow \alpha t \beta$ is the $i$th production, $M(A, t) = \text{ifpush}(\alpha t \beta, i)$.
      For any $B$ in $N$ where $M(B,t)$ is still undefined, $M(A,t)$ is defined, and $B \rightarrow \gamma \Lambda \delta$ is the $j$th production, $M(B, t) = \text{ifpush}(\gamma \Lambda \delta, j)$. This process terminates when $\text{WITHIN}(A,t)$ implies that $M(A,t)$ is defined.
      For each $\Lambda$ in $N$ where not $\text{WITHIN}(A,t)$, $M(A,t) = \text{emit}$.

5) For all $X$ and $a$ for which $M(X,a)$ is not otherwise defined, $M(X,a) = \text{error}$.

Figure 5.5 — DP Parsing Table Algorithm

Indication as to what a reasonable correction should be. COPF places any unattached expressions into PUT statements (this makes inspection of values for debugging convenient), but this choice is clearly not derived from the grammar.

The interface with the expression parser provides an excellent opportunity for single-symbol lookahead which resolves many ambiguous expression occurrences. In order to recognize an expression, the expression parser must already have encountered a keyword (or the end of the entry). Under these circumstances, it is quite simple to resolve ambiguities by a single-symbol lookahead in the expression parser. It is hard to tell from the examples presented in [22], but insertion-only correction seems to have similar problems, even though some additional information is available. Of course, an insertion-only-based top-level parser could also be created from the grammar. More space would be used for tables (though the two-level specification keeps the tables quite small), but the extensions described could be adapted to the alternate LL(1) parser.
5.8. Alternative Parser Choices

The hierarchical separation of the parser admits a wide variety of choices for both the structural and expression parsers. COPE uses the modified insertion-only parser presented above for the structural parser, though the version presented in [22] could also have been adapted quite easily. Two expression parsers have been used: an initial yacc-based [33] version which performed almost no correction, and subsequently a version of the PL/C expression parser [16].

One of the advantages of the hierarchical parsing separation is that it partitions the input stream into short segments that can be handled separately. This suggests the possibility of using a minimal-distance correction mechanism in one or both of the parsers.

For the expression parser, the shorter segments make the $O(n^3)$ penalty tolerable and the enforced locality of correction makes it extremely likely that correction will seem reasonable to the user. (Teitelbaum [55] presents a minimal-distance algorithm for FORTRAN expressions that is $O(n^2)$; this seems applicable to PL/CS expressions, but not to PASCAL or Ada.) For interactive purposes, the speed of the parser is less relevant than the quality of correction provided (so long as perceptible delays are not created). Further investigation is warranted to test the conjecture that a minimal-distance expression parser can be constructed that requires only moderate fixed storage.

By removing the details of expressions, the size of the strings handled by the structural parser is also reduced. In normal use, it is expected that the user will enter very few keywords in each individual entry. This reduction in the size of problems makes minimal-distance a possibility for the structural parser, especially if insertion weights can be designed to assure that "uniquely implied" structures are inserted. A possible advantage of this approach would be the ability to use the same parser (with different tables) at each level to achieve the different goals of each specification.
5.9. Insertion and Modification

Program entry and modification requires somewhat more than the correcting parser described above. The ability to handle random insertions, deletions and modifications is necessary for the COPE entry system; none of the parsing decisions taken would be reasonable in a linear-entry, full-text parser. It is here that the error correction and incremental parsing disciplines combine to make up DP.

5.9.1. Entry Discipline

The basic forms of user entry used in COPE are insertion and replacement. The COPE method for altering an existing unit (a statement or logical group of statements, see 4.1) is to "fetch" the unit into an editing area, line-edit text changes, and "replace" the original. Fetched and user-entered text are treated identically; both can be used for either insertion or replacement. Replacement is logically equivalent to the deletion of the replaced unit, followed by the insertion of the replacement text, though no deletion operation exists. Deletion can be accomplished by replacement with a "cleared" edit area.

The important characteristic of this discipline is its consistency. Existing procedure elements are always fetched or replaced as units that are defined to maintain program structure. User entries are always treated as text to be parsed for inclusion into a syntactically correct procedure. This treatment permits arbitrary structural changes to be made in the fetched text, since the text of the new insertion is reparsed as a new entry. Insertions can never alter existing structure; replacements can never alter structure beyond the current entry.

Regardless of the type of change that is made, all COPE parsing consists of correctly parsing user text for insertion into an existing program structure. Consider the prospect of "moving" the END associated with a loop up into the body of the loop (or, equivalently, moving statements at the end of the loop outside of the loop). This is accomplished by fetching the loop, inserting the END at the appropriate place, and replacing the loop by the edited version. Parsing the replacement text is very simple; the input text has already been
corrected during original entry. The operation would be faster if a special "reparser" were implemented to reclaim the internal form of unmodified text, but the added implementation complexity seems unwarranted for the small improvement in efficiency. Such an addition would be more urgent if parsing correct text were slow enough to introduce noticeable delays.

It is also possible to effect more drastic changes, for instance changing an UNTIL or indexed loop, to an IF or SELECT statement. In each case, the entire unit is fetched, the change to the initial portion of the structure is made and the unit replaced. As for any insertion, the correction mechanisms will assure that the new entry is structurally complete.

The reparsing of fetched text can create small surprises, however. The deletion of the header of a loop (or other compound unit) can leave vestigial keywords in the text to be reparsed (deletion of DO WHILE can leave a corresponding END, IF can leave THEN and/or ELSE, etc.). The parser will interpret each keyword as if it were entered directly, possibly giving it more importance than the user anticipated.

5.9.2. Updating the State of the Parse

The entry discipline used in COPE aids the parser in two ways: it guides the user into structured changes by dealing with structural groups rather than isolated character sequences, and it conditions the user to think in terms of the type of orderly change that will be provided. The latter is very important in determining the type response to particular entries, e.g. isolated END statements. By convincing the user that the structures will be treated as a whole, COPE makes it possible to avoid the more complicated cases of incremental parsing.

In incremental parsing, the problem is how to minimize the time spent reparsing the program when changes are made under the assumption that the result of a series of text modifications is independent of their history. In this context, the problem is to detect which changes are local and which modify the entire structure of the program, as with the insertion of an END in the middle of the program. This discipline assumes that the text of the program is entirely as the user has entered it, and the parsing problem is merely to create a correct parse
with each change. Since COPE actively modifies the program text by its repair mechanisms, it is not clear that the criterion for incremental parsing still makes sense in the narrow technical sense; it certainly is not consistent with the goals of the system.

Many incremental parsing concerns are retained in COPE, however. The state of the parse must still be reconstructed at each modification to ascertain the appropriate action. In COPE, this is done by maintaining the state of the parse with respect to the internal tree form of the program. Reconstructing the parse from a position in the internal tree involves generating the predicted terminals and non-terminals between the current cursor position and the end of the program.

The state of the parse at a particular statement consists of local and global components. The statement component consists of terminals and prompts (unmatched nonterminals) in the current statement. The global component consists of keywords and prompts in containing structures. It is important to notice that statements on the same (or lower) logical level as the current statement are irrelevant to the parse. The local component is available with each statement, since the program is maintained as parse tree. To facilitate determining the global structure, each statement is linked to its containing construct. In this way, the reconstruction of the parse state consists of extracting local state from the current statement, traversing the parent link to the containing statement and repeating the process. The number of steps involved is proportional to the depth of nesting of the control structures of the program.

One of the principle tasks of the parser is to detect the difference between entries that imply the construction of a new structure and those that match predicted structures provided by the parser. The parser detects this difference using the RIGHT function, which simply records the presence in the parsing stack of the particular entry. When an entry has been predicted (exists on the stack), the COPE parser simply pops the stack until the matching entry is found. This prediction mechanism works because the parser is willing to ignore keywords and nonterminals that are not matched. This is inherent in the table construction of Figure 5.5.
Thus, once the parsing stack has been initialized to predict the visible portion of the program that remains, entry of keywords that occur in the stack simply causes the stack to be popped (and the current position in the tree to be changed) until the keyword is matched. In this way, it is possible to decide whether to create new statements or move the cursor. This mechanism works for any symbol in the structural grammar. For PL/CS, it is most commonly useful in finding ENIY's, since there is always a containing END at any position in the program. (This characteristic is enforced by the parser so that it is never possible to enter text past the END of the containing procedure.)

An important characteristic of this method of constructing the parsing stack is that it is possible to construct a parsing stack during modifications that would not be valid in any parse which originated from the initial entry of the program. This makes sense in the context of making changes in the middle of programs and corresponds to filling in prompts in statements which are otherwise already complete. Many of these cases correspond to situations where the COPE parser will move the cursor, repositioning entry, where a traditional incremental parser would try to reconstruct the entire program or signal a terminal error. Within the context of COPE entry, these actions are entirely natural.
CHAPTER 6

Interleaved Execution and Modification

Interactive programming environments offer the opportunity to modify active programs and resume execution without starting over from the beginning. Within the context of the development process, this makes it possible to make necessary changes as they become evident without abandoning the current execution session. The flexibility required to provide this ability is usually associated with interactive languages. Exactly those safety characteristics that make structured, strongly-typed languages attractive also make them more difficult to modify and resume.

The most obvious drawbacks to interleaved execution and modification involve efficiency. A compiler transforms program text into a semantically equivalent form that executes efficiently. The advantages of the compile-time checking provided in structured languages are less apparent when runtime program modification is possible. Ideally, it is desirable to be able to provide arbitrary runtime modification with no loss of efficiency, but it is not always possible to satisfy both criteria.

A criterion that must be satisfied is correctness. In a compiled environment, correctness simply implies that the program text and its corresponding compiled form are both legal representations of the same program. Modifying a program, by itself, introduces no new semantic problems; attempting to resume its execution does. During execution, the program establishes an execution environment in which the meaning of each successive statement is determined. Program text modifications can create inconsistencies between the meanings of statements and the current active environment that could not occur with a static program text.

The purpose of this chapter is to define an informal semantics for modifications, called modification semantics, that is sufficient to determine which statements in a program have
become inconsistent with the active environment, and hence unexecutable.

6.1. Previous Work

Many of the problems in detecting inconsistencies are directly related to the desirable characteristics of structured, strongly-typed languages. Few structured-language systems have allowed interleaved execution and modification, and the semantic problems for interactive languages are not complicated enough to encourage careful study. As a result, little literature exists in this area.

The JOSS system was probably the first system to provide fully interactive modification [51]. The JOSS language is specifically tailored to interactive calculation, avoiding complexities introduced by systems emphasizing stored programs. Whether by design or serendipity, popular interactive languages — APL, BASIC, and LISP — also avoid most of these problems.

Mitchell [43] presents the only careful discussion of the issues involved in interleaved execution and modification available in the literature. The LCC system that he describes supports an Algol-like language (I.CC) that has been modified to make it more interactive. LCC includes "typeless" variables, but allows type declarations and must handle most of the problems introduced by typing. The treatment of interleaved execution and alteration focuses primarily on implementation issues in an interactively compiled environment. Some consideration is given to legal and illegal changes and their resumption, but this is not done within a uniform framework.

The I.CC criterion for interleaved execution and alteration is given in the following definition:

"Visual Fidelity Principle (VFP): 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)."

Applied to inactive programs, the VFP merely states that the language has a consistent semantics and the implementation is correct. The VFP is a good starting point for discussion, but it is not sufficiently explicit to stand by itself as a guide to correct implementation. The
paper itself proceeds to discuss the implementation of specific types of modifications. The most interesting LCC problem discussed deals with consistent treatment of side-effects in active function calls that are modified. The COPE policy of restarting statements from the beginning and the PL/CS prohibition of function side-effects make this problem moot.

The PL/I Checkout Compiler is probably the only generally available system that allows interleaved execution and modification [46]. Individual procedures can be selected for diagnostic compilation and run in concert with others that are conventionally compiled. For a large subset of language statements (notably not including declarations), it is possible to modify the text of the program and resume execution. Unfortunately, the published reports on the system do not discuss how the system determines which changes block resumption [18, 41].

6.2. Overview

The purpose in selecting a formal system is to provide a uniform representation for reasoning. The representation must include enough detail to address the questions of interest. Interleaved execution and modification are not included in semantic models. Before discussing the characteristics of a modification semantics, consider the relation between modification and traditional semantic models.

6.2.1. Relation to Standard Models

Axiomatic semantics [23, 30] provides a way of thinking about the meaning of a particular program. Each construct of the language has an axiom or inference rule that represents its meaning as a function of the current program state. Program proofs are constructed by using the inference rules to represent the transformation of sets of input states to sets of output states. Strict adherence to this view makes resumption of a modified program unlikely. Specifically, any change to the program would have to be consistent with predicates on the current state of the computation. This view is too restrictive for the current purpose.
Denotational semantics [50] deals with the meaning of programming language constructs instead of particular programs. The meaning of a construct is a function that can be composed with the functions for other constructs to represent the meaning of the entire program. The attempt to define the meaning of the entire program complicates the description of constructs that are well-defined, but whose meanings do not lend themselves to function composition (e.g. GO:TO). Such complications are taken as evidence that the corresponding constructs are not easily understood and should be omitted from the language.

Modification semantics is concerned with the meaning of individual statements in an active environment. Program execution (partial or complete) creates a program state consistent with the semantics of the language. Modifications to the program text can render individual statements incompatible with the active environment. Modification semantics provides a way of determining these incompatibilities. Facilitating resumption (where program and active environment are consistent) is a COPF implementation goal; preventing the execution of incompatible statements is a requirement.

6.2.2. Program Text

All forms of semantics define the meaning of a program text by assigning a meaning or interpretation to the objects in the language. Since the semantics deal with syntactically correct programs, there is no need to deal with the program as a string of characters, and there are obvious reasons for not wanting to. As a result, semantics deals with abstractions from the text — statements, expressions, variables. The remaining discussion deals only with the semantics of syntactically legal modifications. Any statement with compile-time detectable errors cannot be executed, regardless of the resumption discipline.

For a modification semantics, it is important that the level of text abstraction for the model and for the system are compatible. In COPFi, the modification unit is a program statement. The user may change only a single character, but the mechanisms for the change involve replacing the entire statement. As a result, the focus for the corresponding
modification semantics is on statements: This is not to say that finer grain distinctions are not made. It is just not possible to change an individual variable reference or expression without changing the statement that contains it. This distinction only matters when the statement being modified is currently “active”.

Alternative choices can be made for the granularity of the semantics. If EMILY (see 1.4.1) supported execution, modification could easily be expressed at a finer grain since each program token is individually placed in the program. The Checkout Compiler modifies statements, but resumes programs. That is, the minimum grain for changes is a statement, but programs are only resumed if no inconsistencies have been detected for the entire program.

COPE uses statement granularity throughout: changes, semantics and resumption policy all deal with complete statements. Each statement is a distinct, identifiable object in the program. Moving or copying statements creates a new object with a different identity and similar, but not necessarily identical, semantics. Replacing a statement is the same as deleting it and replacing it with a new statement reflecting the changes. This is a conservative interpretation that happens to match the COPE implementation. Incorporating a specific notion of change (instead of replacement) can increase, but not decrease, the number of resumable changes.

6.3. Modification Semantics

Constable and Donahue [12] present a denotational semantics for PL/CS in a particularly useful way (and participated in specification of the PL/CS language to facilitate a clean semantics). They present their semantics in hierarchically, separating the language into subsets that minimize interactions between features and simplify presentation. A new level of the hierarchy marks an increase in the level complexity — the introduction of additional functions or domains. The hierarchy serves a similar partitioning function for the discussion of the modification semantics.
This similarity of the structure of the two semantics is not surprising. Language constructs that complicate the meaning of the program also tend to increase the possible interactions between domains. It is precisely this increase in interactions between domains that complicates modification semantics.

6.3.1. While Schemes

While schemes form the basis for structured programming within a single procedure. Figure 6.1, taken from [12], gives an informal syntax of while schemes. Syntactic classes are written with a leading capital letter and members of the class in all lower case. For example, "cond" is a member of the class "Cond" of conditions. Elements of the class of simple statements will be defined as needed. The iterative DO, DO UNTIL, and SELECT statements can be syntactically transformed into combinations of other PL/CS statements and will not be discussed further.

The "meaning" of a statement consists of two components: action, what the statement does, and sequencing, how the next statement is selected. While schemes are concerned with statement sequencing. The sequencing portion of all simple statements is "select the statement immediately following this one in the program text", "the next statement" for short. The compound DO; ... END; are also sequentially sequenced.

\[
\begin{align*}
\text{Stmt} & = \quad \text{SimpleStmt} \\
& | \quad \text{IF ( Cond ) THEN Stmt; ELSE Stmt; } \\
& | \quad \text{DO WILLIF ( Cond ); Stmt; END; } \\
& | \quad \text{Stmt; Stmt; } \\
& | \quad \text{DO; Stmt; END; }
\end{align*}
\]

Figure 6.1 — While Schemes
The DO WHILE and IF statements depend on the value of the associated condition. We defer discussion of the meaning of these conditions until later, relying on intuitive notions of how conditions are evaluated.

The sequencing for the DO WHILE statement is "if the condition is true, select the next; if the loop condition is false, select the statement immediately following the END corresponding to this DO WHILE." The sequencing for the END associated with a DO WHILE is "select the corresponding DO WHILE." The multiple uses of END in PL/I make it necessary to assign differing semantics to the same apparent statement. Since there is a unique correspondence between END statements and the construct terminated, this is only a minor inconvenience.

Sequencing for the IF statement is complicated by the absence of explicit bracketing constructs for the components of the IF. One alternative is to have the IF statement modify the sequencing characteristics of statements subject to the IF. This is an unattractive complication. By inserting "invisible statements", that express the logical character of the IF selection, it is possible to define a static sequence. Figure 6.2 shows the structure of the IF-THEN-ELSE statement and the logical position of the new statements: endTHEN and endIF. The endTHEN and endIF statements never appear in the program, they simply provide a method for simply stating the sequencing for IF. The sequence for IF is "if the condition is

```
IF (cond) THEN DO;
  stmt1;
END;
ELSE stmt2;
```

Figure 6.2 - "Invisible" Sequencers in IF Statements
true, select the next statement; if the condition is false, select the ELSE statement." The sequence for both endTHEN and endIF is "select the statement immediately following the endIF statement."

By defining the semantics in terms of statements as they appear at the time the sequencing choice is made, statement insertion, deletion and replacement introduce few problems. Consider the example in Figure 6.3(a). The execution cursor, "->", marks the next statement to be executed; the statement numbers are for reference. Consider the effect of adding S3.1 after S3, S6.1 after S6, and replacing S5 with S5.1 yields 6.3(b). The sequencing for each of the statements is well-defined and exactly matches the appearance of the modified program.

Two types of changes are potentially complicated: altering the structure of a compound statement (deleting S6), and removing the statement pointed to by the execution cursor (S2). Altering the structure of a compound statement only introduces new complications by

```
(a)   L3: DO WHILE ( I < MAX);   S1
     ->   IF ( A(I) < A(I+1) )   S2
          THEN DO;               S3
          A(I) = A(I+1);          S4
          A(I+1) = A(I);          S5
          END;                   S6
     END L3;                   S7

(a)   L3: DO WHILE ( I < MAX);   S1
     ->   IF ( A(I) < A(I+1) )   S2
          THEN DO;               S3
          T = A(I);              S3.1
          A(I) = A(I+1);         S4
          A(I+1) = T;            S5.1
          END;                  S6
          I = I + 1;             S6.1
     END L3;                   S7
```

Figure 6.3 – While Scheme Modification Example
invalidating the structure of the program in a way that causes a syntax error. COPE does not allow deletions to invalidate the program structure.

Deleting (or replacing) the statement pointed to by the execution cursor raises questions about where execution will resume. This is an implementation decision. A reasonable choice would be to place the execution cursor at the next (or replacement) statement. As long as the execution cursor's position is apparent to the user, the execution sequence is well-defined and properly represented.

Any change to a while scheme program can thus be modified and resumed without difficulty.

6.3.2. Flowchart Schemes

Flowchart schemes introduce only one additional statement, the GOTO. Since PL/CS limits GO TO's to forward transfers within the current procedure to constant labels at a higher level of logical nesting, few additional complexities are added. This simplicity is in direct contrast to the complications introduced into the denotational semantics by the complication of function compositions upon which the denotational semantics is based. The GOTO has no action portion and its sequence portion consists of selecting the statement with the label mentioned in the GOTO. Since in PL/CS the label is syntactically guaranteed to exist and be a proper place to go, this presents no problem.

Moving the execution cursor is equivalent to an unrestricted immediate GOTO. For instance, branches backward into the middle of nested loops are possible. While this sort of branch complicates understanding (static) programs, it poses no new problems for our modification semantics. None of the semantics depends on execution of the beginning of a construct to establish meaning for its components.
6.3.3. Recursion Schemes

Recursion schemes introduce procedure declaration, CALL and RETURN. Up to now, definitions have dealt solely with the text of the program. The execution of program statements did not affect the semantics of the individual program statements (while schemes assume the existence of a "meaning" for boolean expressions, but no details are specified). The definition of recursion schemes requires saving "environments", which retain the state of the computation. Environments are used below to represent variables; we defer discussion of parameters until the meaning of variables has been defined.

A procedure declaration associates a name with a program text. (The PROC statement of the procedure is the only form of procedure declaration in PL/CS; functions and procedures cannot appear in DCL statements.) Each statement in the procedure text has a unique "identifier" that can be used to designate that particular statement in semantic discussions. These identifiers are not evident to the user, they are merely for clarity of presentation (as in Figure 6.3). Uniqueness of these identifiers is maintained over the life of the program. New statements receive new identifiers, as do modified versions of existing statements.

The procedure declaration has no action portion (at this level). The sequencing component selects the first executable statement of the procedure text.

The action component of the CALL statement inserts the name of the called procedure and a reference to the procedure text into a new, otherwise empty, environment. The statement identifier for the CALL statement is then saved in the current environment and execution switched to the first statement (always the procedure declaration) of the called procedure. Execution of a series of nested CALL statements creates an environment chain that closely resembles the standard stack implementation. The sequencing component of the CALL statement selects the immediately following statement.

The action component of the RETURN statement deactivates the current environment and re-activates the environment of the corresponding CALL. The sequencing component of
the RETURN statement is undefined: Following the reactivation of the CALL statement, its action component has already been executed, and sequencing leads to the next statement.

These definitions are intended to be a careful, but otherwise unexceptional, explanation of standard procedure call and return. No attempt will be made to give examples of how they work for static program texts.

6.3.3.1. Resumption

The introduction of explicit environments provides the first opportunities for inconsistencies between the program text and the state of the program.

For parameterless procedures, the only function of the procedure declaration is to associate a name with a procedure text. Changing the procedure name causes future CALL statements to the old name to reference its replacement or to fail if no replacement is provided. Attempts to re-establish active environments (through RETURN statements) will fail since the name of the procedure no longer corresponds to the original procedure text. Even if a new procedure exists with the same name, the statement to be resumed cannot exist (because of the uniqueness of statement identifiers). (Note that the modification units used by COPE make it impossible to alter the procedure declaration without replacing the entire procedure.)

Any modification to an active CALL statement makes it impossible to resume execution in the environment from which the CALL was issued. The RETURN statement always succeeds, since the definitions guarantee that the environment of the CALL will always exist. Within the environment, the statement to be resumed is the CALL that has been deleted, making resumption impossible. (Note that since execution is returned to the correct environment, it is simple for the user to restart execution at any statement in the associated procedure by simply selecting a new statement to be executed.)
6.3.4. Variable Declaration

Variable declaration is a central aspect of structured languages; their omission is a feature of interactive languages. APL and LISP avoid declarations by interpreting the type of a variable at runtime — the type of the variable is the result of execution history. Type errors can occur, but there is no difference between the type of errors caused by incorrect program usage and program modification. BASIC statically associates types with names, making it impossible to introduce inconsistencies.

A PL/CS declaration (excluding parameters) associates attributes with a variable name:

1. **Type.** FIXED, FLOAT, CHARACTER, BIT.

2. **Mapping.** Scalar, vector, 2-dimensional array, etc. Array bounds and declared length of CHARACTER items. (This choice intentionally departs from that of Pascal in which bounds of arrays are part of the type.)

3. **Accessibility.** EXTERNAL means that the name is known outside of the procedure and must exactly correspond other EXTERNAL uses of the same name; otherwise the name is local to this procedure. STATIC (implied by EXTERNAL.) means that the same name references the same instance of the variable across instantiations of the procedure; otherwise, each instantiation corresponds to a new variable.

4. **Initial value.** The value associated with the variable until explicitly changed. If the INIT clause is absent, the special value "uninitialized" is used.

Before the first executable statement of the procedure is executed, each variable is instantiated by the creation of a (type, mapping, location) triple that describes the variable. Type and mapping are as shown above; the assignment of a location is performed in a way that guarantees that the accessibility constraints are satisfied. The appropriate initial value is assigned to the location associated with the variable. The initial value is always assigned for non-STATIC variables. For STATIC variables, the value is only assigned if this is the first activation of this procedure. For EXTERNAL items, the initial value is not assigned if the
variable has been otherwise initialized: Inconsistent declaration, including initialization, of EXTERNAL variables is an error.

6.3.4.1. Modification

Changes to the program text can introduce new declarations, and delete or modify existing ones. The addition and deletion of declarations potentially introduce syntactic problems that are not relevant to the current discussion — name conflicts, undeclared variables, etc. Otherwise, these are both special cases of modifying an existing declaration.

At least two different views of modified declarations are possible. The more conservative of the two insists that any modification to the declaration results in a new variable. This new variable is assigned a new (type, mapping, location) triple and the location is assigned the appropriate initial value. Execution can proceed normally, assuming that uninitialized variable references are checked. Under this discipline, any previous computation on the variable is lost, unless the user takes specific action to save it.

The more radical approach is to attempt to retain as much of the information in the variable as possible. To accomplish this value retention, we will characterize declaration changes by how they change each of the attribute categories (type, mapping, accessibility, and initial value).

Changing the INIT attribute of a variable only changes the value of the associated location if no assignment to the variable has occurred. For automatic variables, this can require changes in each of the active environments associated with this procedure. For EXTERNAL variables, the additional condition of consistency among declarations must be met. One declaration with the INIT attribute is sufficient to initialize the variable. If more than one declaration of the same variable has the INIT attribute, the PL/I and PL/CS language definitions require that the values given must be identical.
Changing the accessibility of a variable can be complicated by implied changes to the possible number of instances and/or references to the variable. Changing a STATIC or EXTERNAL variable to automatic requires that a copy of the variable and its current value be created for each procedure instantiation. Changing automatic to STATIC may be complicated by more than one instantiation of the procedure, each with different values for the variable. The conversion to STATIC implies that a single location be used for all instantiations and it is not possible for both of them to be assigned to the same location. The conversion to EXTERNAL is similar, except that a previous value may already exist for the EXTERNAL version (the existing EXTERNAL value must be retained). Note that some care must be taken when the last reference to an EXTERNAL variable is removed so that subsequent re-introduction of the variable as an EXTERNAL does not conflict with vestigial values.

The criterion for handling changes to mapping characteristics is simple. As long as an item is represented in both the old and the new mapping, it should retain its pre-modification value. For example, if the array A(1:10) is redimensioned to be A(-10:5), A(1) should still reference the same value. A similar argument holds for character strings. Shortening the declared length of the string can cause string-value truncation, but the characters present in both declarations should be retained.

No conversion is possible between one-dimensional and two-dimensional arrays. Even if the modification could be defined, such a shape change would cause syntax errors for all but trivial usages (such as array-scalar assignment).

Changing the type of a variable can be accommodated as long as the normal PL/CS conversion rules are not modified. That is, a FIXI:D scalar can become a FLOAT scalar with the "same" value. Most items can interconvert in PL/CS, so this extends the range of permitted changes.

Note that none of these changes must be allowed. It is sufficient in all cases to assign "uninitialized" values to all variables whose declarations are in any way altered. This provides
adequate warning that some difficulties may exist. However, properly performed, each of these transformations retains a meaning consistent with appearance and execution history of the program. Further, each can save the user considerable additional work copying values to accommodate simple changes.

6.3.5. Parameters

Parameter passing can be viewed as a process of passing the (type, mapping, location) triples used to describe variables. Syntax restrictions in PL/CS guarantee that the type and the mapping shape of the formal and actual match. PL/CS semantics define the dimensions of the formal to be the same as those for the actual. The compatibility of the call and entry side triples must be retained to permit continued reference to the corresponding parameter.

Direct modification of the procedure declaration to change the order or number of parameters certainly invalidates references to these parameters. (Currently, changing the PROC statement of a COPIE procedure is done by replacing the entire procedure, so this question is somewhat moot.) While it is possible to envision a more detailed characterization of the effects of shifting parameters, etc., there is no hope that such a characterization could predictably produce "reasonable" results.

Modification of an active CALL statement does not interfere with the progress of the called procedure, but blocks execution on return — exactly as it did in the parameterless case.

More subtle problems are introduced by the possibility of modifying the declarations of either formal or actual parameters. To retain a consistent semantics, changes to both formal and actual parameter declarations must leave each consistent with its environment and with the other. The types of changes that can be made are the same as those presented for variable declarations. Additional complication results from the fact that both the formal and actual declarations cannot be changed simultaneously, introducing the possibility of temporarily incompatible states that greatly complicate the detection of a sustained correspondence.
Aliasing presents a particular problem for parameter passing. For instance, an easily envisioned change to formal-actual pairs (without causing problems) is the redimensioning of the actual variable. Since the formal inherits these dimensions the correspondence is automatically retained, but the actual array may have to be relocated. If other parameters reference the same array, or individual elements of it, care must be taken to keep the correspondence correct, added by the correspondence.

If a temporary is generated to match an actual parameter and its formal, there is no way to alter the actual once the call is made. This is particularly confusing if the actual is not obviously an expression, (variable default converted to match types). The actual is derived from the argument as it appears in the call, but is not identical to it. Unfortunately, the syntax suggests a more direct correspondence. (A similar confusion is common among students who expect such an actual to be modified when the formal is assigned to — both problems could be solved by prohibiting such parameter conversions.)

6.4. Implementation Note

A major implementation problem introduced by interleaved modification and execution is the allocation of variables. Adding and deleting variable declarations from procedure bodies violates the stack allocation assumptions that are used in most procedural language implementations. LCC [43] allocates each variable separately, stacking its instantiations as needed to handle recursive calls, in essence associating the storage for the variable with it symbol table entry. Although this scheme makes it possible to add and delete variable instances with complete freedom, it introduces additional allocation overhead and an additional level of indirection to all references.

The COPE solution uses fixed offsets from a local stack pointer (essentially the standard method for traditional implementations) for efficiency. Each stack frame is marked with the schema identifier representing the format used to construct it. For allocation and call, the only additional overhead is marking each frame with the appropriate schema identifier. On return,
it is only necessary to compare the schema identifier of the current frame with that being used by the procedure currently. If the schema identifiers are the same, the sequence proceeds as in normal stack management.

If the schema are not the same, there must be sufficient information to reconstruct the stack as it should be. The most straightforward way to accomplish this reconstruction is to have the schema descriptors point to explicit representations of the format of the stack in each instance. Given these schema, the stack can be reconstructed in time proportional to the number of stack elements in the resulting schema. The advantages of this organization are:

(1) Significant execution overhead is only incurred when the schema has been changed.

(2) There is no reshuffling of stacks with parameters referencing them. (Note that this would not be true if PL/CS had pointers, since references could be between local variables.)

(3) There is no permanent residual effect on the stack-frame schema caused by execution history. No empty spaces are created.

(4) Saved execution sequences can be restarted even when the procedures have been modified between save and restore. (This does not apply to COPE, but could apply to a similar system with multiple workspaces and shared procedures.)

This method has the obvious drawback that additional space must be allocated for marking the stack and storing the explicit schema. The space required to mark the stack is proportional to call depth. The current copy of the stack schema is necessary for normal system operation; only as many additional schema's must be saved as there are suspended procedure activations. For reasonable usages, it is expected that this scheme takes less space than individual allocation. When no modifications are made, it is very efficient. It is presumably not possible to make changes fast enough to have schema reformatting be a major runtime overhead.

This method of maintaining local reference schema depends heavily on the stack nature of the procedure call. PL/1 ON conditions (not present in PL/CS) violate the stack assumption
by making it possible to execute statements in previous active environments without returning back through the call history. PL/CS parameters potentially present a similar problem, since the parameter can reference environments that have been modified but not yet reformatted. However, no newly declared (since the procedure was last active) can be parameters since they were not declared when the CALL was issued. Even so, otherwise innocuous changes to variables that are active actual parameters must be carefully checked. For instance, if A(1) were passed as a parameter and the array bounds subsequently adjusted (in a way that A(1) is still defined), it is necessary to make sure that references to the parameter still reference A(1).

6.5. Summary

Our informal modification semantics shows a number of cases in which the implementation must be careful before allowing resumed execution. The number of such cases is, however, considerably smaller than would be suggested by the absence of such features from current systems. Furthermore, the cases that do present problems can be detected during program editing (with an appropriately language-cognizant editor) and the appropriate execution safeguards provided without undue complication of the runtime environment. Building on this logical base, the COPP execution supervisor is able to provide an effective and useful modification facility while avoiding the introduction of semantic errors into implementation.
CHAPTER 7

User File System

The addition of a general file processing capability to COPE makes the system useful in a wider variety of instructional situations, for a richer class of problems. Our objective was to define a permanent file system that would be both simple enough for use by the beginning student and yet powerful enough to be attractive to a more experienced user.

Although the file system is essentially independent of the implemented user language (PL/CS, Pascal, Ada), all language-specific references will be to PL/CS. There are several reasons for this, not the least of which is that our first implementation was for PL/CS and more experience is available with that language. Perhaps the most important distinction between PL/CS and other candidate languages is the availability of variable-length strings and type conversion in file operations. As will be seen below, the proposed file system uses both of these notions extensively. As a result, Pascal programs would, without fixing the string representation, have difficulty making real use of many of the facilities.

7.1. Levels of File Access

The COPE file system is designed in four levels, one of which is assumed to be provided by the underlying operating system or machine environment.

(1) At the lowest level, the environment that is assumed provides services reading and writing specific pages of secondary memory. Minimal interaction with existing file or directory structures is expected.

(2) The lowest level implemented within COPE provides a rudimentary storage management facility based on pages allocated using the services in (1). Although some file management is handled at this level, everything but the storage
management facilities could easily be replaced by a virtual memory facility that allows file pages to be directly accessed in memory. Such facilities were not available for the target machine, and will likely not be generally available on "personal" machines in the near future.

(3) Each basic system object in the system is represented in the manner that makes the most sense for the particular object and its associated processing routines. In addition to whatever processing routines are required to provide the necessary object functionality, each object is provided with an editor, a decoder, and a set of basic file sequencing primitives.

(4) The user level file system is designed to provide a uniform view of the files in the system independent of the implementation differences of the underlying representation. Any non-privileged routine (i.e. any routine that does not deal with the object in its internal form) receives the text form of an object from the appropriate decoder and provides text input to the appropriate editor. In this way, the user can manipulate any system object in precisely the same form as it is normally displayed.

Most of this organization has been outlined in the preceding chapters. The top level provides access to the various objects for display windows and the processing routines deal directly with the internal representation. User access to the files is at the same level as the communication control.

The first and second levels are not novel and are not discussed further. The third-level structures differ depending on the function served and the types of objects represented, but are not generally interesting enough to warrant exhaustive coverage. The role of the internal representation for programs and the runtime environment in the design and implementation of the execution supervisor is described in [5]. This chapter focuses on the representation for user files and the type of access to these files that is provided by the system. Any object appears to
be a user data file if read like one, so at this level the distinctions between the representations are not generally important.

7.2. Unrestricted File Organization

The Unrestricted File Organization (UFO) is the logical file system for COPE. UFO is a file system, comparable to VSAM; it does not pretend to be a full-function database system. VSAM is a flexible, large-scale production file organization whose design has been documented [60, 63, 40]; UFO is a highly flexible, small-scale, interactive file organization. VSAM subordinates flexibility and simplicity to efficiency; UFO subordinates everything to simplicity, and subordinates efficiency to flexibility. The two systems bear some resemblance in general structure and operation.

Figure 7.1 defines the basic terms used in the UFO descriptions. Most of these have normal usages that are slightly different from, and more general than, the ones used in the UFO.

7.2.1. Flexibility

The UFO supports direct-access and sequential file operations. Operations may span multiple records of a file, or apply within a single record. Any file may be accessed or modified by any of the available mechanisms. No feature depends on the file declaration; there are no file variables.

UFO files can contain any mixture of keyed and unkeyed records (which is not allowed for most file systems). Records are accessible by either record number or key-value. Records with keys are stored in key-value order, following the standard rules for character string ordering (lexicographically). Unkeyed records are assumed to have the key-value of the preceding keyed record. Record numbers are consecutive integers starting with one.

An index is required for either keyed or random access. The access time for a record does not depend heavily on the density of keys in the file. The index, including record
file system
A collection of files. All files exist on the same level of the file system. There is no notion of levels of directories.

file
A sequence of records named by a string of printable characters. The file type is an intrinsic characteristic of the file. No naming convention signals the type of the file. File names are global — known to all programs run in the user's system. These values do not conflict with identifiers in any program.

record
A sequence of items. Each record is numbered by its sequential position within its file. The number of a record changes whenever a record is added to or deleted from a lower-numbered position in the file. Optionally, each record can be named by a "key-value". The key-value is any character string value consisting of printable characters. The key-value is unique for the file. The key-value of a record cannot be changed (though the same contents can be rewritten with a new key).

item
One of the distinct objects that make up a record. Items are the basic unit of information transferred during read and write operations. For the simplest case (corresponding to "LIST" operations in PL/CS) an item corresponds to the value of a scalar variable that is read or written.

current record pointer
The record number of the current position in the file.

current item pointer
The number of the next item in the current record of the file.

Figure 7.1 — UFO Definitions

numbers, is easily maintainable during item/record insertion/deletion.

The system is "record-oriented", since records are a useful concept for file access; random access to one thousand single-item records will be faster than random access to the one thousand items stored in a single record. The number of items in a record can vary from record to record.

7.2.2. Simplicity

UFO is the sole file organization available to the COPE user. Each user has an independent file system. User files cannot be shared between users, and users are allowed only a single active process, greatly simplifying concurrency and consistency control.
Although no explicit implementation restrictions force files to remain small, fast response to individual operations on small files is more important than high-throughput for large files. A structure to support many hundred-record files well and a few thousand-record files adequately is different from an efficient structure for million-record (or larger) files. Most user files will require little of the flexibility built into the UFO, so simple sequential access must be relatively efficient.

The target environment presumably offers only limited secondary storage, but a dedicated processor. As a result, UFO is willing to trade processing time for space efficiency. Little or no "room for expansion" is left for update operations. Extensively updated files are less efficient than sequentially created ones.

7.2.3. Incrementality

UFO file operations are isolated and incremental. The basic unit of data transfer is an "item", not an entire record (as would be more standard for file systems). Although access support structures must be made to accommodate finding records, each record is not written as a whole. In particular, this means that the system must work well for each of the atomic operations without disastrously harming overall efficiency. Moreover, the user is able to use any of the basic file operations in "immediate mode", a particularly isolated form of file access. Ideally, the cumulative effect of a well-defined series of operations to a particular locality of a file should closely resemble what could be expected given foreknowledge of the operations to be performed. This principle does not extend to files created in a completely random order.

7.3. File Structures

Figure 7.2 gives an example of a section of a possible file. Each record is represented by a new line. For convenience, items are shown separated by commas; records are separated by sharps (#). Five records are shown and numbered 15-19. Four of these records have keys, and the keys are in collating sequence order. One record (18) has no items.
<table>
<thead>
<tr>
<th>Key</th>
<th>Number</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ace</td>
<td>15</td>
<td>item 1, item 2, item 3#</td>
</tr>
<tr>
<td>lob</td>
<td>16</td>
<td>item 1, item 2#</td>
</tr>
<tr>
<td>serve</td>
<td>17</td>
<td>item 1#</td>
</tr>
<tr>
<td>smash</td>
<td>18</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>item 1, ..., item n#</td>
</tr>
</tbody>
</table>

*Figure 7.2 — Sample File Contents*

If a record were written with a key-value of "forehand", it would be inserted as record 17, causing the record with the key "lob" to be numbered 18, etc. Unkeyed records are considered to be subsidiary to the immediately preceding keyed record (if any exists) for the purpose of record placement.

7.3.1. Basic File Structure

The UFO assumes an underlying fixed-length sector secondary storage medium. The actual sector length and total device capacity determine the sizes of pointers and other fields, but do not affect the overall design philosophy. Typical sector sizes for currently available devices range between 128 and 512 characters; file capacities of 64K sectors are within reason. The file system expects to be able to read, write, and allocate sectors. It is not conceptually important whether sectors are logically addressed within files or physically for the entire device. A logical number would presuppose a more sophisticated device manager. Whichever addressing method is used, the file structure is prepared to take advantage of consecutively addressed sectors; consecutive sectors are not required, however. Minimization of sector accesses for common operations is a system design goal.

A physical file consists of two logical components: an index and a data area. The data area is maintained in logical sequential order; it is possible to read the data portion of the file sequentially without reference to the index. The index is a B*-tree modified to keep track of
record numbers efficiently and handle variable-length keys [35]. Keys are stored entirely in the index. Index blocks correspond directly to sectors; data area records are independent of sector boundaries. The ability to add or delete items (effectively changing the lengths of existing records), coupled with the implicitly inherited key structure make a "static directory" (as proposed in [29]) undesirable for the current system. Fortunately, the system requirements do not introduce the problems of concurrency and secondary indices which such directories help to alleviate.

7.3.2. Data Area

The data area consists of a series of items separated into records. Conceptually, each of these items is simply a varying character string. Consistent with the "print" representation of a list of items, we will associate item separation with a tab character (shown above as "\t") and record separation with a newline character (shown above as "\n"). The actual characters may not be tab and newline, these are logical associations. These two characters are examples of "distinguished" characters, characters which are used to indicate a special function. In order that these characters can occur in data items, an escape sequence is used whereby the escape character (whatever is chosen) causes its immediate successor to be taken literally (including the escape character). This escape process is entirely within the file system and need not concern the user, to whom it is transparent. Efficient character representation dictates that the escape and all other distinguished characters occur only rarely in normal usage. Consider the following section of a file data area:

```
ab, cde\# fghi, j\# #, #
```

There are six items and four records. Wherever it is unambiguous, the tab marking the end of the last item is dropped. The last record, a single null item, is the only situation in which the abbreviation would be ambiguous. The next to the last record is null, i.e., it contains no items.

Items (and consequently records) are immune to sector boundaries. Items overlap consecutive sectors without any indication; consecutive sectors are assumed to be logically
adjacent. Chaining of logically consecutive, but physically separated, sectors is accomplished by a pointer at the end of the sector. The mechanism links sectors; it does not affect the end of items. The pointer indicates the number of the next logical sector. All data is stored within a sector "left-justified"; any free space in the sector is represented by trailing null characters. Insertion within a sector is accomplished by moving data within the sector (presuming the insertion fits). To minimize the upward (into the index) impact of such shifts, all records starting within a block must be under the same index sector.

7.3.3. Item Representation

The representation depicted so far has assumed that items would be stored as character strings. The chief attraction of the character representation is its universality; not all items are character strings, but they can all be converted to character strings. Further, virtually all input/output facilities for casual programming involve streams of characters and facilities of varying sophistication for converting that stream into internal representations.

There are good reasons for this dependency on character representations. Consider a closed system with only one form of input and output, similar to PL/CS GET and PUT. All user data entry must be done using characters. As a result, the GET statement must be able to read character representations. The PUT statement is the only way in which to display information as output from the machine, so this too must be in terms of characters. As a result, all data external to the system ends up being stored in a character representation. Although this logic seems compelling, it should be noted that the quoting conventions of PL/I make it extremely difficult to read a file written with PUT LIST statements using GET LIST statements. For whatever reasons this was done, it makes the PL/CS file representation inappropriate as the base of a user file system.

The COPE structure makes it unnecessary for the file to be stored in a directly-readable form. Since all user inspection of files will be through system-provided routines to make it readable, numeric items that are written out can be represented in internal (or other
abbreviated form) without the user being aware that this is occurring. Similarly, by keeping track of the item boundaries, it is possible to present files in the more attractive unquoted style (used by PUT) without making it impossible to read the same entries. In addition, the internal representation for numeric items is significantly shorter than the standard PL/I string representation, and can easily be kept shorter than the more readable (and shorter) print representations used. In addition, internal-form representation minimizes the number of surprises that the user will get from writing a value, reading it back and finding they are not identical.

To effectively integrate strings (already in essentially internal form) and other types, and internal coding scheme is used which depends on the use of a distinguished character to indicate the presence, type and length of the internal item followed by the fixed-length value itself. For BIT items, the value is actually coded in the distinguished character (PL/CS only allows BIT(1)). No trailing item separator is needed for these values. The coded representations preserve the types of the items, but need not exactly match the internal representation. For instance, many small numeric values are stored in a single character rather than at full length. This method accomplishes significant space savings at only moderately increased processing complexity. It is somewhat machine-dependent, but transferability of files in internal format is not a system objective.

A variety of different methods for encoding text (string data) to reduce its size have been proposed; [54] provides an excellent categorization of the types of techniques that have been used and places them in perspective. Unfortunately, the files that are being processed here are not well suited to most of the more interesting methods. Since each item can be updated independent of the remainder of the record, it seems impossible to adopt any method where the coding for one item depends on the contents of any of the others. This limits any encoding to individual items whose length is likely to be short (limited to 256 characters by the language implementation). As a result, most of the more powerful encoding schemes are relatively
unattractive for the current purpose.

A fixed encoding scheme, based on presumed character frequencies was considered, but it was hard to predict what reasonable character frequencies would be for an unknown mix of user files. For similar reasons, a CCITT (often called Baudot) coding scheme, based on short characters and shift sequences to change character sets, was also rejected. A scheme for compressing strings of characters was adopted. The processing cost is not high, and the use of the method can be reserved for those cases where space is actually saved. The method is tuned to make blank compression particularly effective, though repetitions of more than four of any other character can be compressed.

7.3.4. Index Area

The file index is a modified B*-tree, each node of which occupies a sector. Three types of information are stored in the index: keys, record numbers, and pointers (to either another index level or to data). The sectors of the file are structured into a strict hierarchy by the index; all data records starting in a sector appear in the same index sector. Note that this can cause problems when key lengths approach sector lengths and records are short; this is not considered a common file configuration. Not all data records are indexed. Keyed records are always indexed. Every data sector in which a new record starts has an index entry corresponding to that first record, keyed or not.

All keys are stored in the index as variable-length character strings with explicit terminators. Keys are compacted using a high-order abbreviation technique. Keys that share a common prefix with the previous key begin with distinguished character(s) indicating the length of this common prefix. We will represent the distinguished character for an n-character match as n. Figure 7.3 shows both the compressed and complete versions of an index for a sample set of keys. The distinguished character 0 indicates a complete key match. Remember that unkeyed records are considered to have the same key as their predecessor, so this is a common encoding. The first key of an index sector is implicitly inherited from the next higher level, if
Record numbers are stored incrementally in the index. Each index entry includes the number of records associated with it, i.e. the number of records in its subtree. To determine the number of a record reached by search through the index, it is necessary to accumulate these counts as the index is searched. Note that the set of counters accumulated exactly corresponds to the set of entries that must be scanned to find the record. This distributed representation of record numbers minimizes the number of index sectors which must be accessed to perform record insertion or deletion.
Figure 7.4 illustrates the index and data structure of the file excerpt given in Figure 7.2. The order of entries in the index, when all are present, is: key, pointer, count. Data areas are marked by "record number" to help decode the picture. The insertion of three records, one keyed "service" and two unkeyed immediately following, result in Figure 7.5 (assuming no data overflow or index splitting).

7.3.5. File Update

A file can be modified in a variety of ways; items or entire records can be added or deleted, items can be changed (possibly altering their length). Depending on the organization of secondary memory provided by the system, freed space (in block quantities) could either be released or retained in the file for future expansion. For the current purpose, we will ignore problems caused by deletion (or shortened length).

Insertion of a new record causes parallel changes in both the index area and the data area. Index update in B*-trees is explained in [35]. Linkage of data and index blocks can make update operations within a locality faster [11]. We assume a buffer cache makes such

---

Figure 7.5 – Updated File from Figure 7.4.
index linkage unnecessary. The basic file update operation that we will consider is adding an item. A common special case is adding to the end of the file. Adding new records is done by a creation operation that affects only the index, followed by adding items. Changing an item can be thought of as deleting/adding the item (though this is not quite the implementation).

Consider adding an item to a record. The item must be inserted into the data area in sequence. If the item does not fit, space must be found in one of two locations: the next data block or a newly-allocated data block (or blocks). If sufficient space can be found in the next block, data is shifted over to it. If not, data in the current block, but after the new item is moved to a new block (along with any portion of the new item which will not fit). Whenever a choice is available, free space is left immediately following the most recently added item. Movement of records to new positions or blocks can necessitate index changes. Since the file is strictly hierarchical by index block, it is always true that the index entries to be updated are in the same block at the start of the operation. Moving records can necessitate new index entries (corresponding to unkeyed, “implicitly” indexed records), potentially causing index splitting.

7.3.6. Implementation

The UFO is implemented by a set of procedures that provide all necessary access to files, records and items. Conceptually, the entire file system is accessible at all times and is maintained in an updated form on secondary storage. The implementation keeps sectors of the file system in memory on a Least Recently Used (LRU) basis. The file system does not keep track of the current user position in files. Rather, a file descriptor (FID) is created and updated for each file in the user's area. It is the responsibility to the user (the COPE system routines) to make sure that the correct file is to be accessed. From outside the UFO, the FID is a capsule to be saved and passed with requests, but never examined or changed. File status information is obtained by explicit procedure call. The assumption of an informed, trustworthy user is consistent with the single-user, COPE architecture.
The FD consists of the index search path for the current data record, its record number, key and item number. The entire FD is not always kept current. Consider a sequential file reading program; typically, no index or key information will be required. Efficient implementation of such sequential operations demands that the index level pointers and keys be ignored until required. The UFO is capable of interpreting the current information state of the FD to respond to requests. Depending on the recent usage patterns, a request for the current key of record n may require no secondary storage accesses, or quite a few. Under "normal" conditions, assuming some locality and consistency of access, the number of sector operations should be small. A brief explanation of the routines provided by the UFO and how they are used to define the language constructs is given in Appendix C.

7.4. Language Interface

Defining a language interface to correspond to the underlying strategy is an excellent exercise in conflicting goals and objectives. Since PL/CS is a deliberately simple language (it has a total of fourteen distinct statement type, including END and null [13]), the addition of even a few new statement types represents a substantial relative burden to the language. On the other hand, general input/output facilities are difficult to fit into the procedure paradigm. Pascal [32] handles its input/output by system procedures, but these procedures have a special status with respect to the numbers of arguments and their types which makes it impossible for the user to replicate them. If such procedures could be rewritten to have modified or expanded function for the particular use, the maintenance of a uniform access syntax might be more attractive. The extensions which would make this possible (most of which are available in Ada [1]) are not appropriate to the novice user. Further, the system-procedure approach typically involves a sequence related procedure calls to accomplish a single purpose, simply to overcome the flexibility provided by optional statement components.

PL/CS does not currently support structures. Without structures, the standard PL/I record-oriented constructs are particularly unattractive. Further, considerable expertise, use of
sophisticated storage allocation facilities, and attention to detail are required to process records with variable numbers of items each of variable length in PL/I.

The language extension attempts to provide a mix of flexibility and simplicity very similar to that provided by the underlying UFO. This leads to very different constructs than exist in PL/I. This is not surprising considering that PL/I facilities were designed to handle file organizations where efficiency was paramount and the available access methods were less flexible than VSAM.

7.4.1. Language Additions

Figure 7.6 summarizes the statements and phrases added to PL/CS. A more detailed description is available in Appendix C. Only one statement type (DELETE) was added. The ability to reference files (the FILE phrase) has been added to GET and PUT, and a "start new record" capability, NEXT (similar to PL/I SKIP), has been added to the GET statement (PL/CS GET does not have SKIP as an option). NEXT is used in place of SKIP to increase the syntactic separation of GET and PUT to facilitate PL/CS error repair.

In traditional PL/I input/output, operations on the file are performed by language statements and most of the available status information is accessed through special exception handlers (ON conditions). To achieve a complete file system without the plethora of PL/I

---

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>new statement: deletes a file or a series of records from a file</td>
</tr>
<tr>
<td>GET</td>
<td>FILE phrase: allows multiple files.</td>
</tr>
<tr>
<td></td>
<td>NEXT phrase: starts next input record.</td>
</tr>
<tr>
<td>PUT</td>
<td>FILE phrase: allows multiple files.</td>
</tr>
<tr>
<td></td>
<td>PAGI phrase: clears the data area of the display screen.</td>
</tr>
<tr>
<td></td>
<td>SKIP phrase: move up or down in file.</td>
</tr>
</tbody>
</table>

Figure 7.6 — Additional Statement Constructs for PL/CS
statements (OPEN, CLOSE, READ, WRITE, etc.) and phrases (KEYTO, KEYFROM, GENKEY, etc.), a small, but powerful, set of pseudo-variables and builtin functions (builtins for short) access and modify the components of the file's environment is employed. These builtins are summarized in Figure 7.7.

Each statement or function deals with some file. The name of the file is specified in statements by the \texttt{FILE} phrase and in builtins by an argument (the first). The name of the file is merely a character string, which can be specified by a constant or variable. In the initial implementation, the file name is a string preceded by a designated file character. This representation removes the distinction between the file name and the variable containing the file name. The design and implementation retain the additional flexibility of handling string variables.

The current record position and item position are available through the \texttt{REC} and \texttt{ITEM} builtins, respectively. The number of items in the current record is referenced by \texttt{COUNT} (no

<table>
<thead>
<tr>
<th>REC</th>
<th>function and pseudo-variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Function returns current record pointer.</td>
</tr>
<tr>
<td></td>
<td>Pseudo-variable assignment sets current record pointer.</td>
</tr>
<tr>
<td>REMAIN</td>
<td>function Returns the number of records until end-of-file.</td>
</tr>
<tr>
<td>KEY</td>
<td>function and pseudo-variable</td>
</tr>
<tr>
<td></td>
<td>Function returns the key-value of the current record.</td>
</tr>
<tr>
<td></td>
<td>Pseudo-variable assignment adjusts the current record pointer.</td>
</tr>
<tr>
<td>FIND</td>
<td>function</td>
</tr>
<tr>
<td></td>
<td>Sets current record pointer to first record with key $\geq$ key-value.</td>
</tr>
<tr>
<td></td>
<td>Returns the character-string value of that key.</td>
</tr>
<tr>
<td>COUNT</td>
<td>function and pseudo-variable</td>
</tr>
<tr>
<td></td>
<td>Function returns number of items in the current record.</td>
</tr>
<tr>
<td></td>
<td>Pseudo-variable assignment sets number of items in record.</td>
</tr>
<tr>
<td>ITEM</td>
<td>function and pseudo-variable</td>
</tr>
<tr>
<td></td>
<td>Function returns the current item number in the current record.</td>
</tr>
<tr>
<td></td>
<td>Pseudo-variable assignment sets the current item pointer.</td>
</tr>
</tbody>
</table>

Figure 7.7 — New Builtin Functions and Pseudo-Variables
relation to the PL/I function of the same name). The number of records following the current one is referenced by REMAIN. REMAIN is always maintained so that REC+REMAIN is the number of records in the entire file. The lack of reference symmetry is intentional, corresponding to very different usage patterns.

Together, COUNT and ITEM provide the user with the control necessary to deal with variable-length records and the selection of fields within these records. Similarly, REC and REMAIN provide the ability to deal with the entire file as a logical unit, without being tied to sequential access.

The user program is expected to check for unusual return values from these functions to detect end of file, key not present, etc. This method is both easier to understand and control than the PL/I ON mechanism, which can then be completely removed from PL/CS.

```pli
/** determine file name, start and stop keys */
GET LIST(INDEXED), START, STOP;
/** find the first record to be printed */
START = FIND(INDEXED,START);
/** print key and items with KEY <= STOP */
W3: DO WHILE ((KEY(INDEXED) <= STOP) &
(REMAIN(INDEXED) > 0));
GET FILE(INDEXED) NEXT;
PUT SKIP LIST( KEY(INDEXED) );
/** print each of the list items */
I4: DO I = 1 TO COUNT(INDEXED) BY 1;
GET FILE(INDEXED) LIST(STR);
PUT LIST(STR);
END I4;
END W3;
```

Figure 7.8 — Print Specified Section of Given File.

Additional builtins reference the keys of the file. KEY references the key of the current record and may be used to insert a record at the appropriate position in the file. FIND positions the file to the first record with a key greater than equal to its second argument, providing access by key value. FIND returns the value of the key actually found to facilitate
checking that the record was the one being searched for.

7.4.2. Comparison with PL/I

The file processing capabilities provided by COPE are syntactically and algorithmically simpler for the user than those provided by PL/I [45]. Syntactically, PL/I provides a statement or statement phrase to handle each variation of function desired. The syntax reflects the data management philosophy of large, batch operating systems which allow (and frequently require) the user to make myriad detailed decisions about file allocation, representation and processing. For the single-user, interactive environment, this "flexibility" is more an obstacle than an opportunity.

All PL/I file processing (other than the simplest sequential) is done using record-oriented constructs, most commonly with PL/I structures. The user has three major options for this type of processing.

1) Predetermine data format, type and length. This allows fixed-length records and access to all processing capabilities. Over-allocation is commonly used to allow room for growth and modification. Mistakes in allocation are rectified by "conversion" programs which expand the file records to meet the newly perceived requirements.

2) Use BASED or CONTROLL:ED storage to allocate structures of differing sizes to meet differing needs. Except where efficiency is vital, only the most experienced (or ambitious) user will choose this method over the one above.

3) Use varying length character strings for all data transmission. Record structure can be superimposed (using DEFINED or BASED). Alternatively, the record can be "structured" by GET/PUT STRING operations. Logically, this is the closest alternative to the PL/CS solution.
DECLARE INDEXED RECORD SEQUENTIAL ENV(VSAM GENKEY);
DECLARE FIELD(10) ... ; /* assumed fixed length records */
/* intercept "invalid" start values */
ON KEY(INDEXED)
  BEGIN;
    /* give up — code to proceed too complicated */
    PUT SKIP LIST("*** the key \', ONKEY(INDEXED),
    is not a valid starting point");
  STOP;
END;
/* detect end of file in case stop value too large */
ON ENDFILE(INDEXED) GO TO ENDPRINTING;
/* determine file name, start and stop keys */
GET LIST(FILENAME, START, STOP);
OPEN FILE(INDEXED) TTITLE(FILENAME) INPUT;
/* find the first record to be printed */
READ FILE(INDEXED) KEY(START) INTO(FIELD);
/* cannot get first key so use START */
INDEXEDKEY = START;
/* print key and items with KEY <= STOP */
DO WHILE (INDEXEDKEY <= STOP);
  PUT SKIP LIST(INDEXEDKEY, FIELD);
  READ FILE(INDEXED) KEYTO(INDEXEDKEY) INTO(FIELD);
END;
ENDPRINTING:
CLOSE FILE(INDEXED);

Figure 7.9 — PL/I Version of Figure 7.8.

Algorithmically, COPE also provides significant simplification. For the novice (or casual) user, this simplification is important. A primary area of potential usefulness of a personal computing system lies in its file capabilities; the complexity of the user's files should be governed by the task that he wishes to accomplish, not by the difficulty of fitting the task to an efficient implementation.

COPE simplifies algorithms by its increased range of permissible file operations. The intent is for COPE to allow any file processing operation that can be simply and unambiguously stated. Records can be expanded, truncated, modified, overwritten or deleted at will. Files can be created or destroyed under program control without knowledge of system structure or a separate control language. All user files are capable of supporting all operations
at all times.

The simplified handling of exceptional conditions makes realistic processing much more straightforward. The interactive nature of COPE itself aids in handling exceptional conditions. Just as importantly, it is possible to detect any possible "exceptional" condition before it occurs. A key reason why real PL/I file processing programs frequently have complicated ON units is that most exceptional conditions cannot be predicted; PL/I makes it impossible to detect errors in advance. The most commonly encountered examples of this are the ENDFILE and KEY conditions.

Accessing approximate keys is singularly difficult in PL/I. The only mechanism that approaches it (GENKFY) only recognizes records whose keys contain the proffered string as a prefix. The programming required to implement equivalent of the FIND function is depressingly complicated, requiring nested ON conditions.

7.5. Summary

A COPE file system is designed for use in a highly interactive environment. The facilities provided allow for a wide variety of file formats and access patterns in a general manner. The emphasis has been on simplicity of use without limitation of function. Many of the provided

```plaintext
/** add items from F2 to records in F1 */
DO WHILE ( (REMAIN(F1) > 0) & (REMAIN(F2) > 0) );
GET FILE(F1) NEXT;
GET FILE(F2) LIST(D);
PUT FILE(F1) LIST(D);
END;

/** mark unmatched records from F1 with "missing" field */
DO WHILE ( REMAIN(F1) > 0 );
GET FILE(F1) NEXT;
PUT FILE(F1) LIST('MISSING');
END;

Figure 7.10 — Merge F1 and F2 by adding item from F1 to F2.
```
DECLARE (F1, F2) FILE RECORD SEQUENTIAL;
DECLARE 1 REC1, /* assumed fixed length record */
               2 A ..., 2 B ..., 2 C ..., ...;
DECLARE 1 REC2,
               2 D ..., ...
/* detect end on file conditions on F1 and F2 */
ON ENDFILE(F1) GO TO ENDPRECESS;
ON ENDFILE(F2) GO TO MARKMISSING;
/* add items from F2 to F1 */
OPEN FILE(F1) UPDATE, FILE(F2) INPUT;
DO WHILE ('TB '):
   READ FILE(F1) INTO(REC1);
   READ FILE(F2) INTO(REC2);
   REC1.C = REC2.D;
   REWRITE FILE(F1) FROM(REC1);
END;
MARKMISSING;;
/* mark unmatched records from F1 with "missing" field */
DO WHILE ('TB '):
   REC1.C = 'MISSING';
   REWRITE FILE(F1) FROM(REC1);
   READ FILE(F1) INTO(REC1);
END;
ENDPROCSS:
CLOSE FILE(F1), FILE(F2);

Figure 7.11 — PL/I Version of Figure 7.10.

capabilities are available in "production" file systems only at great expense and programmer ingenuity. Using these capabilities, a novice user should be able to construct and utilize file structures which would otherwise be beyond his grasp.
CHAPTER 8

Summary, Conclusions, and Future Research

8.1. Summary

The preceding chapters have dealt with the unique characteristics of the COPE system and the issues that its design and implementation raised. Chapter 1 explains the program development problem and places the COPE solutions in the context of previous work. Chapter 2 presents the design principles that have guided the COPE effort. Chapter 3 provides an architectural overview of the COPE implementation that creates the context in which the novel features of the COPE environment can be expressed. Chapter 4 details the COPE user interface. Chapter 5 explains differential parsing, the method used by COPE to provide radical, interactive error correction. Chapter 6 provides a context in which to evaluate implementation decisions regarding the modification of active programs and subsequent resumption of these programs. Chapter 7 presents the design of a novel user file system that places a powerful set of processing capabilities within the conceptual reach of novice users.

8.2. Conclusions

As this is written, COPE is demonstrable, but not yet fully serviceable. Since its objective is to permit evaluation of a novel user interface, it will not be possible to conclude how well this goal has been achieved until substantial user experience has been accumulated. Consequently, some of the conclusions presented are more predictions or conjectures that we believe COPE will support.
8.2.1. User Interface

(1) The full power of an integrated, interactive program development environment can be offered without requiring the user to learn much more than the host programming language. No special entry protocol is required to support structured program entry and no special debugging language is required to support source-level debugging.

(2) A generalized "immediate execution" facility is a natural concept (even in a block-structured language) that is simple to understand and use, and very powerful.

(3) A general recovery facility (for system commands) can be simple to understand and use, and very convenient for the user. The extension of recovery to include execution intervals provides an efficient approximation of reverse execution.

(4) Contrary to the conventional wisdom that error-repair is inappropriate in interactive systems, automatic repair is especially useful and effective when each repair can be submitted to the user for acceptance or rejection. When combined with a convenient UNDO facility, error-repair allows both the user and the system to be relatively bold in this regard. The consequence is a cooperative effort in which the user can make deliberate errors of omission and rely on the system to generate some elements of the program.

8.2.2. Differential Parsing

(1) The separation of structural and expression parsing provides a natural basis for program construction across a broad spectrum of entry styles. The user is able to choose keyword-activated template entry or direct text entry or any style in between.

(2) Keywords form a natural basis for error-correction in programming languages by providing concrete bound on relevant parsing scope for both user and system.

(3) Active, but predictable, error-correction leads to the development of personal entry styles that focus attention on program contents instead of syntax.
(4) The use of the corrected program as the basis for all interaction simplifies the communication of corrections between the user and the system by providing a uniform frame of reference that is always current.

(5) Syntactic prompts provide a natural way to express omissions, provide guidance, and allow testing of partially completed programs.

(6) Focusing program entry on the addition of new constructs to the corrected program simplifies both the processing and understanding of program modifications by maintaining the existing program structure until it is explicitly replaced.

8.2.3. Interleaved Execution and Modification

(1) The ability to modify program contents and resume execution is a powerful asset to program development and debugging.

(2) The resumption of execution after program modification is feasible over a wide range of situations, most of which can be supported in compiled as well as interpreted systems.

(3) Although there exist cases where resumed execution is theoretically difficult or impossible, the primary obstacle to providing such facilities is conception of programs as static objects.

8.2.4. User File System

(1) It is possible to provide a powerful set of keyed and random-access file capabilities to novice users without forcing the user understand or accommodate the underlying realities of file storage.

(2) Linking terminal input/output to a general file system provides an attractive way of providing the user with the advantages interaction and the repeatability of stored data.

(3) The provision of adequate status information greatly simplifies both the explanation and utilization of file-processing facilities.
8.3. Future Research

The COPE implementation and the research on related issues has been focussed toward the specific goal of providing a friendly programming environment for novice users. In each of the associated research areas, two separate sorts of new research are possible: continuation of problem solving for the novice audience, and translation of the goals of the COPE system to more sophisticated user communities.

8.3.1. Empirical Evaluation

We think that the COPE environment provides a uniquely attractive environment for programming instruction, but it is certainly not the only alternative. At Cornell alone, there exist four different systems that are being used to teach students to program in PL/C or PL/CS (batch PL/C, PL/CT, the Synthesizer, and COPE). Early experience suggests the students prefer more interaction to less, though there is no concrete evidence that learning is improved. Perhaps the most interesting comparison is between the Synthesizer and COPE. Although tied to similar technologies, the two systems reflect very different outlooks and future students would be served by careful study of which characteristics of each system are most conducive to learning to program.

8.3.2. Parsing

The entire motivation for Differential Parsing was to create an error-repair strategy that could provide as friendly a language interface as possible. While it is possible to ascertain the kinds of omissions that are appropriately corrected empirically, a user-oriented description of how correction decisions are made would be useful. Such a description could help explain seemingly anomalous corrections in a way similar to that in which syntax messages attempt to explain what should be fixed in conventional systems. It would also be useful in allowing the user to develop a personal entry style without sole reliance on trial and error.
Although the particular parsing method used in COPE yields attractive correction performance, other methods based on the separation of grammars are also possible. In particular, it would be interesting to explore the feasibility of generating minimal distance parsers for each grammar partition, taking advantage of the greatly shortened program segments to overcome the non-linear running times associated with these methods.

8.3.3. Program Display

Screen space is particularly scarce resource in a screen-oriented programming environment. Newly developed displays have up to three times the effective display area of the devices for which COPE was designed. The new displays present the challenge of using the additional space and flexibility without complicating the user interface. In particular, the current interface uses the separate entry area to distinguish candidate entries from correct components. While this distinction is useful, it is inconvenient for large changes. A more flexible window algorithm would allow the entry area to grow and shrink as the need arose. Additional display area could also be used on the execution screen to allow sections of a call chain or the contents of multiple data files to be examined simultaneously.

8.3.4. Production Environments

Because COPE is specifically tailored to the needs of novice programmers, efficiency has been subordinated to facility. Even so, the extremely interactive nature of environment and the particularly user-oriented display of the program during execution would be also useful for more experienced programmers. The performance of an interpretive system is a problem for more sophisticated users, especially to the degree that it implies two systems -- one compiled and one interpreted -- that the user never completely believes are really identical. The solution to this dilemma would seem to be a high-quality incremental compilation system with runtime support that is only expensive when monitoring features are being used. Such a system may never compete with highly optimized code generation, but should be able to provide adequate
performance for all but the most computationally intensive tasks.

4. General Conclusions

Although the price/performance of computing hardware continues to drop, there will always remain tasks for which the ultimate test is execution efficiency. However, the number and relative importance of these tasks is diminishing. In light of the increasing importance and expense of software, it seems inevitable that some of the focus of programming systems must shift to making programs easier to develop, not merely faster to execute. We feel COPE is an important step in that direction.
APPENDIX A

COPE-PL/CS Language Summary

The following is a brief summary of the COPE-PL/CS language. A more detailed presentation of PL/CS is available in [13]. COPE-PL/CS is an extension of PL/CS (which is a disciplined subset of PL/I) by the addition of input/output statements to support a generalized file system and statements to control interactive execution. The statements for controlling input/output have been extended to allow them to form the basis for all COPE file operations; the details of this extension are presented in Appendix C. Lexical details such as character set, identifier construction, punctuation, etc., are comparable to PL/I. The semantics of statements, except as noted, are also equivalent to PL/I.

A.1. Procedures

(1) All procedures are external; no nesting of definitions.
(2) All procedures are recursive.
(3) Any procedure is executable as the "main" procedure of a program.
(4) Function procedures can have no side effects. They cannot contain GET, PUT or CALL statements, or assign value to any STATIC or EXTERNAL variable.

A.2. Declarations

DECLARE (name-list) attributes:

For array declaration: name(lowerbound:upperbound, ...)

Attributes are:
FIXED
FLOAT
The default type is FLOAT, regardless of identifier spelling.

CHARACTER (length) VARYING
The default length is 80.

BIT(I)
STATIC
EXTERNAL
INITIAL (value-list)
READONLY
This is not a standard PL/I attribute.
It prevents the variable from being the target of assignment.

Variables and parameters cannot appear in the same declaration.
All variables and parameters must be explicitly declared.
Declarations appear at the beginning of a procedure, before any executable statement.
A.3. Statements

Assignment

\[ \text{variable} = \text{expression}; \]

Logical expressions must be parenthesized.

\[ \text{array} = \text{constant or array}; \]
\[ \text{SUBSTR( } \ldots \text{ )} = \text{expression}; \]

IF (condition)
\[ \text{THEN } s1; \]
\[ \text{[ELSE } s2; \] \]
where \( s1 \) and \( s2 \) are either simple statements or DO; ... END; compound statements

The condition must be parenthesized.

select-name: SELECT;
\[ \text{WHEN (condition) } s1; \]
\[ \ldots \]
\[ \text{OTHERWISE } s_n; \]
\[ \text{END select-name; } \]

OTHERWISE and at least one WHEN are required
All conditions must be parenthesized.
The select-name is required.

loop-name: DO control;
\[ \ldots \]
\[ \text{END loop-name; } \]

Control is one of following:
\[ \text{WHILE (condition) } \]
\[ \text{UNTIL (condition) } \]
\[ \text{index} = c1 \text{ TO } c2 \text{ BY } c3 \]

Control phrases cannot be combined.
After an indexed loop, the index-variable is uninitialized.
All conditions must be parenthesized.
The loop-name is required.

LEAVE loop-name;
Explicit specification of the loop-name is required.

GOTO target-label;
\[ \ldots \]
\[ \text{target-label; } \]
Forward reference only.
Target-label is only on a null statement.

CALL procedure-name [ (argument-list) ];
RETURN [ (expression) ];
GET [FILE(filename)] [NEXT] LIST | EDIT(...);

The FILE and NEXT phrases are not included in the standard PL/CS language, and are not available in either of the other PL/CS implementations. Neither phrase is compatible with PL/I.

The FILE phrase permits multiple input files and appears to be similar to PL/I, but the filename given is the name of the file, not the name of a variable of type FILE.

PUT [SKIP(expr)] [PAGE(expr)] [FILE(filename)] LIST | EDIT(...);

The FILE and PAGE phrases are not included in the standard PL/CS language, and not available in either of the other PL/CS implementations. The FILE and PAGE phrases are not compatible with PL/I.

The PAGE phrase controls the output area of the execution screen:
- PAGE(n) assigns n lines to the output area, and clears the area.
- PAGE clears the output area without changing its size.

The FILE phrase permits multiple output files and appears to be similar to PL/I, but the filename given is the actual name of the file, not the name of a variable of type FILE.

DELETE FILE(filename) [RECORD [e]] ;

This is not a standard PL/I statement. If the RECORD phrase is present, delete the specified number of records from the specified file, starting with REC(filename).

The default filename is .TEMP; the default value of e is 1.

If the RECORD phrase is absent, delete the entire file (or the top level of .TEMP).

Note that this is the only means of deleting a file in COPE. There is no "delete" command — instead the user must execute an immediate DELETE statement.

ASSERT (condition) [quantifier];

Quantifier is:
- FOR ALL index = e1 TO e2 BY e3
- FOR SOME index = e1 TO e2 BY e3

This is not a standard PL/I statement. It is a diagnostic statement used to monitor the truth of the assertion during program execution. The result of an unsatisfied assertion is a pause in execution with an appropriate message in the message area of the execution screen.

PAUSE;

This is not a standard PL/I statement. Pause execution; return control to terminal for user command.

SLOW(e);

This is not a standard PL/I statement. Slows the speed of execution by limiting the frequency with which the execution screen is redrawn.

Arithmetic expression e gives the minimum time (in tenths of a second) between redraws.

The default value for e is 5.
SLOW(0) permits execution to run at full speed.

TRACE(n);

This is not a standard PL/I statement. Limit execution tracing to n levels of indentation. TRACE(0) suppresses tracing (and checking). If TRACE(0) is the first executable statement in a procedure, the trace area is not redrawn for this procedure. (The trace of the calling procedure remains on the screen, with the execution-pointer on the calling statement.)

The trace area is not allocated on the screen until actually needed, so if the main procedure is TRACE(0) the lines normally used for the trace area are automatically added to the output area. They are reclaimed when some procedure without TRACE(0) is called.

The default is TRACE(2).

NOCHECK(list of variables);

This is not a standard PL/I statement. Exempt the variables listed from the checking process. NOCHECK statements are not cumulative — each NOCHECK replaces its predecessor.

NOCHECK or NOCHECK() restores checking of all variables.

NOCHECK(A1,..,A11) suppresses all checking.

Checking is subordinate to tracing — TRACE(0) suppresses checking as well as tracing.

Default is NOCHECK — that is, all variables are checked.

A.4. Comments

(1) Ordinary comments can appear only between statements or between the names in a declaration.

(2) "Statement comments", denoted by /** ... */ are used to describe the function of a group of statements. The statements in the body are indented from the header comment.
APPENDIX B

COPE Command Summary

There are only twenty-three “commands” in COPR. Each command initiates a system action. There are no “modes” — any command can be given at any time — the system automatically switches to the appropriate form of display screen for the command entered.

Note that COPI is not a full-screen editing system. User entries are received in a fixed “entry area”, and the commands are defined accordingly. Commands deal with program “units”, not lines or statements. For simple statements such as assignment, GFT, PUT, etc., the unit corresponds to a single statement. But for complex statements such as loops and conditionals, the unit consists of the initiating statement and all lines indented with respect to that statement.

Every command is entered using one of the special keys on the keyboard (or in some cases, by a combination of keys). Commands cannot be entered by their keyword names. That is, the EXECUTE command can only be entered with the special EXECUTE key, and not by typing the word “execute” on the alphabetic portion of the keyboard. The command names are shown in uppercase BOLDFACE in this description to emphasize their association with special keys.

Many of the commands allow an argument (the REPLACE command requires an argument). An argument is a string of characters typed on the normal alpha-numeric portion of the keyboard. The argument is always entered before the command. The argument of a command that allows an argument is the content of the entry area at the time the command is given. The system response to such commands includes clearing the entry area in preparation for receipt of the next command.

On the other hand, commands that do not allow an argument (for example, UP, ERASE, EXPAND, etc.) do not use or affect the entry area. The content of the entry area remains intact as an argument for the next command.

B.1. Notation

The following notation and terminology are used.

- entry the character-string argument
- [entry] an optional argument
- exec-ptr the execution pointer on the execution screen
- edit-ptr the edit pointer on the edit screen
- cursor the character pointer in the entry area

B.2. Commands
[entry] EXECUTE

Switch to execution screen, if not already on screen.

If entry is a file-name:
  Clear .OUTPUT file.
  Reset pointer in .DATA file to beginning.
  Begin execution of specified file (as a main procedure).

If entry is null:
  Resume execution of current procedure at exec.ptr.

If entry is present, but not a file-name:
  Execute entry as (an) immediate statement(s).

[entry] FILE

If entry is null:
  Display edit screen for file of current procedure.
  Set edit.ptr to position of exec.ptr.

If entry is a file-name:
  Switch to edit screen for specified file. If this file is the current procedure,
  set edit.ptr to position of exec.ptr. Otherwise, edit.ptr is preserved from
  last edit of this file.

If entry is present, but not a file-name:
  If necessary, switch to edit screen (for current procedure, with edit.ptr set
  from exec.ptr).
  Insert entry after edit.ptr.

[entry] ENTER

If exec-screen is on display, and execution is paused for data input:
  Append entry to end of whatever input file caused the pause. Execution
  remains "paused for data input", so this command can be repeated.

If exec-screen is on display, and execution is not paused for data input:
  Exactly equivalent to EXECUTE.

If edit-screen is on display:
  Exactly equivalent to FILE.

QUIT

Preserve system state, and terminate session.

UP

Move exec.ptr (for execution screen) or edit.ptr (for edit screen) up 1 line.
Scroll display if necessary.
DOWN
Move exec-ptr (for execution screen) or edit-ptr (for edit screen) down 1 lines.
Scroll display if necessary.

BACK PAGE
Switch to edit screen if necessary, setting edit-ptr from exec-ptr. Then move
edit-ptr back (toward beginning of file) by n lines (where n is the number of
lines in the text-area of the edit screen). This command does not alter the
position of the exec-ptr.

FORWARD PAGE
Switch to edit screen if necessary, setting edit-ptr from exec-ptr. Then move
edit-ptr forward (toward end of file) by n lines; (where n is the number of lines
in the text-area of the edit screen). This command does not alter the position
of the exec-ptr.

LEFT
Move cursor one position left in entry area.
Scroll entry area right if necessary.
In moving the cursor a "prompt" is considered a single character.

RIGHT
Move cursor one position right in entry area.
Scroll entry area left if necessary.
In moving the cursor a "prompt" is considered a single character.

LEFT END
Move cursor to left end of entry area. Scroll entry area right if necessary.

RIGHT END
Move cursor to position following rightmost character in entry area.
Scroll entry area left if necessary.

CLEAR
Clear character at cursor position in entry area, drawing all right-side characters
left one position.
In clearing characters a "prompt" is considered a single character. If the cursor
is positioned anywhere in a prompt the CLEAR command clears the entire
prompt.
ERASE

Erase all characters in entry area from cursor position to right end.

WORD TAB

Move cursor to right to first character of next word.
Scroll entry area left if necessary.

STM T TAB

Move cursor to right to first character of next statement.
Scroll entry area left if necessary.

CONDENSE

Reformat the innermost uncondensed unit containing the edit-ptr or exec-ptr into condensed form.

Condensed form has all statements in a single line (instead of in multiple-line, indented form). However, on the screen all statements after the first on a line are represented with an ellipsis — "...".

EXPAND

Reformat the outermost condensed unit containing the edit-ptr or exec-ptr into normal indented (uncondensed) form.

The format of a particular unit (condensed or normal) is a relatively permanent property. It persists until altered by a CONDENSE/EXPAND command. It applies to both the edit and execution screens.

[entry] FETCH

Switch to edit screen if necessary, setting edit-ptr from exec-ptr.
Insert entire unit denoted by edit-ptr into cursor position in entry area.

entry REPLACE

Switch to edit screen if necessary, setting edit-ptr from exec-ptr.
Replace unit at edit-ptr with entry.
Note that entry is required, hence a null entry is significant and will delete unit at edit-ptr.
[integer-entry] [file-name-entry] MOVE
Switch to edit screen if necessary, setting edit-ptr from exec-ptr.
Clear specified file (default file is .TEMP).
Move units, starting with edit-ptr unit, to specified file.
The number of units is specified by integer-entry; the default is 1.
Regardless of value given, the move will not continue past END of
containing unit.

[file-name-entry] COPY
Switch to edit screen if necessary, setting edit-ptr from exec-ptr.
Copy entire contents of specified file (default is .TEMP) after edit-ptr.

UNDO
Restore the system to the state that existed before the previous command.
The previous command is shown in the previous-command-area of the screen.
Whenever a command appears in this area, the UNDO command can be used
to negate the effect of the command shown. The UNDO command itself is
never displayed in this area, and is ignored in determining what was the
"previous command".
The system will always allow at least the most recent command to be undone;
in many situations it will allow the last several commands to be undone.

B.2.1. Command Summary

| [e] EXECUTE | [e] FETCH | [i] MOVE |
| [e] FILE | [e] REPLACE | [i] COPY |
| [e] ENTER | UNDO | |
| QUIT |

| UP | DOWN | LEFT | RIGHT |
| BACK PAGE | FORWARD PAGE | LEFT END | RIGHT END |
| WORD TAB | STMT TAB | CLEAR | ERASE |
| CONDENSE | EXPAND |
APPENDIX C

File System Extension

1. Program Statements

Brief descriptions of the syntax and semantics of the GET, PUT and DELETE statements follow. Independent of input form, PL/CS always prints out program statements in a canonical order; that order is reflected by the order of the optional phrases in the prototype statements.

UT [FILE(fn)] [PAGE(ps)] [SKIP([n])] [LIST(list)];

UT [FILE(fn)] [PAGE(ps)] [SKIP([n])] [EDIT(list)(list)];

1. If the file name fn is new, this constitutes an implicit declaration of a new file-name and creates a new file. If the file-name is a constant, the value of the constant is the file name; if the file-name is a variable or expression, the value of the variable or expression is the file name. An uninitialized character string variable used as a file-name variable is an error; the name of the variable is not assumed to be the name of the file.

2. If the file name is known, the statement applies to this existing file. If the first statement or pseudo-variable referencing the file is a PUT, the current file pointer is initialized to filesize+1, and the current item pointer to 0.

3. If the SKIP phrase is omitted, items are written starting at the end of the current record, i.e. at REC(file-name), COUNT(file-name)+1.

4. If the SKIP phrase is specified, output begins at the current record plus the skip count n. If the skip count is positive, this causes n-1 null records to be created. If the skip count is zero, output begins at ITEM(file-name), updating existing fields or extending the record as needed.

5. If the FILE phrase is omitted, the statement refers to the standard system output, .OUTPUT.

6. The PAGE phrase controls the output area of the execution screen. PAGE(ps) assigns ps lines to the output area, and clears the area. PAGE clears the output area without changing its size.

7. Both the LIST and EDIT phrases can be omitted. This construction can be used to create null records or to truncate or extend the current record.
GET [FILE(file-name)] [NEXT] [LIST(list)];

GET [FILE(file-name)] [NEXT] [EDIT(list)(list)];

1. If FILE phrase is omitted, GET applies to the standard input, .DATA, the default file for receiving terminal input. If the first statement or pseudo-variable referencing the file is a GET, the current file pointer is initialized to 0.

2. If NEXT is present, REC(file-name) is incremented and ITEM(file-name) is set to 1 before data transmission.

3. LIST format items are read starting with REC(file-name), ITEM(file-name). Record boundaries are ignored. Null records are ignored when searching for the next item of the data list.

4. EDIT format input operates on a the text image that corresponds to either the actual input (text strings) or the print representation of the input (items written by LIST output). ITEM(file-name) is incremented by 1.

DELETE FILE(file-name) [RECORD](n)];

1. If the RECORD phrase is present, this deletes the number of records specified (1 is the default) starting with REC(file-name). REC(file-name) remains unchanged, but refers to the first record following those deleted. The number of records to be deleted must be positive.

2. If the RECORD phrase is absent, this statement deletes the entire file.

C.2. Builtins
Four functions/pseudo-variables (builtins) are provided to position records in the file: REC, REMAIN, KEY, and FIND. A brief definition of each follows:

REC(file-name) – function and pseudo-variable

Returns the current record number (from 1 to the size of the file). The value returned for an unreferenced file is 0; for an undefined file -1.

Assignment to the pseudo-variable causes the current record pointer to be set to the numbered record. Values may be assigned which arbitrarily extend the file: negative values are errors.

REMAIN(file-name) – function

Returns the number of records until end-of-file is encountered on the file. A 0 value means that the current record is the last record in the file. For a file that has not been read, REMAIN(file-name) is the number of records in the file; -1 if the file is not present. For an existing file, REMAIN(file-name) + REC(file-name) equals the number of records.
KEY(file-name) — function and pseudo-variable

Returns the character string value of the key for the current record in the file, file-name. If the current record is unkeyed, KEY returns the key of the most closely preceding keyed record, or LOW if no preceding keyed record exists.

Assignment to the pseudo-variable sets the current record pointer to the record with the assigned key-value. Assignment of a key-value that did not previously exist causes a null record with that key to be created.

FIND(file-name, key-value) — function

Sets the current record pointer at the first record whose key is greater than or equal to key-value and returns the character-string value of that key. If no record exists with a key-value greater than or equal to key-value, HIGH is returned and the current file pointer is set to "end of file".

The UFO is based on records containing variable numbers of items. To take advantage of this facility, two function/pseudo-variables, COUNT (no relation to the PL/I function of the same name) and ITEM, are provided:

COUNT(file-name) — function and pseudo-variable

Returns the number of items in the current record for the file-name.

Assignment to the pseudo-variable changes the number of items in the current record, either truncating or extending the record with null items.

ITEM(file-name) — function and pseudo-variable

Returns the current item number in the current record of the file. The first value for any record is 1.

Assignment to the pseudo-variable moves the current item pointer to the specified position within the record. It is an error to assign a value to ITEM(file-name) that is greater than COUNT(file-name) or less than 1.

C.3. Summary of File Interactions

The effects of each of the file system statements, pseudo-variables and the FIND built-in function are summarized below. Each table entry represents the effect of the execution of the left-hand column construct on the column header.

The value of REMAIN(FileName) will always be the total file size minus REC(FileName). Error conditions are always referred to the user; no function values are changed by errors.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Keyword</th>
<th>Result</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>equal</td>
<td>corresponding record number</td>
<td>1</td>
<td>unprintable character</td>
</tr>
<tr>
<td>REC</td>
<td>key of closest &lt;= record</td>
<td>=</td>
<td>1</td>
<td>argument &lt; 0</td>
</tr>
<tr>
<td>ITEM</td>
<td>unchanged</td>
<td>unchanged</td>
<td>=</td>
<td>argument &lt;= 0 or &gt; COUNT</td>
</tr>
<tr>
<td>COUNT</td>
<td>unchanged</td>
<td>unchanged</td>
<td>COUNT+1</td>
<td>argument &lt; 0</td>
</tr>
<tr>
<td>FIND</td>
<td>key of closest &gt;= record</td>
<td>corresponding record number</td>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td>GET</td>
<td>key of closest &lt;= record</td>
<td>current record pointer</td>
<td>updated</td>
<td>attempt to read past end of file</td>
</tr>
<tr>
<td>PUT</td>
<td>unchanged</td>
<td>current plus SKIP count</td>
<td>updated</td>
<td>none</td>
</tr>
<tr>
<td>DELETE</td>
<td>key of closest &lt;= record</td>
<td>unchanged</td>
<td>1</td>
<td>records to be deleted &lt;= 0</td>
</tr>
</tbody>
</table>
APPENDIX D

Unrestricted File Organization

The UFO is central to the implementation of the extensions to the PL/CS file handling facilities and to the implementation of the entire system. The first part of this appendix describes the access method routines that implement the UFO. The second part uses the UFO routines to implement the file system constructs.

D.1. Access Method

Three file management routines are provided:

- **Create:** find a file; creating new if necessary
- **Access:** reference an existing file; error return if none exists
- **Destroy:** remove a file from the directory

In addition, the directory is accessible as a read-only file containing a record for each file. The directory is readable in the same form as any user file; file-names form the set of keys of the directory record; the index information for the file are the items of the records (its all done with mirrors).

File manipulation is carried out using the following routines. All routines have an FD parameter not shown. The routines are intended as a system, not a direct user, interface. As a result, only the primitives required are supported; some “reasonable” logical operations require more than one primitive call. Basic communication is controlled by record number and item number. Items are treated generically for simplicity, even though the coding algorithms discussed above would require type information. Parameter abbreviations are: C (count), D (data item), I (item number), K (key), and R (record number). All parameters are assumed to be passed by reference and are updated as required.

CreateRecord(R, C):
Create C new records immediately after the one numbered r; R is updated accordingly.

CreateKey(K, R):
Creates a new, keyed record at the appropriate position in the file; returns the position of the record in R. If the key already exists, the new record is not created, but R is set.

LocateKey(K, R):
Finds the position in the file corresponding to where the key K would go; returns the position in R; updates K if the passed value is not present.

ReadKey(R, K):
Returns the key associated with record R in K.
ReadItem(R, I, D):
Reads the next item, starting at item I of record R into D; for the item actually read, returns the record in R and one past the number of the item in I.

WriteItem(R, I, D):
Writes the value D at item I of record R. If the item exists, it is replaced. Otherwise, the item is written at the end of record R; I is updated to one past the number of the item actually written.

FindItem(R, I):
Checks for the existence of item I. If item I does not exist, I is set to the last item that does exist; a non-positive value for I assures that the last item of the record will be found. If record R does not exist, R is set to the last record in the file and I is appropriately updated.

TruncateRecord(R, I):
Truncate/expand record R so that it contains exactly I items.

DeleteRecord(R, C):
Delete C records starting with record R. None of the records need exist; R is unchanged; C is updated to reflect the number of records actually deleted.

D.2. FSE Implementation

The following outline the operations required to implement the FSE. To simplify the code, all cases assume that the file referenced has been previously accessed and that statically discernible errors have been detected. The variables, RecordNumber, ItemNumber, FileSize, and Key are used as if they were specific to the file being processed. The FD parameter has been dropped from each of the UI/O calls.

DELETE(File Name) RECORD(n):

CALL DeleteRecord(RecordNumber, n);
FileSize = FileSize - n;
GET FILE(FileName) [NEXT] [LIST(data items)];

IF "NEXT phrase is present"
THEN DO;
    RecordNumber = RecordNumber + 1;
    ItemNumber = 1;
END;

DO "for all data items, d";
    CALL ReadItem(RecordNumber, ItemNumber, d);
    IF RecordNumber > FileSize
THEN ERROR;
END;

PUT FILE(FileName) [SKIP(n)] [LIST(data items)];

IF "SKIP(n) phrase is present"
THEN IF (n > 0)
    THEN DO:
        /** create n records */
        CALL CreateRecord(RecordNumber, n);
        /** adjust ItemNumber and FileSize */
        ItemNumber = 1;
        FileSize = FileSize + n;
        END;
ELSE; /* SKIP(0); no action required */
ELSE DO:
    /** no SKIP phrase, find the end of the record */
    ItemNumber = -1;
    CALL FindItem(RecordNumber, ItemNumber);
END;

DO "for all data items, d";
    CALL WriteItem(RecordNumber, ItemNumber, d);
END;

COUNT(FileName) built-in function

c = -1;
CALL FindItem(RecordNumber, c);
RETURN(c);

COUNT(FileName) = n;
CALL TruncateRecord(RecordNumber, n);
FIND(FileName, k) built-in function

Key = k;
CALL LocateKey(Key, RecordNumber);
RETURN(Key);

ITEM(FileName) built-in function

RETURN(ItemNumber);

ITEM(FileName) = n;

r = RecordNumber;
tempn = n;
CALL FindItem(r, tempn);
IF ((r = RecordNumber) & (tempn = n))
THEN ItemNumber = n;
ELSE ERROR;

KEY(FileName) built-in function

CALL ReadKey(RecordNumber, Key);
RETURN(Key);

REC(FileName) = n;

IF (n > FileSize)
THEN DO;
CALL CreateRecord(FileSize, n-FileSize);
FileSize = n;
END;
RecordNumber = n;

REC(FileName) built-in function

RETURN(RecordNumber);

REMAIN(FileName) built-in function

RETURN(FileSize-n);
Bibliography


[54] FOCS or SIGACI paper on data compaction.


[57] Teitelman, W., "A Display Oriented Programmer's Assistant", Xerox PARC, CSL77-3


