ENHANCED MANIPULATIVE CAPABILITIES FOR A SYNTAX-BASED EDITOR

by

Leland R. Dykes

B.A., University of California at Riverside, 1964

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School of

Computing Science

SIMON FRASER UNIVERSITY

September, 1988

APPROVAL

Name: Leland R. Dykes

Degree: Master of Science

Title of thesis: Enhanced Manipulative Capabilities for a Syntax-based Editor

Examining Committee:

Chairman: Dr. Binay Bhattacharya

Dr. Robert D. Cameron
Senior Supervisor

Dr. Louis J. Hafer

Dr. Anthony H. Dixon
External Examiner

Date Approved: September 15, 1988

PARTIAL COPYRIGHT LICENSE

Title of Thesis/Project/Extended Essay

Enhanced Manipulative Capabilities for a Syntax-Based Editor

Author:

(signature)

Leland R. Dykes

(name)

15 September 1988

(date)

ABSTRACT

Various tools have been developed to help meet present day demands for more and better software. One such tool is the syntax-based program editor (SBE). SBE implementations feature program generation and utilize the editor's knowledge of target language syntax to facilitate the inspection and selection of program components. Typically, only minimal means for the manipulation of selected components are provided.

Such an editor was constructed to explore the possibility of enhancing the capabilities of the SBE with a more powerful manipulative facility. Though somewhat limited in its overall functionality, the editor has proven successful as a medium for the development of such a facility, and as a platform for its demonstration. Manipulation commands have been collected, implemented, and organized into a number of families which, it is claimed, correspond to some of the kinds of operations which programmers perform on their programs (and those of others). A feature of this facility is the partial integration of the code production process into the manipulative process.

During the latter stages of its development, the editor has been used for the correction and enhancement of its own source code. In this context the new command families have shown promise when evaluated in terms of both objective and subjective criteria.

DEDICATION

For A.M. and M.S.D.

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Dr. Rob Cameron, for inspiration and guidance, and my other examiners, Dr. Lou Hafer and Dr. Tony Dixon, for their criticism and suggestions.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER I

INTRODUCTION

Traditionally - if it is meaningful to speak of tradition with regard to so youthful a discipline as computing science - hardware costs have dominated those of software. Today just the opposite is the case. The 4:1 ratio of hardware costs to software costs that was typical in the 1950's has in all likelihood been reversed [Toy84]. Technology has produced a new kind of computer: small, powerful and, perhaps most important, affordable. The person operating such a computer is increasingly less likely to be a computer professional [SGW81]. More importantly, the accelerated rate at which computers are assuming control of high-risk processes in transportation, the energy and aerospace industries, medicine and so-called defense systems [Lev86], suggests that there is a growing demand for software not only in increasing quantity but of increasingly high quality as well.

Among the approaches that have arisen to deal with this problem is the consideration of tools and environments dedicated to what Warren Teitelman has called automated programmering, i.e.,

> "... developing systems which automate (or at least greatly facilitate) those tasks that a programmer performs other than writing programs: e.g., repairing syntactic errors ..., generating test cases, making tentative changes, retesting, undoing changes, reconfiguring, massive edits, *et al*, plus repairing and recovering from mistakes made during the above."

The aim is to free the programmer from the uninspiring drudgery at which machines excel.

> "When the system ... is cooperative and helpful with respect to these activities, the programmer can devote more time and energy to the task of programming itself, i.e. to conceptualizing, designing and implementing. Consequently he can be more ambitious and productive."[Tei84]

It is to be hoped as well that there will be a reduction in mechanical errors and that as a consequence, more reliable programs will result.

Programs are not simply text; they are highly structured entities. A program development tool which is "knowledgeable" about a given programming language's formal structure, a syntax-based editor [1], would seem to have the potential to be an important component of such an automated programmering environment, even to serve as its "backbone" perhaps.

Indeed, a substantial amount of research and developmental effort has been devoted to such editors. In Chapter II I give a brief survey of such work and identify a neglected area of research which I have termed the *manipulative facility.* An implementation was undertaken to investigate the possibility of developing such a facility. In Chapter III the aims of the implementation are stated more definitely and major issues introduced. Chapter IV is devoted to a description of the basic editor, which serves as an environment for the more innovative facility. Chapter V describes the manipulative facility itself, and in Chapter VI it is seen in action. Finally, in Chapter VII I look at the significance of the project and point out some possibilities for future work.

[1] In this thesis I use the following terms interchangably: syntactic editor, syntax editor, syntax-directed editor, syntax-based editor and SBE. As noted in the text, their referent is the subject of Chapter II of the thesis. "Syntax editing" and "syntactic editing" simply refer to the knowledgeable use of such an editor. "Structure editor" and "structural editing" are somewhat more general terms. I hope that when they appear in the paper, their meanings will be evident from their context.

CHAPTER II

SYNTAX-BASED EDITORS: AN OVERVIEW

Characteristics of Syntactic Editors

Some text editors support structure-based editing to a degree. The extensible display editor EMACS [Sta84] can be programmed to understand some of the syntax of the language being edited and to provide operations particular to it such as automatic indentation. The text-based editor Z [Woo81], using indentation conventions, provides a number of structural editing features, including selection of syntactic entities, zoom-in and zoom-out. Brun *et al* [BBS85] have developed a token-based editor which supports automatic formatting. However, the syntax-based editor proper is characterized by the maintenance of some internal representation of the program's structure, typically a parse tree (attributed in the case of systems which maintain contextual correctness), plus some means of generating and acting upon that internal representation.

During the latter years of the 1970's, several such editors were developed. MENTOR [DHK84] is a general system for the manipulation of structural material. A programming environment for Pascal, including a structure editor, has been implemented under MENTOR. The programming system Pathcal [Wil84] features incremental execution and is distinguished by the use of Pascal, extended by the addition of the data type "Code," and its consequent extensibility. Yet another editor is SED [All83], which manipulates the tree structure by means of tree matching and substitution in a manner at least superficially analogous to the string matching facilities supported by the more advanced line editors. The above three editors share two common characteristics: textual input of programs (with parsing to generate the internal representation) and a user interface analogous to those of conventional (non-visual) line editors.

In contradistinction to this their contemporary, the Cornell Program Synthesizer [TeR81], features program generation by means of the expansion of program templates (supplemented by textual entry). Editing is performed visually, using a pretty-printed textual representation, a cursor whose motion increments are syntactically meaningful, and a small set of editing commands (*Clip*, *Insert* and *Delete* in the CPS terminology). The syntax editor provided by the SUPPORT programming environment [Zel84] is quite similar, but has "browsing" facilities which are more sophisticated, i.e., holophrasting (the suppression of textual detail), zoom and multiple windows. These, then, are what I would describe as the basic characteristics of the fully developed syntax-based editor: template-based generation of syntactically, and sometimes semantically, correct programs, generally represented internally by a parse tree, full-screen editing with various syntax-directed aids to the inspection and selection of program parts, and a primitive set of manipulation commands.

Research Directions

Various editors have been implemented with some or all of the above features, including at least one commercial effort [San87] designed for use on personal computers. Research involving these syntax-based editors has gone in a number of identifiable directions.

Integration issues

Typically, syntax-based editors do not stand alone. Rather, they are associated with, or integrated into, software development environments. A number of researchers have concerned themselves with what Leon Stucki has referred to as "CAD/CAM for software" [Stu84] - that is, the creation of integrated software engineering environments with supporting methodologies and providing improved management/technical project analysis and control capabilities.

Editor generators

Editor generators are programs designed to automate the implementation of editors for particular target languages. Encouraged by their results with the "hand-made" Cornell Program Synthesizer, Reps and Teitelbaum developed the Synthesizer Generator [ReT84]. Under their methodology, the editor designer prepares an attribute-grammar specification for the target language, from which the Generator creates a full-screen, syntax-directed editor. Another such effort is the PSG System [BaS86] which generates an environment consisting of a language-based editor, an interpreter and a fragment library system from an entirely non-procedural specification of a language's syntax, context conditions and dynamic semantics.

Browsing

Some researchers have concentrated on the user interface and methods of inspection and selection. For example, Schneiderman *et al* [SSS86] have implemented what they call *embedded selection* to access detailed information about a given symbol, and an *hierarchical browser*, which suppresses detail providing a top-down view. The PECAN system [Rei84] provides (simultaneously, if desired) *many* views of a given program, e.g., the syntax-directed editor view, a Nassi-Schneiderman view, a symbol table view, a data type view, etc.. The declared aim of the PECAN project is support for graphical programming.

Syntactic editing style

It is evident that the work mentioned immediately above is concerned with the evolution of what might be termed a "syntactic editing style." With a possibly similar motivation, Alberga *et al* [ABL84] have incorporated into their editor/environment a self-monitoring facility to determine which commands are being used and which are not, and to detect associations between various commands (e.g., frequently used sequences).

Constraint enforcement/language extension

The syntax based editor can be used to enforce programming policies or constraints - on visibility for example - not supported by a given target language. This approach is exemplified by Yggdrasil [Cap85] an otherwise typical syntax-based editor which imposes upon the abstract syntax tree its own language independent concepts of naming and scope.

More powerful manipulations

As noted above, great attention has been paid to the inspection and selection of syntactic entities. However, when it comes to manipulations of selected entities, editors generally offer little beyond the *Clip/Insert* facility of CPS. Some attention has been given to the matter. Though not a fully fledged syntax editor, as defined above, the MENTOR-based Pascal environment does offer some simple transformations. Similarly, Atkinson *et al* [AMN81] have extended the editing commands in their system with a *Qualify/Unqualify* command. In neither case, however, has there been a systematic effort to develop the notion further. This would appear to be a neglected research area.

My Approach

Of the three dimensions of syntactic editing - code generation, inspection/selection, and local manipulations - the third seems to have been the least explored. Consequently, it has become the focus of my research effort. The editor supporting this effort is in the tradition of those described in the first section of the chapter, though not so powerful, outside the realm of manipulations, as many of these. Certainly, a long range aim of this research has been to encourage the emergence of a structure-oriented editing style. These topics will be dealt with at length in subsequent chapters.

CHAPTER III

INTRODUCTION TO THE IMPLEMENTATION

Aims of the Implementation

The implementation consists of a syntax-based editor with certain features emphasized. In previous chapters I hinted at the motivations behind the implementation. They are listed more fully here.

1. I have pointed out the lack of attention given to the manipulative capabilities of SBE's. The principle aim of my research has been to address this lack by looking at the development of such capabilities.
2. Underlying this aim is the more general desire to contribute to the evolution of a "structure-based editing style" by providing an environment which supports such a style.
3. In the section on infrastructure, below, I describe the system and methodology underlying the editor. Any non-trivial implementation serves to exercise this system/methodology, demonstrating its capabilities and, perhaps, revealing some of its shortcomings.
4. Finally, it is always to be hoped that a research project will encourage serendipity, that the effort of the implementation will facilitate the emergence of happy accidents providing insights into matters peripheral to the main goals of the work.

It seemed reasonable to hope, as well, that one of those "happy accidents" might consist of the implementation's serving as a prototype component for software development environments.

Motivated by the above considerations, I have undertaken the construction of an editor. In the remainder of the chapter I discuss some decisions which had to be made before implementation could begin, the system underlying the editor, the various functional components of the editor, and the issues associated with each of those components.

Preliminary Design Decisions

The hypothetical user

When developing any tool, it is essential to identify a target user and the use to which the tool will be put. Certainly an editor aimed at naive users and designed to help them to learn good programming practices will look different from one aimed at experienced programmers involved in the production of useful software. I have chosen as my hypothetical target the latter case.

This choice has a number of consequences. For one, imposition of a coding methodology upon the user is not desirable. The programmer should be free to work with any syntactically meaningful fragment of code, developing in a top-down, bottom-up or inside-out fashion and combining fragments at will. For another, emphasis shifts away from program generation to maintenance and enhancement of programs, since these account for as much as 80 percent of real-world software costs [Toy84]. This necessitates flexibility in the mode of entry of code into, and output of code by, the system. Moreover, any such alteration threatens to introduce new errors into the code [1]. A manipulative capacity which helps to reduce textual entry - and the mechanical errors which attend it - is, therefore, desirable.

Concepts and terminology

In keeping with the stated goal of flexibility it was decided that the editor should accept for editing not just complete programs, but any syntagm, i.e., any syntactically meaningful program fragment. It follows from the editor's syntax-based nature that the elements of interest within the syntagm being edited will themselves be syntagms. The internal representation of a program or program fragment is a tree, the abstract syntax tree, and the elements of interest correspond to its nodes.

----

[1] Adams [Ada84] estimates that for each fix installed the probability of introducing a new error is some 15 percent.

Consequently, it seems natural to think of, and refer to, these elements by the term *node*.

If actions are to be performed on or using nodes, there must be some means to designate the targets of these actions. What is necessary is a conceptual entity which moves about the parse tree in a manner analogous to that of the cursor on a terminal's screen. I have chosen to call the entity the *syntactic cursor*. At any given time the node designated by the syntactic cursor is the *current node*, and as such will serve, typically, as the operand for manipulations performed by the editor. An operation might conceivably require one or more nodes in addition to the current node. A stack seems a reasonable abstraction for keeping track of such nodes, so provision was made for a *node stack* onto which references in the parse tree might be pushed.

The issue of correctness

A fundamental tenet of syntax-based editing is that syntactic correctness, at least in the context-independent sense, is maintained. As well, semantic or contextual correctness is also generally maintained, or at least checked. This presents problems if one desires, as I do, to retain the capability to deal with arbitrary (but syntactically meaningful) program fragments. Bahlke and Snelting [BaS86] have implemented an ingenious solution which considers nodes in the internal tree to have associated with them relationships of attributes. Provision for such a capability entails considerable overhead and may introduce problems. Encumbering the parse tree with contextual (or other) information may hinder or even preclude implementation of some editing operations [ABL84,AMN81]. Since the issue is peripheral to the primary aims of the research, my approach has been to restrict myself to syntactic correctness of the context-independent sort. In the case of editing operations which require context-dependent knowledge, the responsibility for assuring that the necessary context is present is placed upon the user. If he/she fails to do so, the operations simply fail, albeit gracefully and with appropriate messages.

Infrastructure

Underlying a syntax-based editor is some internal representation of the target program. For my implementation this internal representation and the means to deal with it have been provided by Multi MPS [Cam86], a package of subroutines that have been generated in accordance with the methodology known as GRAMPS [Cal84], the GRAMmar-based MetaProgramming Scheme. Under GRAMPS, if one wishes to develop a metaprogramming system, i.e., a system which facilitates the writing of programs which take other programs as their data objects, one begins by codifying the grammar of the language in which the data-programs are written (the *target language*), in an augmented BNF form. One then constructs, *based on that grammar*, a package of subroutines in the language in which metaprograms are to be written (the *host language*). These routines include parsers and unparsers and the recognizers, selectors, constructors and basic editing routines necessary to manipulate the parse tree. In effect, a GRAMPS-based system like MPS enables the programmer to deal with the program's parse tree as an abstract data type. MPS has been particularly suitable for my implementation in that it has facilitated the extension and experimentation which were used to develop more powerful manipulation capabilities for the editor.

At the time of my project's inception, MPS was supported only on the Michigan Terminal System (MTS), running on the university's IBM 3081 GX mainframe. Consequently, my editor runs on this system as well. MTS provides a package of routines which facilitate, or at least enable, the implementation of full-screen applications utilizing the various CRT terminals on the campus-wide system. This package was used to construct the editor's interface. This environment imposed some limitations upon the implementation, particularly the development of the interface. Since these limitations, described in Chapter IV, had no immediate impact upon the primary research area (the manipulative facility), they were deemed to be acceptable.

The target language for my implementation is Pascal [JeW85], a language with wide acceptance in academia and a degree of acceptance in industry. In accordance with the GRAMPS methodology underlying Multi MPS, the language-independent core of the editor can be applied to other target languages as their grammars are defined and the corresponding sets of routines constructed. So far this has only been done for Modula2.

The host language is Pascal as well. Having the same language as host and target in an experimental implementation of a software tool can be advantageous. One is assured of a supply of test code, the implementation itself, and as the power of the tool grows, "bootstrapping" becomes a possibility.

The Editing Facility

A syntax-based editor has a number of identifiable features. In each of the three sub-sections below I identify such a feature and briefly sketch its associated issues. Since it is the *raison d'etre* of the implementation, the fourth feature, provision for more powerful local manipulations, has been allocated its own section.

Interface

As pointed out above, the syntax-based editor actually works upon the internal representation of a target syntagm, but some external representation must be provided to the user. Though various tree-like or graphical representations might be devised, it has been my experience that programmers are generally not enthusiastic about such representations. Consequently, the conservative approach of representing programs as pretty-printed text has been used. That is, code is displayed with conventions of indentation and capitalization enforced. The full-screen, visually oriented approach is now the norm in text editing, and has served as the model for my editor's interface. Issues that have had to be dealt with include the effective use of the screen, the use of prompts and messages, the input of commands, and cursor motion (see Inspection/selection, below).

Program entry/generation

A standard component of many existing syntax-based editors is the program generation facility. Precisely because this issue has been extensively dealt with and because of the orientation toward the editing of existing code noted above, I have not devoted great attention to the matter of code generation except as it relates to the editors manipulative capabilities. What has been provided for is input and output of syntagms to files, storage and retrieval of syntagms in their internal form, and the integration of text editing capabilities into the editor.

Inspection/selection

Possession of syntactic knowledge gives an editor the potential to go far beyond textual scrolling as a means of "traveling" through code. This potential has not gone untapped. As noted in Chapter II, various researchers have looked at various means including a zoom facility, holophrasting, the hierarchical browser and pattern or structure matching. These are well-researched topics, and I have not concentrated on them. I *have* found that if the editor was to be at all useful, something beyond the minimal capabilities was required. Consequently, a suite of search operations has been proposed (and some of them implemented).

In addition to examining code, it is also necessary to select fragments for manipulation. The conceptual basis for this capability was described in the previous section. In concrete terms, the problem has been to determine what the implementation of the so called syntactic cursor would look like: how it would be set and moved, how it would be represented internally and externally. Means had also to be found of dealing in a natural and unobtrusive way with problems of selection and ambiguity arising from the dichotomy of the internal (tree-based) and the external (text-based) representations.

Manipulations

Higher level manipulations - the problem

As I have pointed out, one neglected area of SBE research, and the focus of my efforts, is that of the manipulative or transformational facility. Using the capacity provided by MPS to construct routines to perform arbitrary transformations, guided by intuition, and borrowing from the realm of transformational programming [PaS83] one could certainly provide extensive functionality in the form of a voluminous library of manipulations. The problem then has been to identify, implement and organize a more manageable set of operations which capture, at least to some extent, the pragmatics of program alteration and enhancement.

It is arguable that *no* built-in higher level manipulations should be provided, that one should simply provide the capacity for extensibility, so that the user could add manipulations when, and as, needed. Certainly some form of extensibility is desirable and is supported in my implementation. I believe, however, that it is not sufficient to do so for the following reasons.

1. Users should not be forced to "re-invent the wheel." If useful and usable operations *can* be provided, why not do so?
2. Even when powerful routines on which to base them are available, writing the code for non-trivial transformations can be a non-trivial task. Lacking examples of the potential of this approach, the user may be hesitant to take on such a task.
3. In light of point #2, above, it would seem appropriate to provide the user with such examples. This re-introduces the subject of investigations into the notion of a "structure-based editing style."

It would be nice to formulate a model of software modification on which to base this facility, but our current knowledge of how programmers actually work is inadequate for this. Research using a "top down", cognitive approach to the problem [Sch76] has suggested that programmers do indeed recode syntactic forms into internal structures that represent the semantic structure of a program, but conclusions as to the nature of that recoding and of the internal structure are tentative at best [Sch80,Sol86]. The "bottom up," analytic approach [GMH85] has given an indication of *where* changes occur but has shed little light on the nature of the changes.

Approach of the current work

Given this lack of a theoretical underpinning, it has been necessary to rely upon experimentation to develop the manipulation facility. To this end, means were developed whereby commands could be added to the editor at load time. Consideration and implementation of commands has been guided by a number of principles. These may be grouped into two general strategies: reduction of the number of commands and organization of the resulting set of commands.

One way of reducing the number of commands is to select a set of formal, low-level transformations [Ars79,BuD77,Lov77]. However, performance of even modestly complex transformations can involve using many such low level commands organized in ways that are complicated and difficult to comprehend [Dar84]. Moreover, this approach is characterized by an insistence upon the strict preservation of correctness, with execution of transformations dependent upon various enabling conditions. This is too limiting for a general purpose program editor.

A second approach, that of the current work, is what might be called the intuitive overloading of commands. In addition to a formal syntactic structure, programs have an informal structure to them, awareness of which can be used to motivate the overloading of commands. For example, programs contain various kinds of lists, and a single command could be applicable to various kinds of list. Moreover, the same command might be applied, with some intuitively meaningful effect, to "list-like" objects such as the else-if clauses of an if statement, e.g., the exchange of the two clauses of

```
IF e0 THEN s0 ELSE IF e1 THEN s1
```

to yield

```
IF e1 THEN s1 ELSE IF e0 THEN s0.
```

Similarly, many diverse objects exhibit "nesting" - record declarations, if...else statements, looping constructs, etc. - and conceivably commands could be extended to cover all. Cameron [Cam87] has used this approach with some success. The problem then has been to detect such structure in programs and to determine what actions are intuitively similar for the differing cases.

Some specialization of commands is necessary if the program is to have sufficient power and usability. It has been found [Bro77] that applications with relatively large numbers of distinct commands can still be effective if the commands can be clustered into smaller sets corresponding to conceptual work units. An attempt has been made to discover such functional clusters. The results of this attempt are discussed in Chapter IV.

The interface is important in this regard as well. Within the limits set by the system on which the editor is implemented, the interface must support and enhance the organization of operations so far as possible. A related issue is that of the integration of commands. The manipulation facility will be effective to the degree that sequences of commands can be composed.

CHAPTER IV

THE EDITING FACILITY

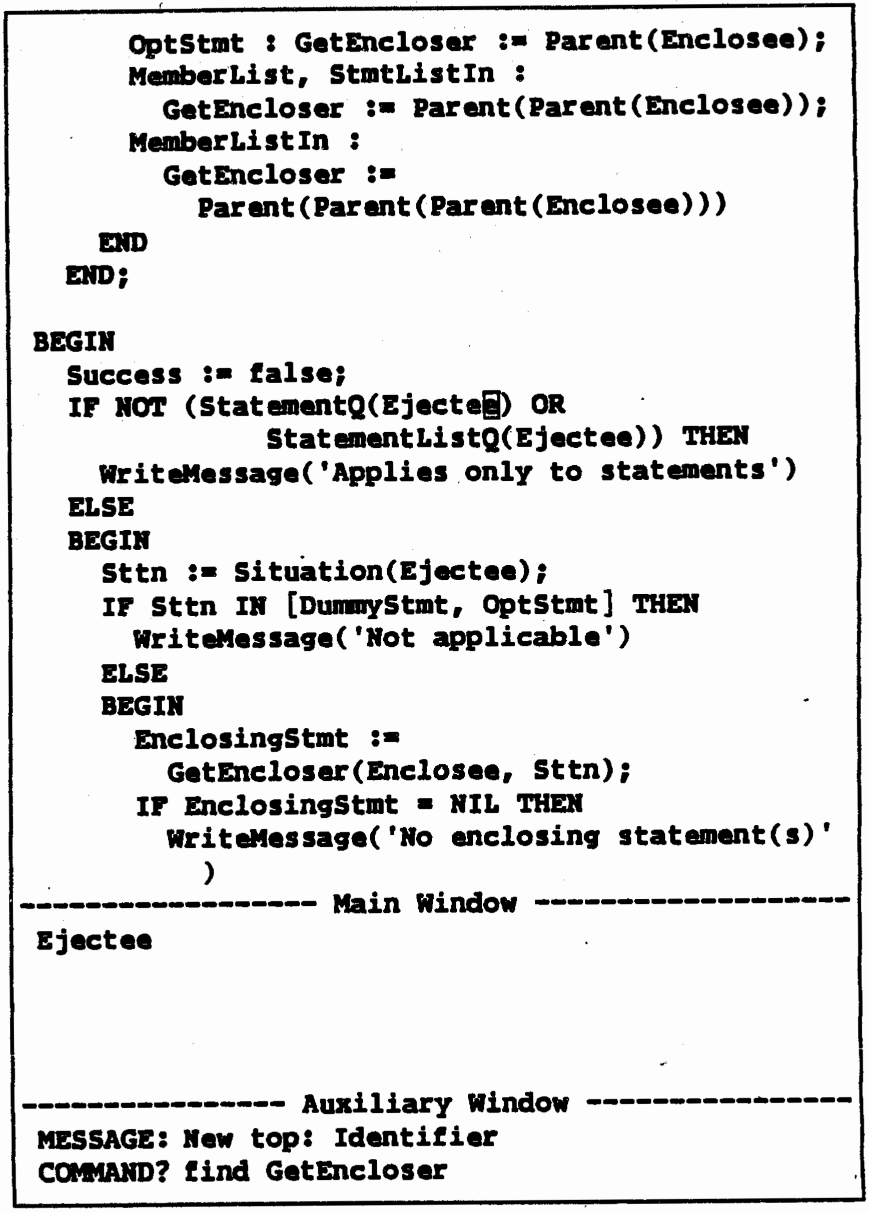
The manipulative facility does not exist in a vacuum. As background, it is worthwhile examining the environment in which it was developed and into which it has been integrated. Therefore, this chapter is devoted to what I have chosen to call the editing facility, i.e., the interface, the code production facilities, and the means utilized for the inspection and selection of programs and their fragments. The typical approach to each of the various components or aspects of the implemention is to consider some of the issues associated with that particular component/aspect, to describe the approach actually taken (generally with some rationale), and finally to suggest any improvements which seem desirable.

Interface

Screen Appearance

The appearance presented by my editor (Figure 4.1) is not altogether different from what one would expect of one of the more advanced line-based text editors. The screen is divided into four horizontal areas or windows. Top-most, and occupying the better part of the screen, is the main or editing window. Conceptually, this is a window into the text resulting from the unparsing (i.e., prettyprinting) of the node currently being edited. The user can, by means of function keys, scroll the window up and down in the (virtual) unparse in a manner familiar to users of conventional text editors. Using this capability, together with other capabilities described in the ensuing sections, the user performs the actual editing operations. Note that only the syntax-based operations provided by the editor may be used. Textual entry is not permitted in this window.

Immediately below this main window is a smaller auxiliary window. This window has two distinct functions. First, it is used for the read-only display of nodes. The editor provides a stack on which to save references to selected nodes. The top



```
      OptStmt : GetEncloser := Parent(Enclosee);
      MemberList, StmtListIn :
        GetEncloser := Parent(Parent(Enclosee));
      MemberListIn :
        GetEncloser :=
          Parent(Parent(Parent(Enclosee)))
    END
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Ejectee) OR
              StatementListQ(Ejectee)) THEN
    WriteMessage('Applies only to statements')
  ELSE
  BEGIN
    Sttn := Situation(Ejectee);
    IF Sttn IN [DummyStmt, OptStmt] THEN
      WriteMessage('Not applicable')
    ELSE
    BEGIN
      EnclosingStmt :=
        GetEncloser(Enclosee, Sttn);
      IF EnclosingStmt = NIL THEN
        WriteMessage('No enclosing statement(s)'
          )
------------------ Main Window ------------------
Ejectee




---------------- Auxiliary Window ----------------
MESSAGE: New top: Identifier
COMMAND? find GetEncloser
```

Figure 4.1: Appearance of the screen [1]

node on the stack is displayed here, either on command or when it is changed by a push or pop operation. In earlier versions of my editor, this window was used for the display and selection of optional nodes. An inline representation of such nodes has been implemented rendering the window's use in this context redundant. A simple "display-current-node" feature, designed to help identify the current node, has

[1] Illustrations for the thesis have been produced by routines which dump the screen contents, inserting text processor commands where necessary, e.g., the commands which draw the outer box representing the boundaries of the screen and those which draw the character-sized box which represents the screen's cursor.

replaced the earlier feature. Implementation of highlighting of the current node would, in turn, render this feature obsolete. The auxiliary window is also used by my editor's on-board text editor. In this context it functions as a conventional, if somewhat small-screened, visual editor.

These windows, and the two others to be described below are resident and static, i.e., they are always present on the screen and their dimensions do not change. On terminals with deep screens (40 lines or more) this is quite satisfactory. The main window is deep enough for the display of a significant portion of code, while the user is spared the potentially distracting screen alterations that would be required if the auxiliary window were transient or dynamic in nature. Moreover, the implementor has been spared the effort of constructing the more complicated system. On terminals with the conventional 24 line screen, the main window is small enough that it is sometimes annoying. It would, therefore, seem desirable to provide the user with the capability to toggle (using a function key) between a *resident mode*, i.e., the functionality of the current implementation, and a *transient mode*, in which the auxiliary window only comes into being when necessary, with the main window expanding and contracting in depth as appropriate.

Below the auxiliary window are two more windows. They are very small, each occupying a single line of the screen. The upper of the two, displays the prefix `MESSAGE:` [2] and the other the prefix, or prompt, `COMMAND?`. The message window is used by the editor to communicate information to the user. The messages are of various types: syntactic information, e.g., the class of the current node, announcements about the internal state of the editor, e.g., `Stack now empty`, and error messages, which are generally formatted in such a way as to help the user utilize the features of the editor, e.g.,

```
Usage: PARSE <filename> <nodeclass> [<nodename>].
```

----

[2] Text which one would expect to see on the screen - static labels or prompts, dynamic prompts or messages, user-entered commands or parameters, or Pascal code - is set in `this typeface` (which looks `LIKE THIS` in uppercase).

The field adjacent to the COMMAND? prompt is the one part of the screen where the entry of characters from the keyboard has been enabled (disregarding, for now, textual editing mode). This area is used for the entry of commands and their parameters and other textual material required from time to time. The user simply positions the screen's cursor, types the required text and hits <RETURN>, or some other relevant key, to cause the data in the command window to be read. Once entered, the text is persistent and may be repeatedly reread by striking the appropriate key.

Interaction issues

In the paragraphs above I have described what the editor looks like and discussed, briefly, how it "talks" to the user. In subsequent sections I will relate how code is generated and inspected and how operands are selected and manipulated. Here I wish to deal in greater depth with the issue of input of instructions to the editor. That is, how the user talks to the editor.

There are a number of criteria to use when assessing a potential mode of command input. It will be seen that in some cases these criteria conflict with one another, and that trade-offs must be considered. The criteria are:

1. Economy - All else being equal, it is desirable to minimize the number of keystrokes required for any operation. Programmers are paid to think, not type.
2. Mnemonic value - Any method used should minimize the amount of memorization required of the user and should be as helpful as possible with respect to that memorization which is unavoidable.
3. Naturalness - This is, admittedly, a subjective criterion. I do believe that some procedures can be seen to be manifestly awkward, and as such they should be avoided.
4. Familiarity - This is closely associated with the criterion immediately above (and *vice versa*). That with which we are familiar tends to

seem most natural. It would seem to be advantageous to stick to the familiar where possible (in the interface) and to save the experience of unfamiliarity for where it is inherent in the problem (editing programs as structured entities rather than mere text).

5. Open-endedness - Imposition of arbitrary limits on the number of commands supported is certainly undesirable.
6. System support - This last criterion while theoretically the least important is, from the practical point of view, most important. What we build is, to a great extent, constrained by the availability of tools and components with which to build it. For example, an interface which would be attractive and effective if implemented on a workstation with support for windowing, menus and the "mouse," may well be messy and unwieldly when implemented on a less sophisticated system (assuming that it could be implemented at all).

Let us consider, in light of the above, the relative merits of three input methods: typed commands, function keys and menus.

Typed commands

The use of typed commands as the sole means of input would seem to defeat the purpose of adopting the visually oriented approach. However, this method does have some advantages. To begin with, communication with the computer by means of command lines is certainly familiar, at least to anyone who has used either of the major systems at this institution, UNIX [3] and MTS. Moreover, it is a most natural form of entry for certain types of commands, i.e. those which do not require selection of an operand from the screen but do require one or more arguments from the user. Unlike (unlabelled) function keys, but not menus, typed commands may have mnemonic value. And this is an open-ended method. Subject to the constraints discussed below, you may have as may such commands as you like.

----

[3] UNIX is a trademark of AT&T.

On the other hand, the entry of character strings is most uneconomical in terms of keystrokes. The number of strokes can be reduced through the use of abbreviation, but only at the expense of mnemonic value. Although their spellings help the user to remember the mappings between string and action, the user still has the burden of remembering the entire set of commands (strings). Finally, this is not, in many instances, the most natural mode of entry. The typical visual editing operation has the form:

-select an operand with the cursor

-invoke some operation.

In this case an awkward procedure would be imposed upon the user. Since the MTS screen support routines permit input only through application-assigned fields in the screen, the user would be required to move the cursor to such a field, or, alternatively, the editor would have to be exited briefly to allow input.

Function keys

Perhaps the most obvious advantage of entry by function key is economy. A minimal number of keystrokes is required in this mode. Of equal importance is its naturalness and familiarity. The place-cursor/strike-key pattern is straightforward (at least for single argument commands) and anyone who has used a visual text editor is familiar with it. Finally, it is easily implemented under MTS. The screen support abstraction provided by that system encourages one to structure an interactive program as an await-function-key/interpret-key/process-instruction loop.

Chief among the drawbacks of this method is the need for extensive memorization. The user simply has to remember the arbitrary mappings between a number of keys and their associated actions. This problem can be alleviated by sticking labels identifying the keys to the keyboard. This solution should be quite effective, provided each key is permanently associated with a single action. Unfortunately, this will not always be the case. Another disadvantage of function key entry is that there is a limited number of such keys available, typically two or three dozen. Since an editor of the sophistication and power of the sort we are concerned with here needs more keys than that, it is necessary to introduce modes of operation or some other method of increasing the number of operations callable by key. In a later section I will discuss this issue in some depth.

Menus

Finally, we come to the last alternative under consideration, selection from menus. This method certainly has the advantage of trendiness. For the growing body of users accustomed to working on microcomputers or workstations which support the mouse/menu paradigm, this is most familiar. There are a number of advantages which account for the popularity of menus. Chief among these is that their use involves selection from an explicit set of alternatives rather than memorization of those alternatives. And they provide mnemonic aids to help the user associate alternatives with actions. Though this method is not completely open ended, fairly large numbers of commands can be handled effectively by means of (for example) hierarchical menus. Finally, it can be a most natural method. I envision a hybrid, "two-handed" implementation where one hand uses function keys to select operands with the screen cursor (or syntactic cursor) while the other uses the mouse to select operations.

Unfortunately, realization of the full potential of this approach requires a high level of system support, starting with bit-mapped graphics and extending to windowing and menu generation facilities, and including provision for a supplementary input device (e.g., the mouse). There are also drawbacks which are intrinsic to the method. Pushed to extremes, the mouse/menu model can itself become awkward and slow. As well, it suffers from the same problems as the function key mode when textual input is involved. For reasons mentioned elsewhere, the current implementation does not run on microcomputer or workstation, but on a mainframe through conventional non-graphics terminals. For this reason, and because there are the advantages to the other methods mentioned above, I have chosen, for now, to forgo the use of menu selection.

Approach of the implementation

I have pointed out that for some commands, typed entry seems most appropriate, and in my implementation I have used this method for such commands. The commands involved are those which do not require selection of an operand from the screen, but do have one or more other arguments. Primarily they control the input and output of syntagms, and the storage and retrieval of nodes by the editor. For example

```
PRINT <filename> [<nodename>]
```

causes a pretty printed representation of the node stored under the name `nodename` to be written to the file called `filename`. The `nodename` argument is optional; if it is omitted the node currently being edited is pretty-printed.

This applies to a minority of the editor's functions, however. Due to its economy and naturalness, I have opted to utilize function key entry for most operations. The result then is an hybrid system with most operations performed by means of function keys, but with typed command lines used where it was deemed appropriate to do so. Among the disadvantages of function key entry mentioned above is the limited number of keys available. The manner in which the implementation addresses this problem is discussed immediately below.

Function key organization

If one is attempting to extend the capabilities of an SBE through the implementation of operations beyond the most basic ones, and if one desires to use function keys for the invocation of those operations, one must deal with the fact that there are not enough function keys to permit a one-to-one mapping of keys to operations. In interactive applications commands frequently cluster conceptually and temporally to yield natural modes of operation, each with its own operative key map. Where my editor is concerned, operational modes have not been easy to define. Although operations certainly may be grouped into *categories* ("travel," selection, basic editing, etc.), sequences of operations generally seem to cross category boundaries. Moreover, my experience, and that of others [4], points to the mode of operation as a source of errors. More exactly, errors tend to be made at temporal boundaries, the commonest error being to forget to change modes, proceeding instead to forge ahead, typing the key which would have been applicable had mode (and key map) been properly changed.

I have used the above mentioned categories as the basis for an approach to keying in commands which is hierarchical and at the same time "mode free." Under this approach, a subset of the available function keys is used (I prefer the keypad; other users might prefer a different locality), and each command is a two key sequence: The first keystroke selects a category; the second selects an operation within that category.

This scheme has the following strong points:

1. Convenience - A given command will always require the same sequence of keystrokes. The user's fingers are not required to remember whether or not the system is in a particular mode.
2. Keyboard localization - Having a small number of physical keys involved limits the area of the keyboard which must be searched/learned.
3. Mnemonic value - Memorization and recall tasks are facilitated by the "chunking" of data items. At each level of the hierarchy the user only has a small set of keys to chose from.
4. Economy - The use of multiple keystrokes would seem to be uneconomical. However, by restricting the set of physical keys used, escape sequences and double-keying can be eliminated, so that the increase in the average number of keystrokes is small.
5. Open-endedness - I have outlined a simple two-tiered arrangement.

[4] Hammond *et al* [HLC80] have found the shift/unshift mode to be implicated in one out of six errors made in an interactive environment.

Multi-leveled hierarchies are also possible.

An observation is in order here concerning my claim of a modeless system. In fact some sets of operations (e.g. textual editing) seem naturally to constitute a mode of operation, and they have been implemented as such. In subsequent chapters I consider these modes further, addressing the problem of transitions between modes.

Hybrid commands

As I have pointed out above, it seems most natural to invoke some operations by means of function key and others by means of typed command lines of this form:

```
COMMAND <arg1> <arg2> ....
```

In practice, there are commands which do not fit nicely into either of these categories. I will cite an example. I have implemented a command which might be called MODIFY_CURRENT_NODE, which permits the user to textually edit any selected node. The procedure is simple and, I believe, quite natural. The user selects a node with the cursor and enters text-editing mode by striking the appropriate function keys. Upon return from text-editing mode, the original node is replaced in its context by the textually modified version. A related command, let us call it CREATE_NEW_NODE, allows the user to textually enter completely new syntagms from the keyboard. The on-board text editor is used here as well, but the resulting node is pushed onto the node stack to be disposed of by the user as he/she sees fit. This does not require selection of a node, but it does require input of the applicable node class (i.e., the sort of structure it is: procedure declaration, statement, etc.). In this sense it is a candidate for implementation as a typed command. On the other hand CREATE_NEW_NODE is clearly related to MODIFY_CURRENT_NODE. In order that related operations may be grouped together into categories, as described above, I have implemented this command (and others as well) as a hybrid operation with textual entry *and* function key invocation.

Given the existence of such hybrid commands it is worthwhile to look more closely at their sequencing. Under what might be termed the "conversational model" of interactive computing, an operation might be performed as follows:

USER: Select node and strike appropriate keys.

EDITOR: Prompt user for necessary textual data.

USER: Enter text and strike function key (typically <RETURN>).

EDITOR: Perform the specified operation and display results or provide an error message if something is amiss.

This sequence has the advantage that the user need not remember all of the details of the command's requirements; the editor spells them out. However, I suspect that experienced users of the editor would be impatient with this dialogue. They would *know* what the command requires, and would prefer the following simpler sequence:

USER: Enter text; select node and strike appropriate function keys. (Note: The command expects to find the textual arguments in the command window.)

EDITOR: Perform operation and display results or produce an error message.

Note that if the error message produced looks like a prompt, which it will if no textual arguments are provided, then the process, as seen by an inexperienced user, is most similar to that of the conversational model, i.e.,

USER: Select node and strike function keys.

EDITOR: Produce error message (i.e., prompt the user).

USER: Enter text and strike function keys again.

EDITOR: Perform operation and display results.

Since this approach assists users when they are new to the editor, while allowing them to take shortcuts as they become more experienced, I have chosen it for the implementation.

Note that currently the implementation imposes the somewhat awkward sequence:

- enter text in command window,
- move cursor to main window and select node,
- strike operative key.

By providing a command which updates the current node (if necessary), then causes the cursor to jump to the command window, I could give the user the option of using the above sequence or the following one:

- select node and jump to command window,
- enter text and strike operative key.

Code Entry/Generation

The editor works by associating a node with the main, or editing, window, displaying all or part of the node's pretty-printed representation in that window, and providing the user a set of tools with which to edit the node. In keeping with the design goal of flexibility, this node may have been created in any of several ways:

1. Parsing a source file,
2. Restoration of a checkpoint file (see below),
3. Entry of text from the keyboard, or
4. Generation by the expansion of templates (at least potentially).

I shall now consider each of these methods for bringing syntagms into the system, together with some methods for outputting then.

Input from files

I have stated in previous chapters that it is most desirable that an editor be capable of processing pre-existing code. To deal with such code the editor supports a command `PARSE` which takes as arguments a file name and a node class. `PARSE` causes the contents of the named file, which must be a syntagm of the given class, to be parsed and the resulting node associated with the editing window. Alternatively, a third, optional, argument allows the resulting node to be stored under a name determined by the user (see below). Since syntagms which have been edited must be written back to files, a PRINT command is supported as well. PRINT unparses the main node, or, optionally, a named node, to a file. Since nodes may be named and stored, there must exist commands to provide access to the stored nodes. Supported are a command to FETCH a copy of a named node into the editing window, a STACK command which puts a copy of a named node onto the node stack, and a STORE command which causes a copy of the top node on the node stack to be stored under a user-specified name.

PARSE and PRINT read and write conventional Pascal code. The underlying MPS system also supports the checkpointing of a node, i.e., writing the node to a file using a representation which is similar to that used internally. Such a checkpoint file occupies no more disk space than the corresponding code and may be restored at a computational cost which is lower than that of parsing the original Pascal code. I have included in my implementation support for commands CHECKPOINT and RESTORE. These commands have not received much use to date, but I believe that in an integrated software development environment, the checkpoint file would be the natural form for storage for programs and would, as well, be the natural means of communication between tools.

Textual entry of code

If it is to be a usable tool, a syntactic editor must be, in reality, a hybrid editor, i.e., it must have some capacity for textual editing. One could support the generation of lexemes by some sort of template expansion, but it is clear that this would in practice be hopelessly awkward. At some level the user must resort to textual entry. The question is: At what level? Various implementors have had various answers: the identifier level, the statement level, the phrase level (whatever that is). My solution, motivated by the principle of flexibility, is to allow textual entry at *any* level. The choice is the user's. If the editor has been properly designed and is being used to best advantage, one would expect that such entry would generally be at a low level (e.g., identifier). Still the user has a choice, e.g., a simple expression or statement may be typed in, while a more complicated one may be generated by expansion (when that facility is available).

To handle such textual entry, the editor has a somewhat rudimentary visually oriented text editor built in. To call this editor the user selects the node to be modified and strikes the appropriate function key. This causes a representation of the selected node to appear in the small auxiliary window. There the user may edit it textually using operations supported by typical visual editors, i.e., textual scrolling and the insertion and deletion of characters and lines. Upon completion, the text is parsed and the resulting node replaces the original in its context. Any syntagm may be edited thus. It can also be seen that *only* syntagms may be created in this fashion. Code fragments which are syntactically incorrect or incomplete will not parse. Striking a different function key allows the user to create code from scratch using the little editor. In this case it is necessary for the user to supply the class of the node to be created, since there is no context to determine it. Under the current implementation it is required that the spellings of the class names correspond to those of the constants of the type NodeClass of the underlying MPS system (e.g., IfStatement, SubprogramDeclList, etc.). In a production implementation it would, of course, be desirable to have a more flexible set of conventions. Upon return from the editor, the resulting node is pushed onto the editor's stack.

It would seem to be more natural for this text editing to take place *in situ*, that is with the command causing a sort of dynamic window to open at the cursor, somehow displacing the older text as characters are entered, but only being parsed when the entry is complete. That this mode of entry is not supported is largely a function of the difficulty of its implementation. However, I have found some advantages to the current scheme in practice:

1. The transition to the auxiliary window is not so annoying as I had anticipated its being. The screen alterations necessitated by insertion in place might actually prove to be more distracting (and slower).

2. The original, unaltered text is always available for inspection, which is sometimes desirable.
3. It emphasizes the distinction between the structural and the textual modes of editing.
4. Because it is clear that this is a distinct mode of operation, errors arising from the transition between modes are less likely.

Both textual entry from the keyboard and input from files require code to be parsed. How does the editor deal with a failure to parse? Currently, this is handled by the underlying MPS routines, which respond to a problem in parsing by offering the user the choice of calling the MTS visual editor to permit correction of the problem, or of terminating the execution of the program (in this case, the editor itself). This is less than satisfactory. It would be preferable, at least in the case of keyboard entry, to use the on-board editor for corrections. In any case, it would certainly be preferable to be able to abort the parsing process without shutting down the editor. Desirable though they may be, these improvements are not critical to the key issues under study, and they have not been implemented.

Code generation

A standard component of many syntax-based editors is the program generation facility. Since this feature is well developed in other implementations, it has not been implemented here. Some of the manipulative routines which I have developed do involve integration of the code generation process into the editing process. I discuss these routines, and some issues which relate to code generation generally, in Chapter V. Template-based code production seems to fit in well with the approach of my research and is a candidate for future implementation. Here I describe the basic facility as I envision it.

To create a template the user strikes the appropriate function key, or perhaps types the command CREATE if command line entry is decided upon, causing the requested template to be associated with, and to appear in, the main window. Since, in accordance with the principle of maximum flexibility, the template may be of any type, the user must specify a node class as well. What I call a template would look very much like the productions in the grammar for the target language. For example, a procedure declaration template would look something like this:

```
PROCEDURE <<Name:Identifier>> (<<Parameters:ParameterList>>)
    <<Block:Block>> ;
```

Placeholders are expanded in their turn by placing the cursor on them and striking the EXPAND key. For example, selecting and expanding the block placeholder above would yield following:

```
PROCEDURE <<Name:Identifier>> (<<Parameters:ParameterList>>)
    LABEL <<Labels:IntegerList>> ;
    CONST <<ConstantDefs:ConstantDefList>>
    TYPE <<TypeDefs:TypeDefList>>
    VAR <<VariableDecls:VariableDeclList>>
        <<SubprogramDecls:SubprogramDeclList>>
    BEGIN
    <<Statements:StatementList>>
    END ;
```

This process continues until the user chooses to actuate a node using the text editor, or he/she reaches the lexeme level (identifier, number, etc.) and *must* use textual entry or replacement by an existing structure of the required type. In most cases a variety of expansions may be possible. For example, both placeholders in the template

```
WHILE <<Condition:Expression>> DO
    <<Body:Statement>>
```

represent alternation domains and can legitimately be replaced by any of a number of expansions. Just how the user is to select from the various alternatives is an interesting problem and is dealt with in Chapter V.

All of the above conforms to the standard model of program generation. I would like to provide the user with additional flexibility by enabling him/her to insert templates into existing code and to replace elements of existing code with templates, those templates then to be expanded as described above. Perhaps it would be appropriate to implement special versions of the basic INSERT and REPLACE commands which would do that. I am also interested in expanding the local manipulation facilities of the program with commands which involve code generation. My experiments along these lines are described in Chapter V.

There are some issues having to do with the implementation of a code generation facility. One is the question of how placeholders are to be represented, both internally and externally. The external representation is not a terribly interesting problem. Certainly the placeholders should be distinct in appearence from key-words and actualized code (though not outlandish), and any text they contain should help the user to understand just what they represent. The conventions I have used in the examples above seem to be not unreasonable. Questions still remain. For example, how should optional nodes be dealt with? Should they be displayed explicitly, as above? Should they be distinguished by square brackets as is done when defining the grammar? Or should they be omitted, as they are in actual code, only to become visible when specifically selected (see Inspection/Selection, below)?

Questions about the internal representation are more basic. A feature of the GRAMPS approach is that however the external textual representation is handled, the underlying syntagm is correct and, in the context independent sense, meaningful. Once templates and placeholders are introduced, this is no longer the case. A template, by its very nature, is an incomplete syntagm. MPS does have a NodeClass, Meta, which is, so far as the construction of nodes is concerned, compatible with all other NodeClasses, and, as such, is a prime candidate for use in the implementation of templates. However, Meta is not at present fully supported, and its use would, at the very least, subject my implementation to constraints that I do not wish to accept. Most of my experiments with code generation simply use identifiers for placeholders. For example:

```
IF DUMMYExpr THEN
    DUMMYStatement.
```

Here the prefix DUMMY- is used to designate a placeholder, and the gimmick of using a parameterless procedure call for the statement placeholder is used. This approach has permitted me to experiment with extensions to the manipulative facility but suffers from some drawbacks:

1. The placeholders are not so distinct, visually, as they should be.
2. Since an identifier representation is used, there is always the possibility of conflict with existing identifiers.
3. Certain productions cannot be represented, because they have components which cannot themselves be represented using identifiers, e.g.

   ```
   <LabelledStatement> ::=
       <Label:Integer> ":" <Statement:UnlabelledStatement>.
   ```

Another issue is that of the proper way to deal with code containing incompletely expanded templates. Obviously the editor must be able to deal with this case, but do we allow such code to be written back to files? The answer is "yes." Once again we do not wish to needlessly inhibit the user, who might have perfectly good reasons for writing out code in that state. Nodes to be written *should* be checked for unexpanded placeholders before output, and a warning issued if any are found, but the user should have the power to over-ride this warning.

Inspection/Selection

Current node, syntactic cursor and node stack

If operations are to be performed on programs (or program fragments), there must be some means of selecting structures as operands. Central to the editor is the concept of the current node, that is the node of interest in the parse tree being edited in the main window. Typically commands operate upon the current node. (Some commands require additional operands; they will be dealt with later in this section.) To keep track of the current node, the editor maintains what I call the syntactic cursor. The syntactic cursor may be set, tagging some node as the current node, in three ways:

1. The user designates a current node by placing the screen cursor on its textual representation.
2. The user moves the (syntactic) cursor through the parse tree by means of the editor's syntactic cursor movement commands.
3. In the course of their execution, editor commands set the syntactic cursor.

The screen cursor is provided by the MTS system (as part of the screen support abstraction). It appears as an underscore or as a reversal of a character's foreground/background. Control is by means of function keys which are reserved by the system for this purpose. Whenever a command is invoked the editor checks to see whether the user has moved the screen cursor. If he/she has, the most deeply nested node whose textual representation now contains the screen cursor becomes the current node.

Direct movement of the syntactic cursor is by means of function keys defined by the editor. The user may move the syntactic cursor up, making the current node's parent the new current node, or it may be moved down, by convention making the current node's leftmost child the new current node. The syntactic cursor may also be moved left or right across the current node's level in the parse tree, making its left or its right neighbor current node.

Commands themselves move the syntactic cursor in a variety of ways. An important issue in the design of the editor's commands has been that of cursor placement after execution.

As mentioned above, commands sometimes have more than one operand. For this, and other reasons (discussed in subsequent chapters), it is desirable to be able to tag one or more nodes in the parse tree in addition to the current node. It is also desirable to have some intermediary between the node in the editing window and such editor components as the on-board text editor and the node store. For this purpose, the editor maintains a stack of nodes. The user may manipulate the node stack by means of function key commands: SELECT pushes the current node onto the node stack; TOP displays, in pretty-printed form, the top node on the stack; POP discards the top node. An example of a command which uses the node stack is the primitive REPLACE, which causes the current node to be replaced in its context by the top node on the node stack.

Selection problems

There are two major problems that any selection method must overcome. The first is that of ambiguities in selection. A given string of characters representing a node does not necessarily represent a *unique* node. Consider, for example, the statement

```
CallSomeProc.
```

This is a parameterless procedure call. The applicable production in the MPS Pascal grammar is:

```
<ProcedureCall> ::=
        <ProcedureName:Identifier> ["("Arguments:ExpressionList")"].
```

How does one know whether the screen cursor, if positioned somewhere in the text, selects the identifier or the procedure call? According to the algorithm stated above, the most deeply nested node (the Identifier) is selected. How then does one select the

ProcedureCall? Early on I toyed with the idea of somehow using the position of the screen cursor within the text to determine what node should be selected. This was unsatisfactory on a number of grounds:

1. Even if it were possible to come up with some conventions which unambiguously identify every node (which seems unlikely), the scheme would likely be so complicated and counter-intuitive that no one would be able to use it.
2. Since this is a syntax-based editor, I wished to avoid having it carry around non-syntactic baggage.
3. Various of my colleagues seemed to find this notion of a bipartite cursor (node/position) sloppy and/or confusing.

Fortunately the solution is, for selection, straightforward. One simply places the screen cursor on the textual representation of the node one wishes to select, then, if necessary, moves it up to the required level. When it comes to distinguishing the current node in the display, the solution is not quite so tidy. Currently the editor places the screen cursor on the first character of the node's representation and announces the node's type in the message window. The user may also display the current node in the auxiliary window. In practice, this system works quite well. Highlighting the node somehow would probably be better yet, but due to difficulties of a practical nature, this feature has not been implemented. Note that even highlighting is not always sufficient to unambiguously designate a particular node. The example above is a case in point.

At this point it is probably worth asking why, since this is a syntax-based editor, the text-editor-like use of the screen cursor is retained at all. Indeed, some implementors seem to have dispensed with it entirely, relying solely upon cursors which move in syntactically significant increments. My experience has been that the screen cursor is simply too handy to give up. The syntactic cursor is necessary for "fine tuning" selection, as seen above, and its use is certainly appropriate in conjunction with various editing operations (as will be demonstrated in subsequent chapters), but very often simply moving the cursor across the screen is the quickest and most natural way to get from one node to another.

The other problem is that of optional nodes. The parameterless procedure call `CallSomeProc`, above, will again serve as an example. Internally the unactualized Arguments component is represented by a special empty node. Externally, there is no counterpart. How do we select that invisible node if we wish to insert some arguments? The solution, once more, involves the syntactic cursor. When the syntactic cursor is moved to an empty optional node in the parse tree, a special placeholder representation of the node is displayed (with the screen cursor on it). For example, if the syntactic cursor were on the procedure name, and we were to move it to the right, the following would be displayed:

```
        CallSomeProc <<Arguments:ExpressionList>>.
```

The empty, optional node is now the current node and as such may be edited textually or replaced by an expression list node.

Another problem, which I have dealt with only partially, is that of sublists. Programs are composed of a number of list structures: variable declaration lists, statement lists, etc.. These list's sublists are, from the point of view of a programmer editing code, most certainly significant entities. For example, while examining the body of a subroutine or program, the programmer might determine that some sequence of statements ought to be executed only if some condition holds, i.e., that the sequence should be replaced by an if statement whose consequent (i.e., the "if branch") is the original sequence (suitably embedded in a compound statement). The problem is that due to the way the EBNF grammar underlying MPS handles repetitions, sublists, including tail sublists, are not nodes. It follows that a sublist cannot be the current node, and therefore, cannot be *directly* selected as the operand of any of the various editing commands. I have implemented two sublist-handling commands. Both treat the current node and the top node of the node stack as the ends of a sublist. One deletes the sublist; the other stores an element-by-element copy of it on the node stack for further use. In association with the hierarchical keying scheme, described in an earlier section, I have considered a modifier key which could cause the operative command to take as operand not simply the current node but the sublist specified as described above.

A "find" facility

A program editor must provide the user with the means to inspect the textual representation of programs and program fragments. From the realm of text editing my editor has borrowed the notion of scrolling. Employing function keys, the user may slide the screen's windows up and down in the unparse of the nodes associated with them. I have found that some additional capabilities are necessary if the editor is to be at all useful. Moreover, I have found that the features that I have identified - inspection, selection, manipulation, etc. - are not altogether orthogonal. For example, if some manipulative operations are to be used to full advantage, they must be supported by higher level navigation routines.

The resulting commands seem to cluster into a *find* facility. Currently implemented are the DEFINING-OCCURRENCE command, which moves the syntactic cursor to the defining occurrence of a selected identifier, and the FIND_IDENTIFIER command. This latter command takes a character string argument and searches the node being edited for its occurrence *as an identifier*. Invoking FIND_IDENTIFIER actually has three effects:

1. The node being edited is traversed in search of the first occurrence of that identifier.
2. If such an occurrence is found, the enclosing code is displayed.
3. The editor goes into "find mode." In this mode the user can continue the traversal to the next occurrence, terminate the search fixing the present occurrence as the current node, or abort the search, restoring the editor to its previous state.

An optional argument allows the user to restrict the search to the current node. It might be useful as well to be able to restrict the search to *everything but* the current node. Similarly, an option that limits the search to a given scope (not necessarily the current node) might be useful. Another desirable command is one that, given the defining occurrence of an identifier, seeks out all occurrences of the identifier over which that definition holds sway.

CHAPTER V

THE MANIPULATIVE FACILITY

The heart of the implementation, and the focus of my research, is the manipulative facility, my attempt to capture in a manageable set of commands some of the pragmatics of program repair, alteration, and enhancement. Development of this facility has proceeded along two tracks, experimentation and organization. The chapter's first section reflects this, opening with a brief discussion of the editor's extensibility feature and its role in the development of a command set, and continuing with some musings on the nature of the editing activity. The remaining sections (the bulk of the chapter) describe in detail the various families of commands which comprise the editor's manipulative facility.

Toward a Comprehensive Command Set

The extensibility feature

To aid in the development of an extended set of manipulative commands, a convenient method for the testing of experimental operations was required. For the following reasons it was decided to develop a standardized methodology and interface for this purpose:

1. Such a methodology/interface would provide the user with the means to customize the editor (to a certain extent).
2. It would enable anyone working on program transformations, particularly those of a local sort, to utilize the editor as a testing facility.
3. It would facilitate the incorporation into the editor of transformations developed elsewhere.

What resulted was a method by which the user may redefine, at load time, any [1] of

[1] In theory any command may be redefined. In actuality, some editor commands have not been implemented in accordance with the conventions to be described, and hence are not redefinable.

the standard two-key command sequences.

Under this scheme, the editor maps the two-key sequences of function-key-commands onto parameterless procedures whose names have the form Cmd*dd*, where 1 <= *d* <= 9, and the sequence *dd* corresponds to the values associated internally with the keys struck. If one wishes to define (or redefine) a command sequence, one writes an MPS Pascal procedure which performs the desired operation and names it in accordance with this formula. Since the procedure can have no parameters, a package of subroutines serves as the interface with the editor. The routines provided include the function CurrentNode, which returns the editor's current node, RepositionCursor, which resets current node, adjusting windowing if necessary, and ShowUnparse which displays the result. A procedure WriteMessage may be used to print text, such as an error message, in the screen's message window. ShowSelectedNode displays a given node in the auxiliary window. Routines are also provided to perform the typical manipulations on the editor's node stack: empty, push, pop and top [2]. This set of routines is not complete in the sense of enabling the reimplementation of all commands supported by the editor, but it is sufficient for the performance of local transformations and typical editing operations.

The file containing the new command (or commands) is compiled, and, when the editor is to be invoked, the user concatenates the name of the resulting object module to the list of separately compiled files and libraries. When searching for a named routine, MTS uses the first one encoutered with that name, hence the new command definition overrides any in the editor's own library.

In the quest to extend the editor, I myself have made use of the facility for extension to implement and experiment with many commands. As well, an early version of the editor was used as a test-bed for local program transformations. Some of those transformations have contributed materially to the development of the

----

[2] Top is actually implemented as the more general NthPreviousNode (where NthPreviousNode(1) is equivalent to "top").

editor's manipulative facility.

Criteria for the inclusion of commands

A large, amorphous body of commands, however interesting their operations may be individually, does not constitute a manipulative facility. Consequently, these experimental commands have been culled and organized, and extensions to, and generalizations from, them have been proposed to produce a working set of manipulations for the editor. Criteria for inclusion in the set are:

1. The command should correspond to some higher level program editing notion. (See the remainder of this section for an examination of this subject.)
2. The command should be applicable to many different node types (*all* is too much to ask) in an intuitively meaningful way, *i.e.* the precise effect of a given command may vary greatly depending upon node type and context, but that effect will be easily predictable on the basis of the English language description of the editing notion/command.
3. The command should represent an improvement over the performance of the operation by means of more basic commands, either textual or syntax-based. This evaluation should not be based strictly upon a comparison of keystrokes. Other criteria, such as "naturalness" and error resistance are of equal (or greater) importance.

Development of the command set has also been influenced by an attempt to balance two desirable, but opposing, qualities: economy and redundancy. On the one hand, it is necessary to prevent an explosion in the number of editing commands to keep the key-mapping manageable and to ease the user's learning and memorization tasks. On the other hand, insistence that the functionalities of commands be absolutely disjoint is not desirable. Some operations occur frequently enough and are complicated enough that they deserve support, even though they can be performed by means of a series of other, simpler operations. Moveover, a measure of redundancy is desirable for its own sake[Bro77]. Different individuals prefer to do things in different ways.

Editing activities characterized

The idea of high level editing notions is central to what I have been trying to do. It is also distressingly vague. What I have been looking for are descriptions of the sorts of things one might wish to do to code without getting bogged down in concern for characters or lines - something (very) roughly analogous to the high level control structures of high level programming languages. To identify such operations it might be instructive to start by considering motivations for the editing of programs, and then to turn our attention to the means by which the programmer satisfies those motivations.

Let us consider then, the very basic question: What does a programmer do with an editor? Programmer aims seem to fall into four categories: to alter the functionality of the program or subprogram, to improve the readability (or more precisely, the understandability) of the code, to alter the degree of abstraction, and to improve the efficiency of its execution. During the initial development of a program, changes to functionality may include the correction of errors in syntax (not a problem in the SBE realm) and static semantics, where functionality is initially null, corrections of errors in logic, and correction of errors arising from faulty or misinterpreted specifications. Later in the life cycle changes and extensions may be necessary due to alterations in specifications over time. Beyond assuring adherence to conventions of indentation, capitalization and naming, changes aimed at the improvement of readability/understandability include reordering of statements and declarations, association of comments with particular structures, and even substitution of equivalent, but more lucid, logic. Although identification of abstractions is properly a part of the early stages of software development, it is sometimes desirable to introduce an element of abstraction after the fact, or to reveal detail (for the promotion of efficiency, for example). Efficiency promoting alterations are, of course, those aimed at facilitating the execution of the program in less time and/or space while preserving its functionality. Such alterations may occur throughout the life cycle.

When looked at from a structural (rather than a strictly textual) point of view, the means used to achieve these ends fall into the following categories:

1. Basic alterations - Structures may be inserted or deleted. Existing structures may be replaced by other, syntactically equivalent structures.
2. Alterations to nesting - A level of nesting may be interposed between structures, or levels may be removed.
3. Alterations to sequencing - Textual sequencing, that is the ordering of structures at a given level of nesting, may be changed.
4. Transformations - Transformations of a semantics-preserving nature may be effected by a combination of operations from the above categories, but some are sufficiently general, and yet sufficiently intuitive in nature, that they deserve consideration on their own.

These operational categories provide a framework for the organization of the editor's manipulative commands into a number of conceptual families. The editor must, like any editor, perform the basic alterations to code, hence there is a family of basic commands. Alterations to levels of nesting may be of either of two types: those which introduce an entirely new level (the EMBED family), and those which involve transfers between existing levels (the ENGULF/EJECT family). Alterations to sequencing may be of a rotational nature or involve the exchange of objects, leading me to dub the fourth command family ROTATE/SWAP. Finally, there is a family of mathematics-based, largely semantics-preserving transformations corresponding to the last category listed above.

Basic Commands

The first set of operations to be considered consists of those basic text-editing analogues, support for which would be expected from any editor: insertion, deletion and replacement. The motivation for the existence of these commands is self evident. I suspect that the ability to insert, delete and replace *structures* is one SBE feature about which few programmers would have reservations. The functionality and implementation of these commands are straightforward, but there are issues associated with each which I will describe immediately below. In addition to the most basic forms of these operations, there are extensions which would enhance the usability of the editor. These will be described as well.

Insertion acts upon the various sequences or lists (I will be using the terms interchangably) of structures of which code is composed: statement lists, expression lists, declaration lists, etc. The node stack's top node, which must be of a sort compatible with the target list, is the one to be inserted. If the node to be inserted is unattached, that very node becomes a component of the target list. If it is attached, then a copy is inserted, so that circular lists are not a problem. Note that the node to be inserted may itself be a list, so long as it is of a type compatible with the target list. In this case it is "spliced" into the target list.

An issue which needed to be resolved before the basic command could be implemented was that of the way in which the location of its insertion was to be designated. In a preceeding chapter I discussed my decision to implement commands in terms of the current node (and the node stack) without reference to additional spatial information provided by the visual cursor. Given this decision, it was necessary that the current node designate some list element and that the new node be inserted before or after that node. Unfortunately, if the "before" convention were used, it would be impossible to insert an element at the extreme tail end of a list (in a single operation). Conversely, the "after" convention would preclude insertion at the head of such a list. In the end both INSERT-BEFORE and INSERT-AFTER were implemented.

In the previous chapter I suggested that integration of the code generation process into the editing process might be desirable. The insertion facility would seem to provide an appropriate medium for this integration. In addition to the above-mentioned commands which insert the editor's top node, there could be versions of INSERT-BEFORE and INSERT-AFTER which would insert an expandable template (once such templates were available). It can be seen that although the context constrains the type of template which may be so inserted, in many cases more than one sort will be appropriate. In the section below on the ENCLOSE facility I will discuss in more detail the means by which the user may choose between such alternatives.

The DELETE command causes the current node to be deleted from its context. It acts upon optional nodes (e.g., the "else clause" of an if...else statement) and upon list elements. It also applies to constituents of "list-like" structures where something syntactically sound may be considered to remain after the deletion. For example, if DELETE is applied to the element b in

```
a OR b OR c
```

the result is

```
a OR c.
```

Sequences of elements may be deleted as well by placing the element at either end of the sequence on the stack, then selecting the other end with the cursor and invoking the DELETE-SUBLIST command.

Under the current implementation, if one wishes to move a node from one context to another, it is necessary to first stack the node then to delete it. This sequence of operations is common enough, and has a distinct enough identity, that an operation combining the two, SELECT-DESTRUCTIVE, is in order. To extract a sublist one currently must place the element at one end on the stack with SELECT, then move the cursor to the other end and invoke SELECT-SUBLIST, which causes the top node on the stack to be replaced by a copy of the delimited subsequence. It is then necessary to select *that* node, return the cursor to the other end of the sublist and key in DELETE-SUBLIST. This is a time consuming and error-prone series of operations. In a practical editing environment, support for SELECT-SUBLIST-DESTRUCTIVE would be not only justifiable but necessary.

The basic REPLACE command causes the current node to be replaced in its context by the top node, subject to compatibility checks. As in the case of the INSERT- operations, a copy is made if the top node is attached. If the current node is unattached, in which case it must be the node which has been associated with the editing window, the top node is simply substituted for the current node, i.e., it becomes the node under consideration, and its predecessor is lost, unless a reference to it has been saved on the node stack. REPLACE may be applied to an empty editing window, causing it to be initialized to the top node value.

Closely related to the basic REPLACE command is the MODIFY command (see Chapter IV, Code Entry/Generation) which summons the on-board text editor and upon return substitutes the textually edited version of the current node for the original. As well, an as yet unimplemented REPLACE-WITH-TEMPLATE command could serve, along with the INSERT-TEMPLATE operations proposed above, to integrate code generation into the editing process.

The EMBED Family

Frequently, the aim of a set of editing operations is to add a layer of complexity to the logic embodied in some existing code. The programmer is called upon to deal with editing problems which he/she might, for example, express as follows:

I now want this code to be executed only when condition `c` holds.

In addition to these conditions, the `Continue` flag must be tested as well.

> No wonder I'm getting an error! I should be passing the node's *class* to that routine, not the node itself.

Glass [Gla81] has found that circumstances of this sort, involving missing or incomplete logic, account for a large percentage of persistent software errors. It has been my experience that such patterns of alteration are important throughout the development of a given piece of software.

Here, then, is a candidate for implementation as a command, one that effects the interposition of a semantic layer by adding a layer of syntactic nesting. I have chosen to call the concept/command EMBED, since its action consists of the replacement of a node by the node itself suitably embedded in another node. There are three major issues associated with the implementation of EMBED:

1. Applicability - To what nodes, and under what circumstances, should the command be applicable?
2. Alternatives - In most circumstances more than one embedding may be legitimate. How is the user to select from among those alternatives?
3. Placeholders - In many cases the embedding node will have additional, unactualized components. How are these to be represented and dealt with?

In the preceding chapter, I discussed the issue of placeholder representation at some length, and it will not be dealt with further here. In the following two subsections I will look at the other issues, applicability and alternatives. Since my implementation of EMBED introduces a new mode of operation, I will then consider the problem of the transition between modes. Finally, the command/notion complementary to EMBED will be described.

The application of EMBED

I have implemented EMBED on a case-by-case basis to good effect. Any statement may serve as the target for the command and the full range of alterations to control flow may be effected by means of EMBED. It is also applicable to expressions (and to some expression lists), though the set of resulting expressions is not exhaustive. The boolean operations of negation, conjunction and disjunction are provided for, as are a range of arithmetic operations. There are certainly possibilities for this concept's extension. For example, it should be possible to generate expressions which contain set and arithmetic operators. Structured data objects are important in Pascal, and the EMBED concept is applicable to their extension and alteration.

While I was attempting to systematically enumerate the appropriate applications of EMBED, it occured to me that, due to the hierarchical nature of code, the EMBED concept was essentially universally applicable. Therefore, I propose that the command be applied in a mechanical fashion. Under this scheme, any node could serve as the target for EMBED and the set of possible node classes for the replacement node would be the intersection of the set of the target node's possible parent types with the set of its initial parent's possible child types. It should be noted that:

1. The target node may have no parent, i.e., it may be the unattached "main node" in the editing window.
2. The intersection may be empty, i.e., the grammar does not permit another layer of nesting to be "squeezed in."

The former case admits a sort of code generation by "bottom up" expansion, as the target node may be embedded in any of its possible parent types. The latter case should simply lead to an `Inappropriate operation` message.

Two exceptions or, more appropriately, extensions to the intersection rule proposed above are the cases of bracketing and "enlisting." When the existing version of EMBED is applied to an expression, brackets are supplied as necessary, e.g.,

```
        a OR b    -->    (a OR b) AND DUMMYFactor
```

Similarly, when it is applied to a member of an expression list (e.g. the arguments to a procedure call), the target node is automatically enclosed in an expression list to enable production of the function call alternative. In general, target nodes should be preprocessed to permit these operations. If there is a disadvantage to the mechanical implementation, it is that it is tightly bound to the strict, grammar-based approach and, hence, may occasionally exclude what some users might regard as intuitively meaningful operations. The inflexibility of this approach also arbitrarily restrains the placement of the cursor after the operation has been performed. However, always placing the cursor on the first placeholder node (if there is one) corresponds, with very few exceptions, to what I have done in the existing case-by-case treatment.

Presentation of alternatives

Clearly, under many circumstances there are a number of responses to EMBED which are syntactically correct and semantically sound. For example, any statement may be enclosed by another statement of any of eight types: if statement (as the consequent, "then" clause, or as the alternate, "else" clause), repeat loop, while loop, for-to-loop, etc. The problem of how the user is to choose between these various alternatives is an important one. I will look at three approaches to it:

1. Character string entry,
2. Menu selection, and
3. Exhaustive display of alternatives.

Note that although the following discussion is couched in terms of the EMBED command, it also has application to commands which insert or expand generative templates.

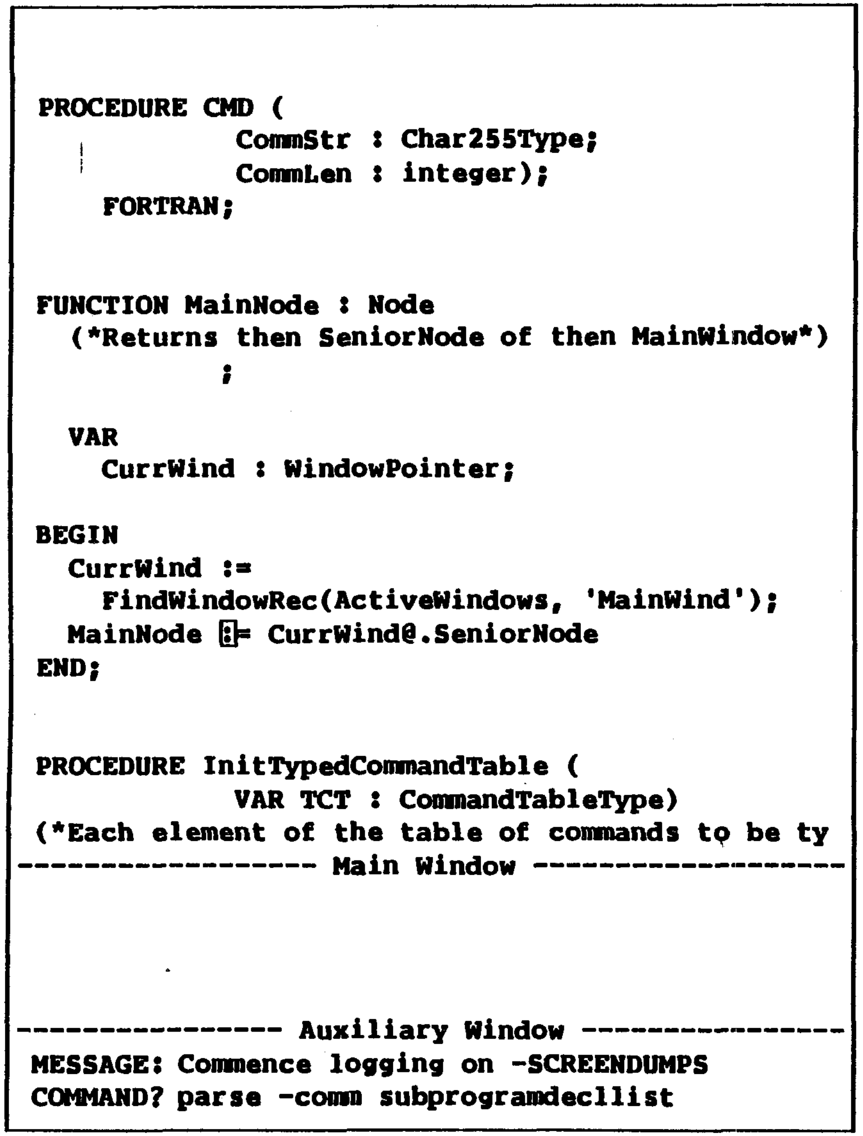
Character string entry, i.e., the typing of the name of the desired node type into the screen's command area prior to invocation of EMBED, has pros and cons similar to those previously cited for typed commands. To its credit this is a flexible, open-ended method. Its disadvantages include the awkward jump to the command window, the time-consuming and error-prone nature of character entry *per se*, and the necessity for memorizing the spellings of node class names and their association with familiar structures. Once again, abbreviation may alleviate the second problem at the expense of exacerbating the third. Provision for alternate spellings and the acceptance of unambiguous prefixes, may alleviate, but not eliminate, the third drawback.

Menus effectively address the issue of memorization of possibilities by substituting selection for specification, but the problem of identifying the correct choice remains. It may be difficult to find labels which are evocative and unambiguous under all circumstances. As well there seems to be some awkwardness inherent in the necessity of simultaneously selecting a node and a menu item. I have pointed out previously that a really effective implementation of the mouse/menu model requires a fair amount of system support.

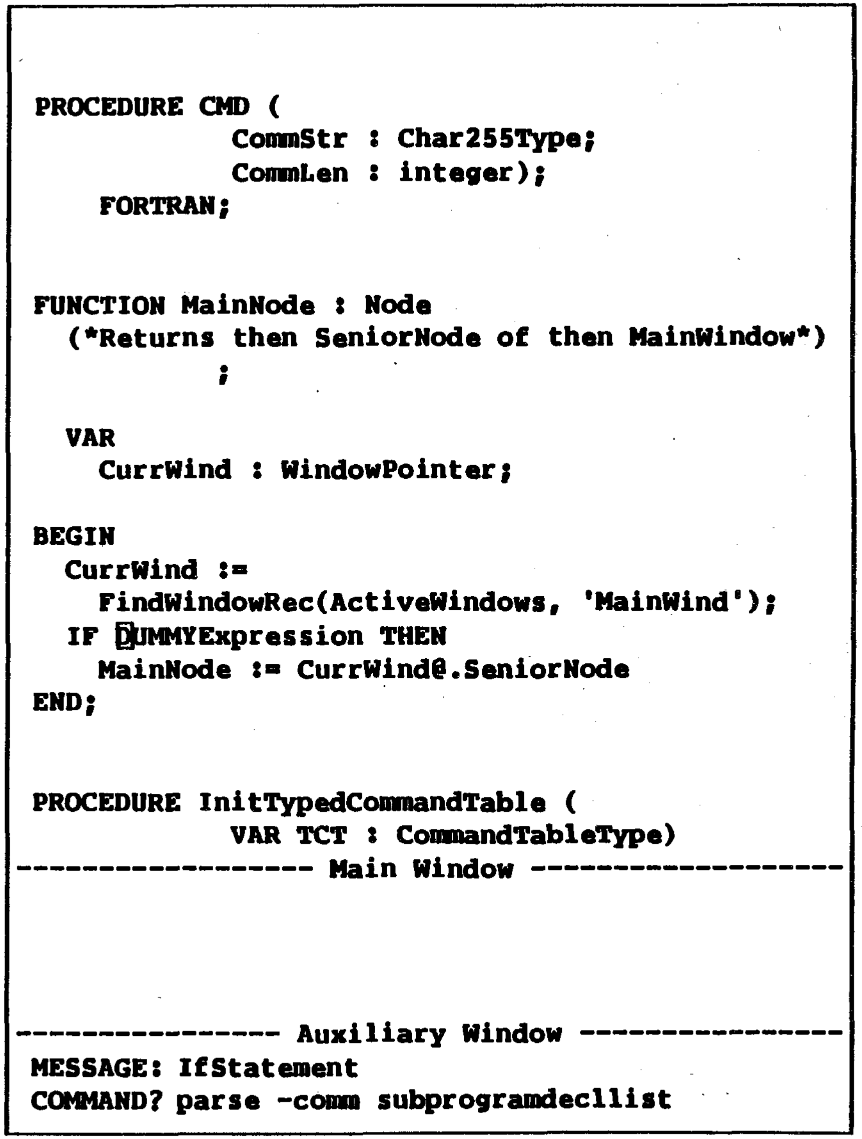
The final approach, exhaustive presentation, does not have a analogue among the command entry methods discussed in Chapter IV. What I mean by "exhaustive presentation" is that the user actually gets to examine each alternative in context. When he/she strikes the appropriate sequence of keys, the editor displays, in context, a possible embedding of the current node and simultaneously goes into "embed mode." The user may then either select that possibility or may, by tapping a function key, leaf back and forth through the various alternatives till the desired one is found and selected (causing exit from that mode). Figures 5.1a, 5.1b, 5.1c and 5.1d [3] show, in sequence, four of the alternatives resulting from an application of EMBED to a statement. Figures 5.2a, 5.2b, 5.2c, 5.2d and 5.2e show some possible embeddings of

----

[3] The figures in this chapter, and the succeeding one, show before-and-after screen dumps. In each, the panel on the left shows the screen before some editing operation is perfomed. The panel on the right shows the screen immediately after the operation. When the operation is one of the SBE-specific commands, the screen dump routines display the name of the command beneath the lower right-hand corner of the "before" screen.



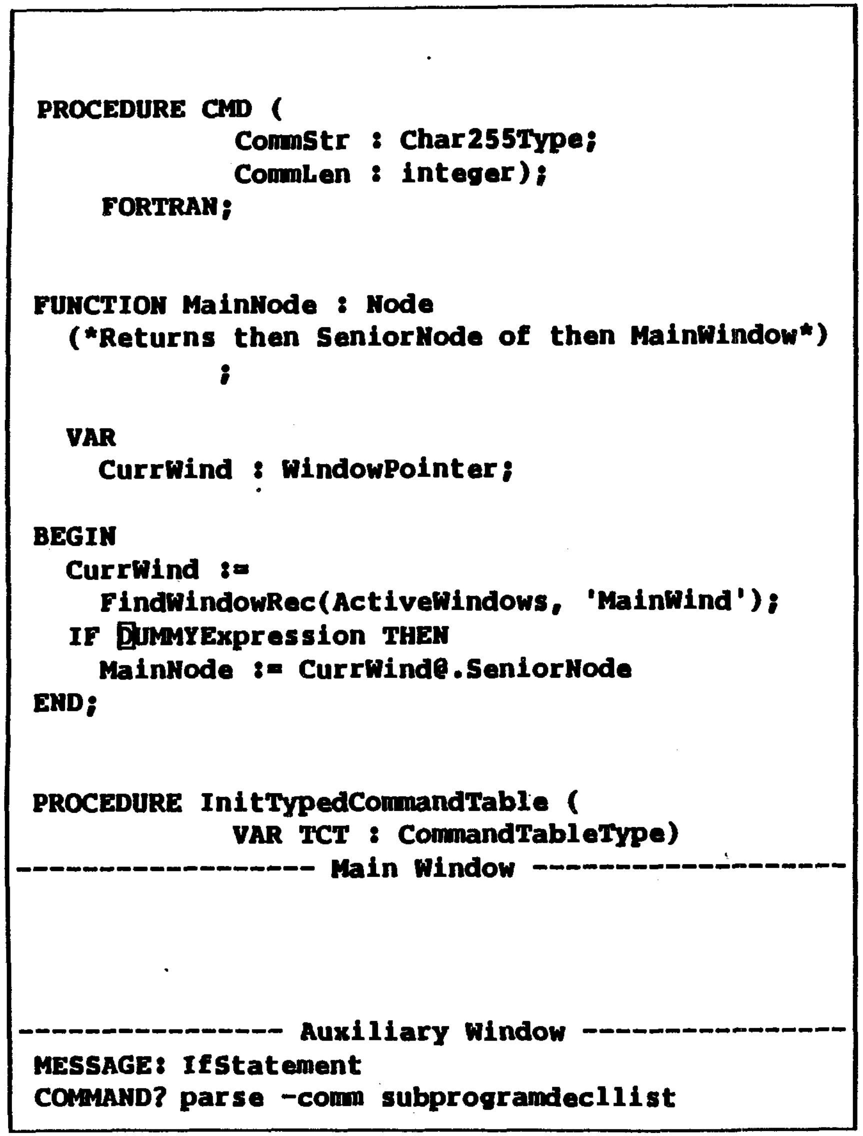
```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
(*Each element of the table of commands to be ty
------------------ Main Window ------------------




--------------- Auxiliary Window ---------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? parse -comm subprogramdecllist
```



```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  IF DUMMYExpression THEN
    MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
------------------ Main Window ------------------




--------------- Auxiliary Window ---------------
MESSAGE: IfStatement
COMMAND? parse -comm subprogramdecllist
```

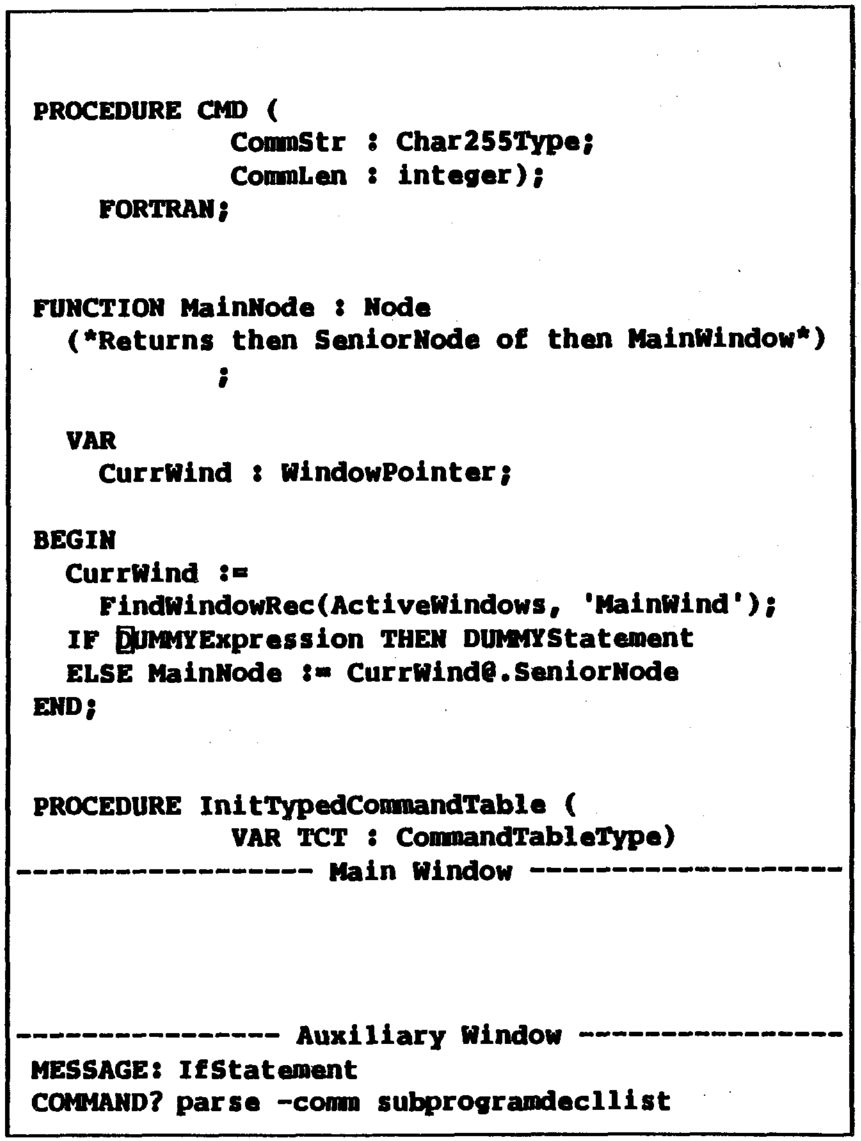
EMBED

Figure 5.1a: EMBED applied to a statement - first alternative.



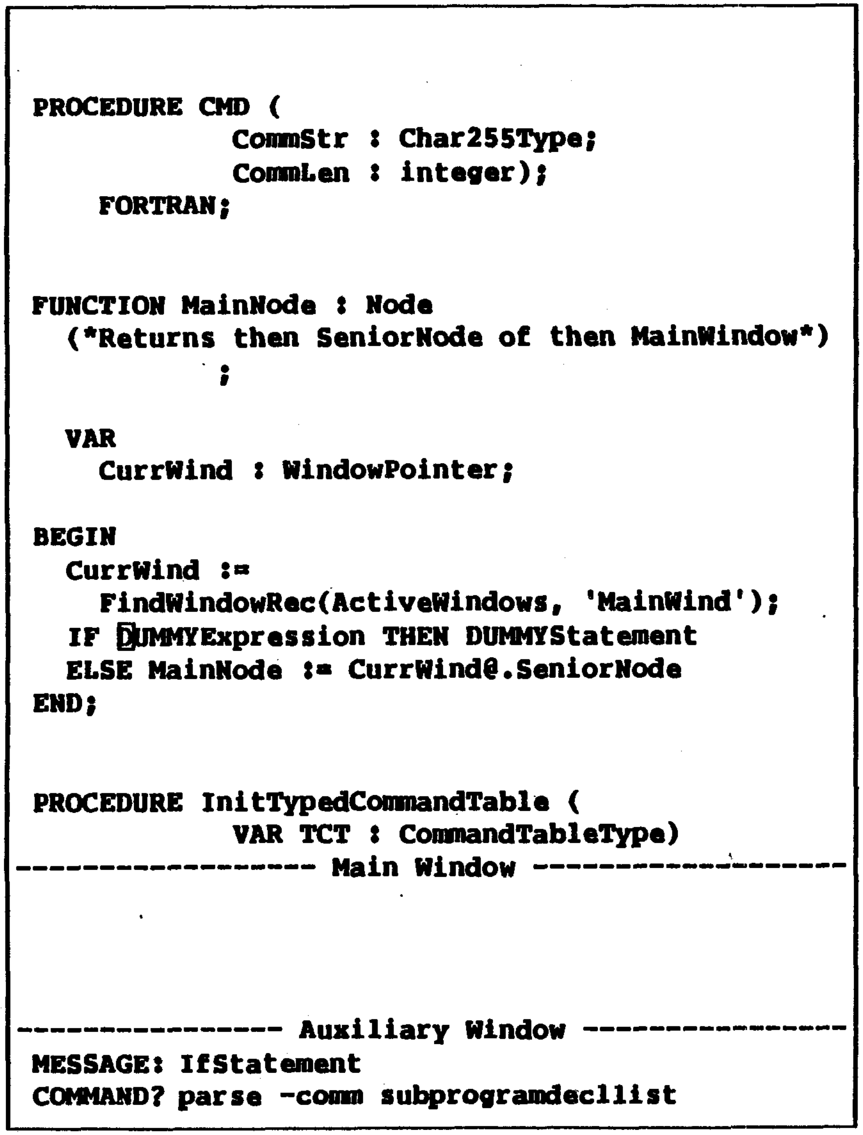
```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  IF DUMMYExpression THEN
    MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
------------------ Main Window -------------------




---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? parse -comm subprogramdecllist
```

NEXT

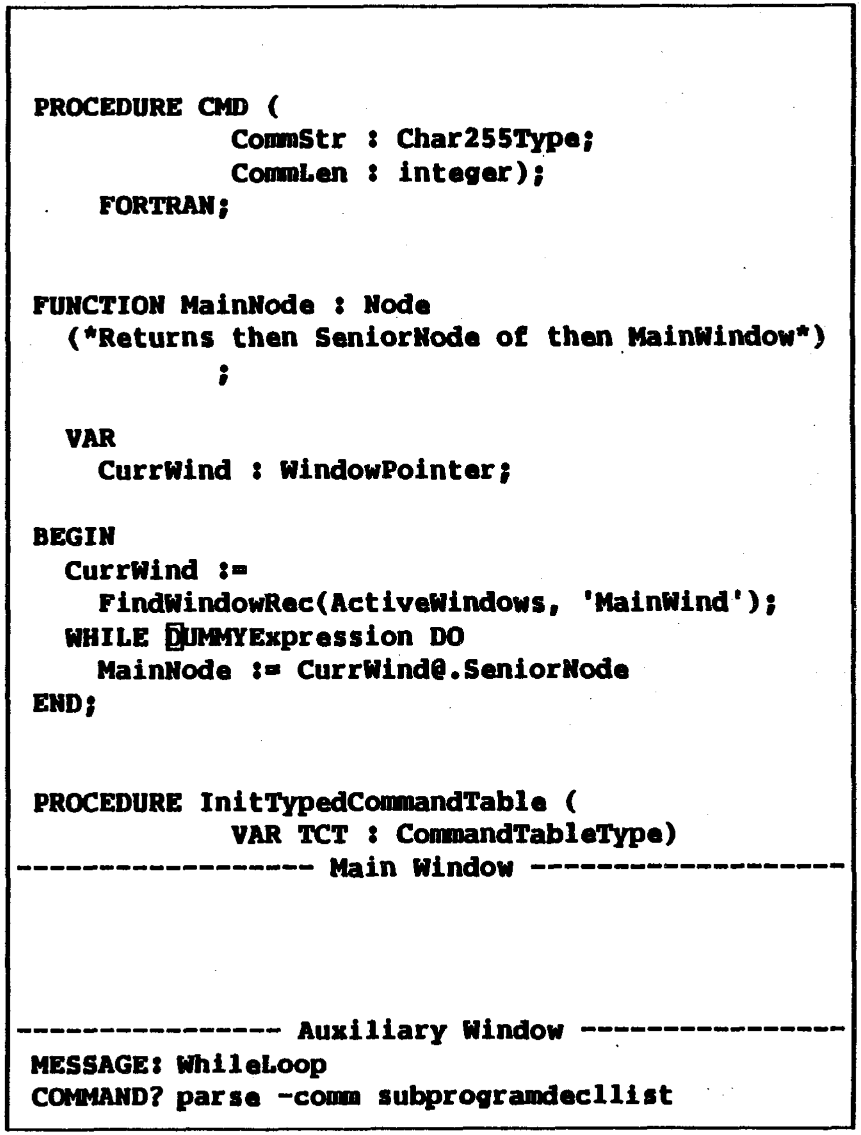


```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  IF DUMMYExpression THEN DUMMYStatement
  ELSE MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
------------------ Main Window -------------------




---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? parse -comm subprogramdecllist
```

Figure 5.1b: EMBED applied to a statement - second alternative.



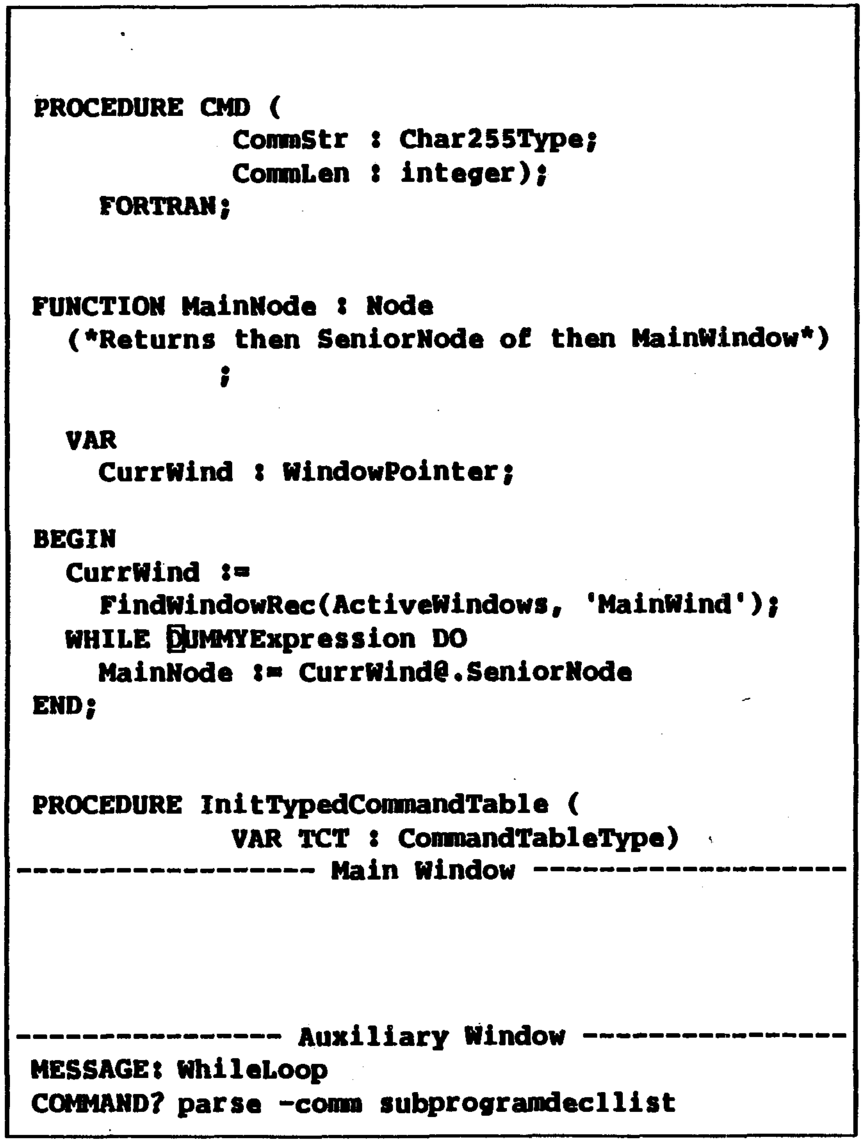
```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  IF DUMMYExpression THEN DUMMYStatement
  ELSE MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? parse -comm subprogramdecllist
```



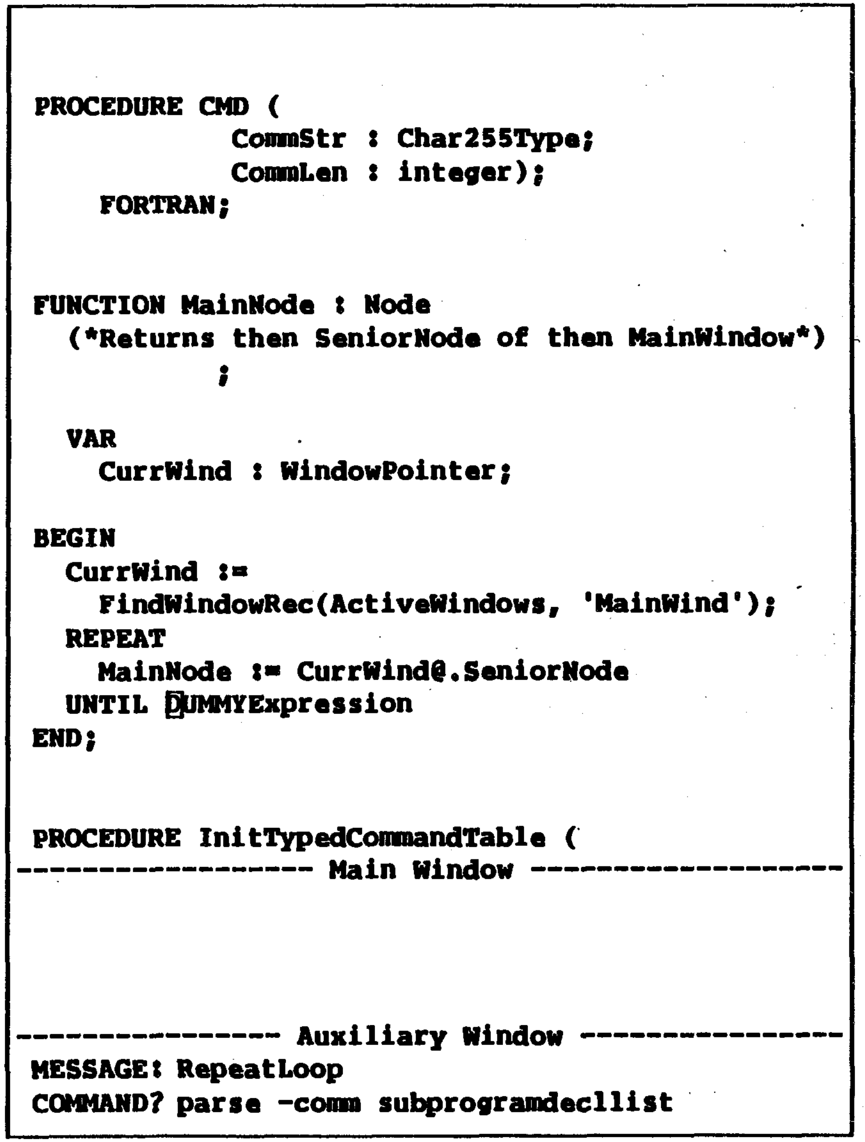
```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  WHILE DUMMYExpression DO
    MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: WhileLoop
COMMAND? parse -comm subprogramdecllist
```

NEXT

Figure 5.1c: EMBED applied to a statement - third alternative.



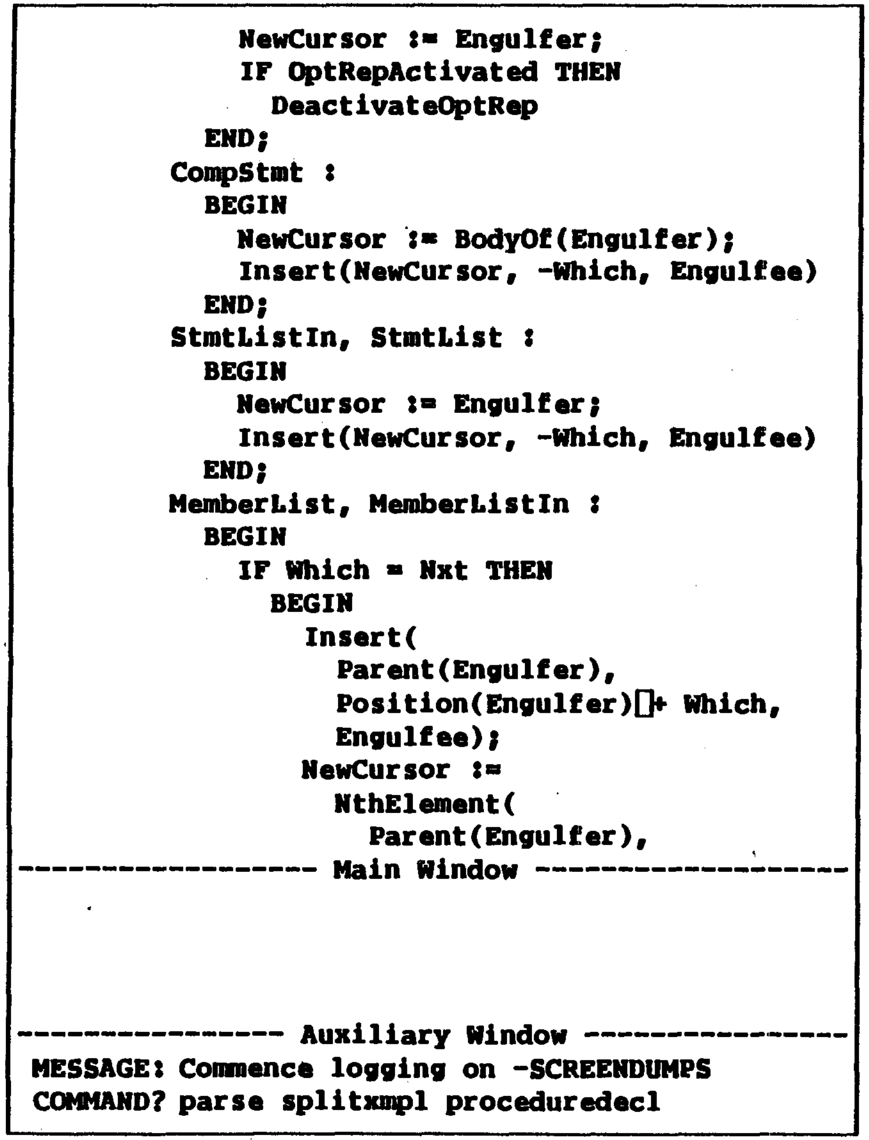
```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  WHILE DUMMYExpression DO
    MainNode := CurrWind@.SeniorNode
END;


PROCEDURE InitTypedCommandTable (
            VAR TCT : CommandTableType)
------------------ Main Window -------------------




---------------- Auxiliary Window ---------------
MESSAGE: WhileLoop
COMMAND? parse -comm subprogramdecllist
```



```
PROCEDURE CMD (
            CommStr : Char255Type;
            CommLen : integer);
    FORTRAN;


FUNCTION MainNode : Node
  (*Returns then SeniorNode of then MainWindow*)
          ;

  VAR
    CurrWind : WindowPointer;

BEGIN
  CurrWind :=
    FindWindowRec(ActiveWindows, 'MainWind');
  REPEAT
    MainNode := CurrWind@.SeniorNode
  UNTIL DUMMYExpression
END;


PROCEDURE InitTypedCommandTable (
------------------ Main Window -------------------




---------------- Auxiliary Window ---------------
MESSAGE: RepeatLoop
COMMAND? parse -comm subprogramdecllist
```

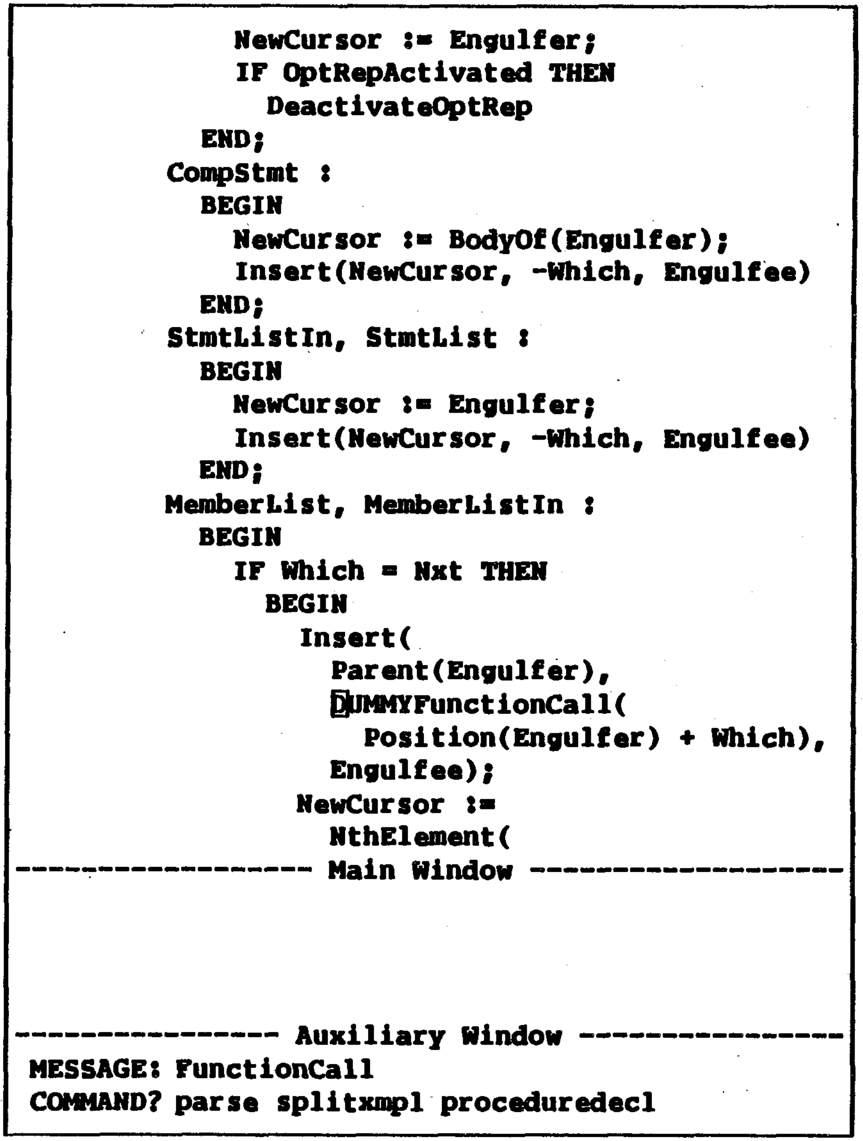
NEXT

Figure 5.1d: EMBED applied to a statement - fourth alternative.



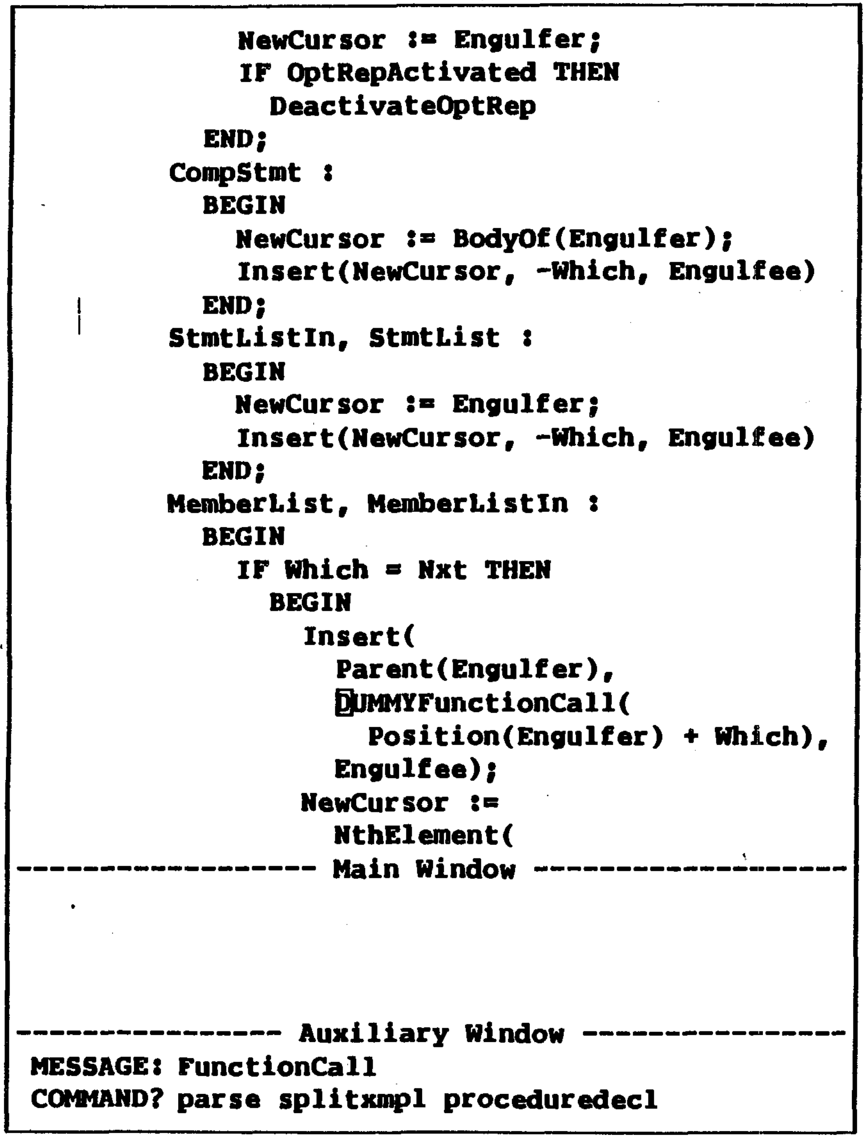
```
          NewCursor := Engulfer;
          IF OptRepActivated THEN
            DeactivateOptRep
        END;
      CompStmt :
        BEGIN
          NewCursor := BodyOf(Engulfer);
          Insert(NewCursor, -Which, Engulfee)
        END;
      StmtListIn, StmtList :
        BEGIN
          NewCursor := Engulfer;
          Insert(NewCursor, -Which, Engulfee)
        END;
      MemberList, MemberListIn :
        BEGIN
          IF Which = Nxt THEN
            BEGIN
              Insert(
                Parent(Engulfer),
                Position(Engulfer)□+ Which,
                Engulfee);
              NewCursor :=
                NthElement(
                  Parent(Engulfer),
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? parse splitxmpl proceduredecl
```

EMBED



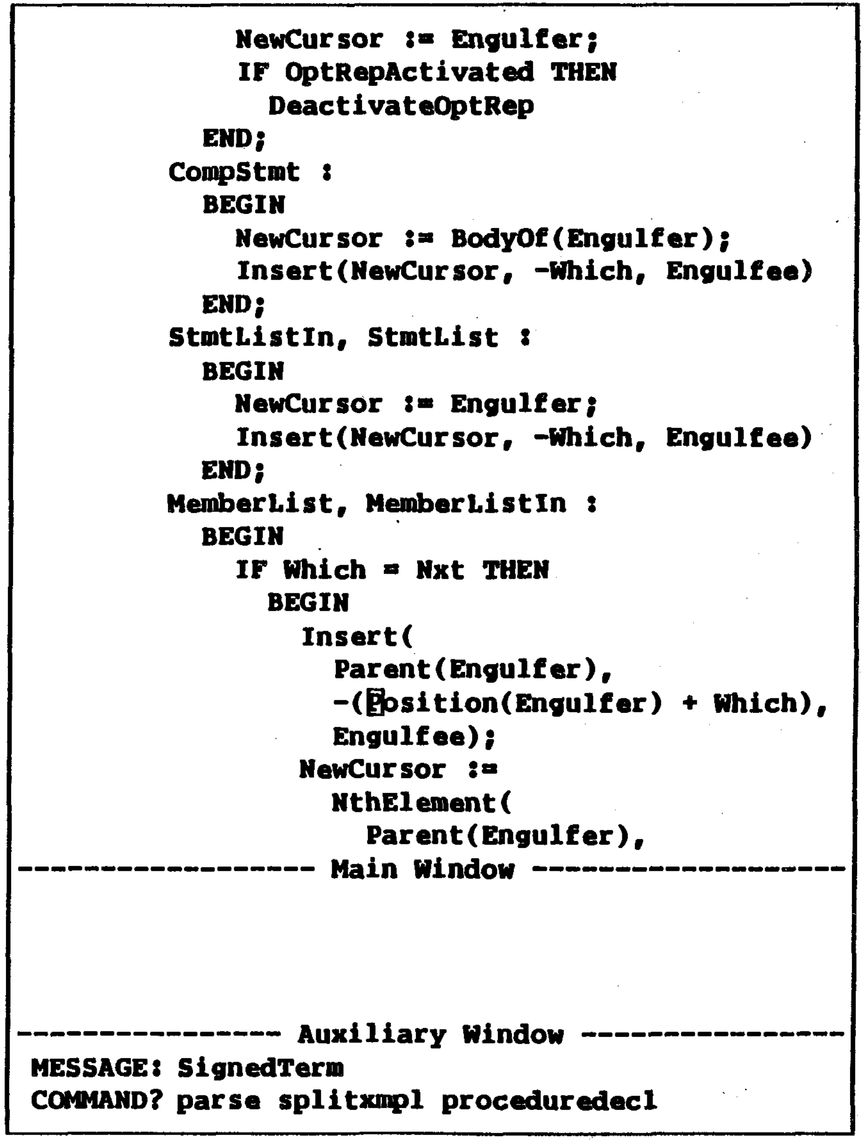
```
          NewCursor := Engulfer;
          IF OptRepActivated THEN
            DeactivateOptRep
        END;
      CompStmt :
        BEGIN
          NewCursor := BodyOf(Engulfer);
          Insert(NewCursor, -Which, Engulfee)
        END;
      StmtListIn, StmtList :
        BEGIN
          NewCursor := Engulfer;
          Insert(NewCursor, -Which, Engulfee)
        END;
      MemberList, MemberListIn :
        BEGIN
          IF Which = Nxt THEN
            BEGIN
              Insert(
                Parent(Engulfer),
                DUMMYFunctionCall(
                  Position(Engulfer) + Which),
                Engulfee);
              NewCursor :=
                NthElement(
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
MESSAGE: FunctionCall
COMMAND? parse splitxmpl proceduredecl
```

Figure 5.2a: EMBED applied to an expression - first alternative.



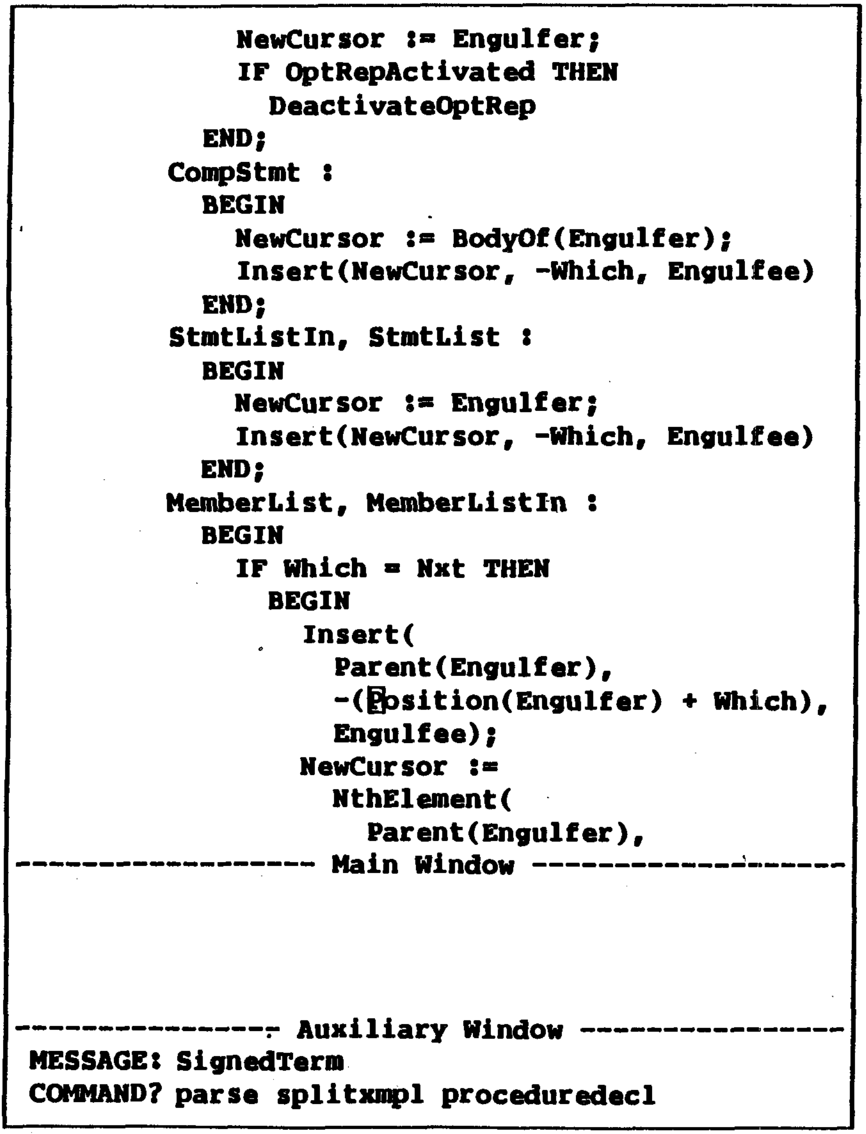
```
          NewCursor := Engulfer;
          IF OptRepActivated THEN
            DeactivateOptRep
        END;
      CompStmt :
        BEGIN
          NewCursor := BodyOf(Engulfer);
          Insert(NewCursor, -Which, Engulfee)
        END;
      StmtListIn, StmtList :
        BEGIN
          NewCursor := Engulfer;
          Insert(NewCursor, -Which, Engulfee)
        END;
      MemberList, MemberListIn :
        BEGIN
          IF Which = Nxt THEN
            BEGIN
              Insert(
                Parent(Engulfer),
                DUMMYFunctionCall(
                  Position(Engulfer) + Which),
                Engulfee);
              NewCursor :=
                NthElement(
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
MESSAGE: FunctionCall
COMMAND? parse splitxmpl proceduredecl
```

PREVIOUS



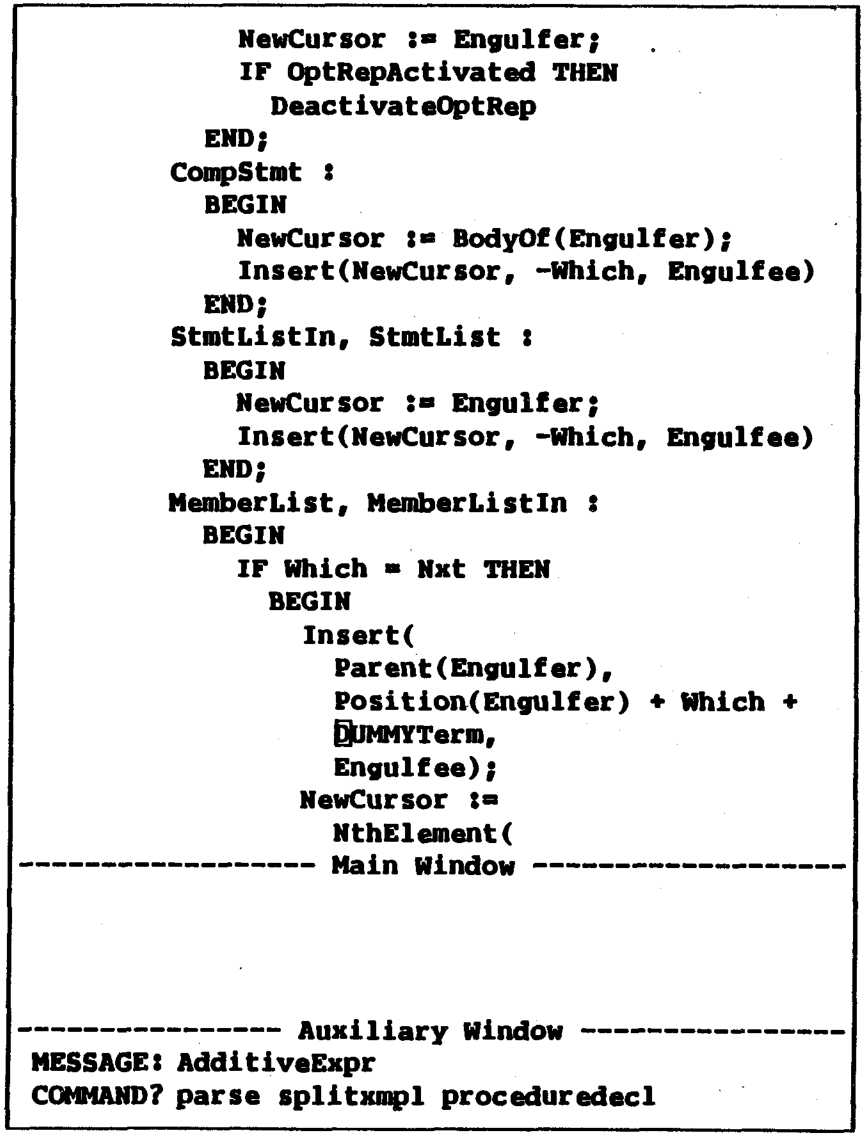
```
          NewCursor := Engulfer;
          IF OptRepActivated THEN
            DeactivateOptRep
        END;
      CompStmt :
        BEGIN
          NewCursor := BodyOf(Engulfer);
          Insert(NewCursor, -Which, Engulfee)
        END;
      StmtListIn, StmtList :
        BEGIN
          NewCursor := Engulfer;
          Insert(NewCursor, -Which, Engulfee)
        END;
      MemberList, MemberListIn :
        BEGIN
          IF Which = Nxt THEN
            BEGIN
              Insert(
                Parent(Engulfer),
                -(Position(Engulfer) + Which),
                Engulfee);
              NewCursor :=
                NthElement(
                  Parent(Engulfer),
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
MESSAGE: SignedTerm
COMMAND? parse splitxmpl proceduredecl
```

Figure 5.2b: EMBED applied to an expression - second (arithmetic) alternative.



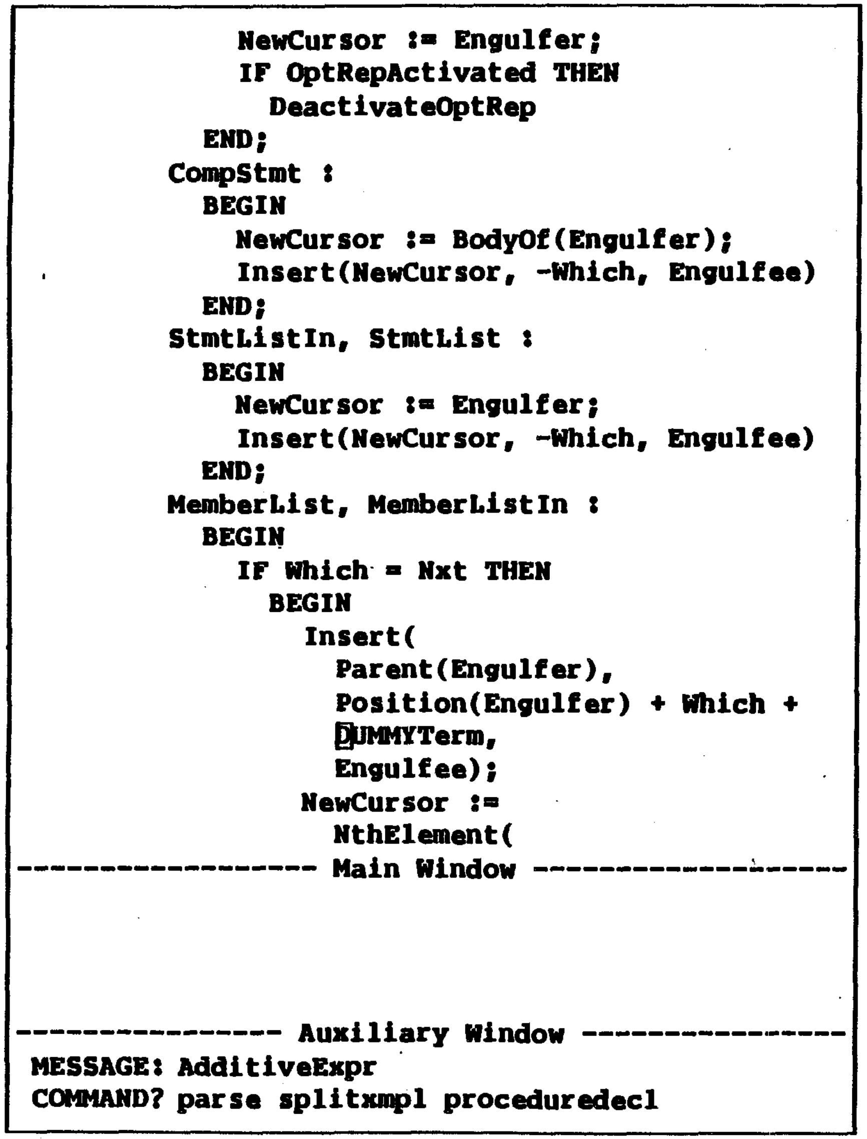
```
             NewCursor := Engulfer;
             IF OptRepActivated THEN
               DeactivateOptRep
           END;
         CompStmt :
           BEGIN
             NewCursor := BodyOf(Engulfer);
             Insert(NewCursor, -Which, Engulfee)
           END;
         StmtListIn, StmtList :
           BEGIN
             NewCursor := Engulfer;
             Insert(NewCursor, -Which, Engulfee)
           END;
         MemberList, MemberListIn :
           BEGIN
             IF Which = Nxt THEN
               BEGIN
                 Insert(
                   Parent(Engulfer),
                   -(Position(Engulfer) + Which),
                   Engulfee);
                 NewCursor :=
                   NthElement(
                     Parent(Engulfer),
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
 MESSAGE: SignedTerm
 COMMAND? parse splitxmpl proceduredecl
```

PREVIOUS

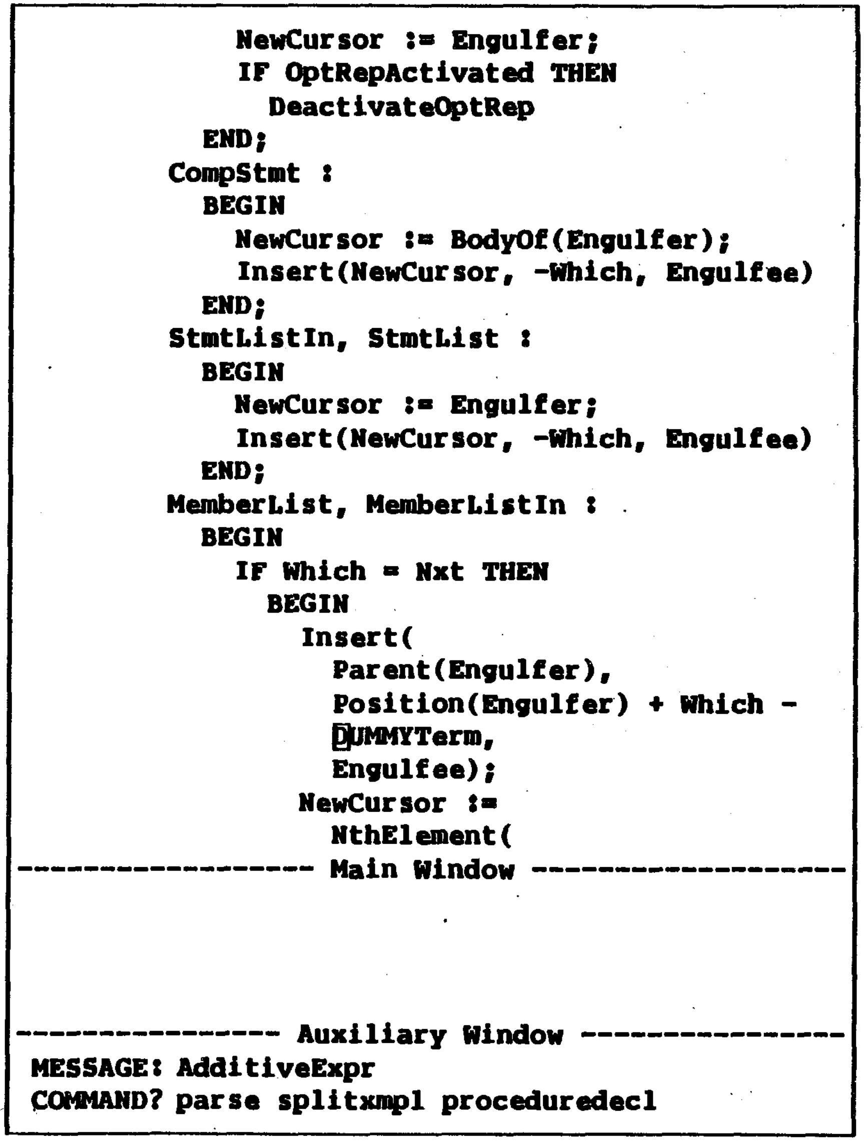


```
             NewCursor := Engulfer;
             IF OptRepActivated THEN
               DeactivateOptRep
           END;
         CompStmt :
           BEGIN
             NewCursor := BodyOf(Engulfer);
             Insert(NewCursor, -Which, Engulfee)
           END;
         StmtListIn, StmtList :
           BEGIN
             NewCursor := Engulfer;
             Insert(NewCursor, -Which, Engulfee)
           END;
         MemberList, MemberListIn :
           BEGIN
             IF Which = Nxt THEN
               BEGIN
                 Insert(
                   Parent(Engulfer),
                   Position(Engulfer) + Which +
                   DUMMYTerm,
                   Engulfee);
                 NewCursor :=
                   NthElement(
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
 MESSAGE: AdditiveExpr
 COMMAND? parse splitxmpl proceduredecl
```

Figure 5.2c: EMBED applied to an expression - third (arithmetic) alternative.



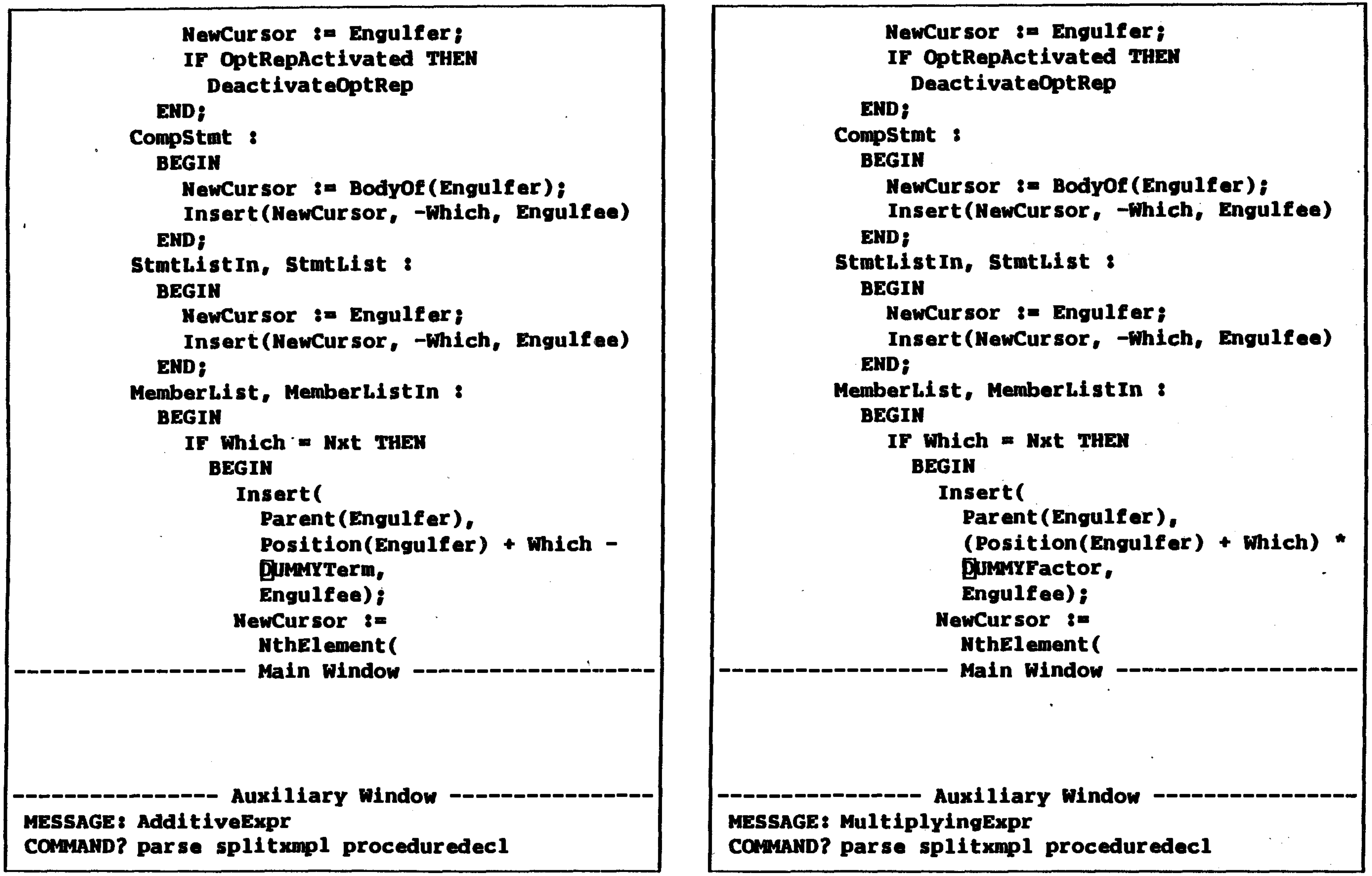
```
            NewCursor := Engulfer;
            IF OptRepActivated THEN
              DeactivateOptRep
          END;
        CompStmt :
          BEGIN
            NewCursor := BodyOf(Engulfer);
            Insert(NewCursor, -Which, Engulfee)
          END;
        StmtListIn, StmtList :
          BEGIN
            NewCursor := Engulfer;
            Insert(NewCursor, -Which, Engulfee)
          END;
        MemberList, MemberListIn :
          BEGIN
            IF Which = Nxt THEN
              BEGIN
                Insert(
                  Parent(Engulfer),
                  Position(Engulfer) + Which +
                  DUMMYTerm,
                  Engulfee);
                NewCursor :=
                  NthElement(
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: AdditiveExpr
COMMAND? parse splitxmpl proceduredecl
```



```
            NewCursor := Engulfer;
            IF OptRepActivated THEN
              DeactivateOptRep
          END;
        CompStmt :
          BEGIN
            NewCursor := BodyOf(Engulfer);
            Insert(NewCursor, -Which, Engulfee)
          END;
        StmtListIn, StmtList :
          BEGIN
            NewCursor := Engulfer;
            Insert(NewCursor, -Which, Engulfee)
          END;
        MemberList, MemberListIn :
          BEGIN
            IF Which = Nxt THEN
              BEGIN
                Insert(
                  Parent(Engulfer),
                  Position(Engulfer) + Which -
                  DUMMYTerm,
                  Engulfee);
                NewCursor :=
                  NthElement(
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: AdditiveExpr
COMMAND? parse splitxmpl proceduredecl
```

PREVIOUS

Figure 5.2d: EMBED applied to an expression - fourth (arithmetic) alternative.



```
      NewCursor := Engulfer;
      IF OptRepActivated THEN
        DeactivateOptRep
     END;
    CompStmt :
     BEGIN
      NewCursor := BodyOf(Engulfer);
      Insert(NewCursor, -Which, Engulfee)
     END;
    StmtListIn, StmtList :
     BEGIN
      NewCursor := Engulfer;
      Insert(NewCursor, -Which, Engulfee)
     END;
    MemberList, MemberListIn :
     BEGIN
      IF Which = Nxt THEN
        BEGIN
         Insert(
          Parent(Engulfer),
          Position(Engulfer) + Which -
          DUMMYTerm,
          Engulfee);
         NewCursor :=
          NthElement(
-------------------- Main Window --------------------



----------------- Auxiliary Window -----------------
MESSAGE: AdditiveExpr
COMMAND? parse splitxmpl proceduredecl
```

```
      NewCursor := Engulfer;
      IF OptRepActivated THEN
        DeactivateOptRep
     END;
    CompStmt :
     BEGIN
      NewCursor := BodyOf(Engulfer);
      Insert(NewCursor, -Which, Engulfee)
     END;
    StmtListIn, StmtList :
     BEGIN
      NewCursor := Engulfer;
      Insert(NewCursor, -Which, Engulfee)
     END;
    MemberList, MemberListIn :
     BEGIN
      IF Which = Nxt THEN
        BEGIN
         Insert(
          Parent(Engulfer),
          (Position(Engulfer) + Which) *
          DUMMYFactor,
          Engulfee);
         NewCursor :=
          NthElement(
-------------------- Main Window --------------------



----------------- Auxiliary Window -----------------
MESSAGE: MultiplyingExpr
COMMAND? parse splitxmpl proceduredecl
```

PREVIOUS

Figure 5.2e: EMBED applied to an expression - fifth (arithmetic) alternative.

an expression. Note that under the current implementation the alternatives are, conceptually, arranged in a circular list. The designations NEXT and PREVIOUS that accompany the figures describe the direction in which the user is searching that circular list.

An advantage of this method is its "what you see is what you get" nature. The user is neither required to memorize character strings nor to choose from (possibly obscure) descriptions. This, moreover, is a method which is suited to both neophyte and experienced user. The former may deliberately examine all alternations until the desired one is found. The latter, knowing the position of the desired node type in the list of alternatives, rapidly taps the correct number of keystrokes. An apparent disadvantage is its slowness and awkwardness. In practice this has not been a problem. I have in fact selected this method for the implementation of EMBED, and it has proven to be reasonably fast and feels most natural. It would be desirable if, at the level of person-machine interaction, the efficiency of this alternative presentation/selection process could be assured. To that end I propose the following implementation.

Although there may be a number of alternatives appropriate to a given operation, there are, generally, a couple of "most popular" choices. If these can be presented first, then this selection method can be rapid indeed. One way of accomplishing this is by means of extensive analysis of code and hand adjustment of the program. Another possibility which I have considered is that of a self-adjusting implementation. Between executions of the editor a table of selections would be maintained (in files) for each operand pair. Each entry in the table would be a list of the applicable alternatives. Associated with each alternative would be a usage field. When EMBED was invoked the alternatives appropriate to the context would be presented in the order in which they appear on the list. Whenever one was selected, its usage field would be incremented and it would be moved ahead of all alternatives in the list with a usage value less than its own. In time popular selections would percolate to the top of the list and unpopular ones to the bottom. The unpredictability inherent in this approach would probably be somewhat disconcerting to the user. To counteract this it might be best to have this feature under user control, so that once the system had stabilized, the user could turn off the self-adjustment.

The method I have described is that of serial presentation. Another possibility is to combine, in a sense, exhaustive presentation and menu selection by displaying all of the alternatives *simultaneously*. This list of alternative embeddings would appear either in the main window or in its own specially generated window. Since the list in many cases would not fit in a single window, the user would either scroll through it or rotate it in a manner somewhat analogous to the use of a "cardex." The screen cursor would be used to select the desired version. I have rejected this approach for the following reasons:

1. It is disruptive to screen integrity i.e., it causes extensive alterations to the screen being edited, and, consequently, may adversely affect the continuity of the editing process.
2. The user does not get to see the alternatives actually embedded in context, at least not so clearly as in the serial method.
3. It is unnecessary, since it was originally proposed as a means of graphically differentiating the "select mode" from the regular syntactic editing mode, an issue with which there are other ways of dealing (see next subsection).

The transition between modes

In the previous chapter I introduced the problem of transitions between modes of operation, in particular the problems encountered when returning to the basic structural editing mode from a specialized one, and I put forward the desire to avoid this problem as a motivation for the development of the editor's scheme for the keying in of commands. Subsequently, I have described several such modes: text editing mode, ENCLOSE mode, EXPAND mode, and a family of search operations. I will now address the issue.

Where the text editor is concerned, no difficulties have been encountered in practice. I believe this to be a consequence of the processes' being sufficiently distinct from one another. Though the structural editing and textual editing processes share function keys, they do not share an entry format. Moreover, textual editing and entry take place in a separate window. Finally, and perhaps most important, text based operations and operations in the syntactic or structural realm are conceptually distinct. As a consequence, it seems most natural to return to that latter realm before invoking the characteristic SBE operations.

This does not appear to be the case for the other modes of operation mentioned above. In each case the mode is characterized by a small set of single-key commands: NEXT instance, PREVIOUS instance (in some cases), ABORT the process restoring the original state, and CHOOSE the current instance. The first two operations are natural and convenient, the third is necessary and has presented no problems, but the last interferes with the flow of the editing process, its necessity leading to errors. (To be more precise, it leads to erroneous keying. Since it is difficult to accidentally return to the main editor the integrity of the target node is generally not compromised.) Though the act of toggling or flipping through possibilities is intuitively appropriate to the situation, these modes are not sufficiently distinct to motivate an explicit selection and return operation. Since the two-stroke scheme for command entry requires only a small set of keys, I have been able to use a set of keys for the modes we are discussing which is disjoint from that set, but the operations are seen to take place within the main editing window, and conceptually they are most definitely of a structurally oriented nature.

One could conceivably alleviate the problem by making the operations appear to be distinct (by allocating special windows for them, for example). I submit that this approach would introduce undesirable elements of artificiality and complexity. I propose, instead, to eliminate the necessity for explicit selection. Under this scheme, each succeeding instance or state would become the current one, with the option of restoration of the original always available. This would enable the user to invoke operations by immediately keying in the approproate two-key sequence. In such an implementation the main control loop would be executed each time a function key was read from the keyboard and, in the case of mode-specific keys, interpretation would be dependent upon the value of some CurrentMode variable. The search operations, which are currently implemented as traversals rather than loops, would be particularly affected by such a reimplementation.

Such a restructuring of the text-editing mode, as it is currently implemented, seems neither necessary nor desirable. However, if the *in situ* text editing suggested in the previous chapter were to be implemented, the distinction between that mode and the main structural editing mode might be lost, in which case the approach of the previous paragraphs could be generalized to include the text editor.

STRIP, the complement to EMBED

Just as it is sometimes desirable to add a level of nesting in the code, so is it sometimes desirable to strip away a level. All of the examples cited at the beginning of the section have their counterparts:

> I now wish this code to be executed unconditionally.
>
> Testing of the `Continue` flag is now superfluous.
>
> No wonder I'm getting an error. I should be passing the node itself to this routine, not its class!

I have looked at routines to unqualify if statements and, more interestingly, if...else statements. I have also experimented with an assortment of commands which, in a sense, partially unqualify various statement types, e.g., reducing the number of iterations by converting a looping construct into an if statement, or reducing the degree of abstraction by converting a case statement into a series of nested if...else statements. These operations, though sometimes interesting in their own right, have not provided the desired symmetry with EMBED. Some have been incorporated into other commands yet to be discussed; some have simply been dismissed as dead-ends. To complement EMBED what has been provided is a mechanical STRIP which, subject to compatibility constraints, substitutes the target node itself for that target node's parent. Though not terribly interesting conceptually, STRIP is a useful function, providing not only the implementation of the notion "un-nest," but providing as well the capacity to undo erroneous EMBED's with a single two-key sequence.

The ENGULF/EJECT Family

There are other operations where the notion of nesting is important, i.e., those which involve transfers of structures between levels of nesting already existing in the code. They come into play in response to programmer discoveries of the following sort:

> No wonder it's producing garbage. This statement belongs within the preceding loop/if statement/ ... .

Here one statement is in effect "engulfed" by another. It might also be desirable to perform the complementary operation, i.e., to "eject" one statement from another. These complementary notions do not apply to statements alone. For example, given the declaration of a number of variables with type in common,

```
VAR a, b, c : SomeType;,
```

it might be desirable to split out or eject a particular variable thus

```
VAR a, c : SomeType;
    b : SomeType;,
```

so that it was textually distinct and could be commented separately. Note that in this case the ejected item takes its associated type with it.

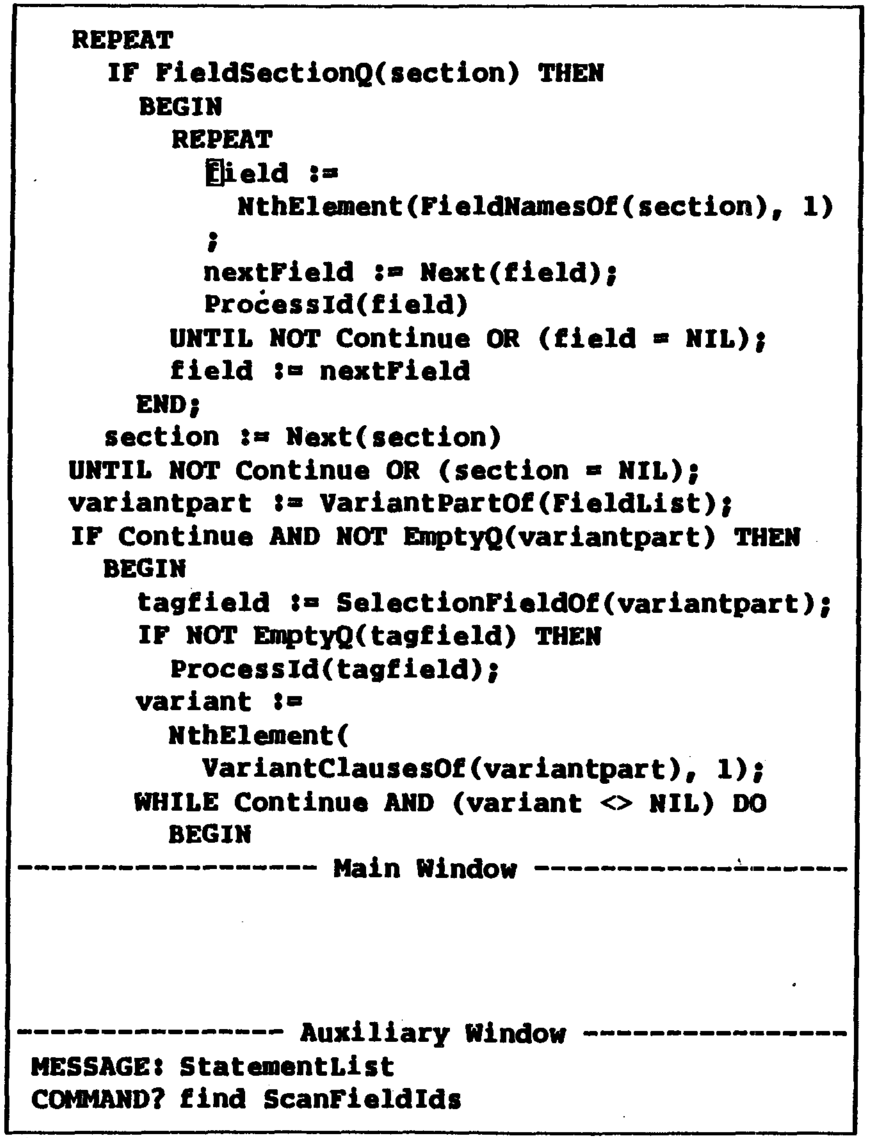
I have named the commands embodying these notions ENGULF and EJECT [4]. It is evident that one may ENGULF either the next item in the textual sequence or the previous one. Similarly, one may wish to EJECT an item either forward or backward. In combination with the capability to swap items (see next section), a single directionality would be sufficient, but, as was the case for the basic command INSERT, it seemed advisable to provide the full set of operations (ENGULF-NEXT, ENGULF-PREVIOUS, EJECT-BACKWARD, EJECT-FORWARD) if smooth, unencumbered editing were to be achieved. Unlike EMBED/STRIP, ENGULF/EJECT is not a notion or .command which may be applied universally in a mechanical fashion. Questions of how it is to be applied, and to what sorts of nodes, have had to be answered on a case-by-case basis. I will now illustrate the facility with a detailed description of its application in the realm of statements, where it is fully operational. Topics to be discussed include operand selection, the operation appropriate to various combinations of operand and context, and the placing of the cursor subsequent to the operation. I will then identify other circumstances in which ENGULF/EJECT is applicable. Areas where the concept is tantalizing but not really feasible will also be pointed out.

The paradigmatic operation

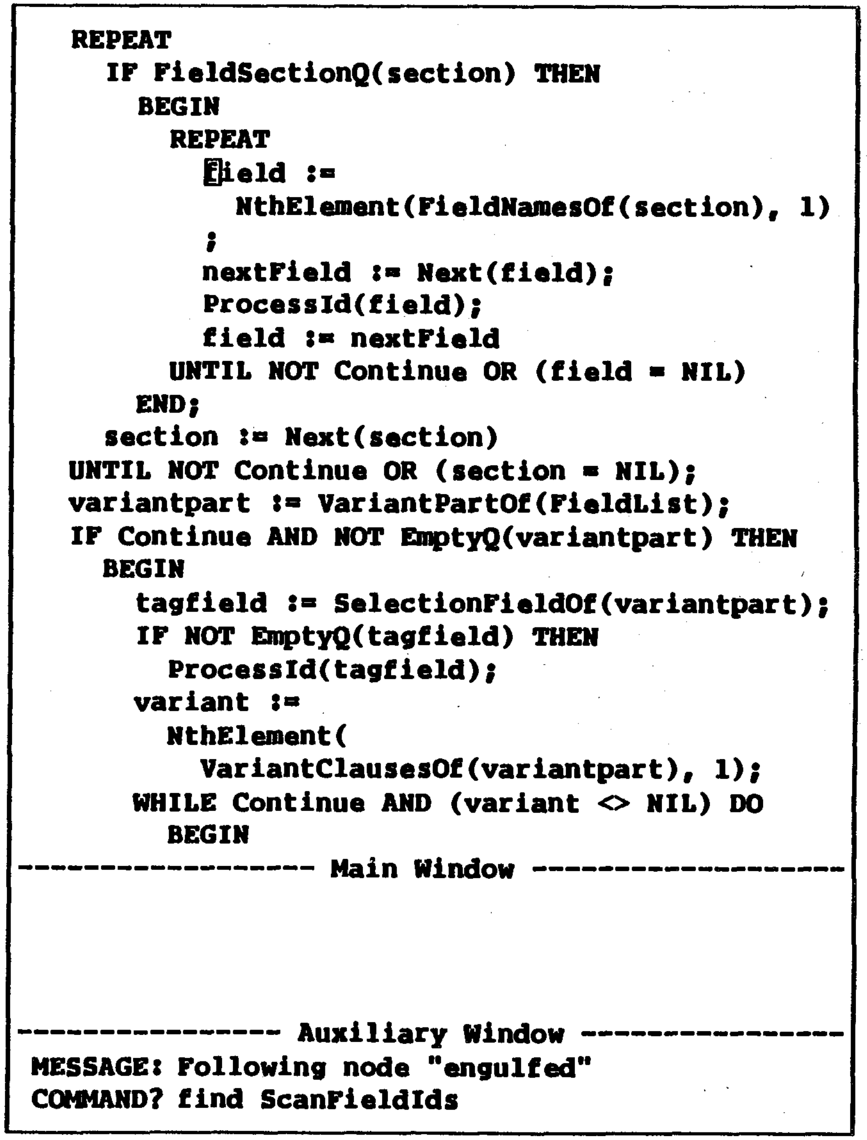
As a starting point let us consider the application of ENGULF-NEXT to the body of some enclosing statement (Figure 5.3). By body I mean the statements to be executed within a repeat, while or for loop, or the consequent or alternate statements of an if statement. Here both implementation and discussion are complicated by the peculiarities of Pascal syntax, which dictates that with one exception the body may be either a simple statement or a compound statement, in which latter case the

----

[4] The family of ENGULF/EJECT operations contains two general sets of operations corresponding to the two kinds of circumstance described above: where the "engulfees" and "ejectees" have some other node associated with them, and where they do not. I have given considerable thought to the idea of assigning different names and different key-stroke sequences to the operations of each of these types. To keep down the number of keying sequences, and because the operations do have something in common, I have not done so.



```
REPEAT
  IF FieldSectionQ(section) THEN
    BEGIN
      REPEAT
        field :=
          NthElement(FieldNamesOf(section), 1)
        ;
        nextField := Next(field);
        ProcessId(field)
      UNTIL NOT Continue OR (field = NIL);
      field := nextField
    END;
  section := Next(section)
UNTIL NOT Continue OR (section = NIL);
variantpart := VariantPartOf(FieldList);
IF Continue AND NOT EmptyQ(variantpart) THEN
  BEGIN
    tagfield := SelectionFieldOf(variantpart);
    IF NOT EmptyQ(tagfield) THEN
      ProcessId(tagfield);
    variant :=
      NthElement(
        VariantClausesOf(variantpart), 1);
    WHILE Continue AND (variant <> NIL) DO
      BEGIN
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: StatementList
COMMAND? find ScanFieldIds
```



```
REPEAT
  IF FieldSectionQ(section) THEN
    BEGIN
      REPEAT
        field :=
          NthElement(FieldNamesOf(section), 1)
        ;
        nextField := Next(field);
        ProcessId(field);
        field := nextField
      UNTIL NOT Continue OR (field = NIL)
    END;
  section := Next(section)
UNTIL NOT Continue OR (section = NIL);
variantpart := VariantPartOf(FieldList);
IF Continue AND NOT EmptyQ(variantpart) THEN
  BEGIN
    tagfield := SelectionFieldOf(variantpart);
    IF NOT EmptyQ(tagfield) THEN
      ProcessId(tagfield);
    variant :=
      NthElement(
        VariantClausesOf(variantpart), 1);
    WHILE Continue AND (variant <> NIL) DO
      BEGIN
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: Following node "engulfed"
COMMAND? find ScanFieldIds
```

ENGULF NEXT

Figure 5.3: ENGULF-NEXT applied to the body of a repeat loop.

structure of interest is the statement list constituting the body of that statement. The situation is complicated further by the case of the repeat statement (the afore mentioned exception) where the body is always a statement list. To select a non-compound statement or a repeat loop's statement list, the user must place the cursor upon the desired node itself. In the case of a compound statement, either the statement itself or its statement list may be selected.

For ENGULF-NEXT to be meaningful the enclosing statement (for loop, if statement, etc.) must be a member of a statement list, and there must be at least one statement following the enclosing statement in that sequence. This statement is removed from that list and appended to the body list to which ENGULF-NEXT has been applied. If the selected statement was a non-compound one, then it is automatically included in a statement list embedded in a compound statement to enable the above operation.

An important consideration in the determination of the proper placement of the cursor after an operation is regard for the operation likely to come next. In the case of ENGULF, the succeeding command is likely to be another ENGULF. With this in mind the cursor is placed upon the body list. (Once ENGULF has been executed there is guaranteed to be such a list.) Note that the sequencing of a sublist incorporated through repeated application of ENGULF-NEXT is maintained.

It may be desirable to bring the succeeding statement into the body list at a position other than the very tail end. To do this the user places the cursor upon an individual element of the body list before invoking ENGULF-NEXT. The engulfed statement then is inserted into the body list at the position immediately following the selected element. Since it is desirable that repeated applications of ENGULF-NEXT maintain the ordering of the engulfed statements, the cursor is placed upon the newly engulfed element.

There are two important special cases of the application of ENGULF where the operand node is actually replaced by the engulfed node. The first is that of an empty optional node, specifically the alternate of an if statement, and the second is that of a placeholder node, specifically the consequent of an EMBED-created if statement (or, potentially, an unexpanded template). This invocation of ENGULF in tandem with EMBED has, in practice, proven to be very useful.

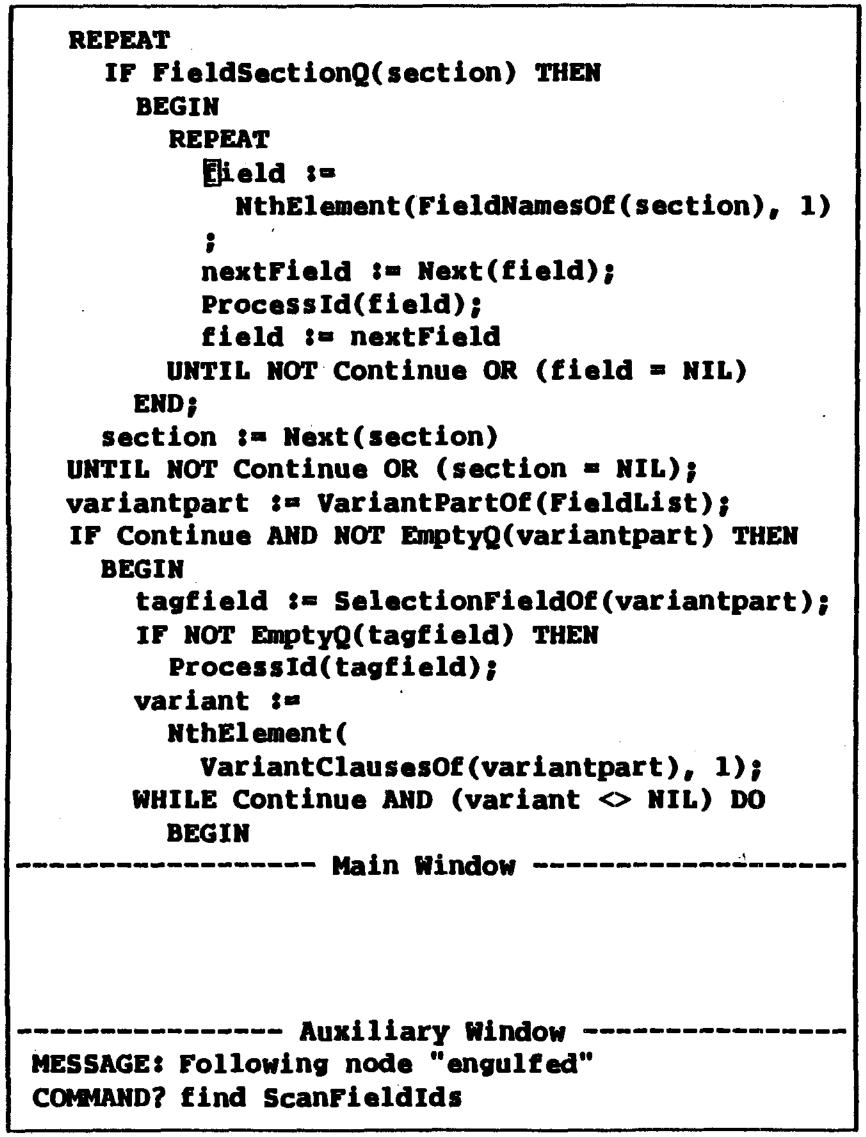
Complementary to ENGULF-NEXT is EJECT-FORWARD. If the body as a whole is selected, then the last statement in the body list is removed and becomes the element immediately following the enclosing statement in the enclosing statement's parent list. If the enclosing statement is not a list element, then a statement list parent (and, if necessary, a compound statement grandparent) is created for it. The cursor is placed on the body list. When EJECT is applied to a singleton body list or a non-compound statement, the statement is ejected and the null statement is substituted within the enclosing statement. If an individual element is chosen, then that element is ejected. In order to preserve the ordering of statements ejected by repeated invocations of EJECT-FORWARD, the cursor is placed on the element which previously preceeded the selected (and ejected) element. This also means that ENGULF-NEXT serves to undo EJECT-FORWARD and *vice versa*. There is an obvious exception, i.e., when the element ejected is the first on the body list. In this case the implementor must either abandon the principle of maintenance of ordering and the desirable inverse property and place the cursor on some available node (e.g., the element which followed the selected one) or introduce a spurious null statement into the code. Since, under the circumstances just described, there is no next statement in the sequence, and since maintenance of the strict inverse relationship has proven to be difficult or impossible on other grounds, I have chosen the former action. Admittedly, this decision is at odds with the policy stated above for ejection of lone statements. I claim that leaving the null statement upon ejection of a singleton statement is generally what the user will have had in mind, whereas introduction of a null statement simply to give the cursor a place on which to rest interferes unduly with the integrity of the code.

For ENGULF-PREVIOUS and EJECT-BACKWARD the operand selection process is identical to that described above. The actions are symmetrical to those that have been cited. ENGULF-PREVIOUS causes the statement preceding the enclosing statement to be moved to the head of the body list when the list as a whole is selected, and to the position immediately preceding that of the selected node if an individual element is selected (with the cursor moving to the newly engulfed node). EJECT-BACKWARD (Figure 5.4) causes the first element of the body list, or the element selected, to be moved to the position immediately preceding the enclosing statement. Once again, the cursor is positioned so as to preserve sequencing over repeated invocations.

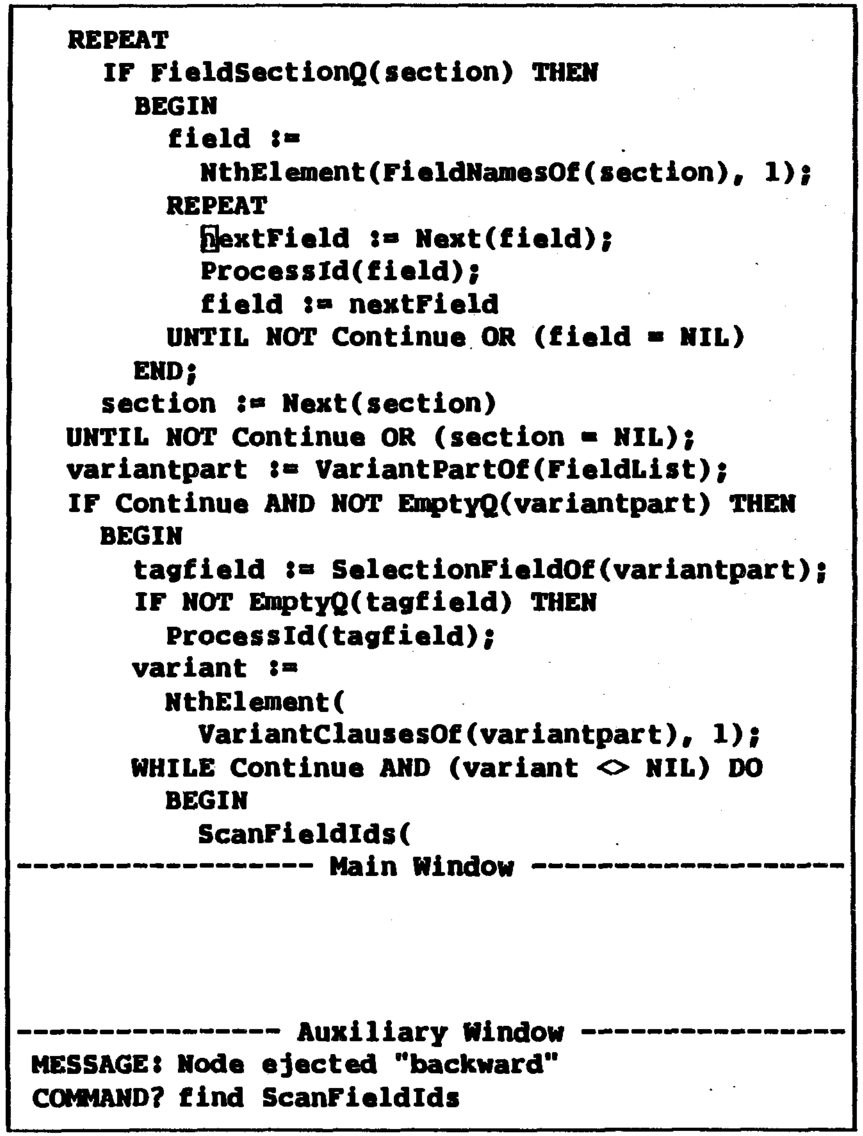
It can be seen that ENGULF/EJECT is not comprehensive in its functionality. One cannot engulf the statement following the enclosing statement onto the head of the body list. Nor can one engulf the preceding statement onto the very tail end of the body list. That is, these operations cannot be performed by means of a single command. They are easily performed using ENGULF in conjunction with other commands introduced later in the chapter. There are also opportunities for application of ENGULF/EJECT in the realm of the statement which I have not covered here, notably the case of enclosure in a case statement or a with statement. Since the intention of this subsection is to illustrate the EMBED/EJECT operation in its most straightforward guise, I will withhold discussion of these anomalous cases until the next subsection.

Further applications of ENGULF/EJECT

The applications of ENGULF to be discussed here all share a very general schema exemplified by the case of nesting statements. The selected node is always a list, or element of such a list, which is embedded (perhaps multiply embedded) in another node, which is itself an element of a list whose other elements have some relation to the selected node. Perhaps they are of the same type as the selected node;



```
REPEAT
  IF FieldSectionQ(section) THEN
    BEGIN
      REPEAT
        field :=
          NthElement(FieldNamesOf(section), 1)
        ;
        nextField := Next(field);
        ProcessId(field);
        field := nextField
      UNTIL NOT Continue OR (field = NIL)
    END;
  section := Next(section)
UNTIL NOT Continue OR (section = NIL);
variantpart := VariantPartOf(FieldList);
IF Continue AND NOT EmptyQ(variantpart) THEN
  BEGIN
    tagfield := SelectionFieldOf(variantpart);
    IF NOT EmptyQ(tagfield) THEN
      ProcessId(tagfield);
    variant :=
      NthElement(
        VariantClausesOf(variantpart), 1);
    WHILE Continue AND (variant <> NIL) DO
      BEGIN
----------------- Main Window -----------------




--------------- Auxiliary Window ---------------
MESSAGE: Following node "engulfed"
COMMAND? find ScanFieldIds
```



```
REPEAT
  IF FieldSectionQ(section) THEN
    BEGIN
      field :=
        NthElement(FieldNamesOf(section), 1);
      REPEAT
        nextField := Next(field);
        ProcessId(field);
        field := nextField
      UNTIL NOT Continue OR (field = NIL)
    END;
  section := Next(section)
UNTIL NOT Continue OR (section = NIL);
variantpart := VariantPartOf(FieldList);
IF Continue AND NOT EmptyQ(variantpart) THEN
  BEGIN
    tagfield := SelectionFieldOf(variantpart);
    IF NOT EmptyQ(tagfield) THEN
      ProcessId(tagfield);
    variant :=
      NthElement(
        VariantClausesOf(variantpart), 1);
    WHILE Continue AND (variant <> NIL) DO
      BEGIN
        ScanFieldIds(
----------------- Main Window -----------------




--------------- Auxiliary Window ---------------
MESSAGE: Node ejected "backward"
COMMAND? find ScanFieldIds
```

EJECT BACKWARD

Figure 5.4: EJECT-BACKWARD applied to the body of a repeat loop.

perhaps they have nodes of the same type embedded within them. The method of selection of engulfer and engulfee together with the rules for positioning nodes and cursor are essentially those described. Therefore, I will omit these details in the ensuing discussion except where there is something distinctive associated with a particular application.

This is probably the appropriate place to address the remaining cases of application of ENGULF/EJECT to statements. They are the case statement and the with statement. Where the case statement is concerned, I am now referring to the statement (or body list) which is a part of each case clause of the case statement. Application to the list of expressions in each such clause will be dealt with shortly. The issue of statements interior to a case statement was not resolved in the preceding subsection because of the complication of added levels of nesting, because I am not absolutely convinced of its usefulness, and finally, because there is implied in the case statement a series of nested "else ifs." In the final analysis, the clauses of the case statement are all at the same level, and when this application of the command is implemented, they should be treated in a manner completely analagous to that used for the other statements. Ejected statements should go to the level of the enclosing statement; engulfees should be drawn from that level.

The with statement of Pascal is something of an anomaly in that, rather than affecting the logic of control, it serves to introduce a new scope. This introduces a new question: Does one ENGULF and EJECT on a strictly textual basis, or is it more appropriate to consider the semantics of the situation when dealing with the variables involved. Certainly the operations discussed so far have been of a semantics altering sort. But are the alterations that would result from simply moving variables in and out of a with statement of a desirable nature. Other operations exist to "qualify" and "unqualify" the field identifiers of record component variables. I submit that in the case of the with statement it is better to preserve semantics. For example, application of ENGULF-NEXT to the body of the with in

```
WITH NamedNode DO
    BEGIN
    NameField := NodesName;
    NodeField := SomeNode
    END;
NamedNode.Next := PtrToNextRec
```

should yield

```
WITH NamedNode DO
    BEGIN
    NameField := NodesName;
    NodeField := SomeNode;
    Next := PtrToNextRec
    END,
```

and a subsequent invocation of EJECT-FORWARD would restore the original version exactly. Of course performance of these operations requires access to the definitions of the record types, and they would not be performed by the editor if the user had not ensured that the necessary environment was in place.

There are other situations quite similar to that of nested statements. One such case, where ENGULF/EJECT would be useful, particularly in conjunction with EMBED, involves the arguments of a function call which is itself one of the arguments of a function or procedure call. For example,

```
NthElement(Append(List1), List2, N)
```

becomes

```
NthElement(Append(List1, List2), N)
```

when ENGULF-NEXT is applied to `List1` (or the argument list of which it was the sole member). Another situation very similar to that of nested statements is that of nested record definitions. For example, given

```
RECORD
    Name : StringType;
    PersonalData : RECORD
        Weight : INTEGER;
        Sex : CHAR;
        SIN : StringType
        END
```

```
          END;,
```

application of EJECT-BACKWARD to the last field section in the nested record would yield

```
RECORD
    Name : StringType;
    SIN : StringType;
    PersonalData : RECORD
        Weight : INTEGER;
        Sex : CHAR
        END
    END;.
```

I will now turn my attention to that class of applications of ENGULF/EJECT so distinct that I have considered giving it its own name and key-sequences. The target objects in this case are lists (and their individual elements) which have associated with them another node, and where the resulting construction may be a member of a list of constructions of the same type. Specifically, I am referring to the variable names of a variable declaration, the case constants of a case statement's case clauses or the variant clauses in the variant part of a record, the field names of a record's field section, and the parameter names of a parameter section (of either the variable or value sort) of a subroutine declaration. Unlike the other applications described in this subsection, which are in the proposal stage, application of ENGULF/EJECT to these lists with associated nodes is fully implemented. The rules for selection of operands, placement of cursor, etc., are the same as those cited for the statement realm. The distinctive aspect of this case is that the element which is moved retains its logical association with the companion node.

Perhaps this is best illustrated by the example of EJECT. When EJECT-FORWARD is applied to the list of variable names `a, b, c, d` in

```
VAR a, b, c, d : SomeType;
```

the result is

```
VAR a, b, c : SomeType;
    d : SomeType;.
```

If EJECT-BACKWARD is then applied to the element b, the situation becomes

```
VAR b : SomeType;
    a, c : SomeType;
    d : SomeType;.
```

There is some conceptual similarity between this instance and that of the with statement, where the associations between record variables and their fields are maintained.

If the appropriate neighbor of the enclosing node has a singleton list, and if the associated nodes are the same (i.e., they are structurally equal), then ENGULF may be applied. For example, the operation shown above could be undone. Or, to cite an example from the field name realm

```
RECORD
    Name : StringType;
    SIN : StringType;
    PersonalData : RECORD
        Weight : INTEGER;
        Sex : CHAR
        END
    END;
```

becomes

```
RECORD
    Name, SIN : StringType;
    PersonalData : RECORD
        Weight : INTEGER;
        Sex : CHAR
        END
    END;
```

upon application of ENGULF-NEXT to the field identifier Name (or its parent list). I have considered less constrained implementations of ENGULF in which, if the structure immediately adjacent to the enclosing node were not appropriate (e.g., the associated node did not match), a search would be made forward (or backward) for a match. I have also looked at allowing ENGULF to "steal" an element from a non-singleton list. I have decided against these versions of the command. Although the attempt to assure that ENGULF and EJECT are, in every case, inverses of one another has been abandoned, it was deemed desirable to maintain their complementary nature so far as possible.

Application of ENGULF/EJECT within case statements and record variants is completely analogous to that described above. So is its application to parameter sections. In the latter case there is an interesting "wrinkle." One would like to be able to use ENGULF for operations such as:

```
    PROCEDURE Proc(VAR Par1 : INTEGER; Par2 : INTEGER);    -->
        PROCEDURE Proc(VAR Par1, Par2 : INTEGER);.
```

But if the value or variable nature of the parameter section is ignored, ambiguities are unavoidable. Consider, application of EJECT to the parameters in the result version above. Should the result be

```
    PROCEDURE Proc(VAR Par1 : INTEGER; Par2 : INTEGER);
```

or

```
    PROCEDURE Proc(VAR Par1 : INTEGER; VAR Par2 : INTEGER);?
```

In order to maintain consistency with the other applications of ENGULF/EJECT, I have decided that the variable case and the value case should be treated as distinct. (This also corresponds to their treatment in the formal MPS Pascal grammar.)

The last instance of ENGULF/EJECT to be considered is its application to the index types of an array type. For example, in the case of

```
        ARRAY [0..Max1, 0..Max2] OF SomeType;,
```

selecting the index list and invoking EJECT-FORWARD would yield

```
        ARRAY[0..Max1] OF ARRAY[0..Max2] OF SomeType;.
```

Although this case does not match precisely any of the schemata discussed above, it should be clear that there are analogies to the whole set of ENGULF/EJECT operations that have been introduced.

I have been tempted to try to increase the power of ENGULF/EJECT by extending its application to still more structures/circumstances. For example, the notion seems applicable to various expressions. Unfortunately analogies must be stretched to accomplish this, and the intuitive appropriateness of these operations is lost. An informal "poll" of five programmers has yielded three or four "appropriate" results for the application of EJECT to the bracketed (sub-) expression in

```
NOT (a AND b).
```

Another puzzling case is that of the output statements WRITE and WRITELN. Here the target list is associated not with a node but with a token and, in the semantic sense, with an action. Perhaps the operation

```
WRITELN(a, b, c)    -->

          WRITELN(a, b);
          WRITELN(c)
```

should be supported. Under Pascal semantics, the first parameter of a standard output statement may, or may not, designate a file, and the changes to the program's functionality would differ markedly depending upon which were the case. In order to avoid such unsavory alterations to semantics, ENGULF/EJECT should here be implemented, if at all, as a context dependent operation with the first argument carrying over with the ejected one if, and only if, it denotes a file, and ENGULF being applicable only when the output of both statements is to a common file.

The SWAP/ROTATE Family

The two editing notions that have been considered so far are oriented toward the issue of nesting and the concomitant semantics, though they may in fact involve a textual reordering of elements. I will now look at operations where the reordering, *per se*, of sequences of elements is the intent. Typically these reorderings will be of elements at the same level of nesting, members of the same list in fact, though there is a major exception which has been placed in this family by virtue of other similarities. Programs are pieced together of various lists or sequences, as well as "list-like" structures. There are a number of reasons why the programmer might wish to manipulate the ordering of these sequences. In some cases the goal is to increase the readability or clarity of the code, e.g. reordering of various declarations or the case clauses of a case statement (in this latter case, the motivation may be the improvement of efficiency as well). In other cases the operations may be intended to effect changes to functionality, e.g., reordering of statements or of actual parameters.

A common example of an operation of the latter sort is that of moving a statement (frequently one implementing the incrementing of a counter variable) in the body of a looping construct from the bottom of the loop to the top. This example will serve to illustrate the first member of this family of commands. Since, as the last element moves to the head of the list, the bulk of the list moves down, or forward, an appropriate name for the command is ROTATE-FORWARD. When applied to a list *element*, this command causes the ROTATE-FORWARD operation to be performed on the tail sublist whose first element is the selected element. When applied to a list, ROTATE-BACKWARD moves all elements backward, except the first one, which goes to the last position on the list. Applied to a list element, it performs the ROTATE-BACKWARD operation on the sublist whose *last element* is the selected one. Both rotational commands leave the cursor positioned in such a way as to facilitate repeated applications. When the list as a whole is the operand, the cursor remains on that list. When an element has been selected, the cursor remains at the *position* originally selected.

Using the two directional modes one can rotate the "front" portion of a list backward or rotate the latter portion of a list forward. One would like, in addition, to be able to perform arbitrary rotations upon sublists, i.e., to rotate them in either direction and to rotate sublists not necessarily bounded by a terminus of the list as a whole. I propose to deal with this in a manner similar to that employed for the primitives SELECT-SUBLIST and DELETE-SUBLIST. ROTATE-SUBLIST (or ROTATE-TOP) would act upon the sublist bounded by the current node and the top node on the stack, which must have the same parent list as the currrent node. The top node would be moved to the location of the current node and the remainder of the sublist would move toward the location formerly occupied by the top node. To facilitate repeated rotations the cursor would remain at the same list position and the top node on the node stack would be replaced by a reference to the node now in the position formerly occupied by the old top node.

The other major way to effect alterations in sequencing is by means of an exchange or "swap." I will introduce this family, or subfamily, of operations by describing its most general form, the arbitrary swap, SWAP-TOP. This command causes the current node and the top node to be exchanged in their respective contexts. The cursor is left at its original contextual location, and now rests on the former top node. The top node on the node stack is replaced by a reference to the former current node. Thus the user may readily gain access to that location which has been significant to the operation but may be remote from the current node location. He/she may also undo the operation by means of another invocation of SWAP-TOP. Note that this command is less rigidly constrained than the ROTATE operation or the other SWAP operations to be described below. Current node and top node need not have a common parent list; only node type compatability is required.

Any exchange of nodes can be accomplished by means of SWAP-TOP; however, a particular class is common enough (in my experience it is the most common sort of swap) to warrant a separate command. This is the exchange of adjacent list elements. One such command, SWAP-NEXT, is sufficient, but, once again, for the sake of convenience and cursor-positioning considerations, support for SWAP-PREVIOUS is justified. By having the cursor stay with the selected element when it is swapped, a bubble-up (or bubble-down) procedure is enabled. Though it is at odds with the SWAP-TOP implementation, this cursor placement strategy facilitates the rotation of small sublists in a fashion which may be more convenient than rotation by means of ROTATE-TOP.

It is not immediately obvious how the SWAP notion might be applied to a list, or whether it is applicable at all. Perhaps it would be not altogether counter-intuitive to have the command produce a complete reversal of the ordering of the elements of the target list.

Commands of a Transformational Nature

The last family of operations to be considered is the one whose members probably best deserve the appellation "high level." These are operations borrowed from the realm of transformational programming. Under the transformational programming model successive, correctness preserving, potentionally automatable transformations are applied to a formal specification, producing, eventually, executable code. Another, looser, formulation of the model has the programmer writing good, modular code, then applying such transformations as necessary to yield code (in the same language) which is capable of more efficient execution.

Certainly, candidates for inclusion as editing commands are to be found here. Transformational programming of the looser kind is an identifiable aim of the editing process. The commands are likely to be powerful, in that a single key sequence may replace an arbitrary amount of textual (or basic syntax-based) editing. And they take advantage of the underlying grammar-based methodology, under which their implementation is relatively straightforward (though by no means trivial).

However, not all such transformations are appropriate to a general purpose program editor. Typically, the circumstances under which thay may be applied are highly restricted, and in many cases only likely to arise as a result of previous transformations. Their potential outcomes are limited as well, aimed, as they are, almost exclusively at optimization. Nor are they generally easy to understand or compose.

The problem, once again, has been to identify categories of operation which are sufficiently intuitive and general in application. The conceptual backbone for this family has been provided by three transformational routines due to Cameron [Cam87] which implement simplification of expressions, constant propagation, and in-line coding of procedure calls. The first two are conceptually and functionally related and seem to constitute one subfamily, SIMPLIFY/PROPAGATE. The third serves as paradigm for another subfamily which I shall call INLINE/ENCAPSULATE [5].

The INLINE/ENCAPSULATE subfamily

INLINE takes its name from the operation of encoding the statements of a subroutine call in line, but I wish to extend the notion to other cases where some program element is replaced by code which is equivalent in functionality, but is expressed at a lower level or, somehow, more explicitly. Though many such operations are, like the procedure call example, aimed at an increase in efficiency, I have not restricted my attention to such cases. In the following sections I will discuss various flavours of the INLINE notion. First, the procedure call case will be briefly examined. Next, I will introduce two other operations, which I have implemented, that are quite dissimilar to the procedure call operation but seem to belong in this family nonetheless. Proposed extensions of different sorts will be presented as well. Finally, the matter of a complement to INLINE will be dealt with.

Replacement of a procedure call by the statements of its body to eliminate procedure call overhead is a standard optimization technique. Its incorporation into the editor is appropriate, since it is an operation which should be applied locally in those situations where analysis (or profiling) has shown it to be beneficial. Though straightforward enough when facilitated by the MPS package, its implementation is not trivial, in that arguments must be substituted for the corresponding formal parameters, declarations must be created for the subroutine's local declarations, and renaming must take place if there are clashes with any existing identifiers. Extension of the operation to function calls is complicated by the necessity for the introduction of intermediate variables, and the effects of such an extension are complicated by the possibility of side effects in the target code.

[5] Other terms which have been used for similar notion pairs are *unfold/fold* and *devolution/evolution.*

A transformation which is not of the optimizing kind is the "in-line coding" of a with statement. By introducing a new scope (or scopes) the with statement may make comprehension of code difficult. Simple withs are generally not a problem, but when withs are nested, and there are multiple instances of field/variable names the situation is somewhat more complicated. For example, given the declarations

```
VAR f1, f2, f3, f4, f5, f6 : t;
    r1 : RECORD
            f1, f2, f3 : t
            END;
    r2 : RECORD
            f2, f3, f4 :t
            END;
    r3 : RECORD
            f3, f4, f5 : t
            END;,
```

what is one to make of the statement

```
WITH r3, r1, r2 DO
    BEGIN
    f1 := f2;
    f2 := f3;
    f3 := f4;
    f4 := f5;
    f5 := f6
    END?
```

Recognizing that WITH r3, r1, r2 DO ... is shorthand for

```
WITH r3 DO
    WITH r1 DO
        WITH r2 DO
            ...,
```

the programmer must, for each variable in the with's body, search the record definitions from the inside out for a matching field. This can be a daunting task, particularly if the declarations are two hundred, or so, lines away. INLINE automatically yields

```
r1.f1 := r2.f2;
r2.f2 := r2.f3;
r2.f3 := r2.f4;
r2.f4 := r3.f5;
```

```
r3.f5 := f6,
```

where, at least, the references are explicit. Such an expansion may also enable the writing of additional code which would otherwise be impossible due to duplication of names, e.g.

```
f3 {the variable} := r2.f3.
```

As implemented, the command acts upon the with's body as a whole, as shown here. It might be preferrable to be able to perform partial in-line codings by selecting a record variable from the with's record variable list (or selecting the body as a whole by selecting the list as a whole). Note that with statements included in the body statements are not affected by INLINE.

Another implemented application which may not increase clarity but does facilitate alterations to the code is transformation of a case statement into a series of nested if...elses, e.g.

```
CASE e OF
    a : s1;
    b, c : s2;
    d,e,f : s3
    END
```

becomes

```
IF e IN [a] THEN
    s1
ELSE
    IF e IN [b, c] THEN
        s2
    ELSE
        IF e IN [d, e, f]
        THEN
            s3,
```

where, for example, predicates could be modified to further qualify selected statements. This transformation definitely has the flavour of the INLINE family. Note, however, that it is not strictly semantics preserving, in that a failure to match e leads to a run time error in the case version, but not in the nested if...else version.

A more conventional INLINE operation is loop "unrolling." This optimization technique, aimed at reduction of the number of tests in critical passages of code, has not been implemented but could be without generating ambiguities in selection. If both the initial and final expressions are constants, a for loop may be unrolled completely, i.e.,

```
FOR i := m to n DO
     s
```

becomes

```
$s_m$;
$s_{m+1}$;
$s_{m+2}$;
.
.
.
$s_{n-1}$;
$s_n$,
```

where s_j represents s with j substituted for all instances of i contained within it. Repeat and while loops may be partially unrolled, i.e.,

```
REPEAT
     sl
UNTIL e
```

becomes

```
sl;
WHILE NOT e DO
     sl,
```

and

```
WHILE e DO
     s
```

becomes

```
IF e THEN
     REPEAT
          s
     UNTIL NOT e
```

The gains are not immediately apparent, but if the value of e is known, then tests (and, sometimes, their associated statements) can be eliminated, an operation facilitated by commands described in the next subsection, and the operations may be (repeatedly) applied to the loops.

I speculate that it might be meaningful to expand the INLINE concept beyond the realm of the statement. Specifically it could be used to substitute a type definition for a type name. For example, given

```
TYPE RA = ARRAY [Min..Max] OF SomeType;
```

and

```
VAR Foo : RA;,
```

application to RA (in the variable declaration) would yield

```
VAR Foo : ARRAY [Min..Max] OF SomeType;.
```

Once more the rationale for this operation is the facilitation of further alterations.

The commands I have introduced in this chapter have typically had complementary operations associated with them, as does INLINE. I have chosen to refer to this notion as ENCAPSULATE. Here we leave the realm of implementation and enter that of speculation. These operations have not actually been implemented. Nonetheless, it is worthwhile to consider briefly such a facility, its potential, and the problems associated with it.

Encapsulation, the hiding of the sort of detail that is exposed by INLINE, should really be a part of the early stages of software production, but programmers may find themselves engaged in this activity at any time in the life cycle, and editor assistance would be helpful. It seems unlikely that the situation would arise where constraints would be satisfied such that nested if...elses could be transformed into an equivalent case statement, but gathering statements containing component variables (i.e., varaiables of the form <Record:Variable>"."<Field:Identifier>) into a with statement may occur in Pascal programming. And it certainly sometimes happens that sequences of statements coded in-line turn out to be required in some other context, in which case it would be desirable to have the automated capacity to replace the sequence with a procedure call and construct a procedure declaration.

If encapsulation routines are to be incorporated into the editor, the problem of the many-to-one nature of the operation must be addressed. Whereas the appropriate INLINE transformation may be unambiguously determined for any given node, the same is not true for ENCAPSULATE. When it is applied to a sequence of statements, for example, is the intent to transform them into a with statement, or to encapsulate them in a procedure call? Moreover, even when the transformation desired is known, there are other choices to be made. If there are a number of potential record variables represented, which is to be the operative one in the with statement, and if more than one is to be used, in what order are they to occur? If a procedure call is required, which variables are to appear as arguments? The method of exhaustive display does not seem to be applicable here. The two-tiered nature of the process complicates the situation. Moreover, since these operations require a great deal of contextual analysis, they are slower than mere template substitution, conceivably slow enough to make the method unworkable. ENCAPSULATE is, perhaps, a candidate for implementation as a hybrid command. Before entering the command sequence, the user would be required to enter pertinent data in the command window. Perhaps a fragment of a with statement (e.g., `WITH r1, r2, r3`) could be used to specify the desire for the with and to identify its record variables. An actual procedure call could specify that alternative.

In the procedure call case, there are other interesting questions. Since Pascal supports the use of both value parameters and variable parameters, it must be determined which parameters are to be of which sort. It is also necessary to determine which variables are to be declared locally within the new procedure. Assuming that the code is initially correct (and complete), these decisions can be made based on analysis of the code. But such analysis could prove so time consuming as to interfere with the interactive editing process. There are also some instances where the programmer will desire to exercise his/her judgement. For example, a read-only variable might automatically be declared as a value parameter, but if it were very large, a competent programmer would declare it as a variable parameter (so that it could be passed by reference). Another question relates to the disposition of the resulting procedure declaration. Should it be inserted into the code automatically, or, to localize the effects of the command, should it not be placed on the node stack to be dealt with by the user as he/she sees fit?

It is not clear that the problems with parameterization can be solve in a fashion which is tidy enough for an editing command. However, a somewhat simpler operation which "factors out" subconstructs, replacing them with automatically generated identifiers, is practical. For example, given the statement

```
a := b + c * d,
```

application of this FACTOR concept do `c * d` would yield

```
cTimesd := c * d;
a := b + cTimesd.
```

Extended to statements, this notion could provide, at least, for their encapsulation in parameterless procedure calls.

The enumeration of possible applications above is not meant to be exhaustive. (ENCAPSULATE/FACTOR could, for example, be applied to types, where it would probably be as important as INLINE, if not more so.) It is meant rather to suggest the possibilities of the notion, on one hand, and on the other, to point out some implementation issues and potential sources of awkwardness.

The SIMPLIFY/PROPAGATE subfamily

The last borrowed operations to be considered are SIMPLIFY and PROPAGATE. SIMPLIFY acts upon expressions, both arithmetic and boolean (and, potentially, set expressions), transforming them into their simplified forms in accordance with established rules, e.g.,

```
1 + 2   -->   3,
x * 0   -->   0,
p AND TRUE    -->    p.
```

It may also be applied to conditional statements, e.g.,

```
IF TRUE THEN p    -->    p.
```

PROPAGATE is applied to variables to which a constant value has been assigned (or, potentially, to constant definitions), causing that value to be substituted for the next occurrence of the variable in the sequence of statements. Sequences of the two operations are readily composed, e.g., SIMPLIFY the right hand side of an assignment, PROPAGATE that value into a subsequent expression, SIMPLIFY *that* expression, etc.

These commands become important when used with one another and with other transformations. One generally does not produce expressions such as 5 + 6 in the course of writing a program, but they do frequently arise as the result of a previous transformational operation. For example, in-line coding of a subroutine with a literal argument will introduce literals into the code where they previously were not present. This may well result in expressions of precisely the type which SIMPLIFY and PROPAGATE are designed to handle. The two commands may then be used to tidy up the code and to remove unnecessary tests of conditions.

CHAPTER VI

THE MANIPULATIVE FACILITY IN ACTION

In the previous chapter I described in detail the families of commands which make up what I have called the manipulative facility. The question remains: How effective are the new commands in a practical editing situation? Some insights into this matter have been gained through application *of* the editor *to* the editor during the latter stages of its development. This included corrections of errors and omissions in large scale additions (which were, for the sake of convenience, composed on the text editor), alterations and enhancements to existing code, and the repair of bugs, old and new.

Based upon these experiences, I have constructed a few examples of the editor in action. Each example demonstrates a solution to a particular editing (and programming) problem. It is hoped that each will be long enough to impart some of the flavour of the editing process. The commentary accompanying each example describes the process, and, where appropriate, makes comparisons with alternate methods based upon more elementary editing operations. The examples have been contrived to demonstrate selected operations from each of the command families. No attempt has been made to demonstrate the full range of applicability of the commands. What is presented instead is a sampling of some of the kinds of operations that I have, so far, encountered in practice. Each example is intended to spotlight one or more of the new manipulative commands. but it is also important to note how these commands interact with each other and with the more basic ones.

Example - Rotation and Transformation

In the ongoing editing of the editor, I have not found myself using what might be called the traditional operations of the transformational family. I suspect that this is a function of the particular stage in the editor's development where the editor has been used. These operations have proven useful elsewhere, and the editor is manifestly a convenient platform for their execution. I *have* found some use for the extensions I have made to the INLINE notion. The following example demonstrates how INLINE may be used to facilitate further alterations.

The SWAP/ROTATE family is largely "sugar." Its operations may be simulated using the basic SELECT and INSERT (or REPLACE) commands together with some cursor motion. Nevertheless, I have found these commands, particularly SWAP-NEXT, to be useful and satisfying. The example shows how ROTATE may be helpful as well.

The problem is to replace case-by-case processing, implemented by the case statement shown in the left screen of Figure 6.1a, by a generic procedure call `Strip1`, while retaining special processing for the `IfStatement` case. In other words, the case statement is to be replaced by code of the following sort:

```
IF NodeType(OldNode) = IfStatement THEN
                      .
                      .
                      .
     {Existing code for the IfStatement case}
                      .
                      .
                      .
ELSE
     Strip1(OldNode, SelectedNode, Marker, Problem).
```

The first step in the process is the production of the new code, i.e., the procedure call. In Figure 6.1a (right screen) the screen cursor is moved to the command line and the node type is specified there, so that the text editor may be invoked (Figure 6.1b). The necessary text is typed in (figure 6.1c) and the resulting node placed on the editor's node stack (Figure 6.1d). The cursor is then returned to the main editing window (Figure 6.1e), where it selects the first item of the case statement's case clause list.

The strategy now calls for replacement of the case statement by an if statement with a series of nested else-ifs. First, however, the IfStatement case must be brought to the head of the case clause list, so that its body statement will form the consequent (then branch) of the if statement. There are various ways of finding and moving that clause, which is located somewhere in the large case clause list. I choose what seems a straightforward expedient, simply rotating the case clause list as a whole until the desired element appears at its head. To accomplish this, the syntactic cursor is moved up (Figure 6.1f) to select the sequence of case clauses as a whole, and the ROTATE command is invoked repeatedly (Figures 6.1g, 6.1h, 6.1i). Since the current syntactic cursor position (on the CaseClauseList) is sufficient to select the case statement for the INLINE command, that command is keyed in as soon as the IfStatement clause is properly positioned (Figure 6.1j).

What remains is replacement of the resulting if statement's alternate (the else branch) with the procedure call which is waiting on the node stack. Navigating the complex of nested ifs and elses by eye would be difficult, so the syntactic cursor is moved down a level in the internal tree (Figure 6.1k), then laterally across that level from the outer if statement's predicate to its consequent (Figure 6.1l), and from the consequent to the desired alternate (Figure 6.1m). Finally the basic command REPLACE causes that node, which comprises a large fragment of code of undetermined length, to be replaced by the previously prepared procedure call.

Though this process requires a number of steps, it is straightforward and effective. It is clear that a simple concatenation of basic, syntax-based commands cannot be substituted. The changes could, of course, be effected textually:

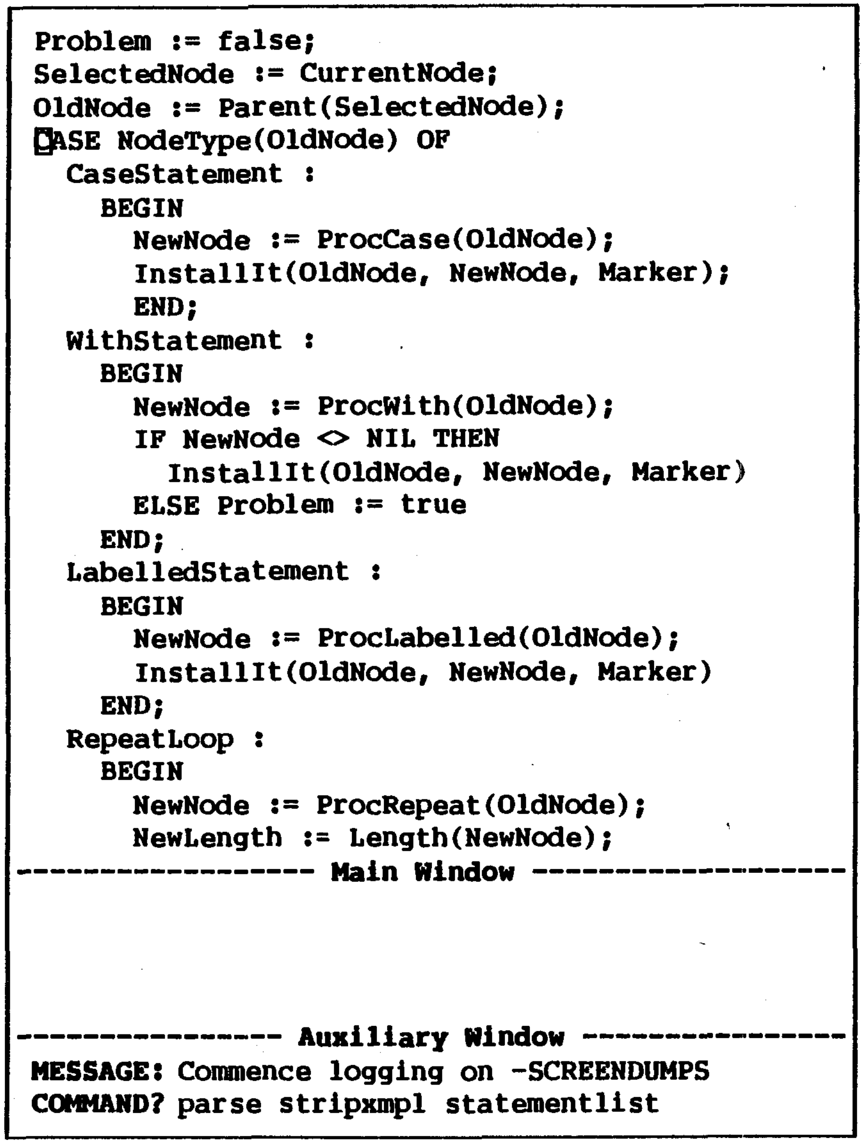
4. Note line number of first line of the case statement.
5. Find the string, "IfStatement."
6. Delete intervening range of lines.
7. Type "IF NodeType(OldNode) = IfStatement THEN."
8. Find the end of the case clause (by scrolling downward) and note its

line number.

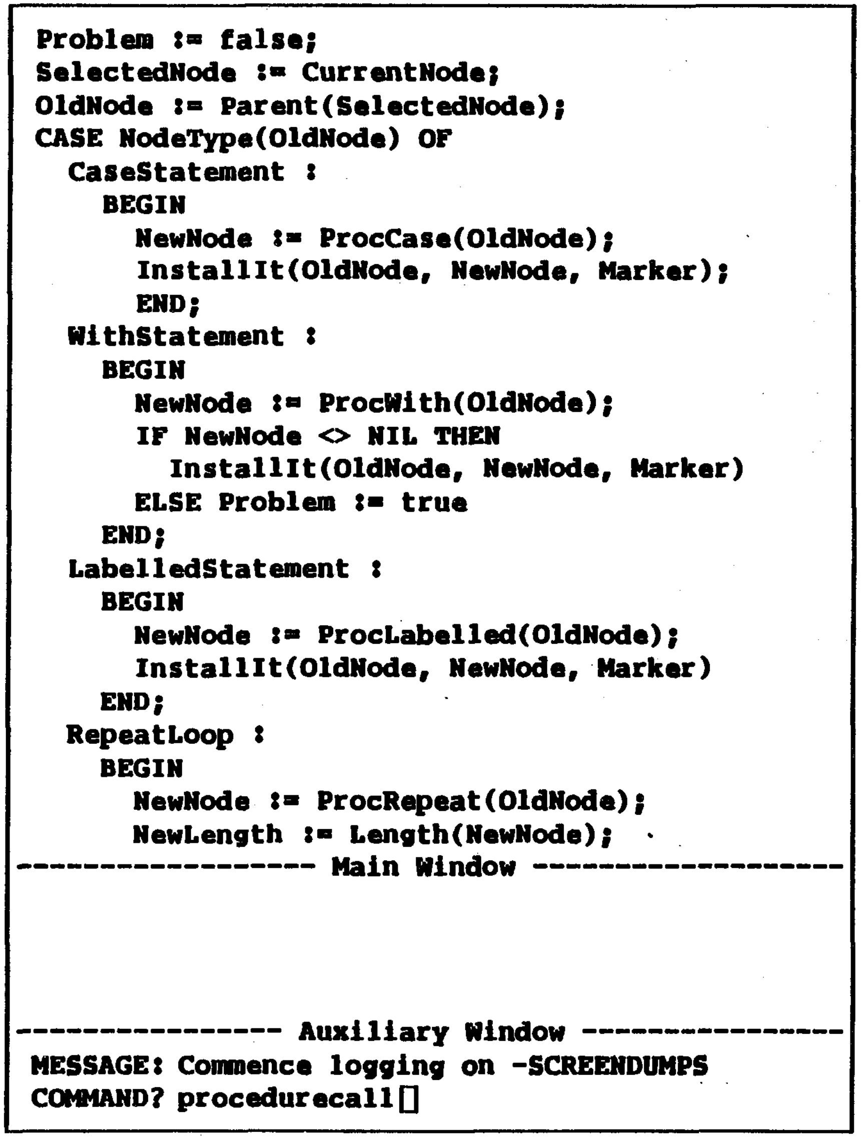
9. Find the end of the case statement (by further scrolling).
10. Delete this range of lines.
11. Delete the semicolon which is about to become superfluous (and erroneous).
12. Type "ELSE Strip1(OldNode, SelectedNode, Maker, Problem);."

Note that the above scenario is simplistic in that no attention is paid to such details as proper indentation. Moreover, it takes advantage of formatting resulting from the abnormally narrow screen (i.e., `IfStatement :` would normally be located on the same line as the first part of the body statement and would therefore have to be deleted one character at a time). This example and the comparison of the different editing approaches leads to some observations:

1. The textual version certainly requires many more key strokes.
2. It is error-prone. (Consider, for example, how easy it would be to forget to delete the semicolon preceding the new `ELSE`.)
3. It requires attention to a great deal of arbitrary detail, whereas the SBE version operates at the level of syntact entities possessing semantic significance.
4. The main component operations of the SBE version emerge naturally from a statement of the problem, whereas there is no such relationship discernable for the textual operations.

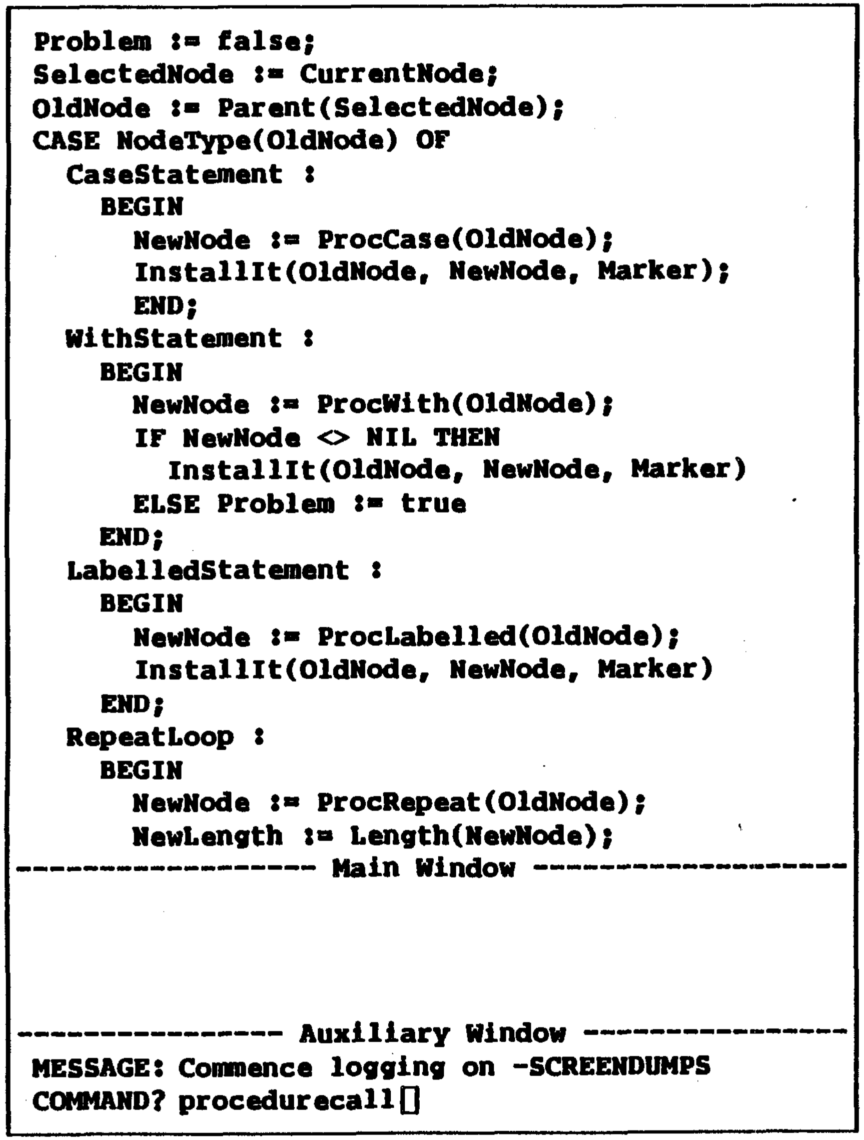


```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
----------------- Main Window -----------------



--------------- Auxiliary Window ---------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? parse stripxmpl statementlist
```



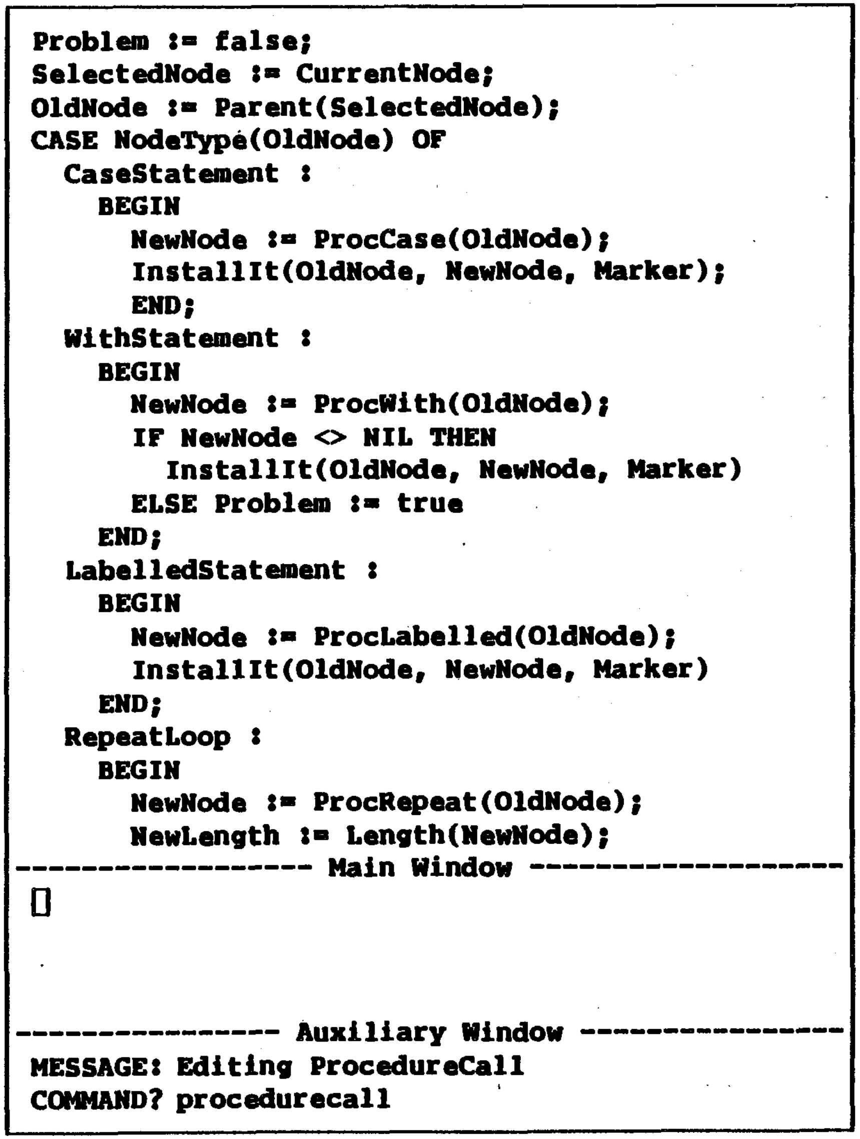
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
----------------- Main Window -----------------



--------------- Auxiliary Window ---------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? procedurecall□
```

6.1a: A node class is entered in the command window.



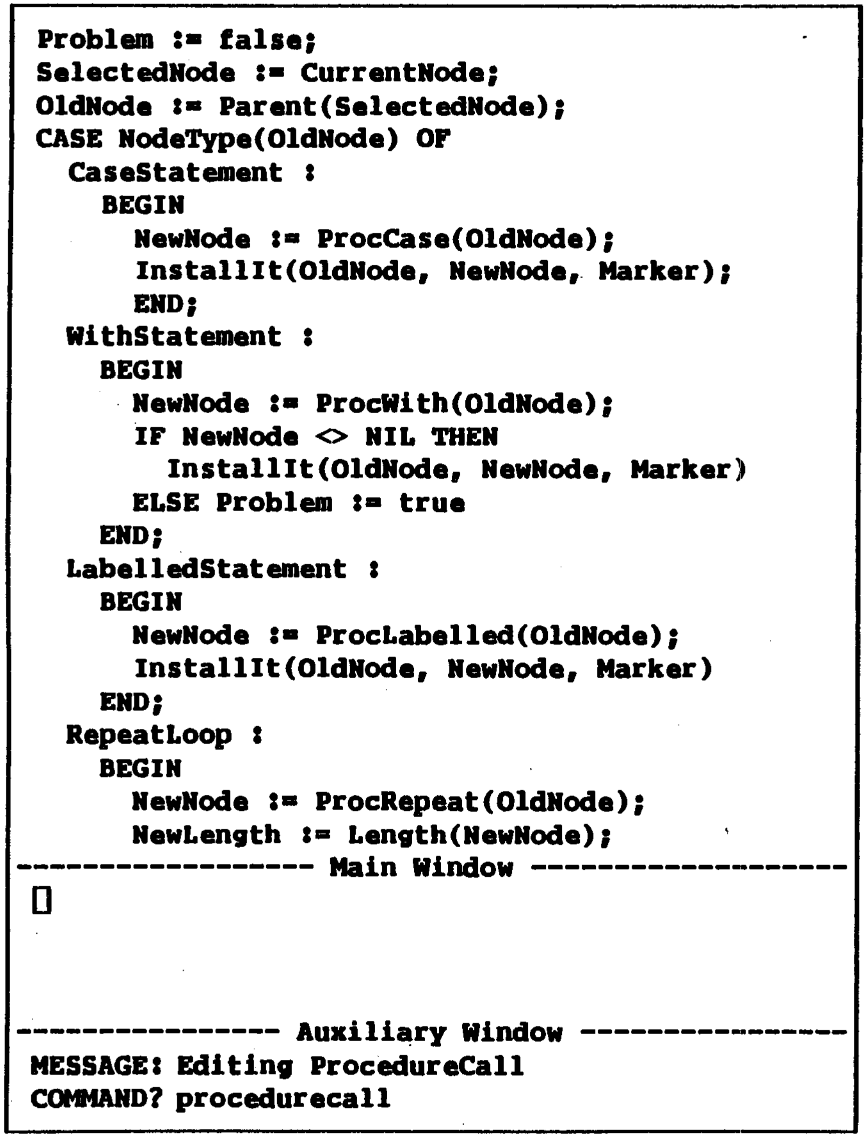
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window ------------------




---------------- Auxiliary Window ----------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? procedurecall[]
```

EDIT NEW

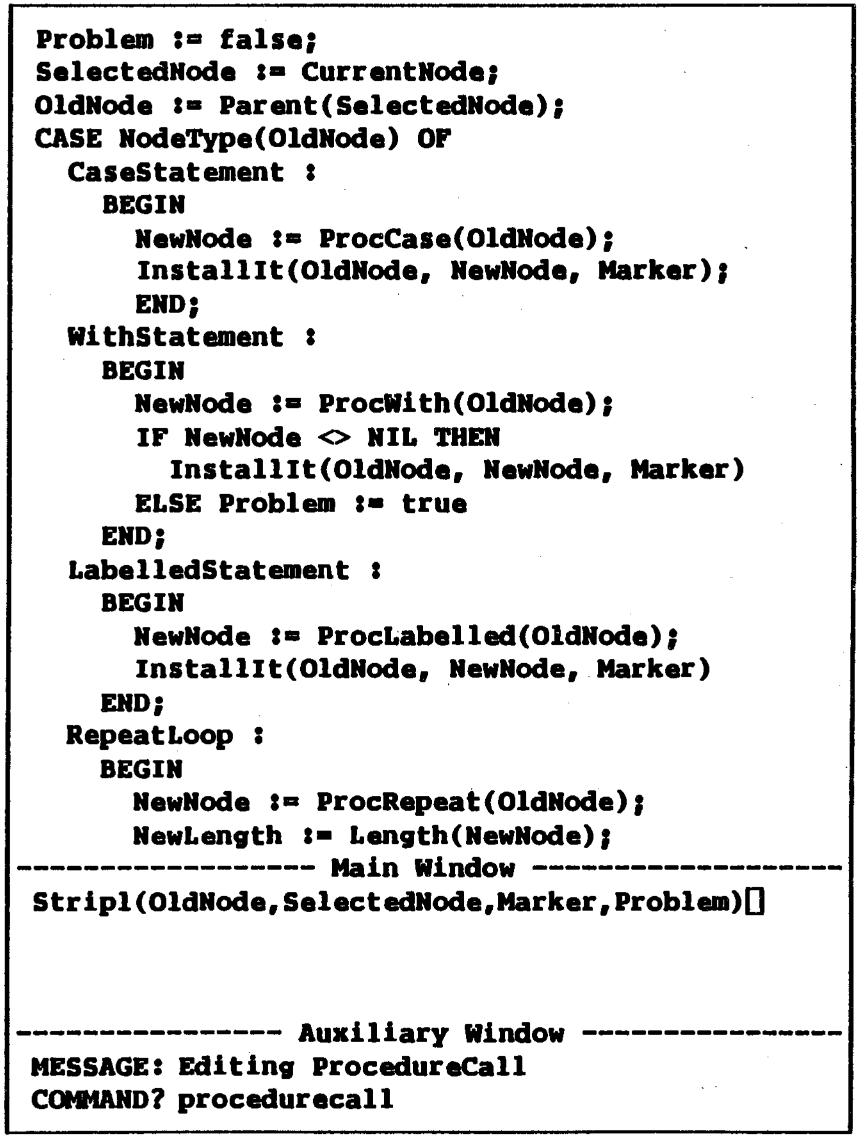


```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window ------------------
[]



---------------- Auxiliary Window ----------------
MESSAGE: Editing ProcedureCall
COMMAND? procedurecall
```

6.1b: The editor is called for production of a new node.

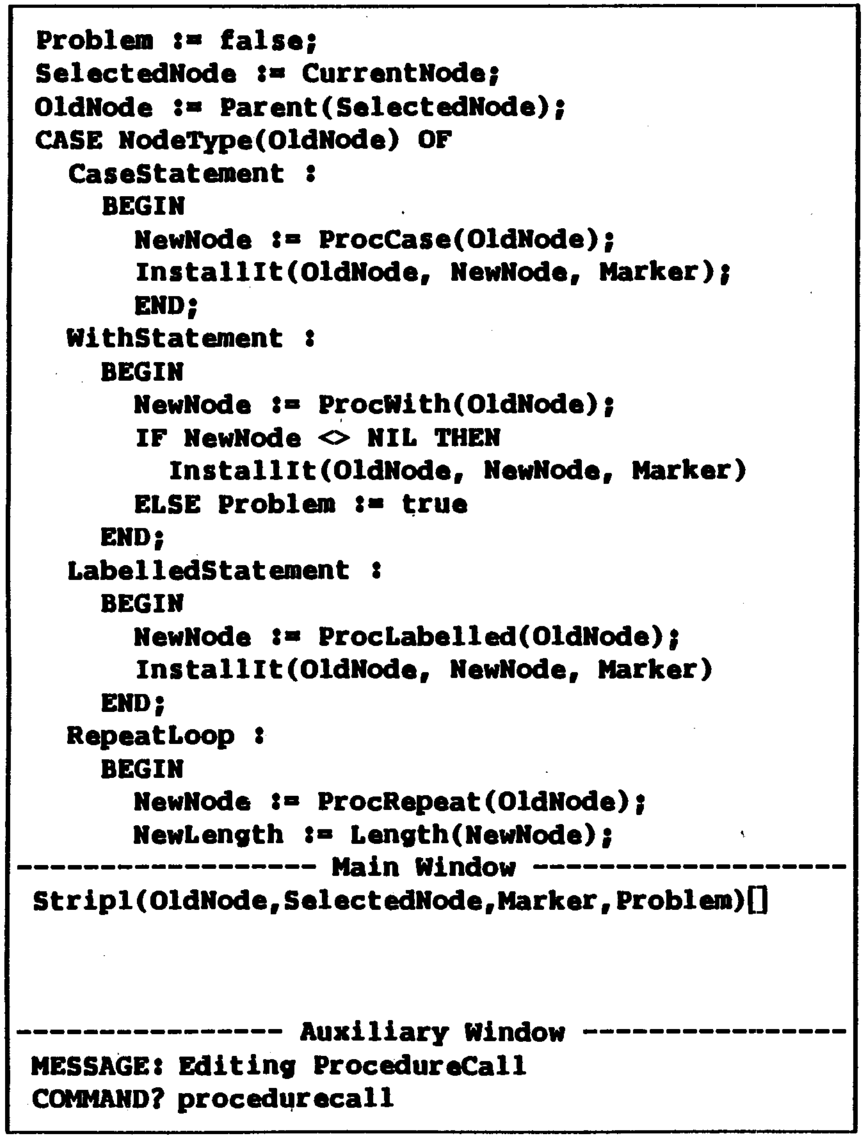


```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window -------------------
□



---------------- Auxiliary Window ----------------
MESSAGE: Editing ProcedureCall
COMMAND? procedurecall
```

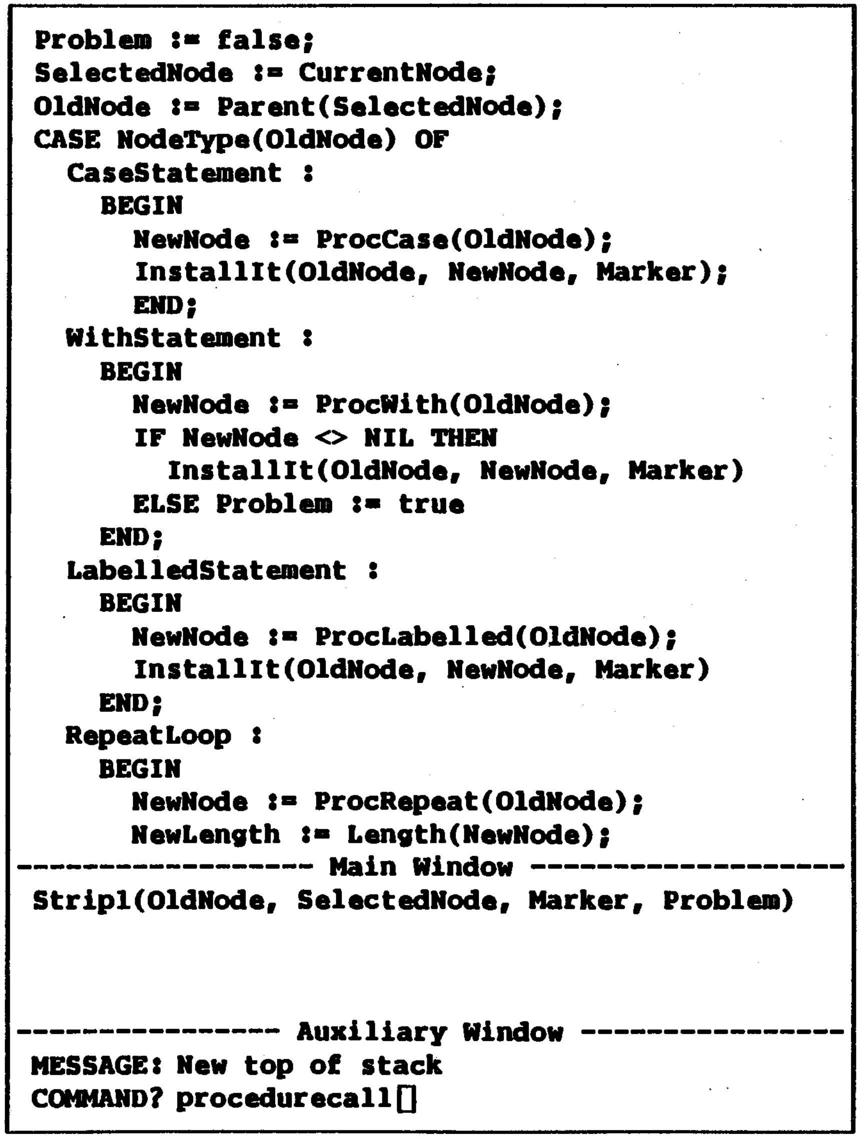


```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window -------------------
Stripl(OldNode,SelectedNode,Marker,Problem)□



---------------- Auxiliary Window ----------------
MESSAGE: Editing ProcedureCall
COMMAND? procedurecall
```

6.1c: The textual representation of the new node is entered.



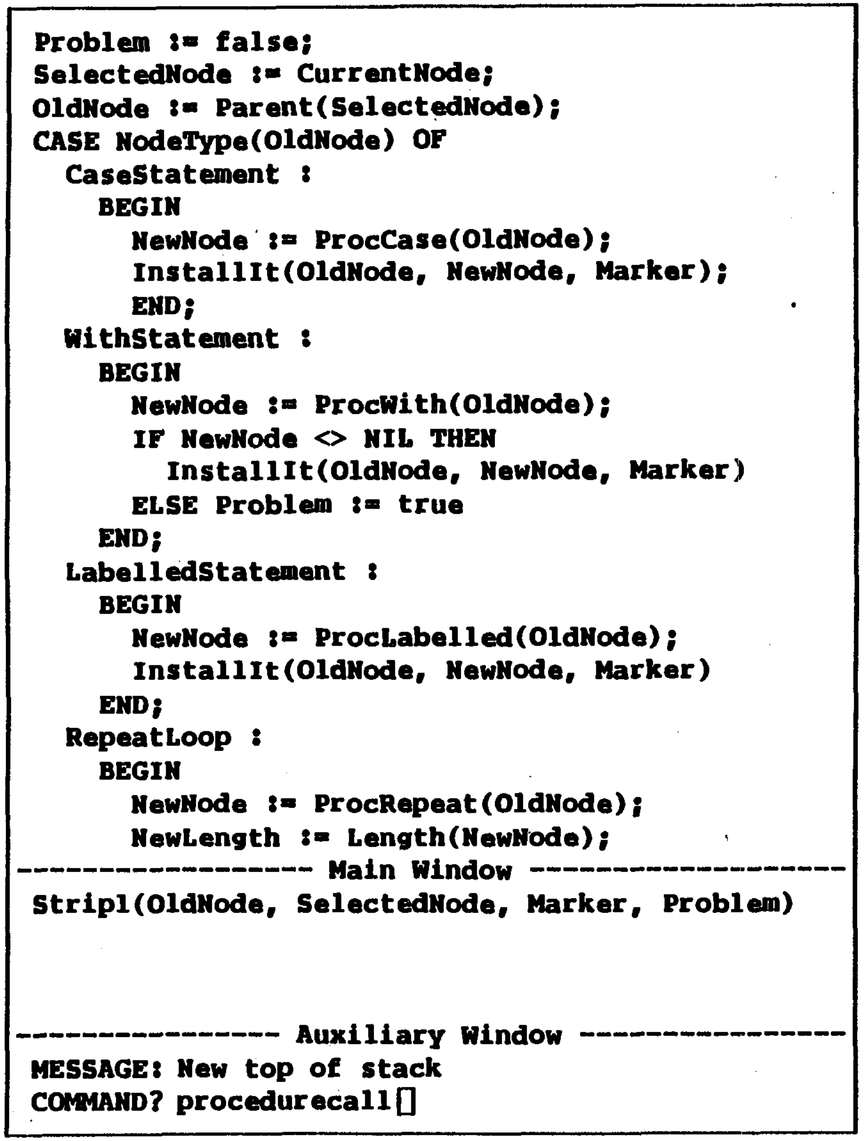
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window -------------------
Stripl(OldNode,SelectedNode,Marker,Problem)□



---------------- Auxiliary Window ----------------
MESSAGE: Editing ProcedureCall
COMMAND? procedurecall
```



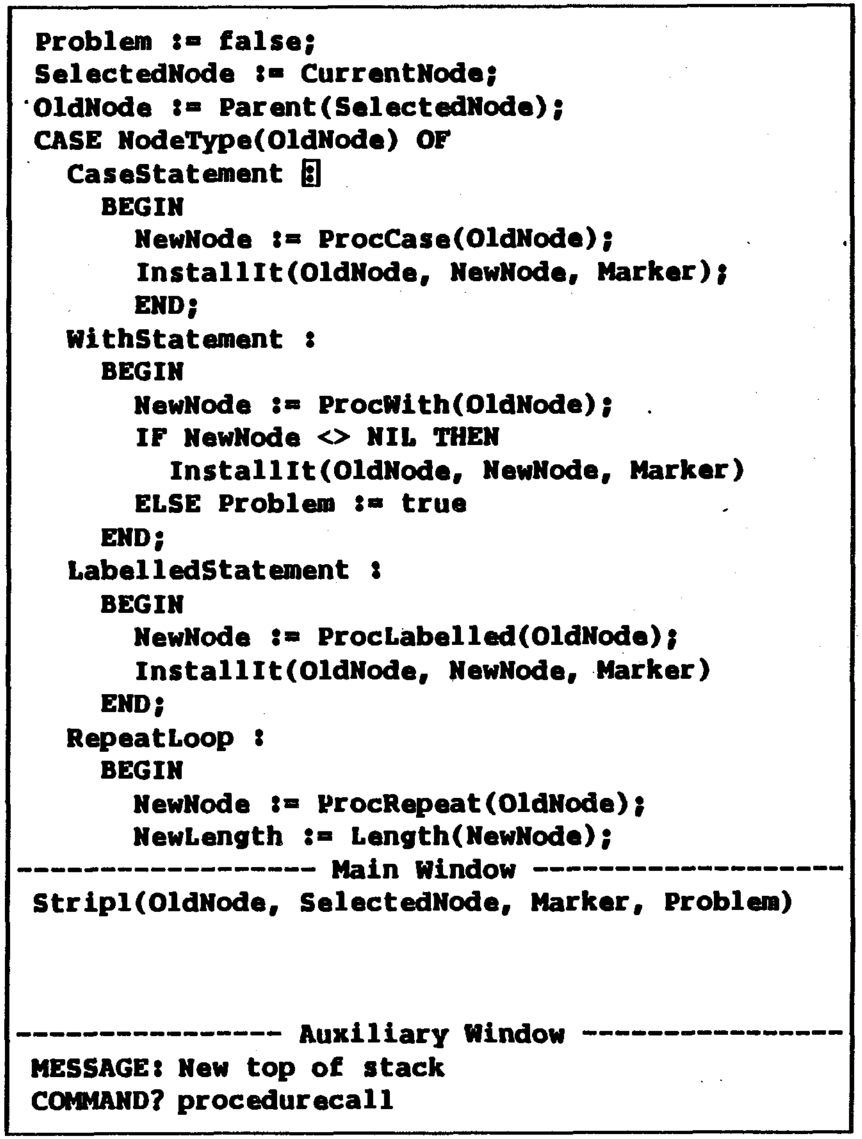
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window -------------------
Stripl(OldNode, SelectedNode, Marker, Problem)



---------------- Auxiliary Window ----------------
MESSAGE: New top of stack
COMMAND? procedurecall□
```

EDIT COMPLETE

6.1d: Parsing yields a node, which goes on the node stack.

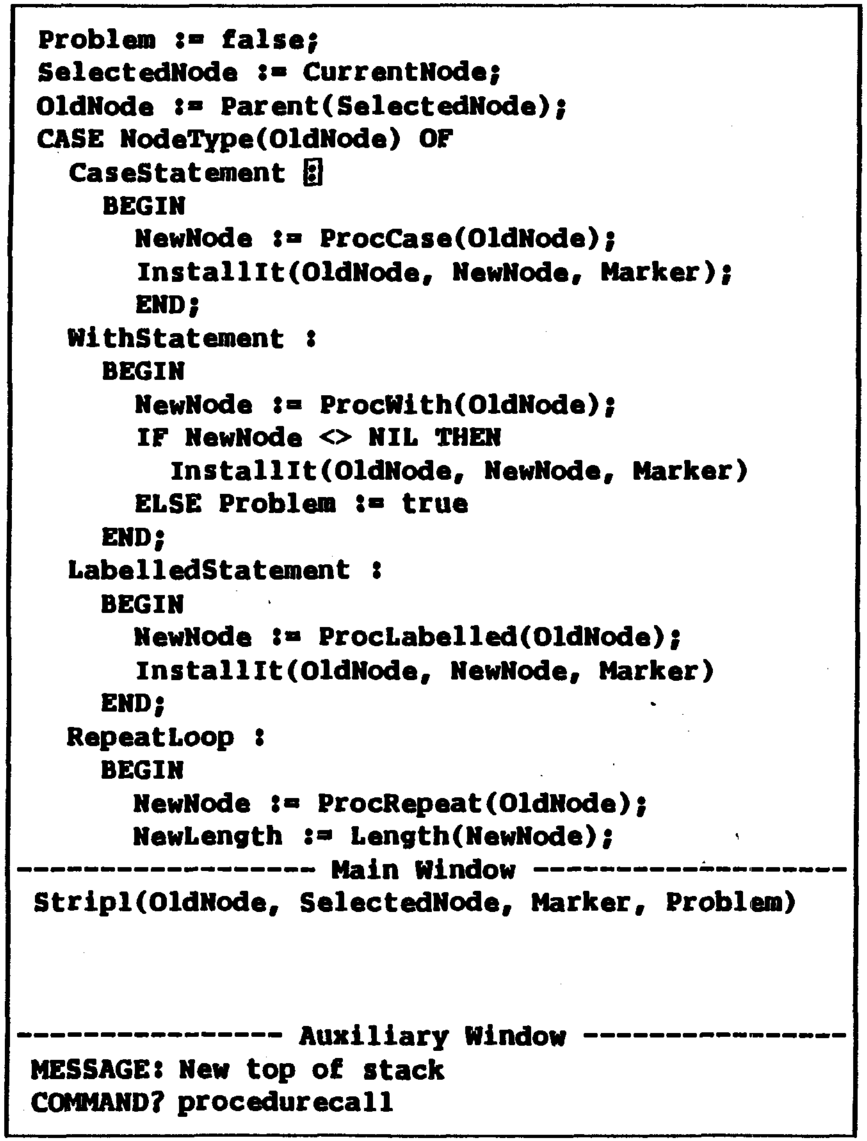


```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
----------------- Main Window -------------------
Stripl(OldNode, SelectedNode, Marker, Problem)



---------------- Auxiliary Window ---------------
MESSAGE: New top of stack
COMMAND? procedurecall□
```

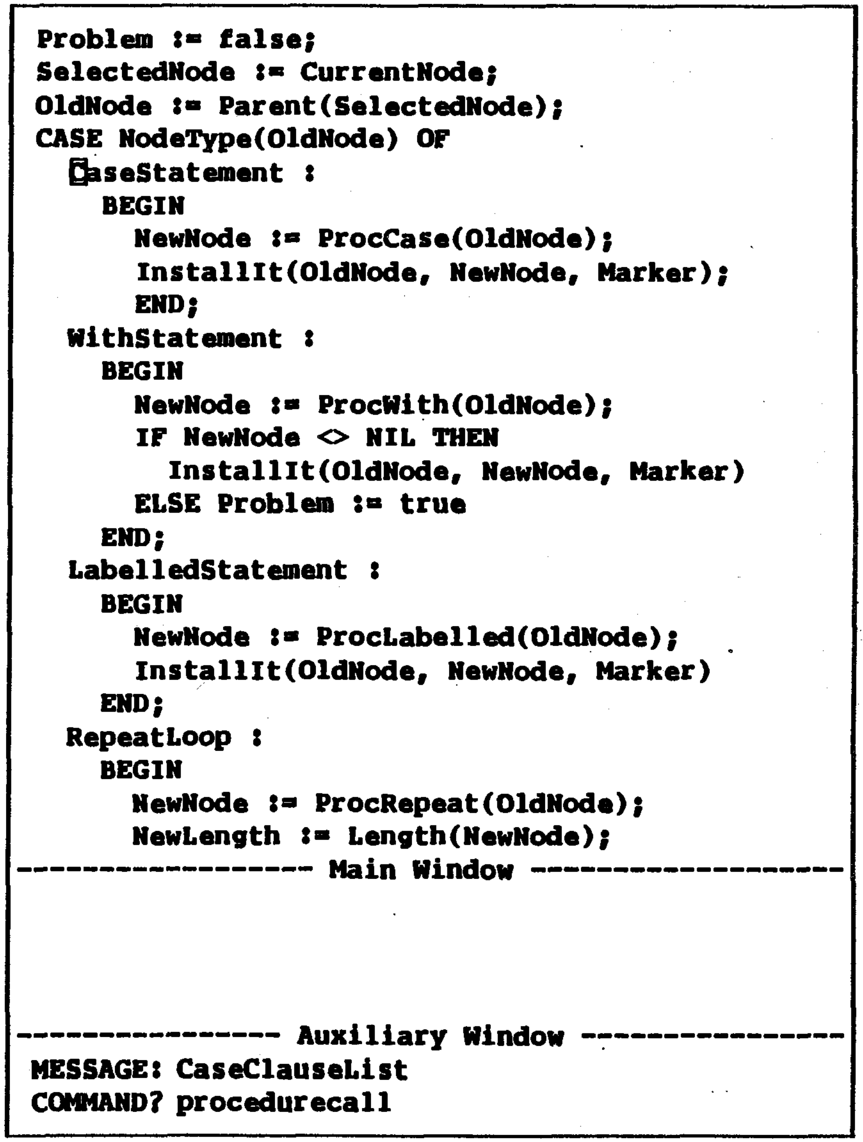


```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
----------------- Main Window -------------------
Stripl(OldNode, SelectedNode, Marker, Problem)



---------------- Auxiliary Window ---------------
MESSAGE: New top of stack
COMMAND? procedurecall
```

6.1e: The screen cursor is returned to the working area.



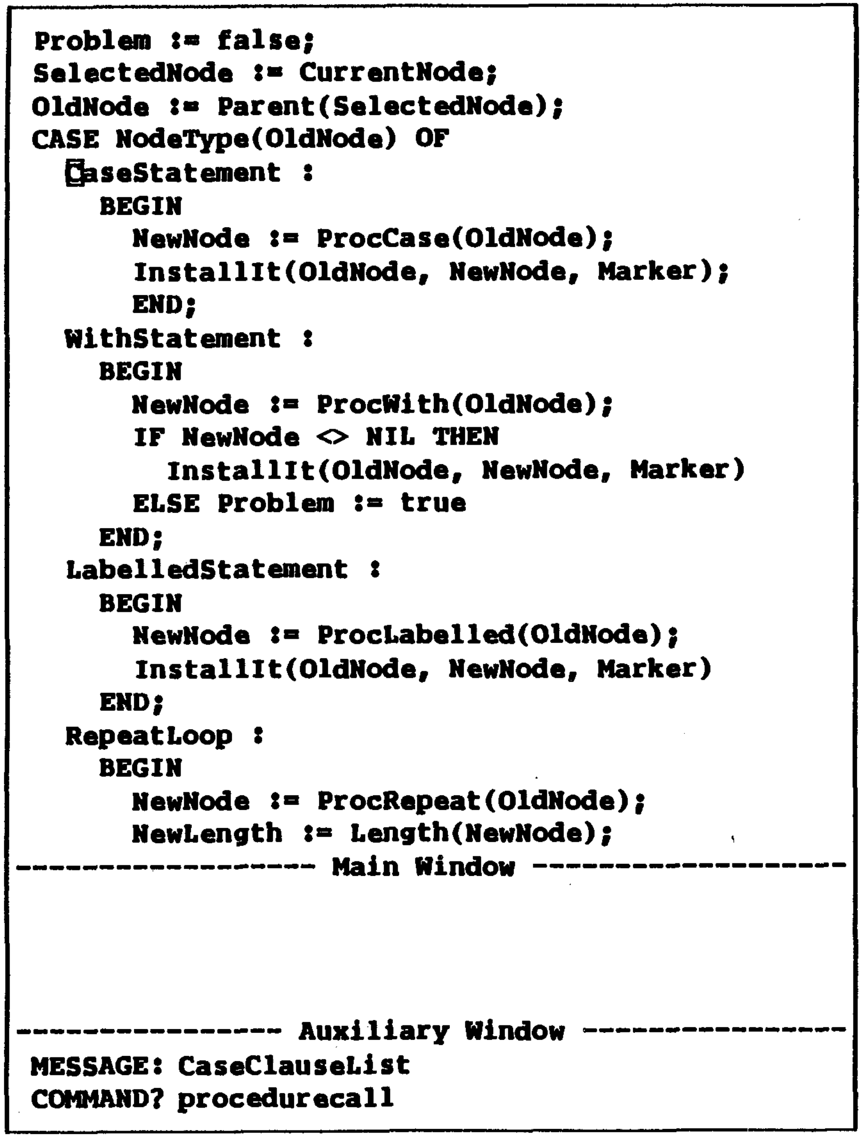
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window -------------------
Stripl(OldNode, SelectedNode, Marker, Problem)



---------------- Auxiliary Window ----------------
MESSAGE: New top of stack
COMMAND? procedurecall
```



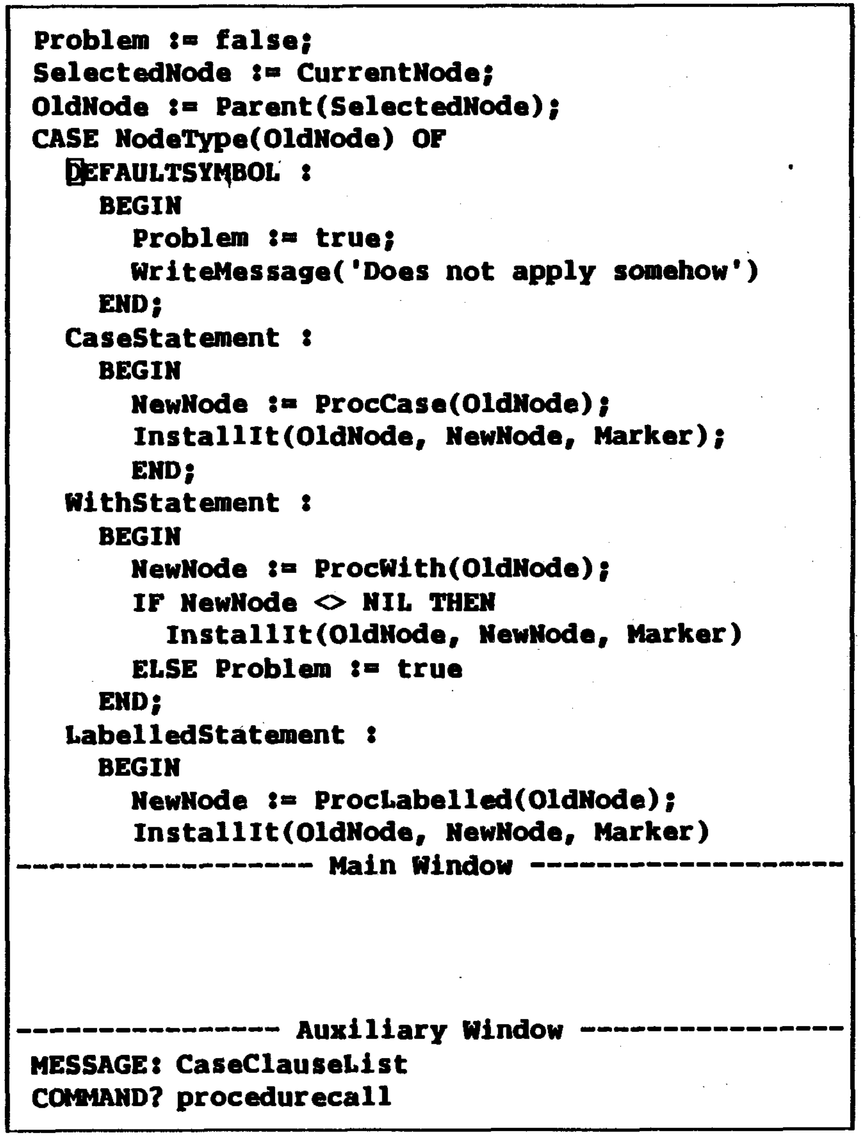
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window -------------------




---------------- Auxiliary Window ----------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```

CURSOR UP

6.1f: The syntactic cursor selects the case clause list.



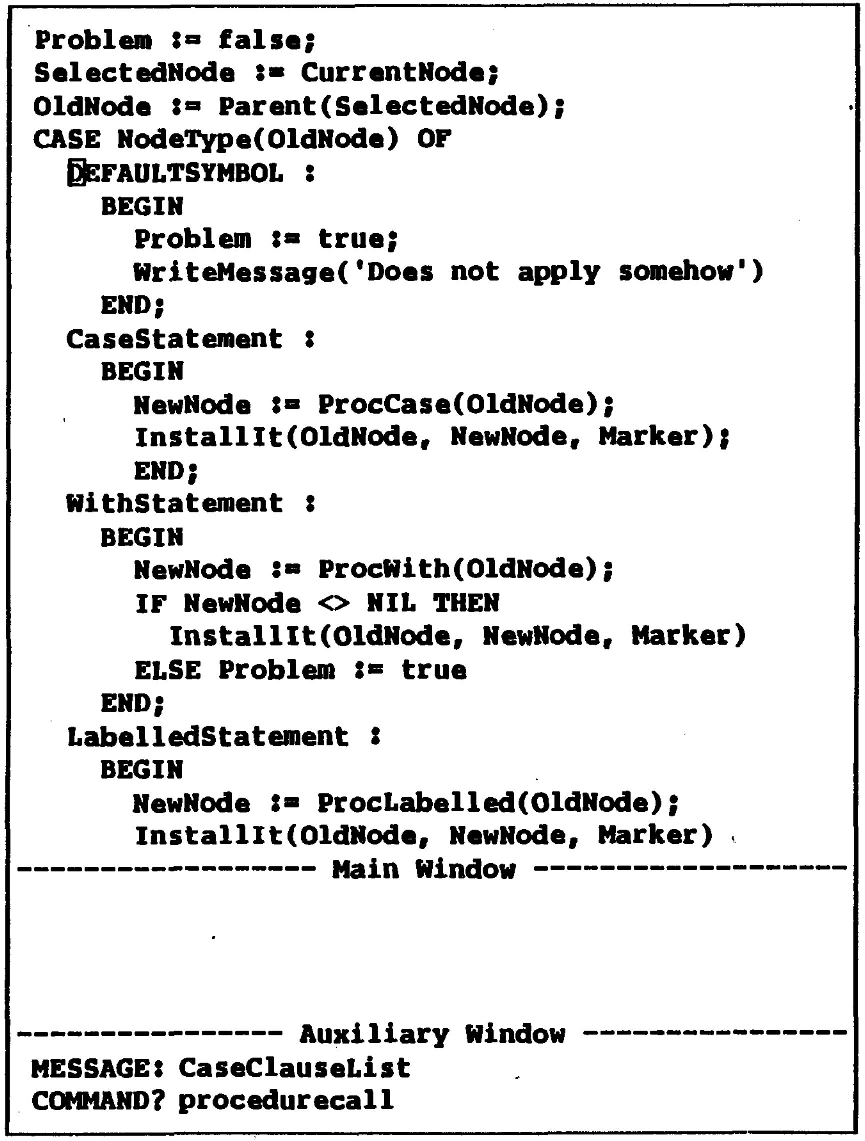
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
    END;
  RepeatLoop :
    BEGIN
      NewNode := ProcRepeat(OldNode);
      NewLength := Length(NewNode);
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```



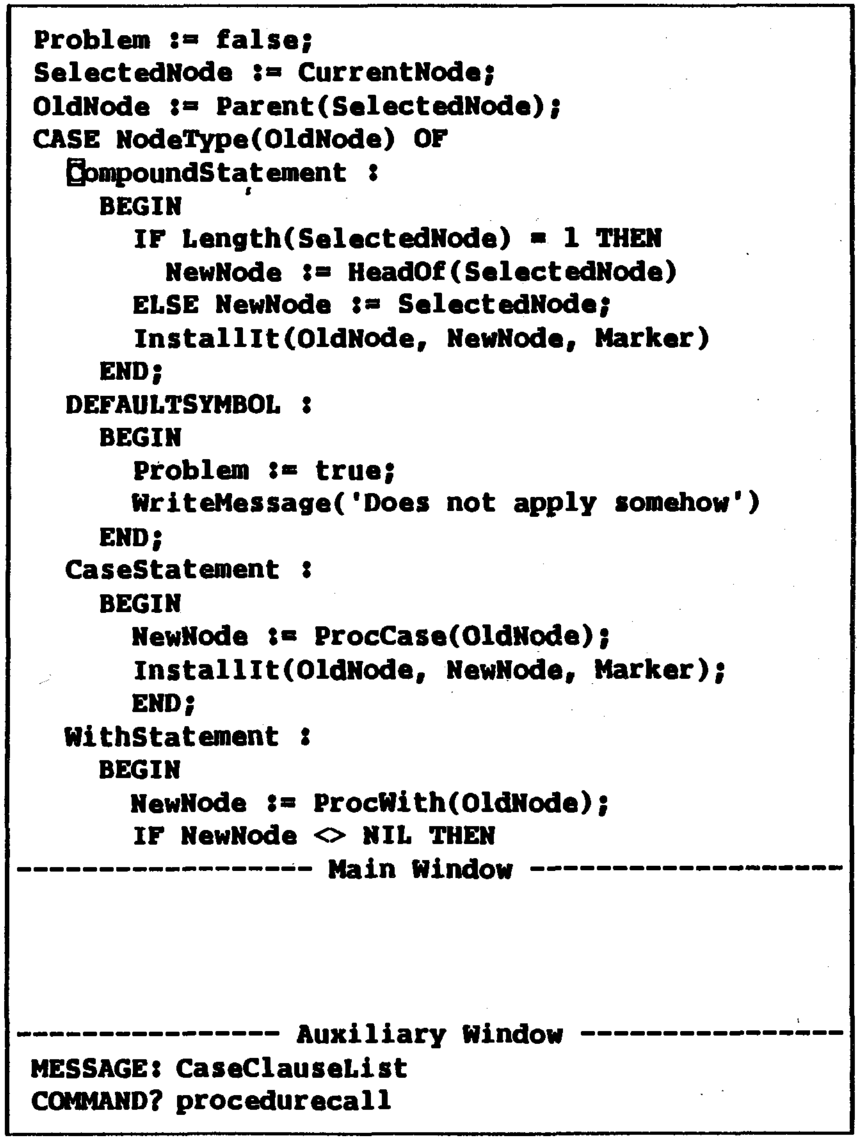
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  DEFAULTSYMBOL :
    BEGIN
      Problem := true;
      WriteMessage('Does not apply somehow')
    END;
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```

ROTATE

6.1g: The list as a whole is rotated.



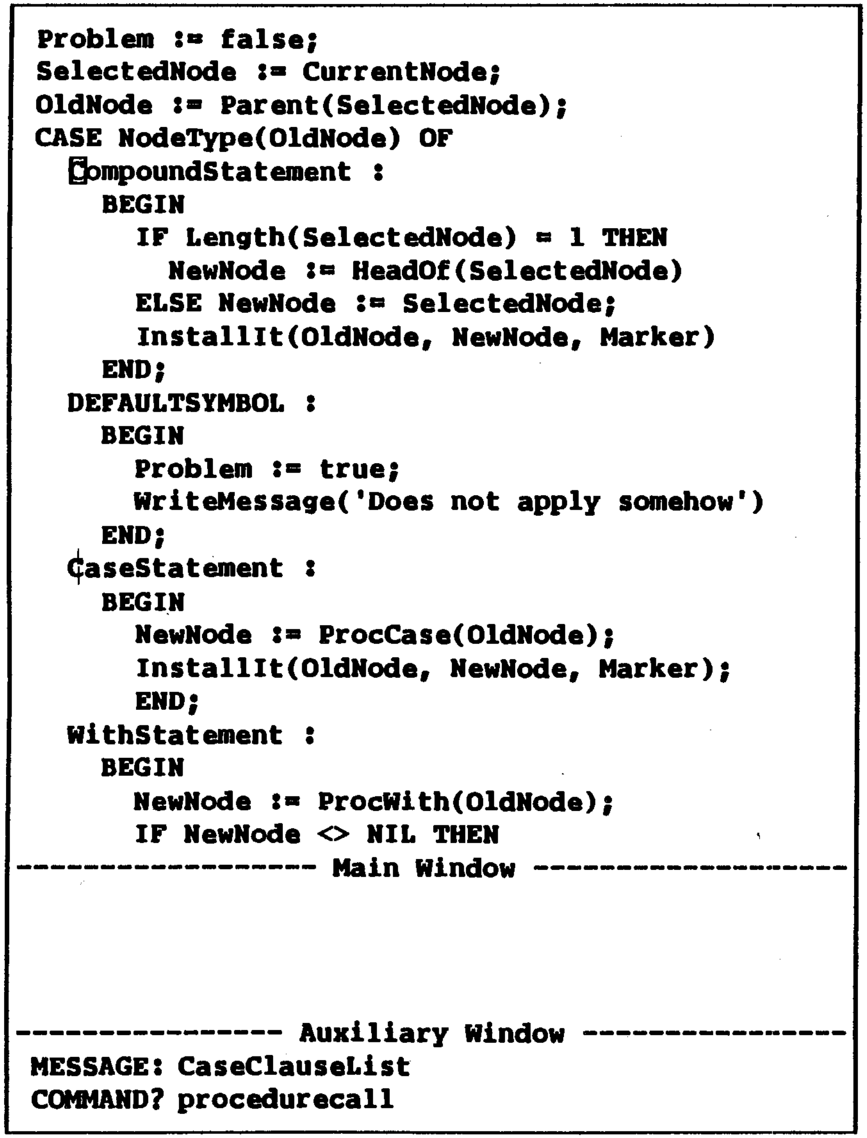
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  DEFAULTSYMBOL :
    BEGIN
      Problem := true;
      WriteMessage('Does not apply somehow')
    END;
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
        InstallIt(OldNode, NewNode, Marker)
      ELSE Problem := true
    END;
  LabelledStatement :
    BEGIN
      NewNode := ProcLabelled(OldNode);
      InstallIt(OldNode, NewNode, Marker)
----------------- Main Window ------------------



--------------- Auxiliary Window ---------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```



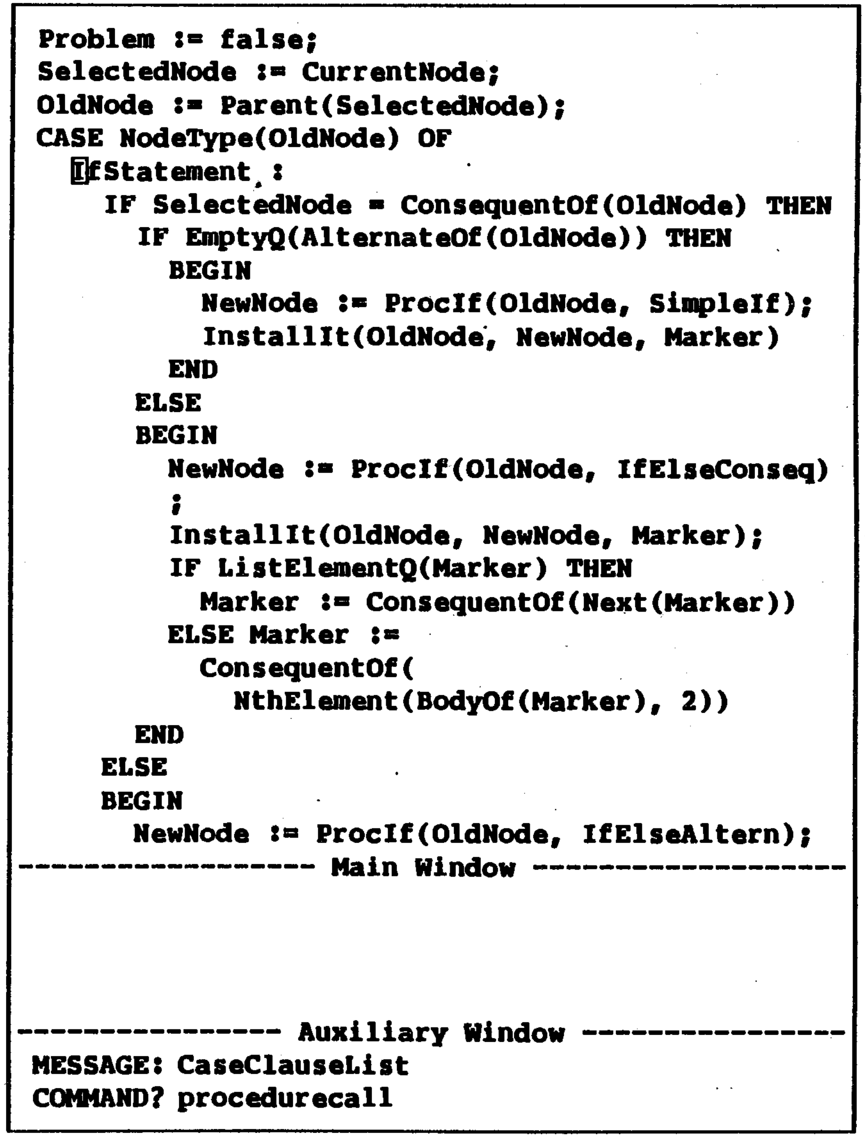
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CompoundStatement :
    BEGIN
      IF Length(SelectedNode) = 1 THEN
        NewNode := HeadOf(SelectedNode)
      ELSE NewNode := SelectedNode;
      InstallIt(OldNode, NewNode, Marker)
    END;
  DEFAULTSYMBOL :
    BEGIN
      Problem := true;
      WriteMessage('Does not apply somehow')
    END;
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
----------------- Main Window ------------------



--------------- Auxiliary Window ---------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```

ROTATE

6.1h: The list is rotated again.



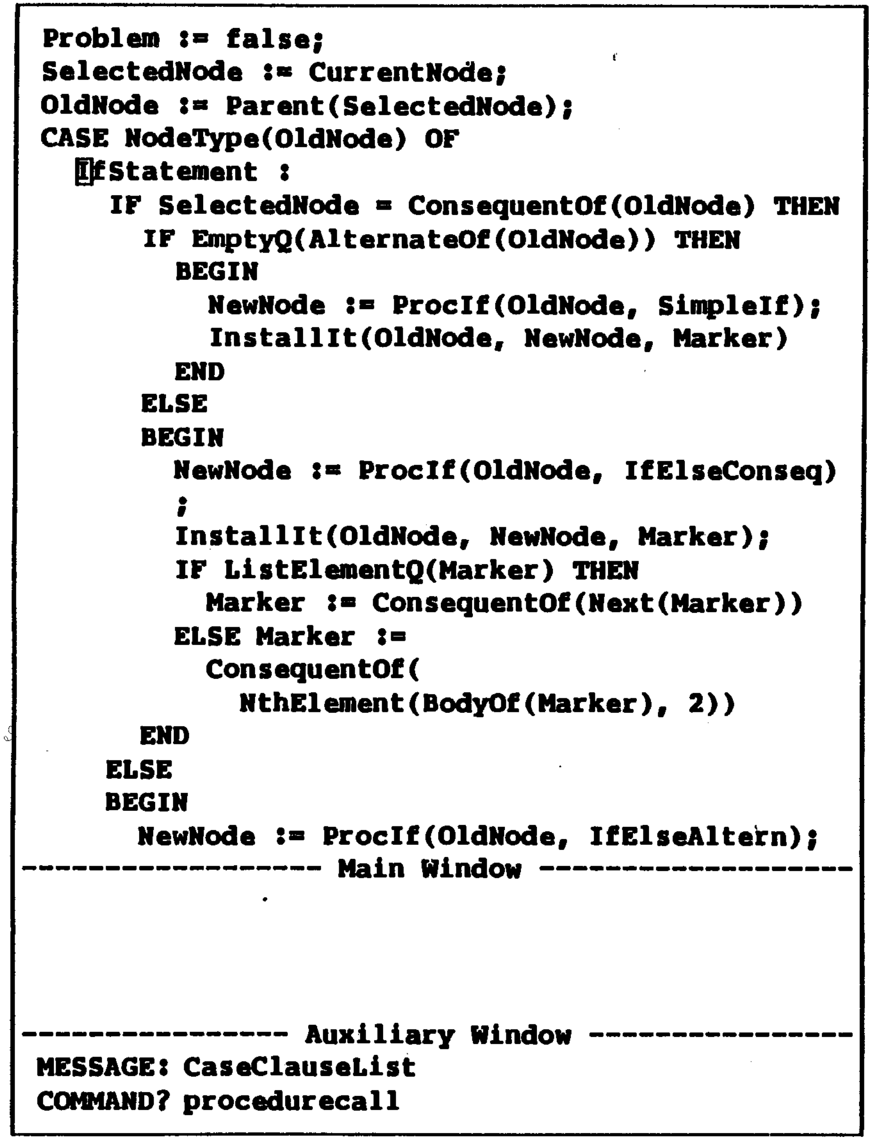
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  CompoundStatement :
    BEGIN
      IF Length(SelectedNode) = 1 THEN
        NewNode := HeadOf(SelectedNode)
      ELSE NewNode := SelectedNode;
      InstallIt(OldNode, NewNode, Marker)
    END;
  DEFAULTSYMBOL :
    BEGIN
      Problem := true;
      WriteMessage('Does not apply somehow')
    END;
  CaseStatement :
    BEGIN
      NewNode := ProcCase(OldNode);
      InstallIt(OldNode, NewNode, Marker);
      END;
  WithStatement :
    BEGIN
      NewNode := ProcWith(OldNode);
      IF NewNode <> NIL THEN
----------------- Main Window ------------------



--------------- Auxiliary Window ---------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```



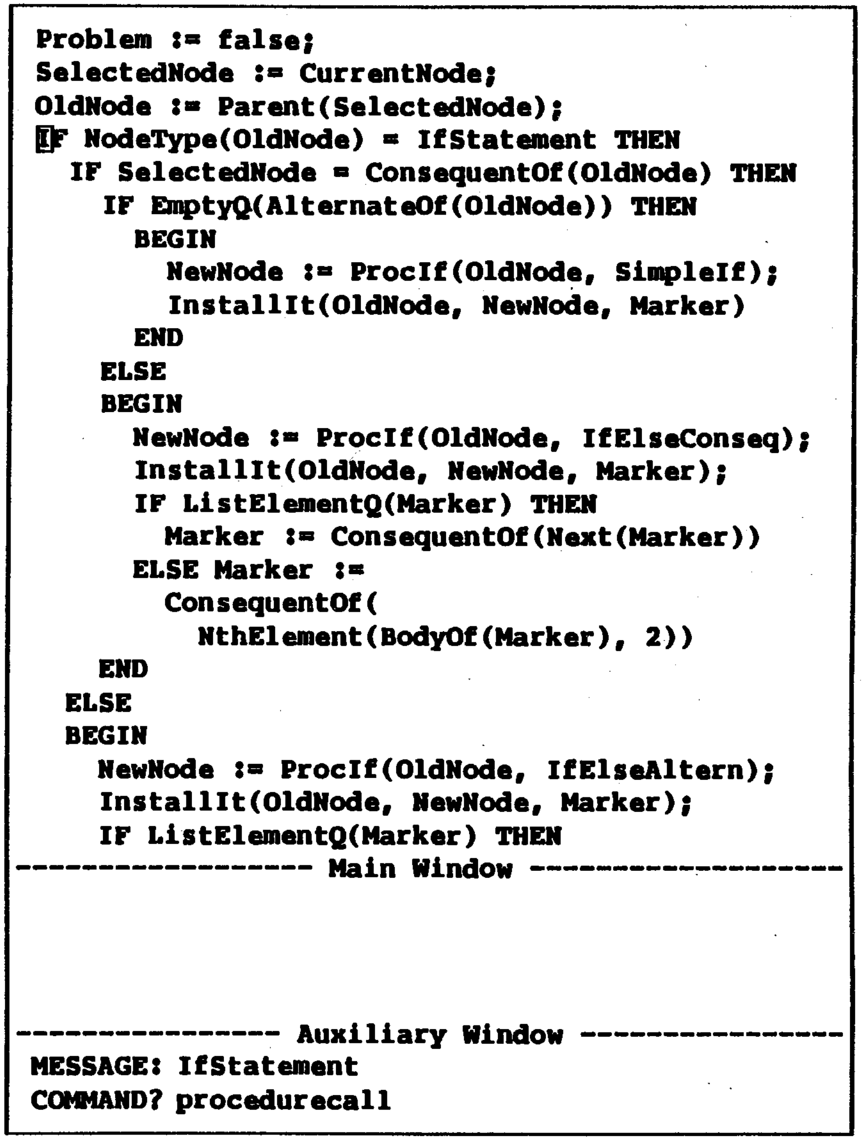
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  IfStatement :
    IF SelectedNode = ConsequentOf(OldNode) THEN
      IF EmptyQ(AlternateOf(OldNode)) THEN
        BEGIN
          NewNode := ProcIf(OldNode, SimpleIf);
          InstallIt(OldNode, NewNode, Marker)
        END
      ELSE
      BEGIN
        NewNode := ProcIf(OldNode, IfElseConseq)
        ;
        InstallIt(OldNode, NewNode, Marker);
        IF ListElementQ(Marker) THEN
          Marker := ConsequentOf(Next(Marker))
        ELSE Marker :=
          ConsequentOf(
            NthElement(BodyOf(Marker), 2))
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseAltern);
----------------- Main Window ------------------



--------------- Auxiliary Window ---------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```

ROTATE

6.1i: Yet another rotation leaves the elements correctly positioned.



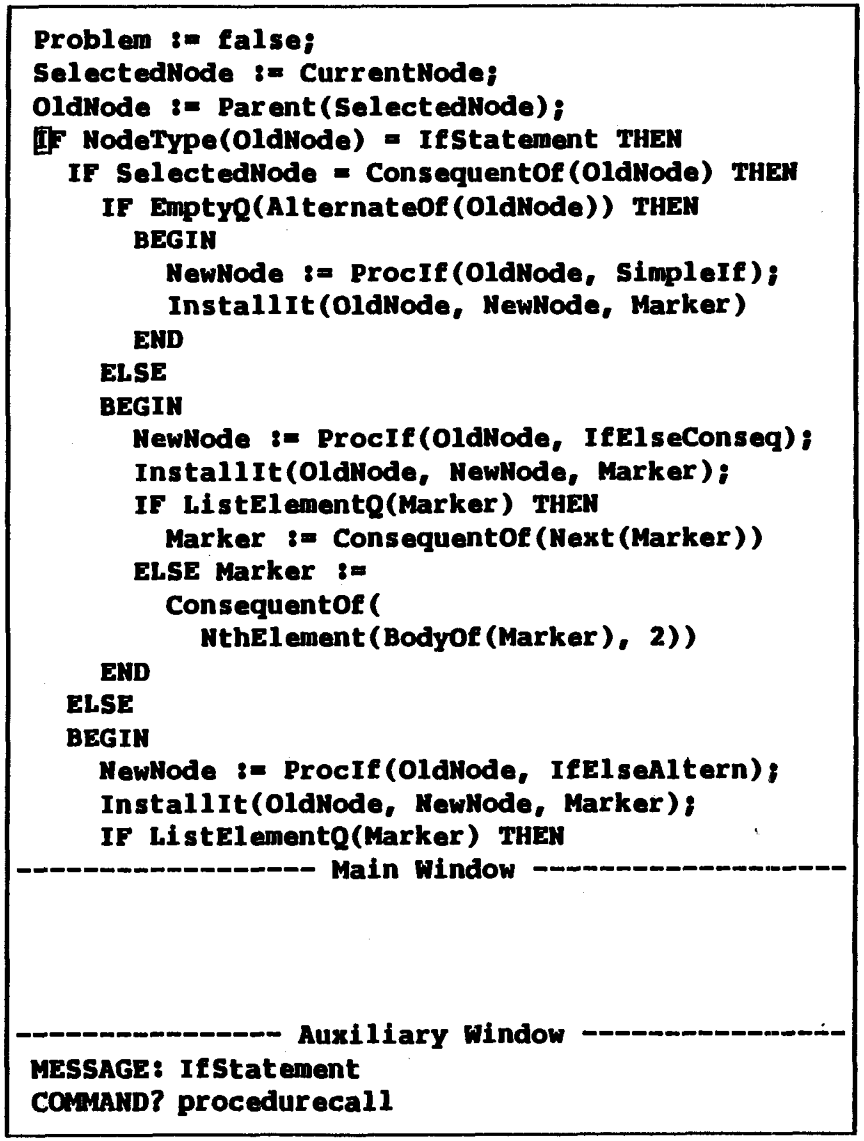
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
CASE NodeType(OldNode) OF
  IfStatement :
    IF SelectedNode = ConsequentOf(OldNode) THEN
      IF EmptyQ(AlternateOf(OldNode)) THEN
        BEGIN
          NewNode := ProcIf(OldNode, SimpleIf);
          InstallIt(OldNode, NewNode, Marker)
        END
      ELSE
      BEGIN
        NewNode := ProcIf(OldNode, IfElseConseq)
        ;
        InstallIt(OldNode, NewNode, Marker);
        IF ListElementQ(Marker) THEN
          Marker := ConsequentOf(Next(Marker))
        ELSE Marker :=
          ConsequentOf(
            NthElement(BodyOf(Marker), 2))
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseAltern);
------------------ Main Window ------------------



----------------- Auxiliary Window ----------------
MESSAGE: CaseClauseList
COMMAND? procedurecall
```



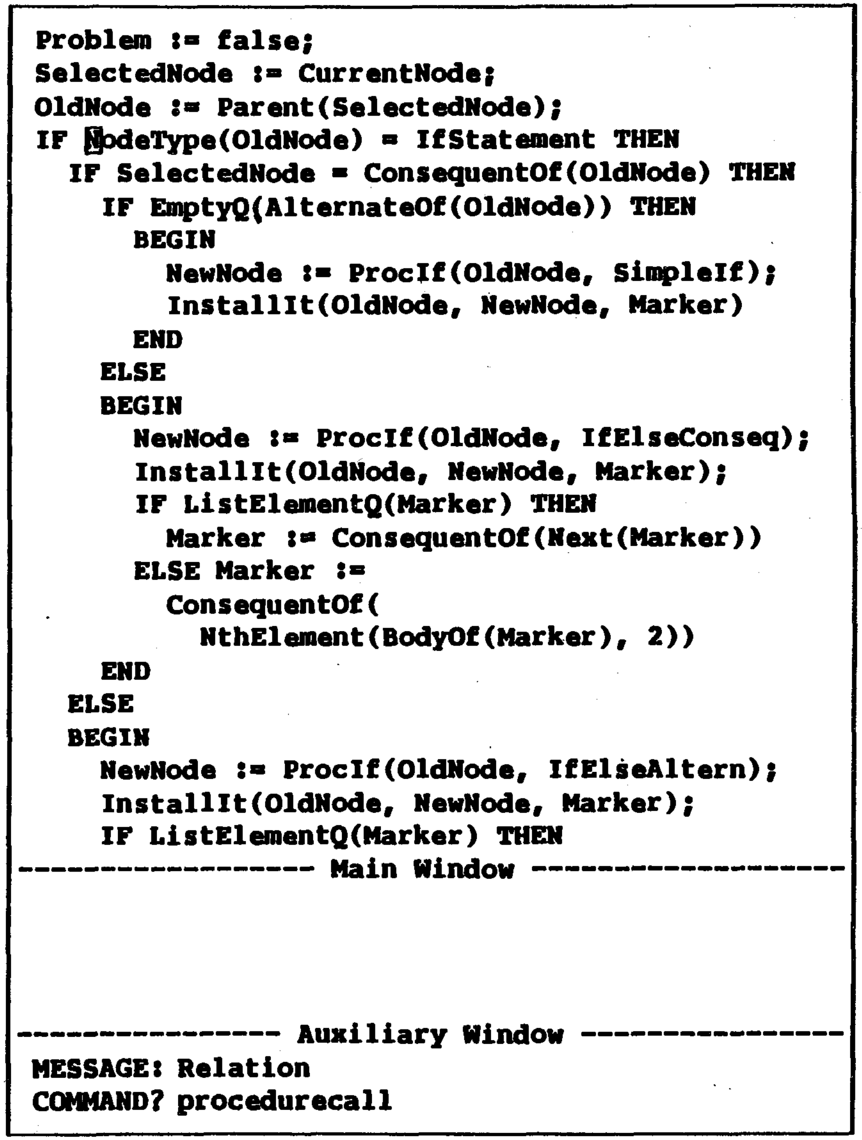
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
IF NodeType(OldNode) = IfStatement THEN
  IF SelectedNode = ConsequentOf(OldNode) THEN
    IF EmptyQ(AlternateOf(OldNode)) THEN
      BEGIN
        NewNode := ProcIf(OldNode, SimpleIf);
        InstallIt(OldNode, NewNode, Marker)
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseConseq);
      InstallIt(OldNode, NewNode, Marker);
      IF ListElementQ(Marker) THEN
        Marker := ConsequentOf(Next(Marker))
      ELSE Marker :=
        ConsequentOf(
          NthElement(BodyOf(Marker), 2))
    END
  ELSE
  BEGIN
    NewNode := ProcIf(OldNode, IfElseAltern);
    InstallIt(OldNode, NewNode, Marker);
    IF ListElementQ(Marker) THEN
------------------ Main Window ------------------



----------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? procedurecall
```

INLINE

6.1j: The case statement is replaced by its "inline coding."



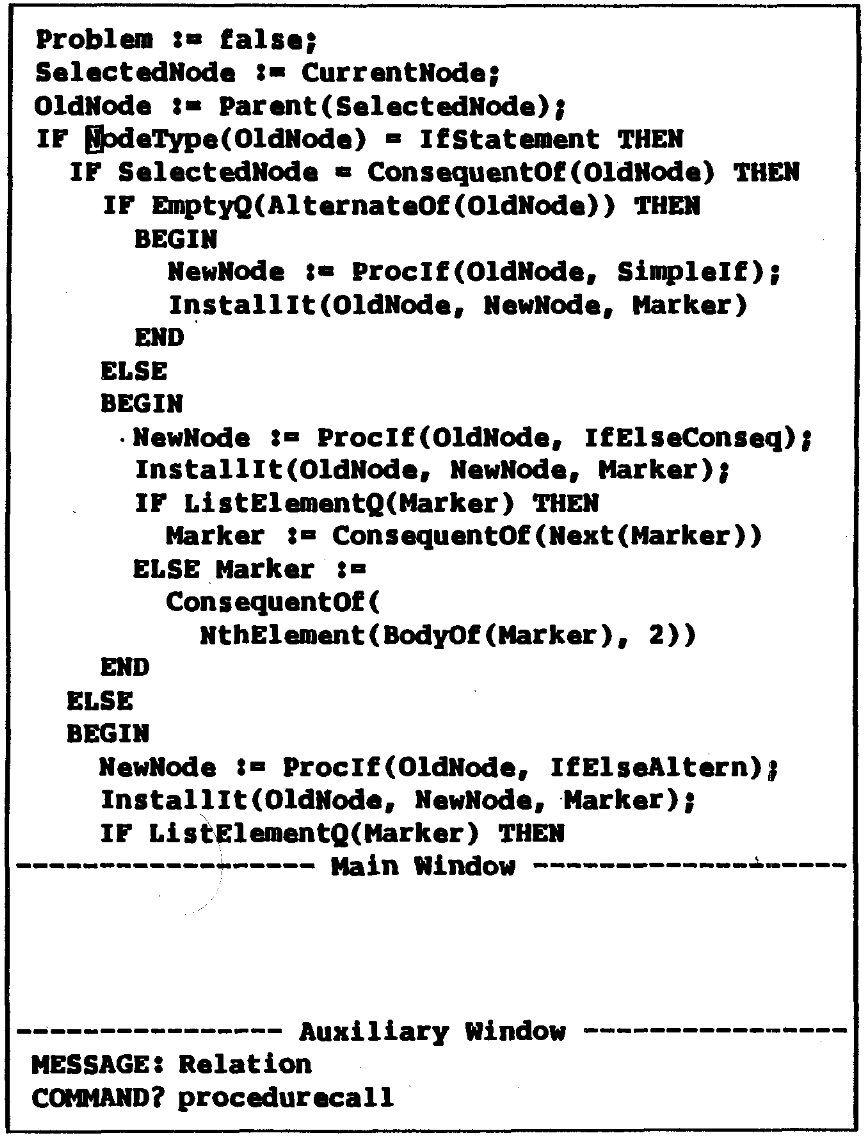
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
IF NodeType(OldNode) = IfStatement THEN
  IF SelectedNode = ConsequentOf(OldNode) THEN
    IF EmptyQ(AlternateOf(OldNode)) THEN
      BEGIN
        NewNode := ProcIf(OldNode, SimpleIf);
        InstallIt(OldNode, NewNode, Marker)
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseConseq);
      InstallIt(OldNode, NewNode, Marker);
      IF ListElementQ(Marker) THEN
        Marker := ConsequentOf(Next(Marker))
      ELSE Marker :=
        ConsequentOf(
          NthElement(BodyOf(Marker), 2))
    END
  ELSE
  BEGIN
    NewNode := ProcIf(OldNode, IfElseAltern);
    InstallIt(OldNode, NewNode, Marker);
    IF ListElementQ(Marker) THEN
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? procedurecall
```



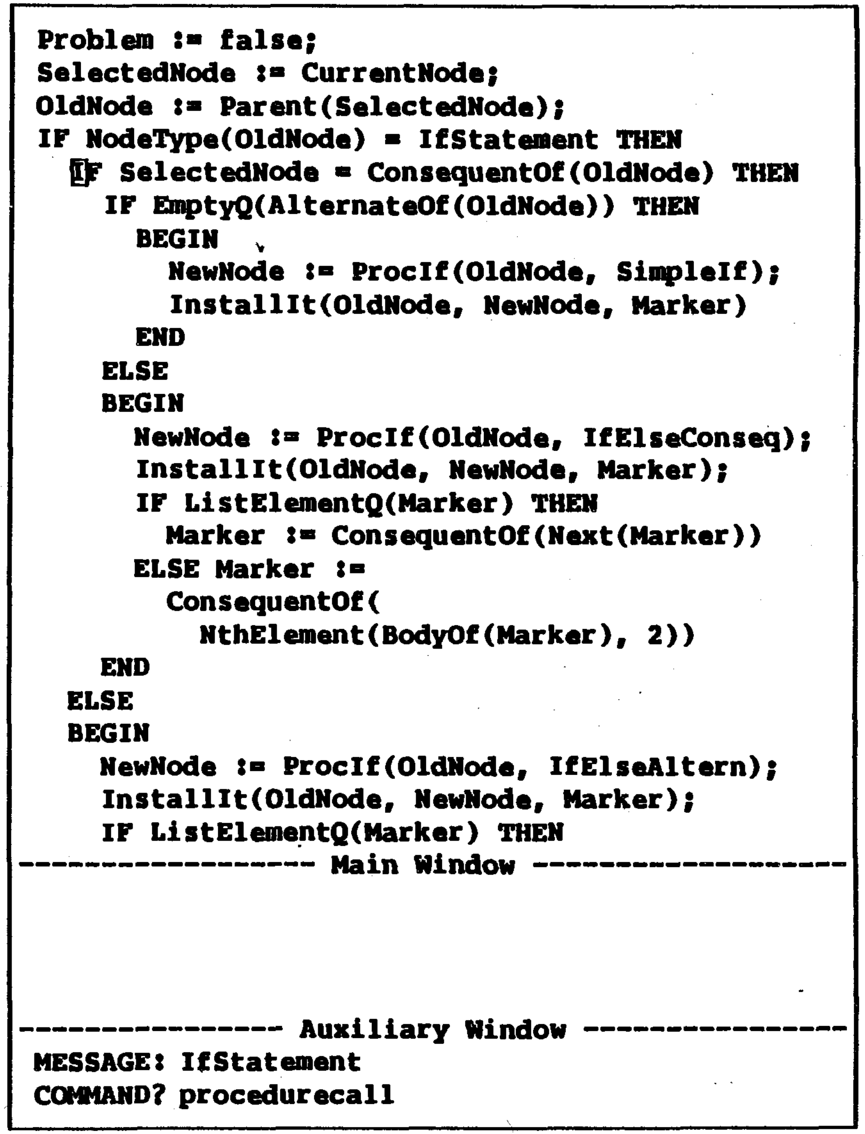
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
IF NodeType(OldNode) = IfStatement THEN
  IF SelectedNode = ConsequentOf(OldNode) THEN
    IF EmptyQ(AlternateOf(OldNode)) THEN
      BEGIN
        NewNode := ProcIf(OldNode, SimpleIf);
        InstallIt(OldNode, NewNode, Marker)
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseConseq);
      InstallIt(OldNode, NewNode, Marker);
      IF ListElementQ(Marker) THEN
        Marker := ConsequentOf(Next(Marker))
      ELSE Marker :=
        ConsequentOf(
          NthElement(BodyOf(Marker), 2))
    END
  ELSE
  BEGIN
    NewNode := ProcIf(OldNode, IfElseAltern);
    InstallIt(OldNode, NewNode, Marker);
    IF ListElementQ(Marker) THEN
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: Relation
COMMAND? procedurecall
```

CURSOR DOWN

6.1k: The cursor is moved to the if's first component (predicate).



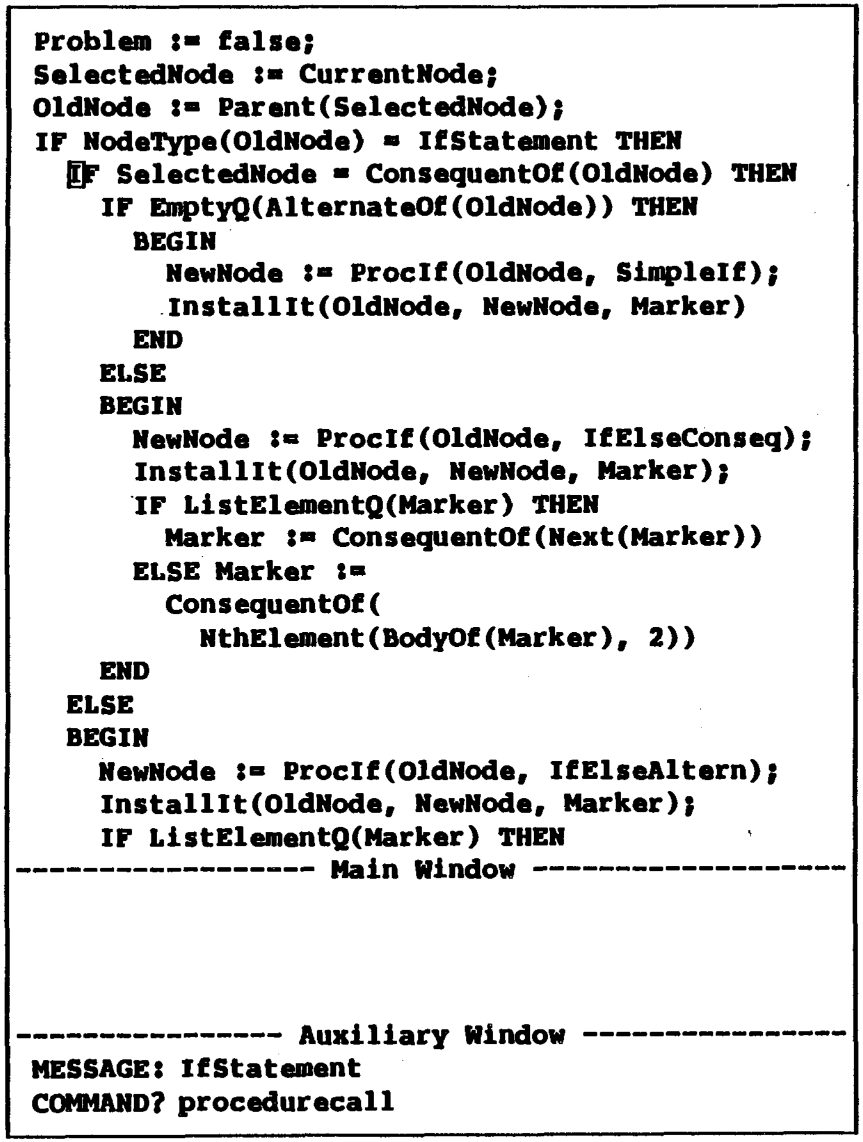
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
IF NodeType(OldNode) = IfStatement THEN
  IF SelectedNode = ConsequentOf(OldNode) THEN
    IF EmptyQ(AlternateOf(OldNode)) THEN
      BEGIN
        NewNode := ProcIf(OldNode, SimpleIf);
        InstallIt(OldNode, NewNode, Marker)
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseConseq);
      InstallIt(OldNode, NewNode, Marker);
      IF ListElementQ(Marker) THEN
        Marker := ConsequentOf(Next(Marker))
      ELSE Marker :=
        ConsequentOf(
          NthElement(BodyOf(Marker), 2))
    END
  ELSE
  BEGIN
    NewNode := ProcIf(OldNode, IfElseAltern);
    InstallIt(OldNode, NewNode, Marker);
    IF ListElementQ(Marker) THEN
----------------- Main Window -------------------



--------------- Auxiliary Window ---------------
MESSAGE: Relation
COMMAND? procedurecall
```



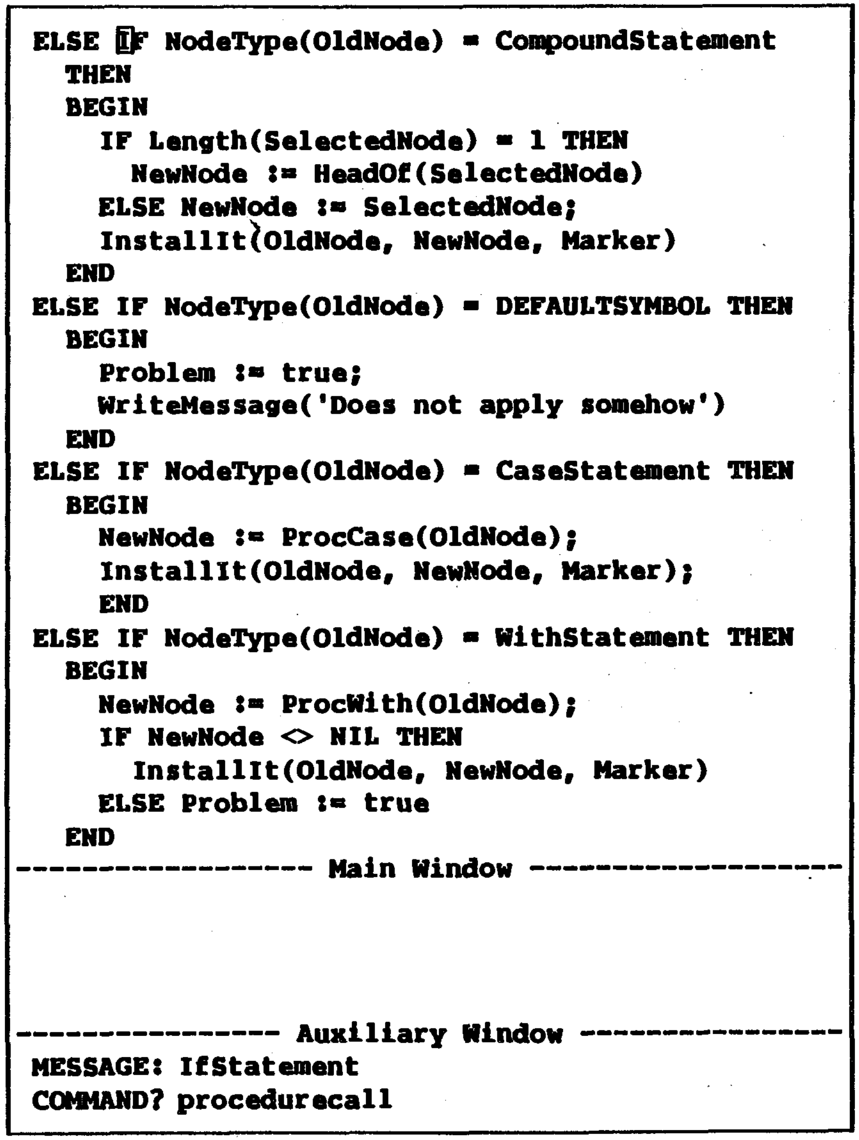
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
IF NodeType(OldNode) = IfStatement THEN
  IF SelectedNode = ConsequentOf(OldNode) THEN
    IF EmptyQ(AlternateOf(OldNode)) THEN
      BEGIN
        NewNode := ProcIf(OldNode, SimpleIf);
        InstallIt(OldNode, NewNode, Marker)
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseConseq);
      InstallIt(OldNode, NewNode, Marker);
      IF ListElementQ(Marker) THEN
        Marker := ConsequentOf(Next(Marker))
      ELSE Marker :=
        ConsequentOf(
          NthElement(BodyOf(Marker), 2))
    END
  ELSE
  BEGIN
    NewNode := ProcIf(OldNode, IfElseAltern);
    InstallIt(OldNode, NewNode, Marker);
    IF ListElementQ(Marker) THEN
----------------- Main Window -------------------



--------------- Auxiliary Window ---------------
MESSAGE: IfStatement
COMMAND? procedurecall
```

CURSOR RIGHT

6.11: The cursor is moved laterally to the consequent statement.



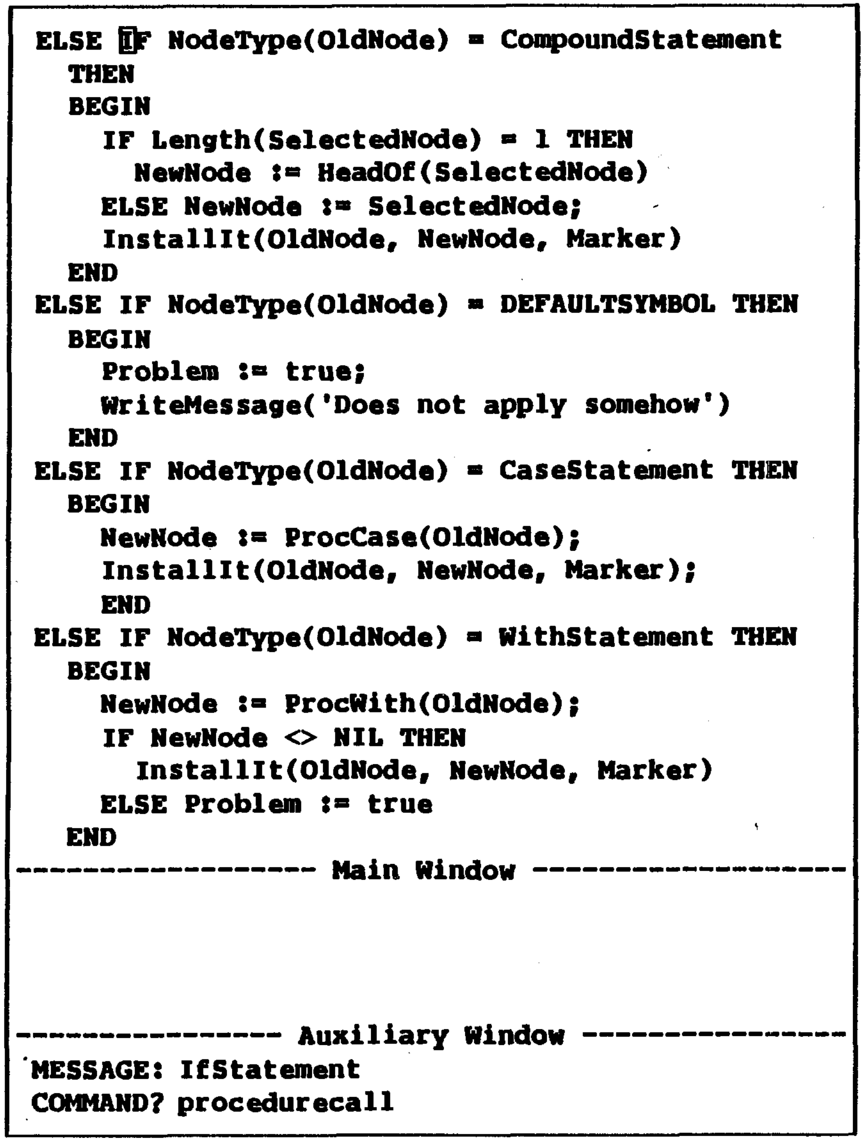
```
Problem := false;
SelectedNode := CurrentNode;
OldNode := Parent(SelectedNode);
IF NodeType(OldNode) = IfStatement THEN
  IF SelectedNode = ConsequentOf(OldNode) THEN
    IF EmptyQ(AlternateOf(OldNode)) THEN
      BEGIN
        NewNode := ProcIf(OldNode, SimpleIf);
        InstallIt(OldNode, NewNode, Marker)
      END
    ELSE
    BEGIN
      NewNode := ProcIf(OldNode, IfElseConseq);
      InstallIt(OldNode, NewNode, Marker);
      IF ListElementQ(Marker) THEN
        Marker := ConsequentOf(Next(Marker))
      ELSE Marker :=
        ConsequentOf(
          NthElement(BodyOf(Marker), 2))
    END
  ELSE
  BEGIN
    NewNode := ProcIf(OldNode, IfElseAltern);
    InstallIt(OldNode, NewNode, Marker);
    IF ListElementQ(Marker) THEN
------------------ Main Window ------------------




---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? procedurecall
```



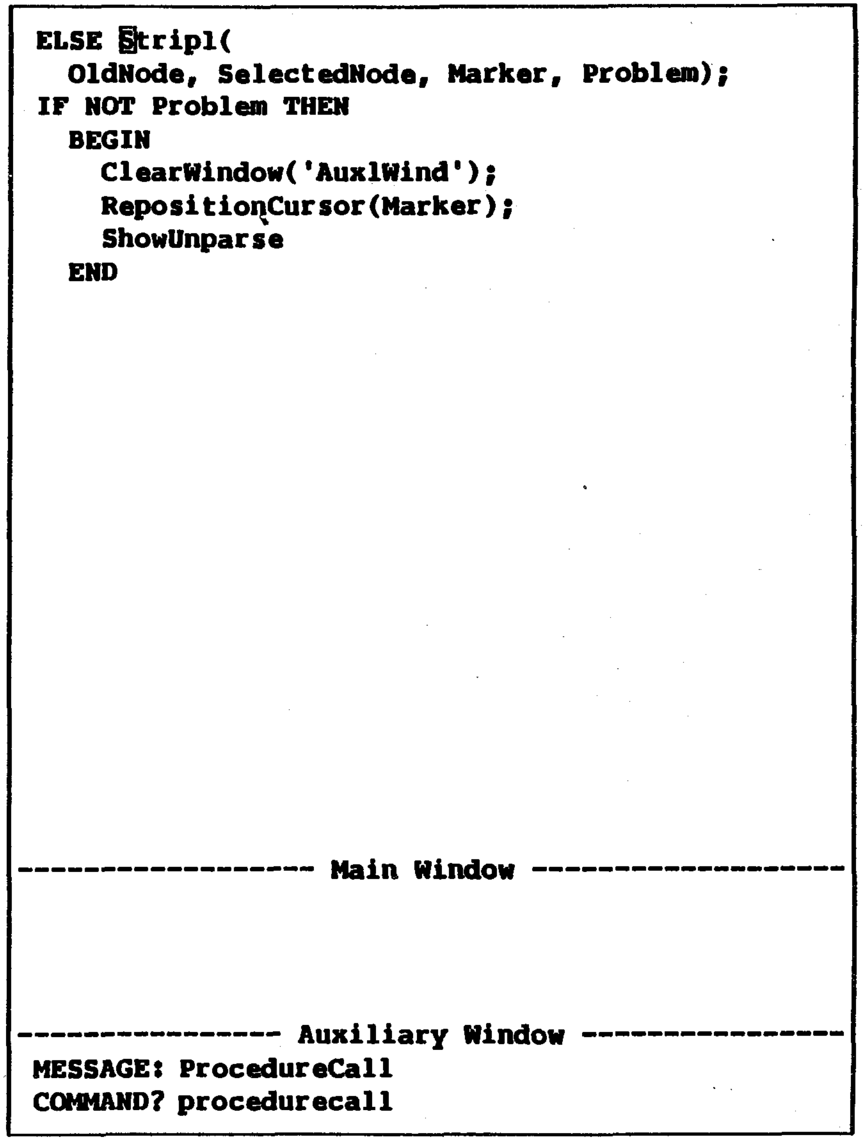
```
ELSE IF NodeType(OldNode) = CompoundStatement
  THEN
  BEGIN
    IF Length(SelectedNode) = 1 THEN
      NewNode := HeadOf(SelectedNode)
    ELSE NewNode := SelectedNode;
    InstallIt(OldNode, NewNode, Marker)
  END
ELSE IF NodeType(OldNode) = DEFAULTSYMBOL THEN
  BEGIN
    Problem := true;
    WriteMessage('Does not apply somehow')
  END
ELSE IF NodeType(OldNode) = CaseStatement THEN
  BEGIN
    NewNode := ProcCase(OldNode);
    InstallIt(OldNode, NewNode, Marker);
    END
ELSE IF NodeType(OldNode) = WithStatement THEN
  BEGIN
    NewNode := ProcWith(OldNode);
    IF NewNode <> NIL THEN
      InstallIt(OldNode, NewNode, Marker)
    ELSE Problem := true
  END
------------------ Main Window ------------------




---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? procedurecall
```

CURSOR RIGHT

6.1m: The cursor is moved again to the desired alternate.



```
ELSE IF NodeType(OldNode) = CompoundStatement
  THEN
  BEGIN
    IF Length(SelectedNode) = 1 THEN
      NewNode := HeadOf(SelectedNode)
    ELSE NewNode := SelectedNode;
    InstallIt(OldNode, NewNode, Marker)
  END
ELSE IF NodeType(OldNode) = DEFAULTSYMBOL THEN
  BEGIN
    Problem := true;
    WriteMessage('Does not apply somehow')
  END
ELSE IF NodeType(OldNode) = CaseStatement THEN
  BEGIN
    NewNode := ProcCase(OldNode);
    InstallIt(OldNode, NewNode, Marker);
    END
ELSE IF NodeType(OldNode) = WithStatement THEN
  BEGIN
    NewNode := ProcWith(OldNode);
    IF NewNode <> NIL THEN
      InstallIt(OldNode, NewNode, Marker)
    ELSE Problem := true
  END
------------------ Main Window ------------------

---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? procedurecall
```



```
ELSE Stripl(
  OldNode, SelectedNode, Marker, Problem);
IF NOT Problem THEN
  BEGIN
    ClearWindow('AuxlWind');
    RepositionCursor(Marker);
    ShowUnparse
  END
----------------- Main Window ------------------

---------------- Auxiliary Window ----------------
MESSAGE: ProcedureCall
COMMAND? procedurecall
```

REPLACE

6.1n: The previously created procedure replaces the alternate.

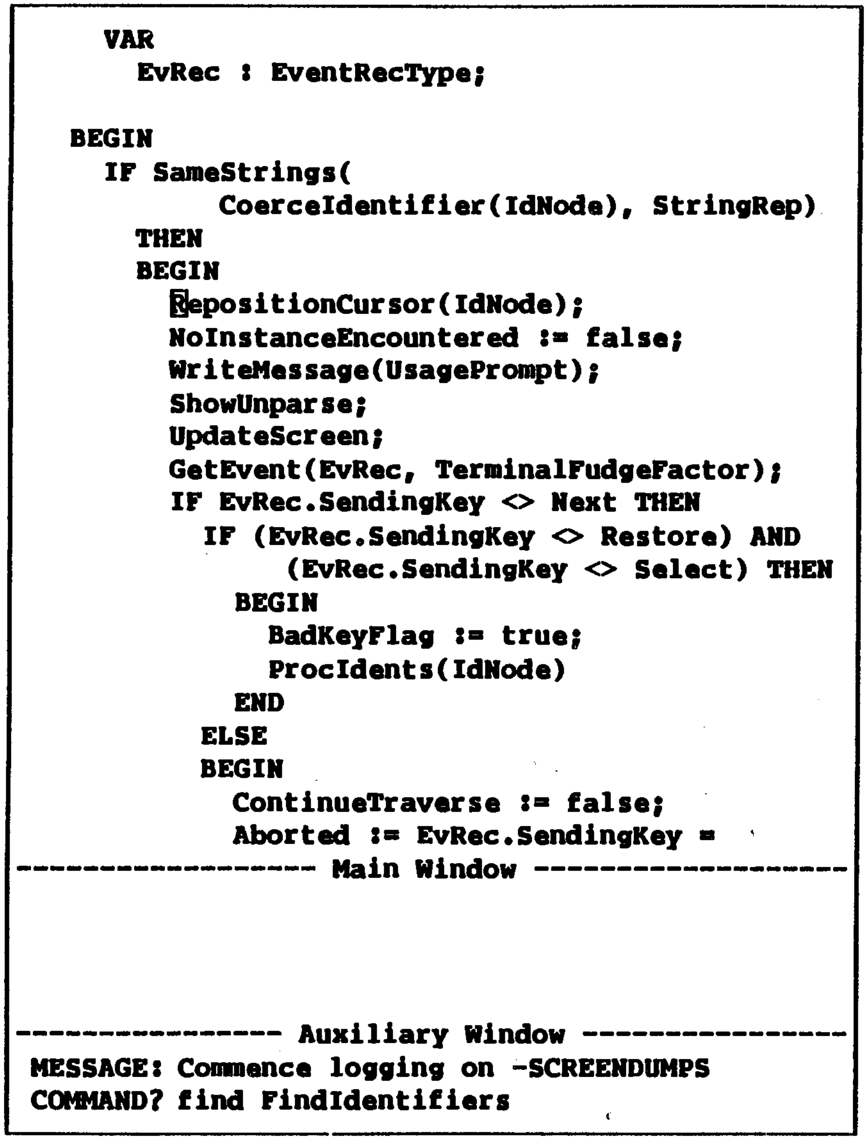
Example - Embedding and Engulfing

In EMBED I believe that I have captured an important editing notion. Of the new commands, it is the one which I have found myself calling upon most often. Application frequently involves the introduction of branching into the code, as in this example. As well, EMBED is often used in tandem with ENGULF.

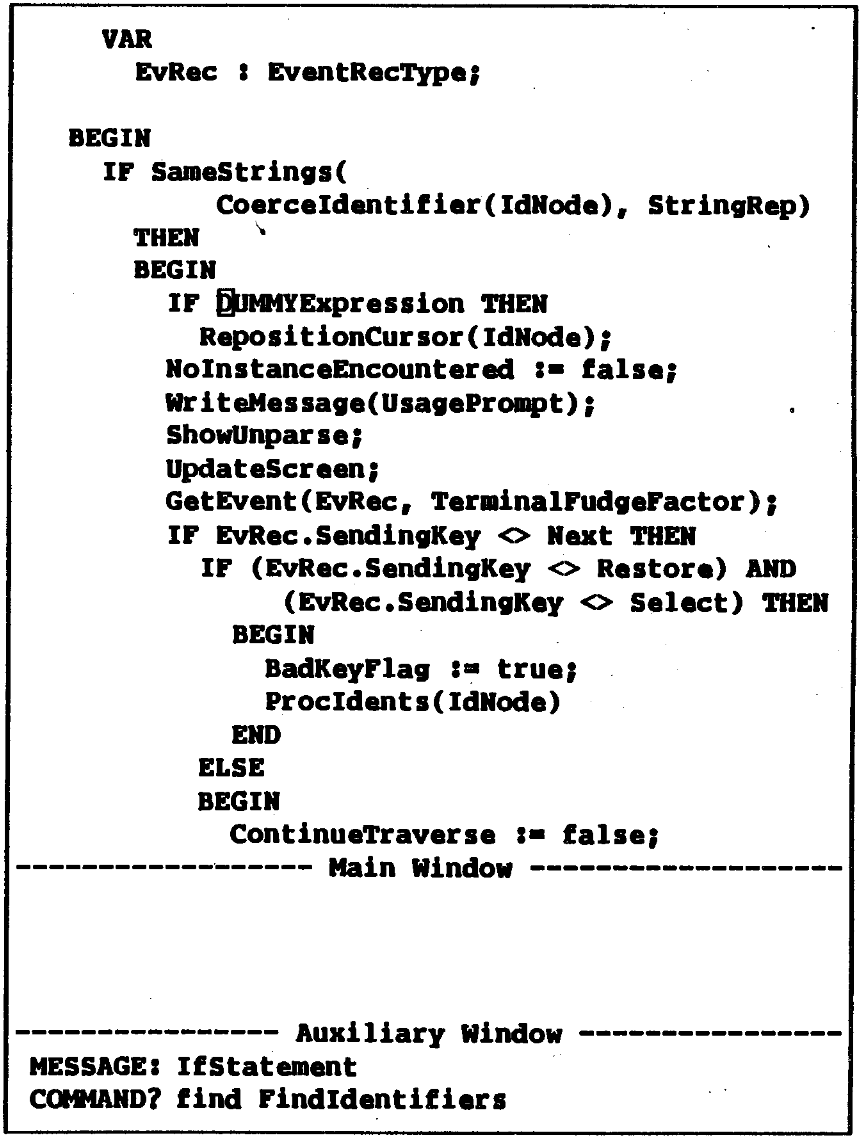
Figure 6.2a (left screen) shows the site of the intended alterations. Before the statement designated by the screen cursor is executed, a flag (`BadKeyFlag`) must be tested. If this flag has value true, then some error handling is performed, otherwise a sequence of statements, consisting of that cursor-designated statement plus the two succeeding statements, is executed.

Since that first statement is to be executed only under certain conditions, EMBED is applied to it (Figure 6.2a). To correspond with the aim delineated above, embedding in the alternate of the if statement is chosen (Figure 6.2b). The next step of the operation is actualization of the statement's predicate. Though the desired identifier could have been selected and stacked previously and now inserted into the placeholder, I have, for the sake of generality, chosen to provide `BadKeyFlag` textually (Figure 6.2c, 6.2d, 6.2e). Similarly, to provide the error handling statements, the cursor is moved to the consequent placeholder (Figure 6.2f), MODIFY is invoked once more (Figure 6.2g), and the new text is entered without any particular regard for formatting (Figures 6.2h, 6.2i), leading to actualization of the placeholder (Figure 6.2j).

It will be recalled that not just `RepositionCursor(IdNode)`, but the succeeding two statements as well, are to be executed if `BadKeyFlag` evaluates to false. To bring about this result, the syntactic cursor is moved to the alternate statement (Figure 6.2k), and two applications of ENGULF-NEXT are keyed in (Figures 6.2l, 6.2m).



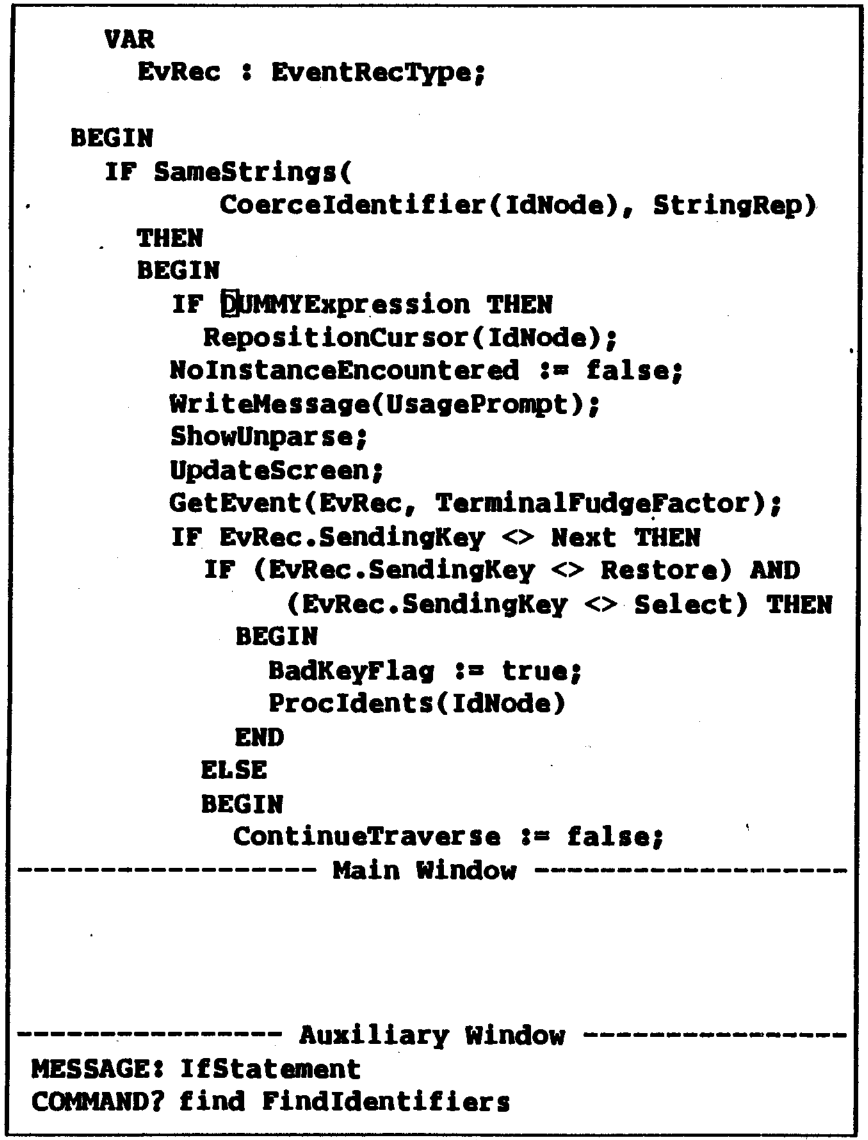
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
            Aborted := EvRec.SendingKey =
------------------ Main Window -------------------



----------------- Auxiliary Window ----------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? find FindIdentifiers
```



```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN
          RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window -------------------



----------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? find FindIdentifiers
```

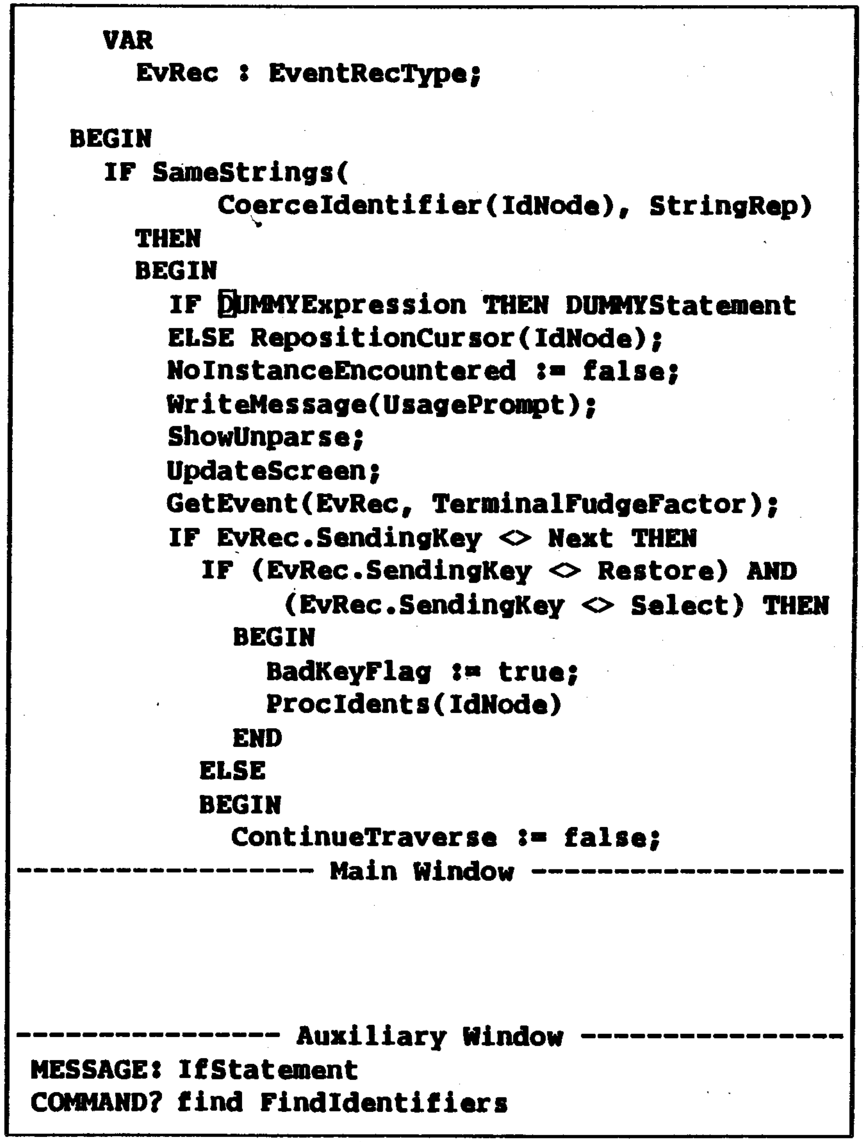
EMBED

6.2a: EMBED is applied to the procedure call.



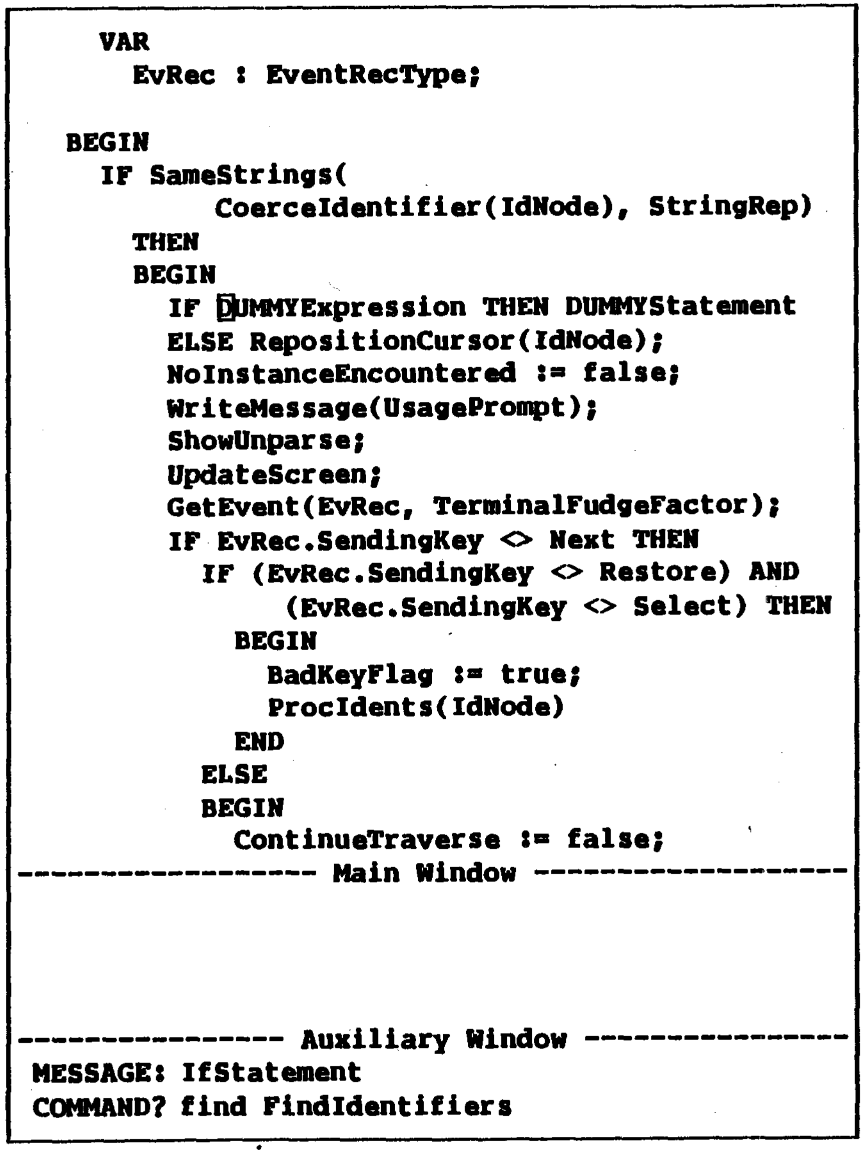
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN
          RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? find FindIdentifiers
```

NEXT



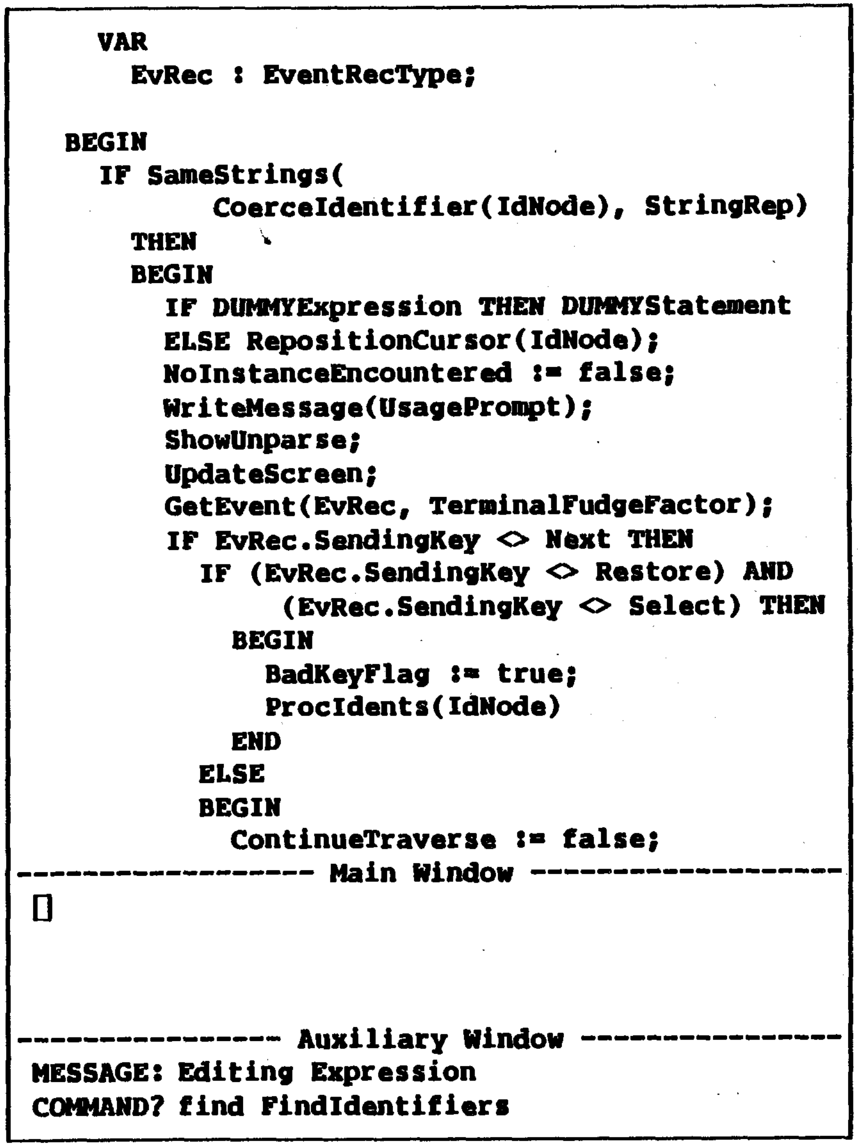
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: IfStatement
COMMAND? find FindIdentifiers
```

6.2b: The desired embedding is located.



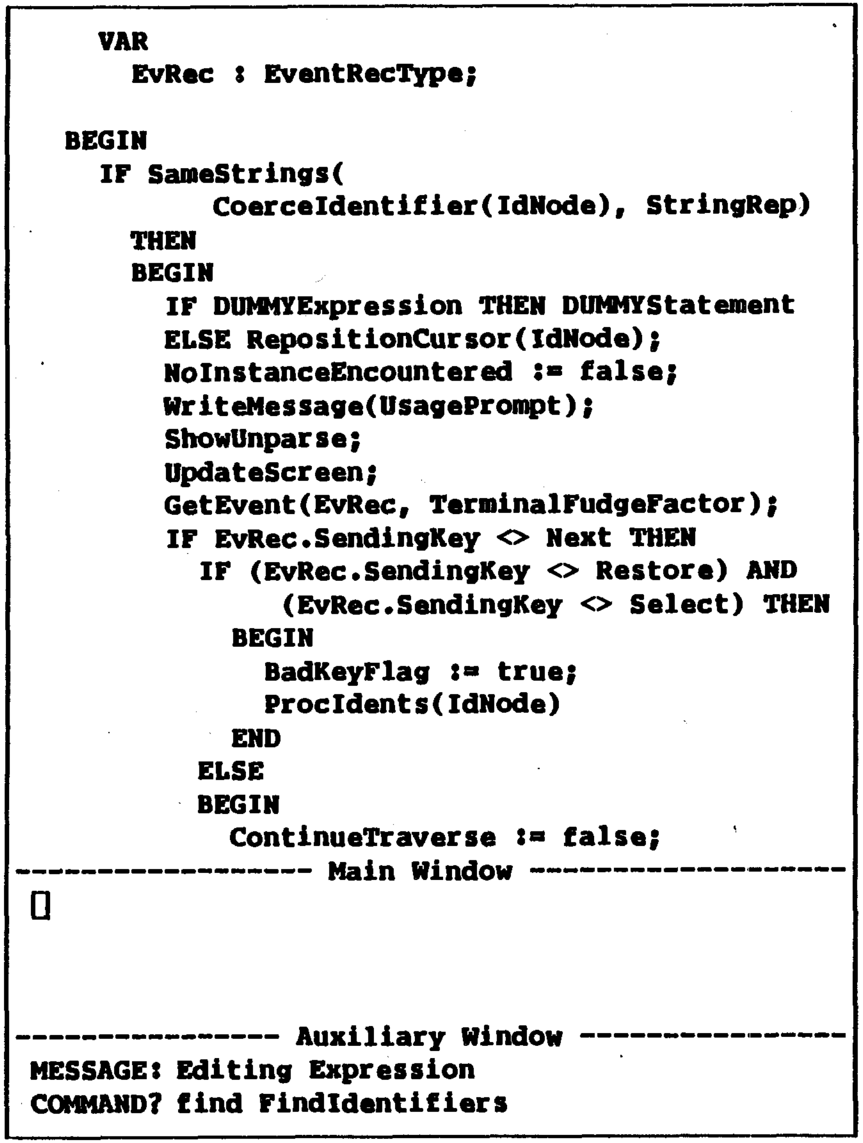
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------




--------------- Auxiliary Window ---------------
MESSAGE: IfStatement
COMMAND? find FindIdentifiers
```

MODIFY CURRENT

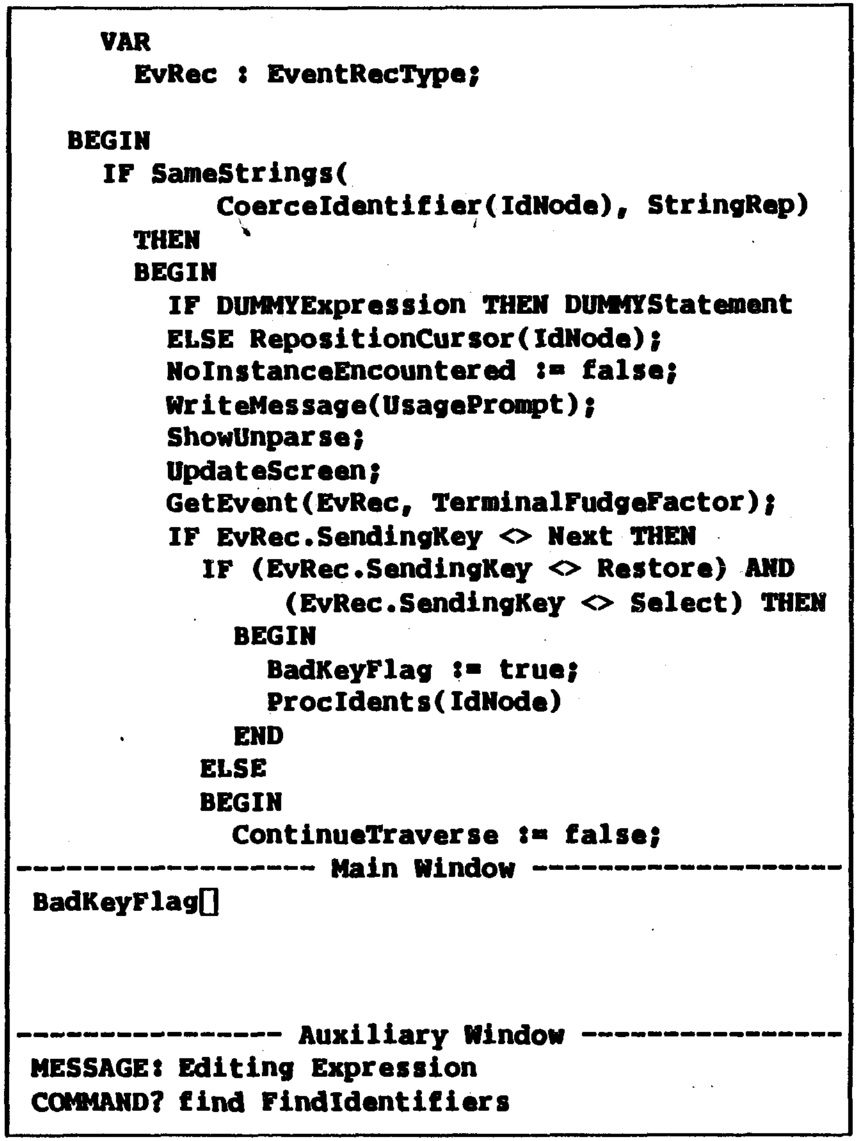


```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------
□



--------------- Auxiliary Window ---------------
MESSAGE: Editing Expression
COMMAND? find FindIdentifiers
```

6.2c: The editor is called to actualize the placeholder.

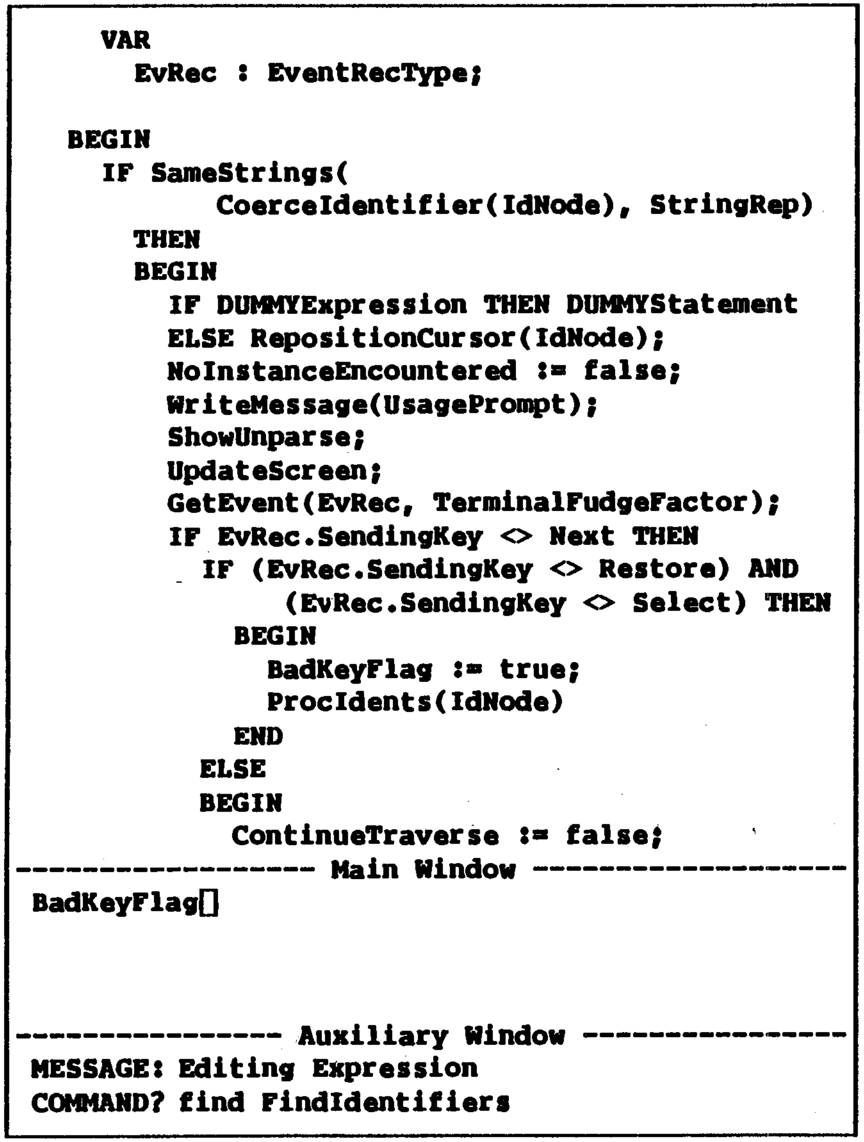


```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
----------------- Main Window ------------------
[]



--------------- Auxiliary Window ---------------
MESSAGE: Editing Expression
COMMAND? find FindIdentifiers
```

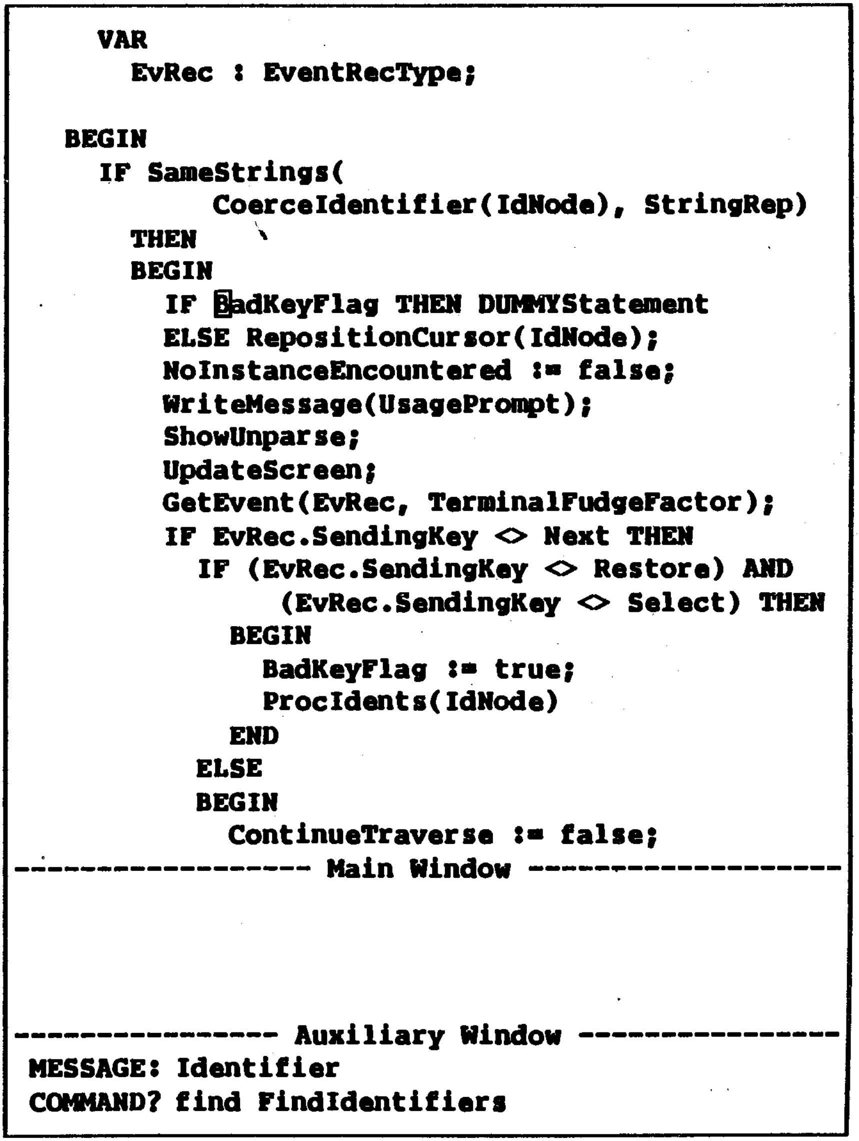


```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
----------------- Main Window ------------------
BadKeyFlag[]



--------------- Auxiliary Window ---------------
MESSAGE: Editing Expression
COMMAND? find FindIdentifiers
```

6.2d: Text is entered.



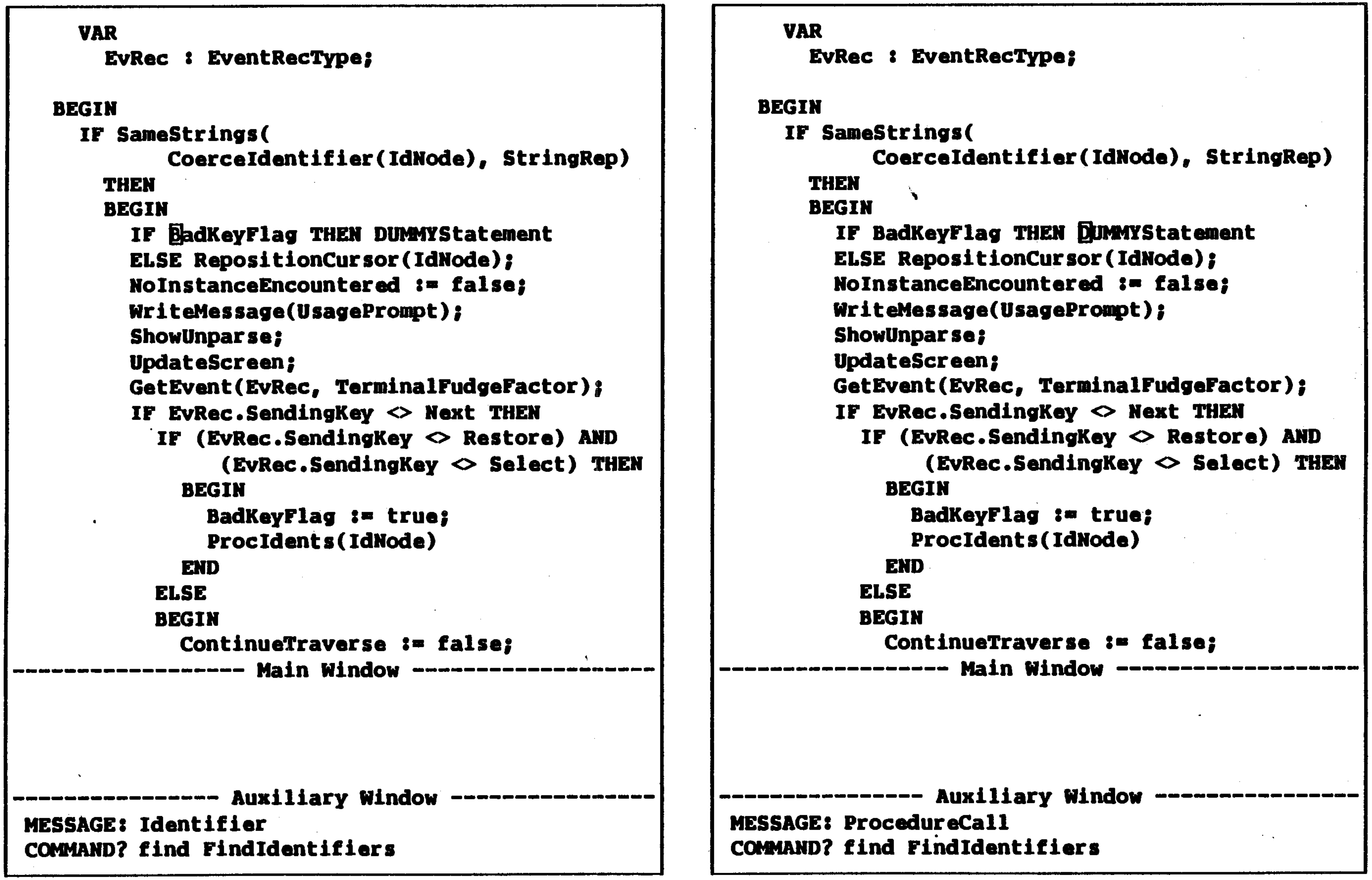
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
          CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF DUMMYExpression THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------
BadKeyFlag□



---------------- Auxiliary Window ----------------
MESSAGE: Editing Expression
COMMAND? find FindIdentifiers
```



```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
          CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: Identifier
COMMAND? find FindIdentifiers
```

EDIT COMPLETE

6.2e: The placeholder is replaced.

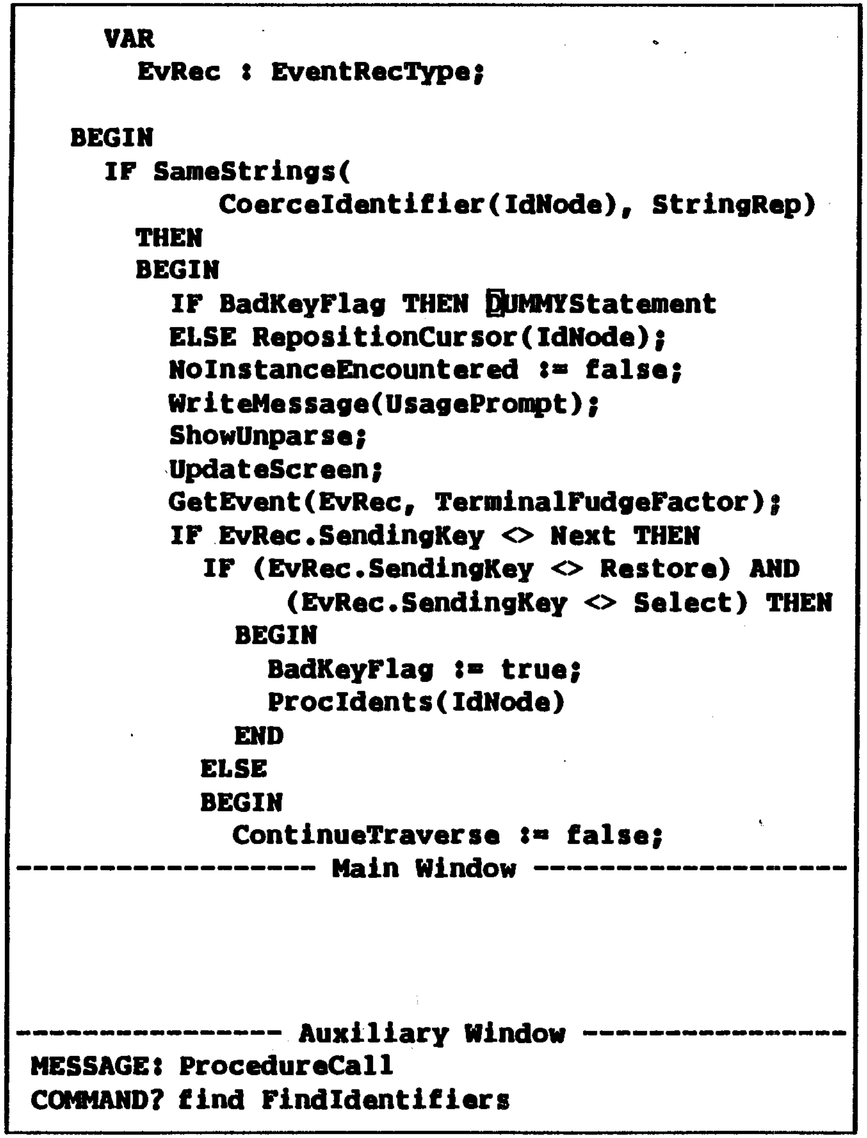


```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
          CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
 MESSAGE: Identifier
 COMMAND? find FindIdentifiers
```

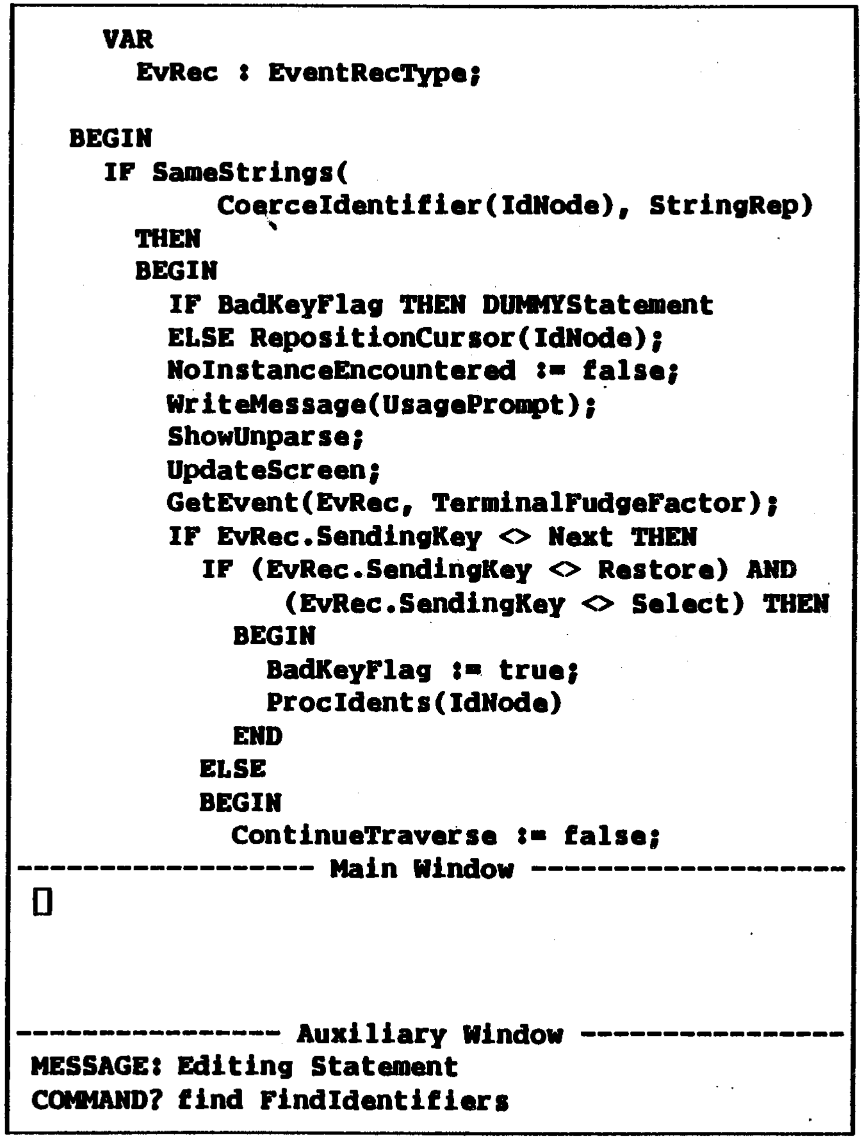
CURSOR RIGHT

```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
          CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
 MESSAGE: ProcedureCall
 COMMAND? find FindIdentifiers
```

6.2f: The next placeholder is selected.



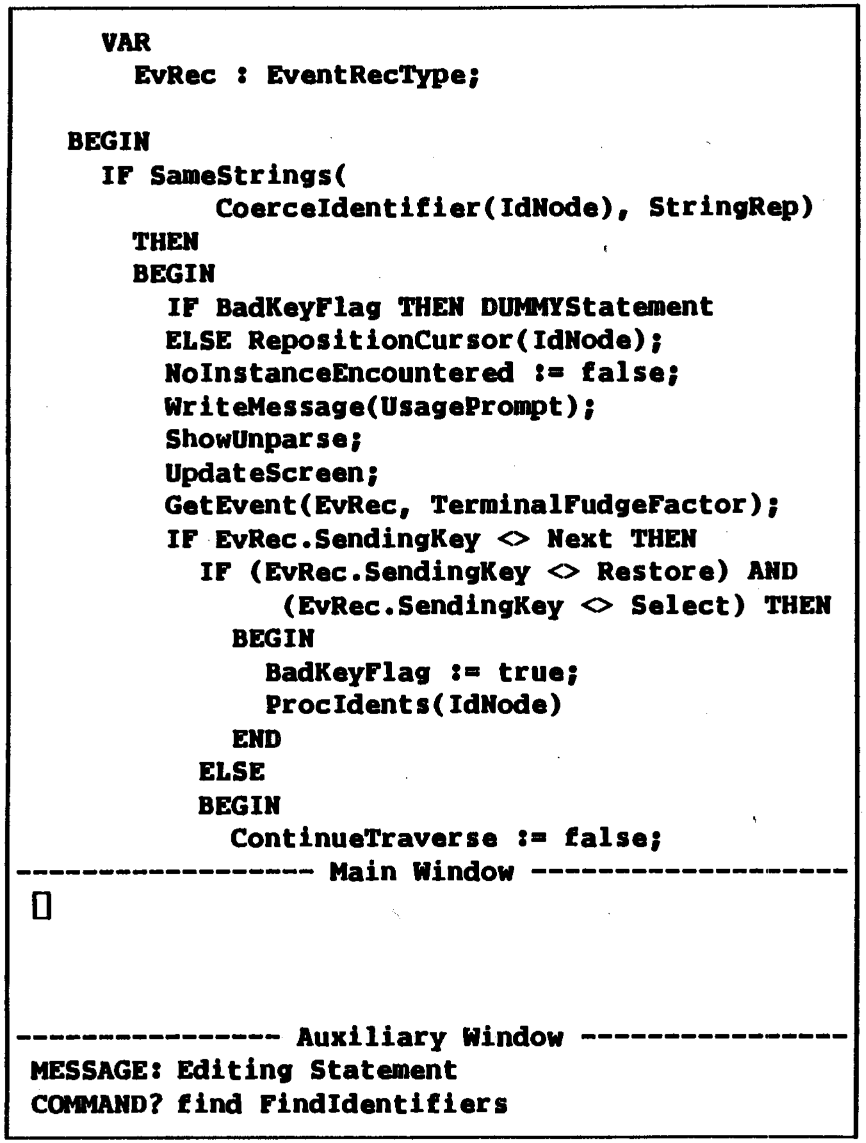
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window --------------------




---------------- Auxiliary Window ----------------
MESSAGE: ProcedureCall
COMMAND? find FindIdentifiers
```



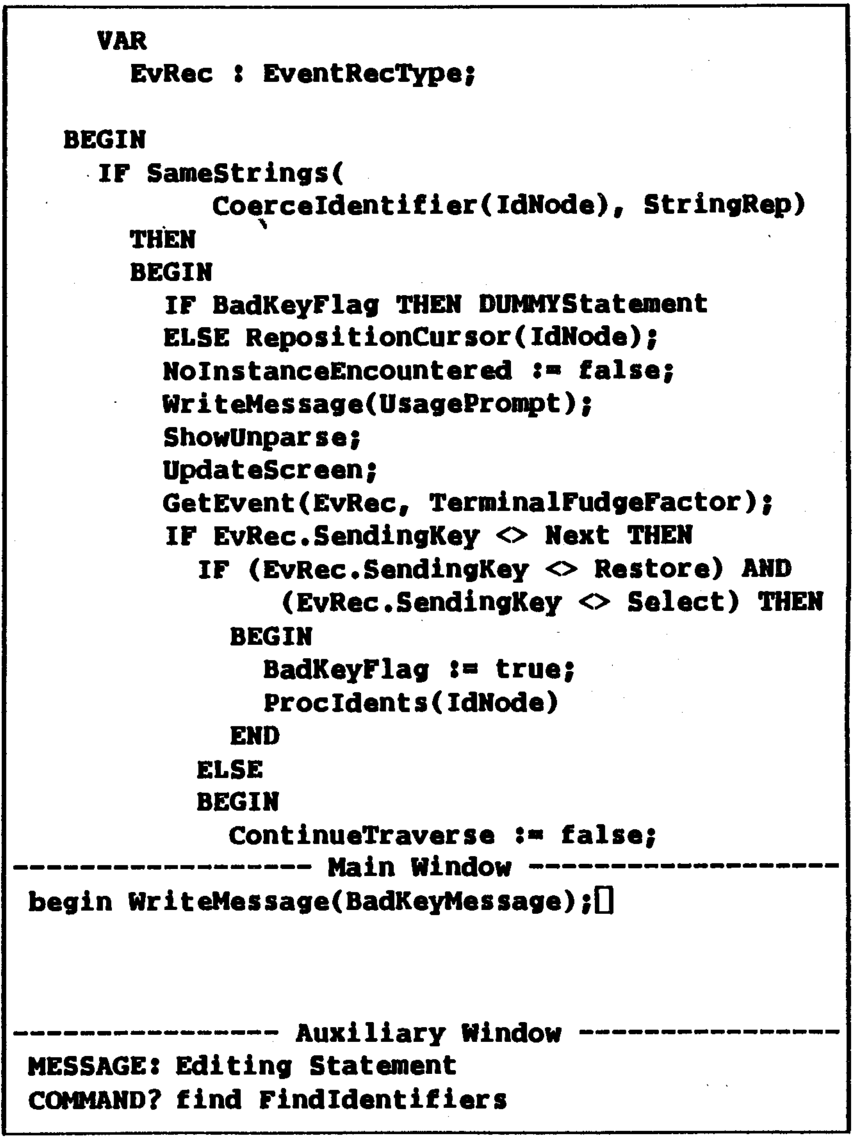
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window --------------------
□



---------------- Auxiliary Window ----------------
MESSAGE: Editing Statement
COMMAND? find FindIdentifiers
```

MODIFY CURRENT

6.2g: The editor is called to fill this placeholder.

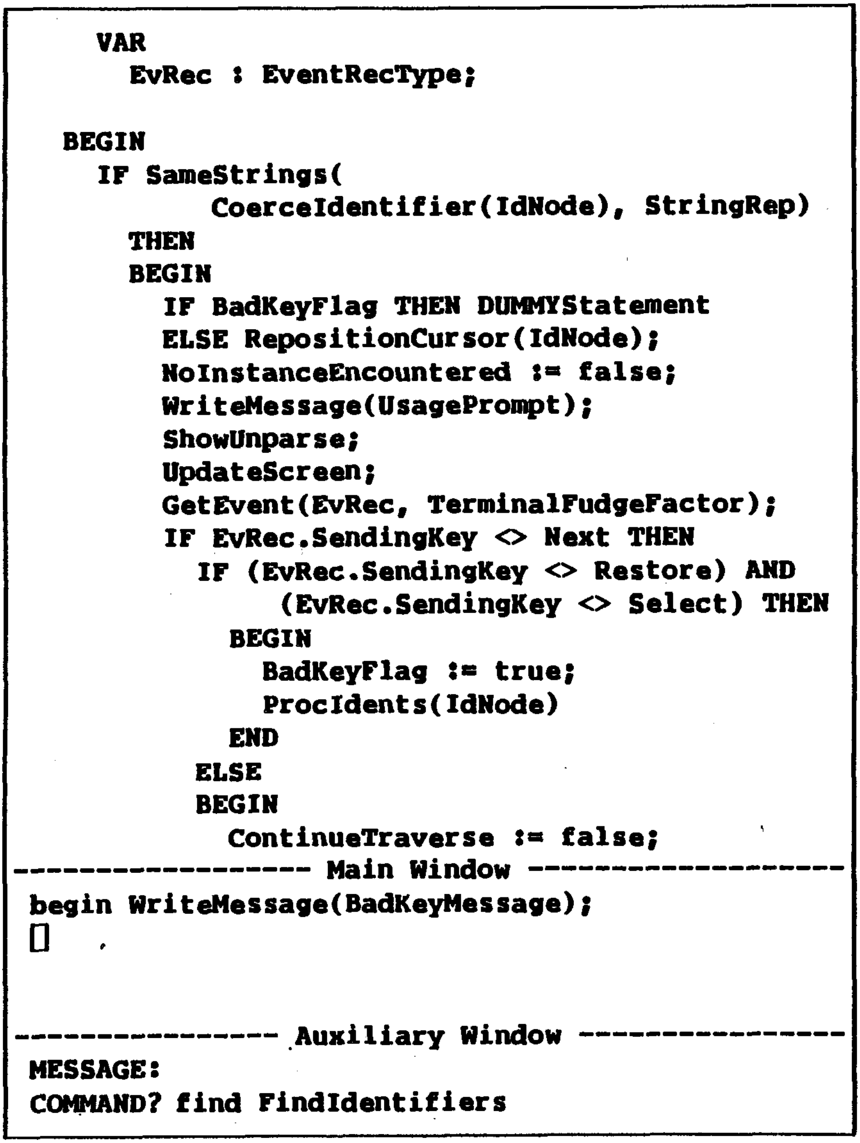


```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window -------------------
□



---------------- Auxiliary Window ----------------
MESSAGE: Editing Statement
COMMAND? find FindIdentifiers
```

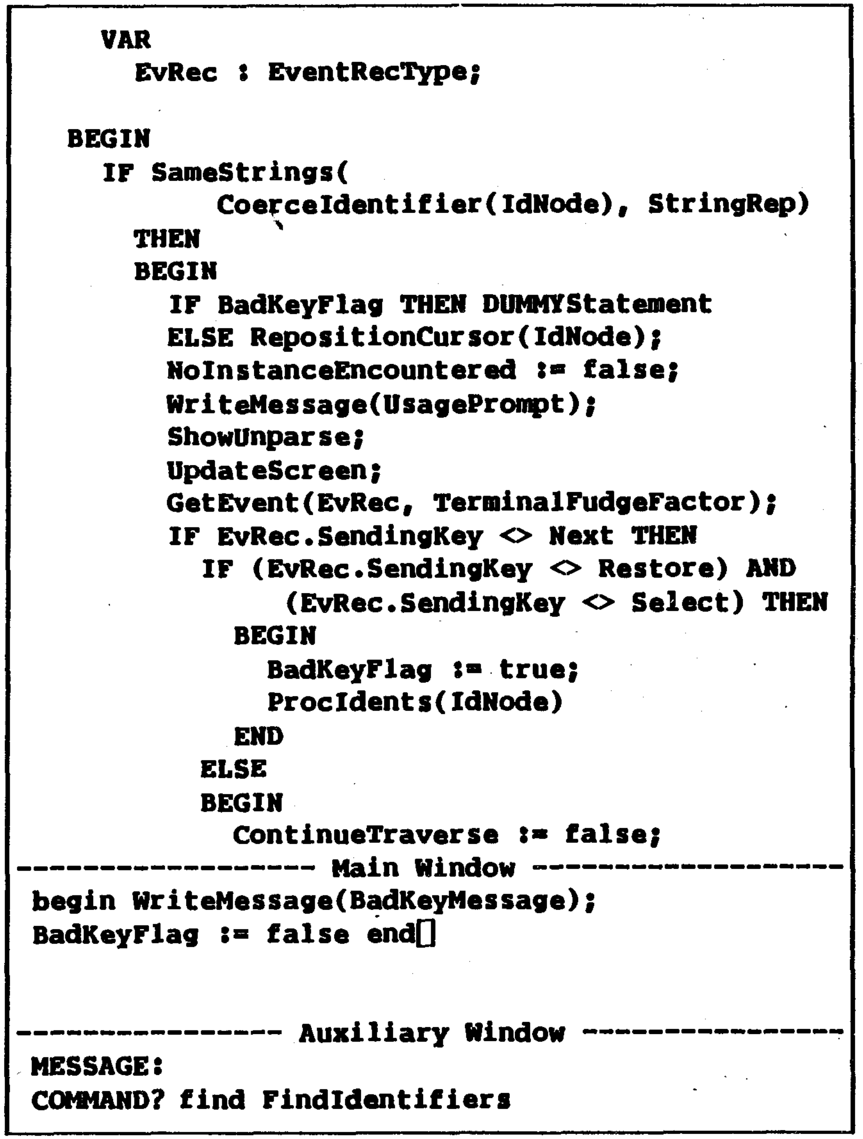


```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window -------------------
begin WriteMessage(BadKeyMessage);□



---------------- Auxiliary Window ----------------
MESSAGE: Editing Statement
COMMAND? find FindIdentifiers
```

6.2h: A line of text is entered.

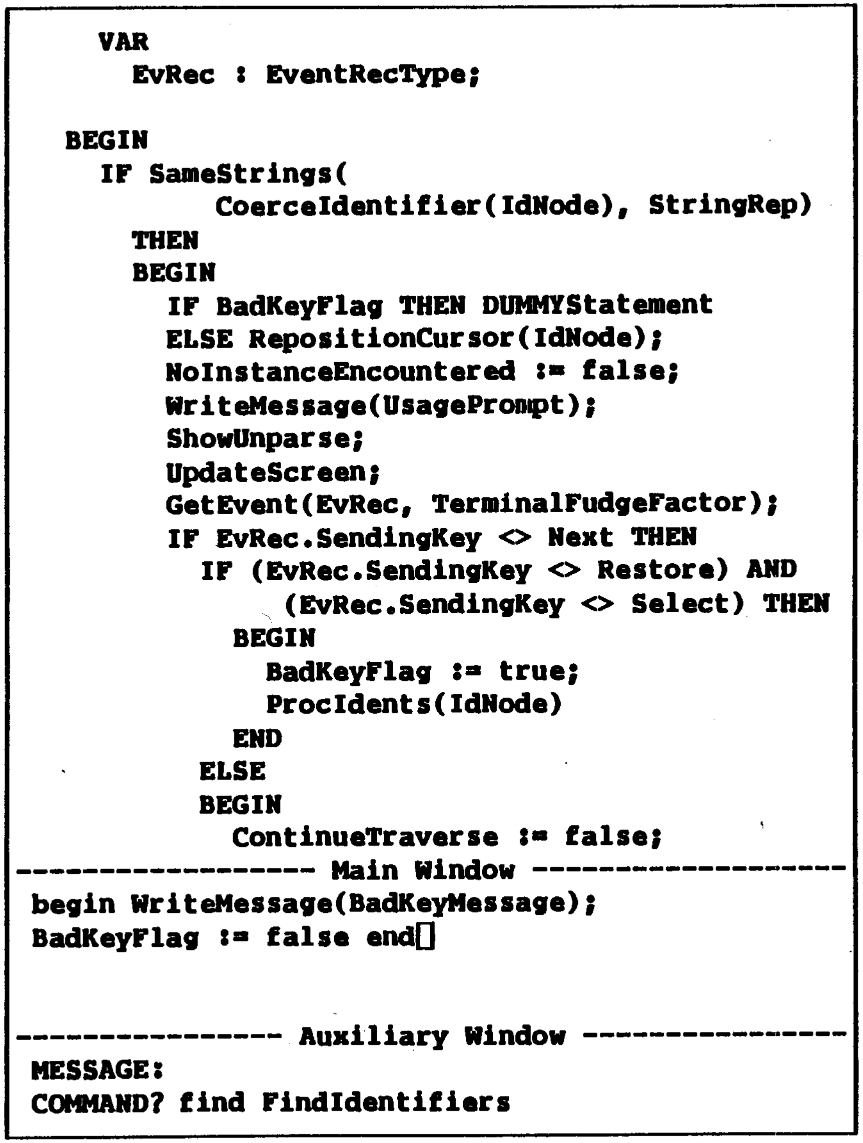


```
  VAR
   EvRec : EventRecType;

 BEGIN
  IF SameStrings(
      CoerceIdentifier(IdNode), StringRep)
   THEN
   BEGIN
    IF BadKeyFlag THEN DUMMYStatement
    ELSE RepositionCursor(IdNode);
    NoInstanceEncountered := false;
    WriteMessage(UsagePrompt);
    ShowUnparse;
    UpdateScreen;
    GetEvent(EvRec, TerminalFudgeFactor);
    IF EvRec.SendingKey <> Next THEN
     IF (EvRec.SendingKey <> Restore) AND
        (EvRec.SendingKey <> Select) THEN
      BEGIN
       BadKeyFlag := true;
       ProcIdents(IdNode)
      END
     ELSE
     BEGIN
      ContinueTraverse := false;
----------------- Main Window -----------------
begin WriteMessage(BadKeyMessage);
[]


--------------- Auxiliary Window ---------------
MESSAGE:
COMMAND? find FindIdentifiers
```

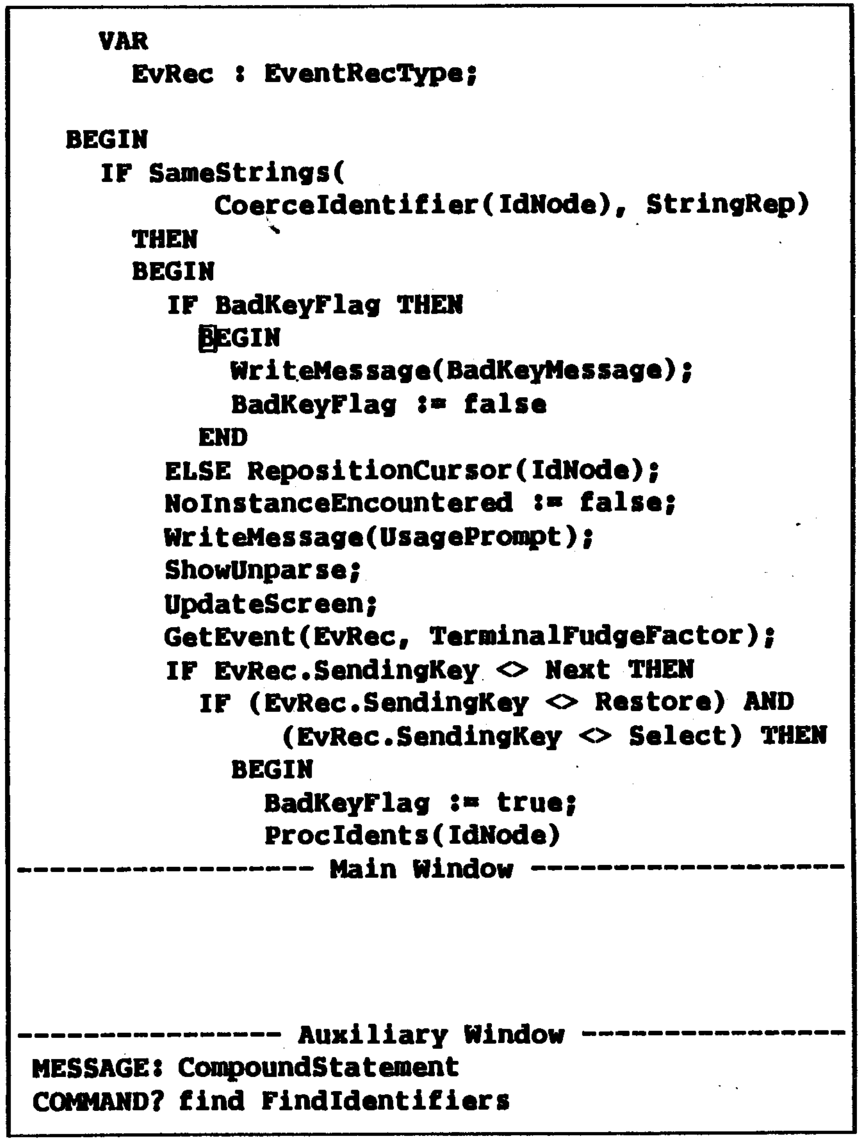


```
  VAR
   EvRec : EventRecType;

 BEGIN
  IF SameStrings(
      CoerceIdentifier(IdNode), StringRep)
   THEN
   BEGIN
    IF BadKeyFlag THEN DUMMYStatement
    ELSE RepositionCursor(IdNode);
    NoInstanceEncountered := false;
    WriteMessage(UsagePrompt);
    ShowUnparse;
    UpdateScreen;
    GetEvent(EvRec, TerminalFudgeFactor);
    IF EvRec.SendingKey <> Next THEN
     IF (EvRec.SendingKey <> Restore) AND
        (EvRec.SendingKey <> Select) THEN
      BEGIN
       BadKeyFlag := true;
       ProcIdents(IdNode)
      END
     ELSE
     BEGIN
      ContinueTraverse := false;
----------------- Main Window -----------------
begin WriteMessage(BadKeyMessage);
BadKeyFlag := false end[]


--------------- Auxiliary Window ---------------
MESSAGE:
COMMAND? find FindIdentifiers
```

6.2i: Another line is entered.



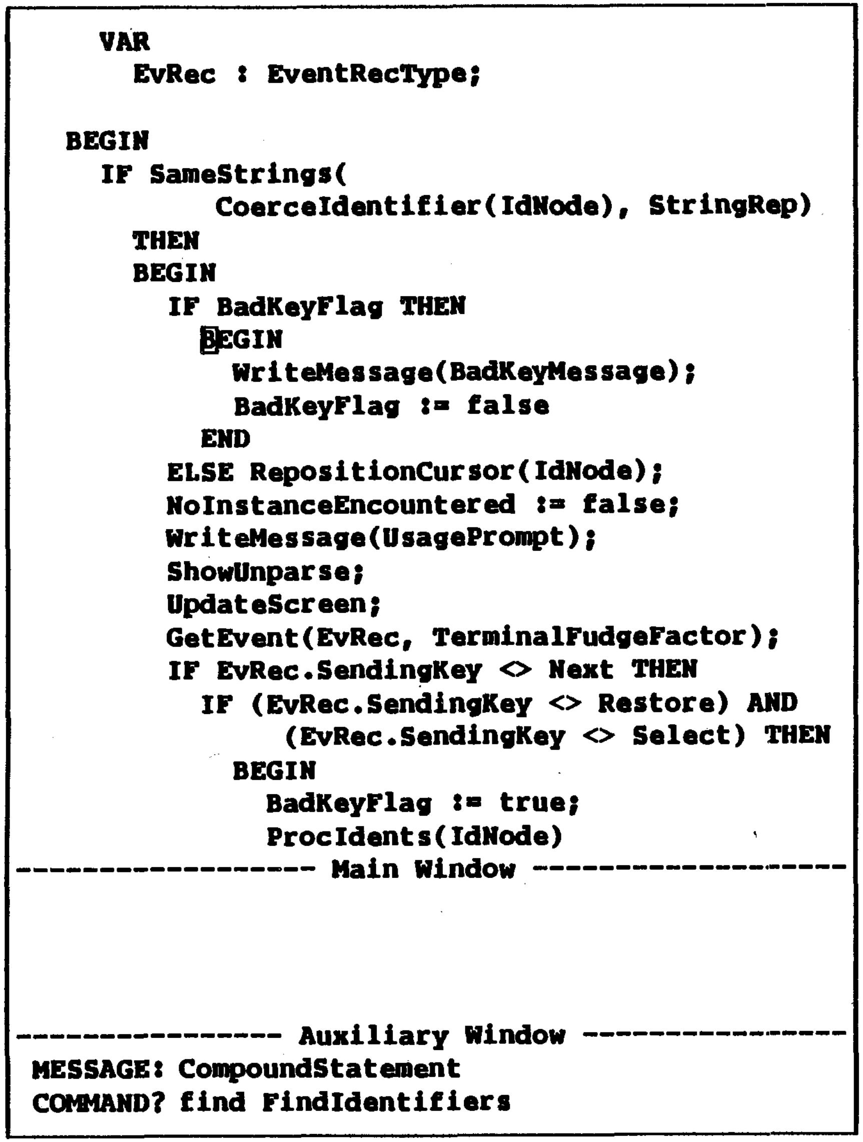
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN DUMMYStatement
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
            END
          ELSE
          BEGIN
            ContinueTraverse := false;
------------------ Main Window ------------------
begin WriteMessage(BadKeyMessage);
BadKeyFlag := false end[]


---------------- Auxiliary Window ----------------
MESSAGE:
COMMAND? find FindIdentifiers
```



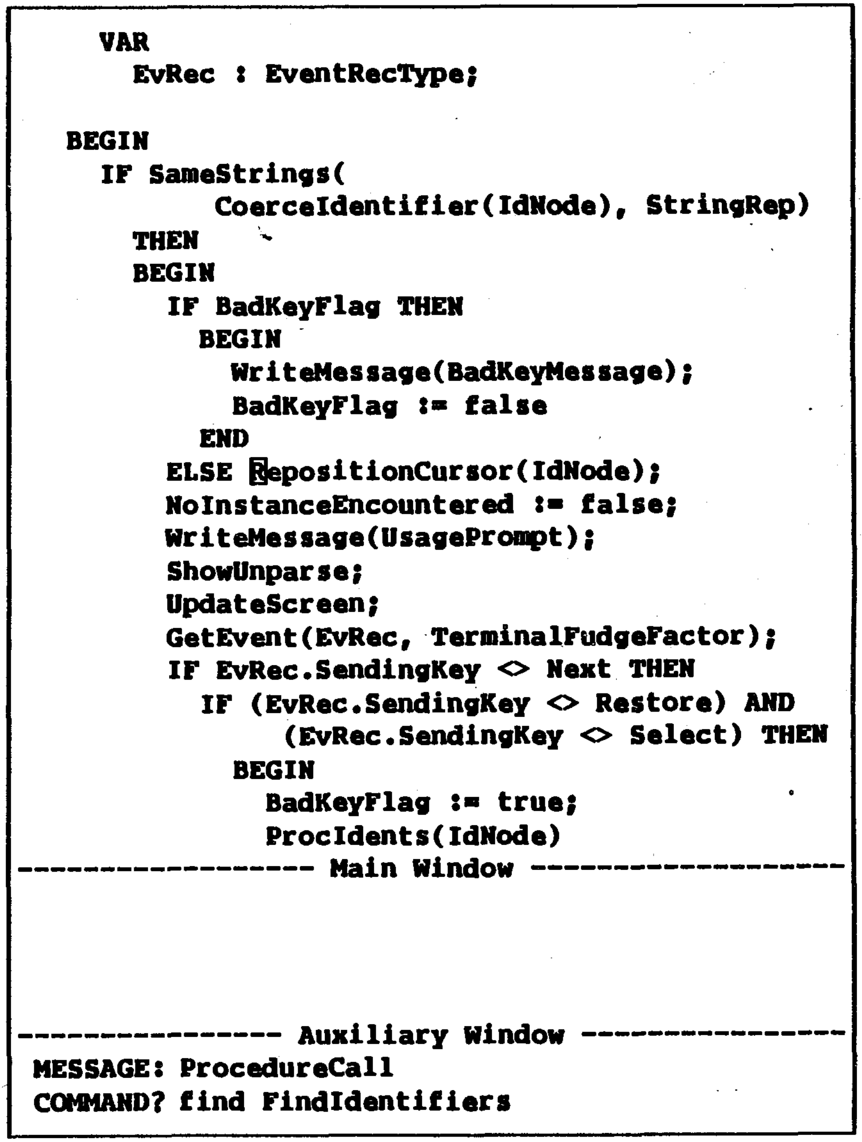
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN
          BEGIN
            WriteMessage(BadKeyMessage);
            BadKeyFlag := false
          END
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: CompoundStatement
COMMAND? find FindIdentifiers
```

EDIT COMPLETE

6.2j: The component is actualized.



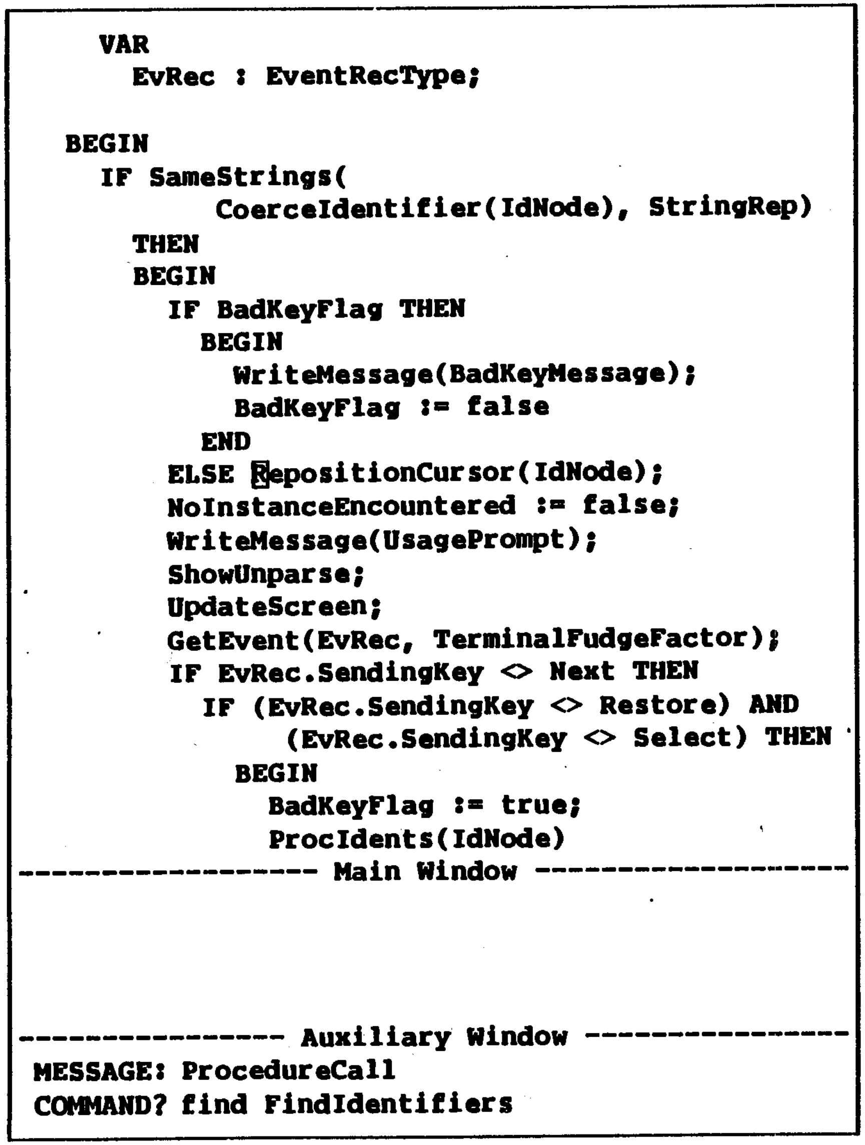
```
  VAR
    EvRec : EventRecType;

BEGIN
  IF SameStrings(
          CoerceIdentifier(IdNode), StringRep)
    THEN
    BEGIN
      IF BadKeyFlag THEN
        BEGIN
          WriteMessage(BadKeyMessage);
          BadKeyFlag := false
        END
      ELSE RepositionCursor(IdNode);
      NoInstanceEncountered := false;
      WriteMessage(UsagePrompt);
      ShowUnparse;
      UpdateScreen;
      GetEvent(EvRec, TerminalFudgeFactor);
      IF EvRec.SendingKey <> Next THEN
        IF (EvRec.SendingKey <> Restore) AND
             (EvRec.SendingKey <> Select) THEN
          BEGIN
            BadKeyFlag := true;
            ProcIdents(IdNode)
------------------ Main Window -------------------



----------------- Auxiliary Window ----------------
MESSAGE: CompoundStatement
COMMAND? find FindIdentifiers
```



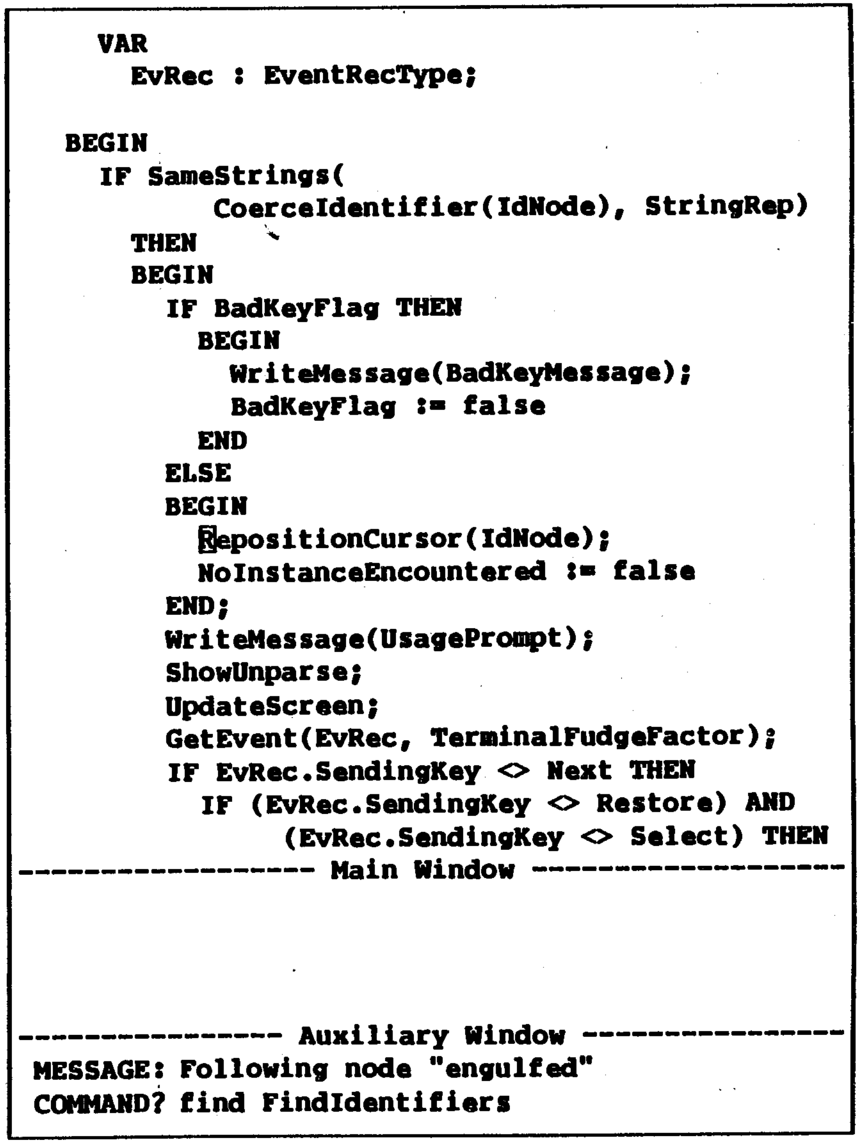
```
  VAR
    EvRec : EventRecType;

BEGIN
  IF SameStrings(
          CoerceIdentifier(IdNode), StringRep)
    THEN
    BEGIN
      IF BadKeyFlag THEN
        BEGIN
          WriteMessage(BadKeyMessage);
          BadKeyFlag := false
        END
      ELSE RepositionCursor(IdNode);
      NoInstanceEncountered := false;
      WriteMessage(UsagePrompt);
      ShowUnparse;
      UpdateScreen;
      GetEvent(EvRec, TerminalFudgeFactor);
      IF EvRec.SendingKey <> Next THEN
        IF (EvRec.SendingKey <> Restore) AND
             (EvRec.SendingKey <> Select) THEN
          BEGIN
            BadKeyFlag := true;
            ProcIdents(IdNode)
------------------ Main Window -------------------



----------------- Auxiliary Window ----------------
MESSAGE: ProcedureCall
COMMAND? find FindIdentifiers
```

CURSOR RIGHT

6.2k: The syntactic cursor is moved to the alternate statement.



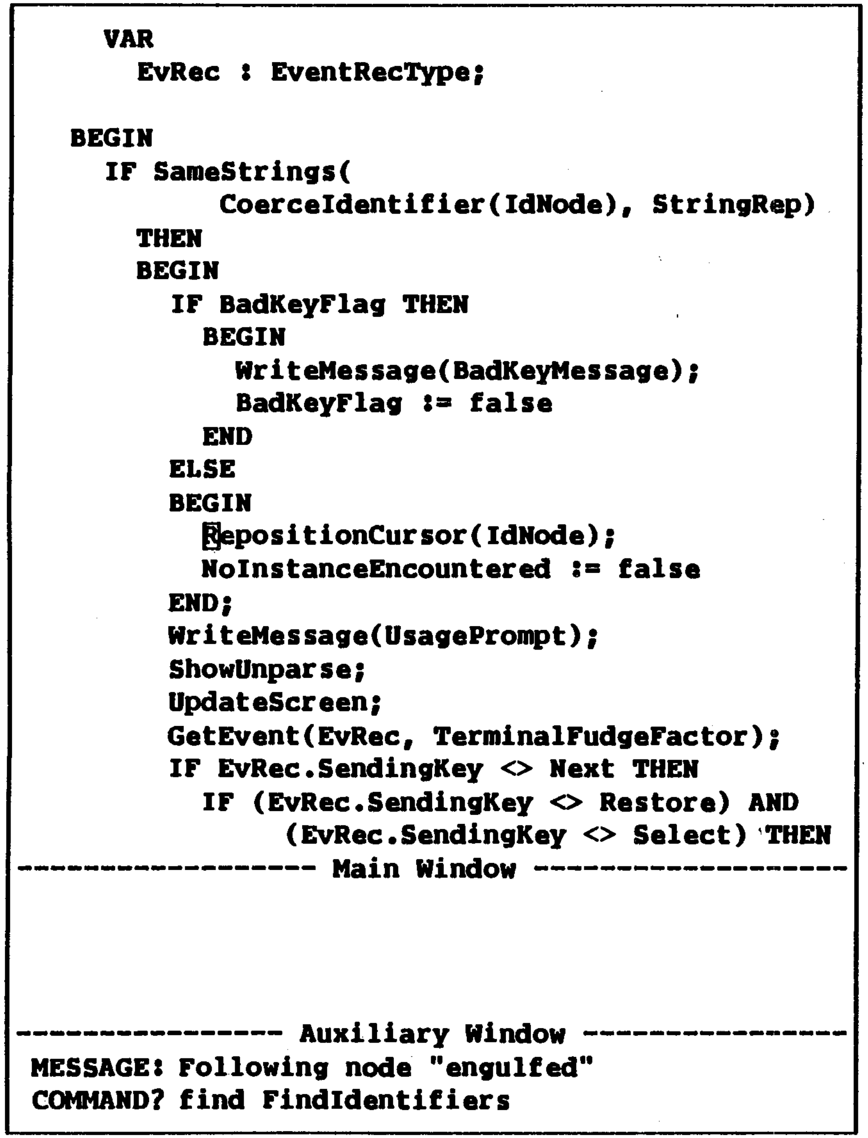
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN
          BEGIN
            WriteMessage(BadKeyMessage);
            BadKeyFlag := false
          END
        ELSE RepositionCursor(IdNode);
        NoInstanceEncountered := false;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
            BEGIN
              BadKeyFlag := true;
              ProcIdents(IdNode)
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
 MESSAGE: ProcedureCall
 COMMAND? find FindIdentifiers
```



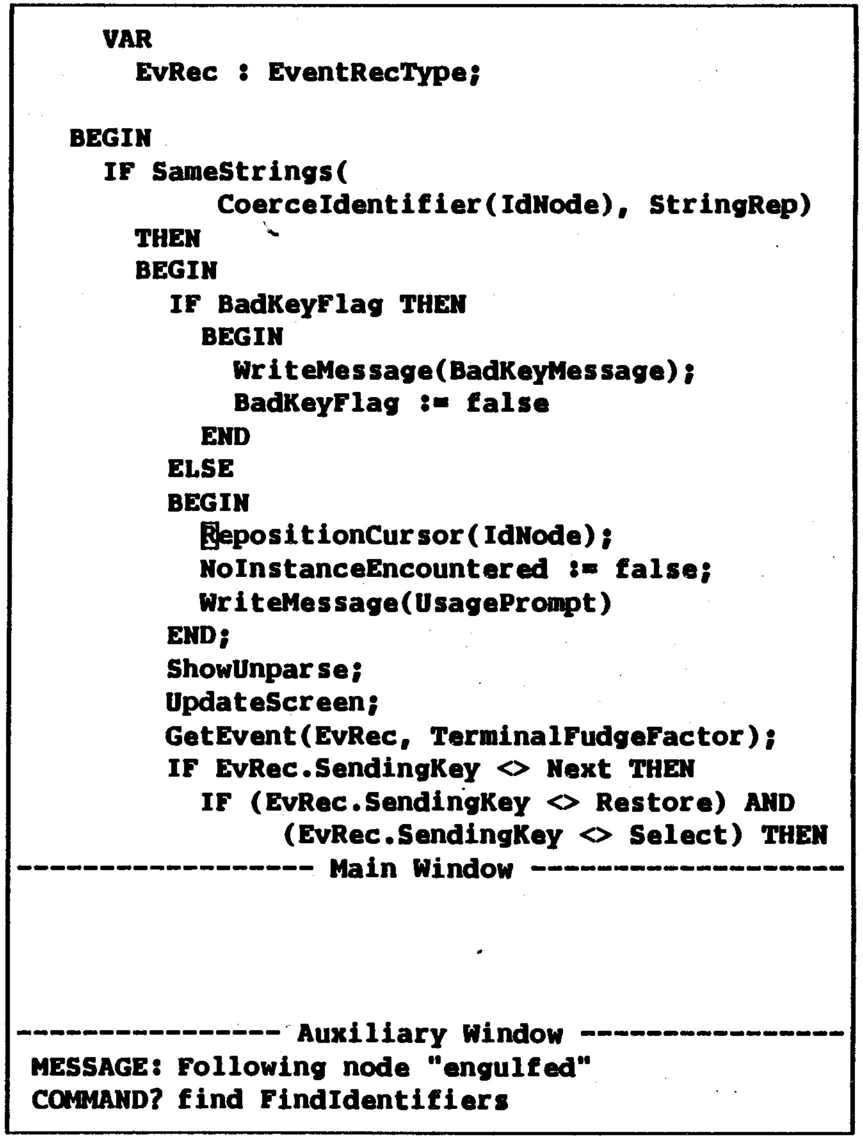
```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN
          BEGIN
            WriteMessage(BadKeyMessage);
            BadKeyFlag := false
          END
        ELSE
        BEGIN
          RepositionCursor(IdNode);
          NoInstanceEncountered := false
        END;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
 MESSAGE: Following node "engulfed"
 COMMAND? find FindIdentifiers
```

ENGULF NEXT

6.21: The next statement is engulfed.



```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN
          BEGIN
            WriteMessage(BadKeyMessage);
            BadKeyFlag := false
          END
        ELSE
        BEGIN
          RepositionCursor(IdNode);
          NoInstanceEncountered := false
        END;
        WriteMessage(UsagePrompt);
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
 MESSAGE: Following node "engulfed"
 COMMAND? find FindIdentifiers
```



```
    VAR
      EvRec : EventRecType;

  BEGIN
    IF SameStrings(
           CoerceIdentifier(IdNode), StringRep)
      THEN
      BEGIN
        IF BadKeyFlag THEN
          BEGIN
            WriteMessage(BadKeyMessage);
            BadKeyFlag := false
          END
        ELSE
        BEGIN
          RepositionCursor(IdNode);
          NoInstanceEncountered := false;
          WriteMessage(UsagePrompt)
        END;
        ShowUnparse;
        UpdateScreen;
        GetEvent(EvRec, TerminalFudgeFactor);
        IF EvRec.SendingKey <> Next THEN
          IF (EvRec.SendingKey <> Restore) AND
               (EvRec.SendingKey <> Select) THEN
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
 MESSAGE: Following node "engulfed"
 COMMAND? find FindIdentifiers
```

ENGULF NEXT

6.2m: Another statement is engulfed.

This is a particularly effective method for engulfing short sublists. Only one selection must be made, it is easy to determine precisely what is being engulfed, and the process is efficient and error resistant in that the same key sequence is repeatedly struck. For longer lists it might be preferable to select and move the sublist as a whole, since the ENGULF approach is linear in the number of elements, whereas sublist selection takes, as a first approximation, constant time. A precise determination of the sublist length where the two methods require equivalent effort is not possible, since the act of locating the remote terminus of the sublist and moving the cursor to it is *not* independent of sublist length. Under some circumstances, it may be advantageous to engulf even fairly large sublists.

Example - Eject with Associated Node

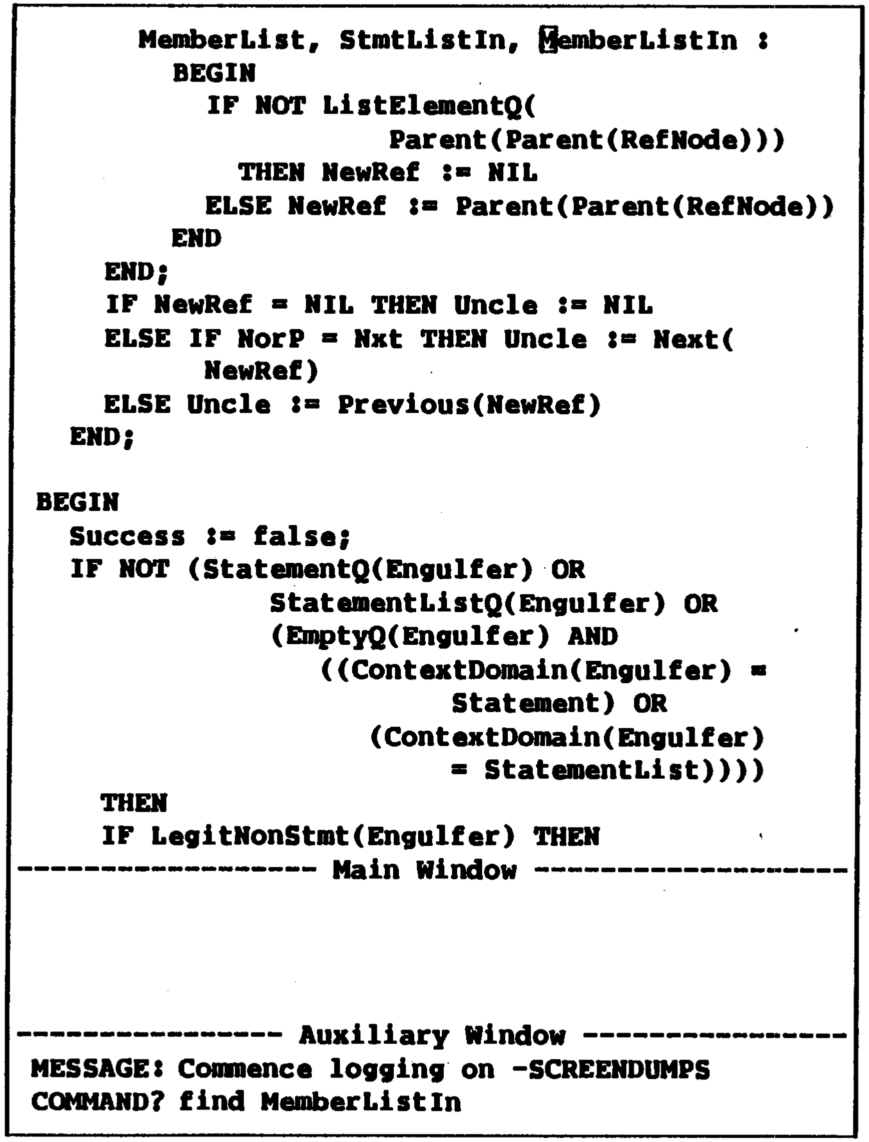
ENGULF/EJECT is a command family with a particularly wide range of applicability. Here it is seen to set up the desired alteration. Note also the incidental use of EMBED.

This example involves alterations to a case statement as well, but, whereas in the previous instance the case statement was, in a sense, simplified, here the case becomes a little more complex. The problem is that the case of `MemberListIn` (designated by the screen cursor in Figure 6.3a, left screen) requires processing which is different from, though related to, that of its list-mates. Recall that EJECT, when applied to such a list with an associated node (or a member of such a list), causes a list member to be split out in combination with the associated node. This is precisely what is required here. Figure 6.3a shows the application of EJECT-FORWARD to `MemberListIn`.

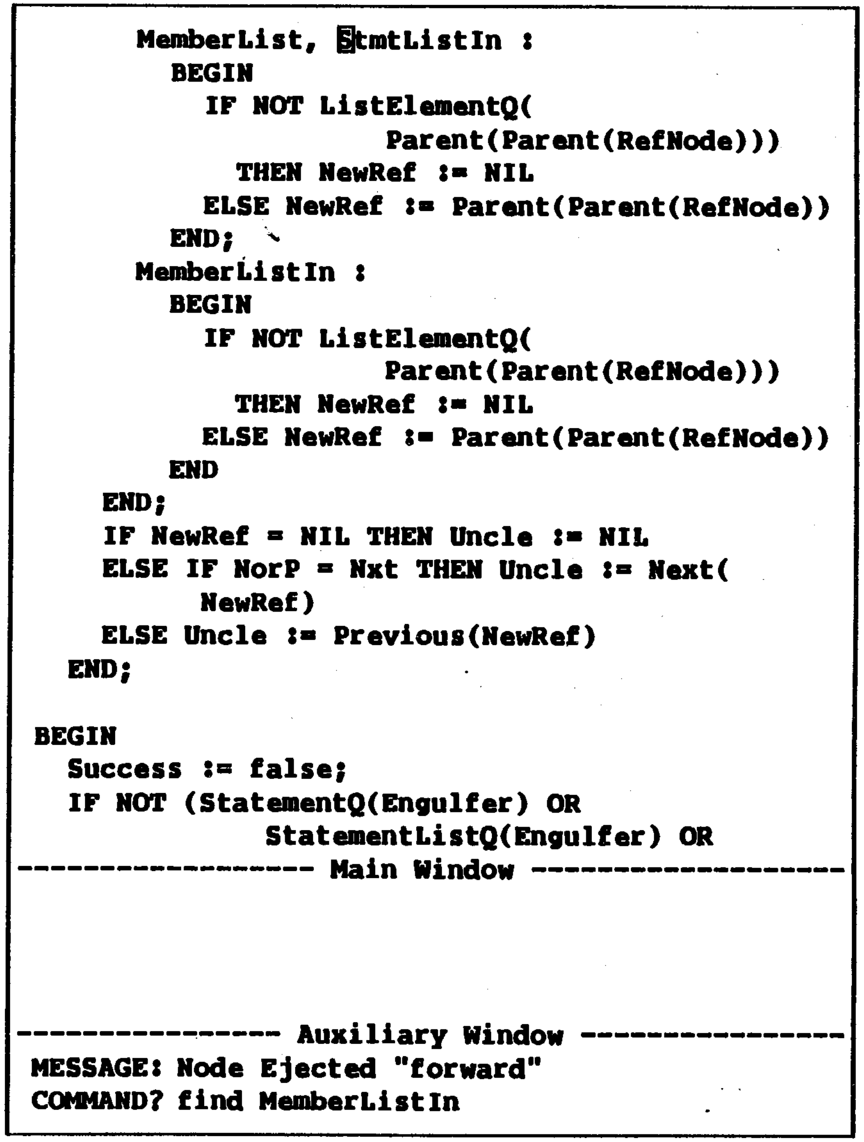
The alternate procedure for carrying out this operation would be essentially the same whether a text editor or primitive SBE commands were used: Duplicate the entire case clause; delete the extraneous cases labels. The EJECT approach certainly saves a few keystrokes. More important, perhaps, the command nicely captures the essence of the desired change to the program's structure and semantics: Treat this instance as a separate case.

Having split out the special case it is now possible to continue the example. The reason that different handling is required in the `MemberListIn` case is that there is an added level of nesting around `RefNode` (I am now referring to the internal structure of the target program). The nested calls to `Parent` in both the predicate of the if statement and its alternate statement must be embedded in an additional call to `Parent`. To accomplish this, the screen cursor is used to select the outermost function call (Figure 6.3b) and EMBED is applied (Figure 6.3c). Since the first choice presented by EMBED is the desired one, it is possible to proceed immediately to textual entry of the function name (Figures 6.3d, 6.3e, 6.3f). In this particular example it would have been possible to have taken advantage of the fact that the instances of the required function call are already present, and to implement the correction with a single SELECT followed by REPLACE. In the course of an actual editing session I would have done so, but for demonstration purposes the more generally applicable invocation of the EMBED command seemed preferable. Though in terms of keystrokes its use is essentially a break-even proposition when compared to simple textual editing, EMBED does offer some advantages. On the practical side parentheses are taken care of, and formatting is automatic. On the conceptual side, the command captures at least the structural essence of the operation: Embed one function call in another.

The example is completed in an opportunistic fashion: The syntactic cursor is used to select the outermost call to `Parent` (Figure 6.3g, 6.3h), the screen cursor is moved to the outermost call to `Parent` in the alternate (Figure 6.3i), and that node is replaced by the correct one (Figure 6.3j).



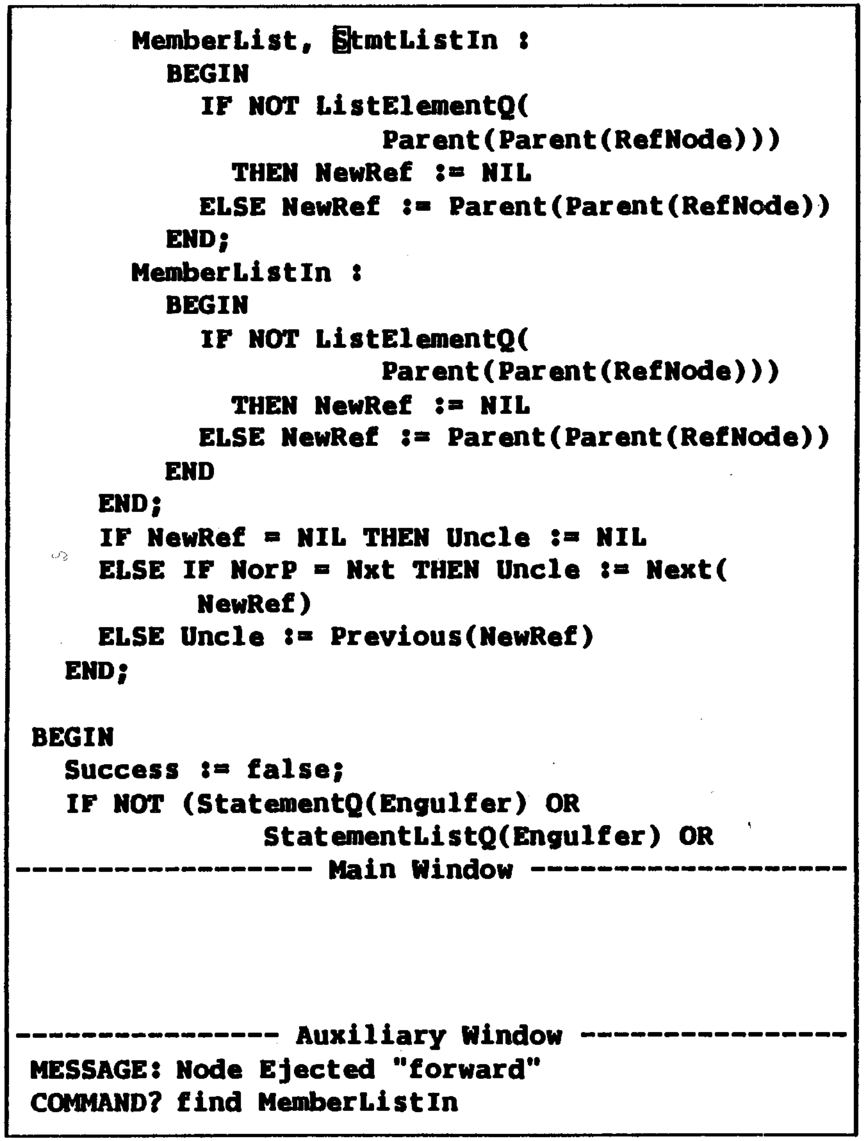
```
   MemberList, StmtListIn, MemberListIn :
    BEGIN
     IF NOT ListElementQ(
           Parent(Parent(RefNode)))
      THEN NewRef := NIL
     ELSE NewRef := Parent(Parent(RefNode))
    END
  END;
  IF NewRef = NIL THEN Uncle := NIL
  ELSE IF NorP = Nxt THEN Uncle := Next(
     NewRef)
  ELSE Uncle := Previous(NewRef)
 END;

BEGIN
 Success := false;
 IF NOT (StatementQ(Engulfer) OR
       StatementListQ(Engulfer) OR
       (EmptyQ(Engulfer) AND
         ((ContextDomain(Engulfer) =
             Statement) OR
          (ContextDomain(Engulfer)
             = StatementList))))
  THEN
  IF LegitNonStmt(Engulfer) THEN
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: Commence logging on -SCREENDUMPS
COMMAND? find MemberListIn
```



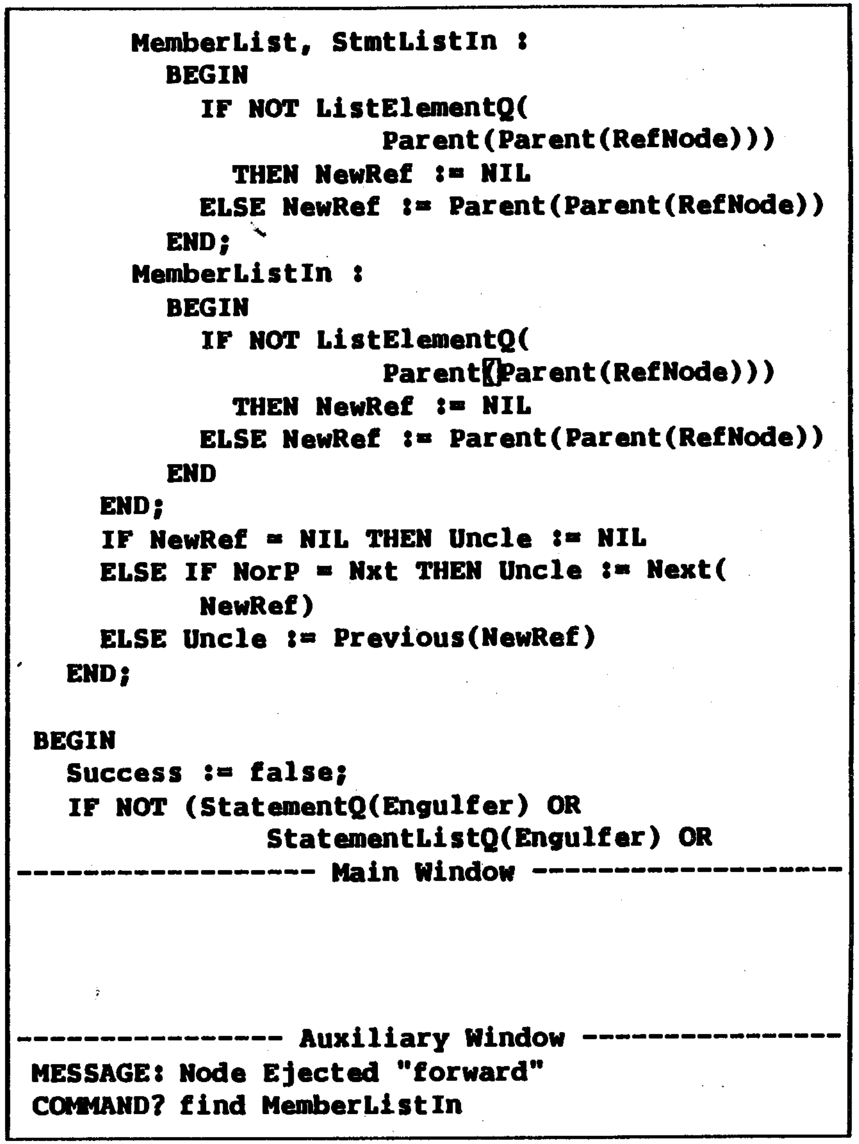
```
   MemberList, StmtListIn :
    BEGIN
     IF NOT ListElementQ(
           Parent(Parent(RefNode)))
      THEN NewRef := NIL
     ELSE NewRef := Parent(Parent(RefNode))
    END;
   MemberListIn :
    BEGIN
     IF NOT ListElementQ(
           Parent(Parent(RefNode)))
      THEN NewRef := NIL
     ELSE NewRef := Parent(Parent(RefNode))
    END
  END;
  IF NewRef = NIL THEN Uncle := NIL
  ELSE IF NorP = Nxt THEN Uncle := Next(
     NewRef)
  ELSE Uncle := Previous(NewRef)
 END;

BEGIN
 Success := false;
 IF NOT (StatementQ(Engulfer) OR
       StatementListQ(Engulfer) OR
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: Node Ejected "forward"
COMMAND? find MemberListIn
```

EJECT FORWARD

6.3a: The `MemberListIn` element is split off.

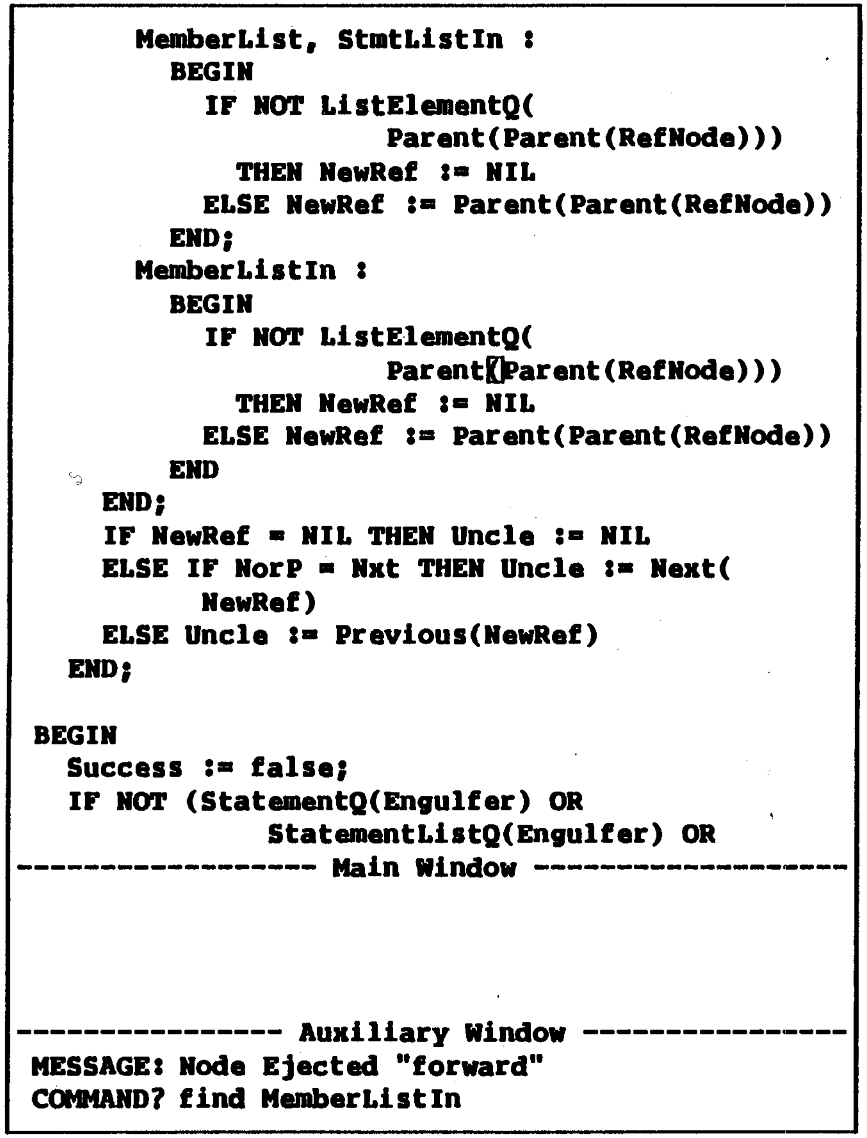


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
            StatementListQ(Engulfer) OR
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
MESSAGE: Node Ejected "forward"
COMMAND? find MemberListIn
```

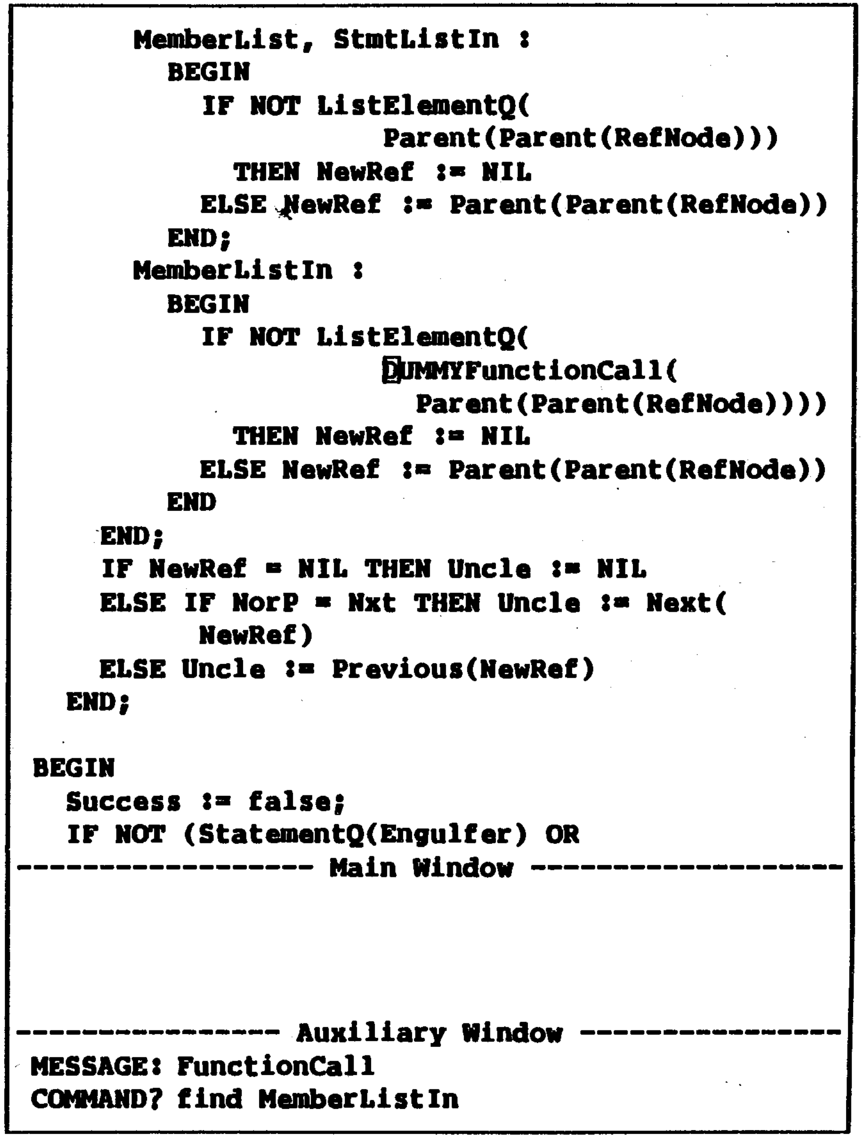


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
            StatementListQ(Engulfer) OR
----------------- Main Window ------------------



---------------- Auxiliary Window ---------------
MESSAGE: Node Ejected "forward"
COMMAND? find MemberListIn
```

6.3b: The screen cursor is moved to a function call.



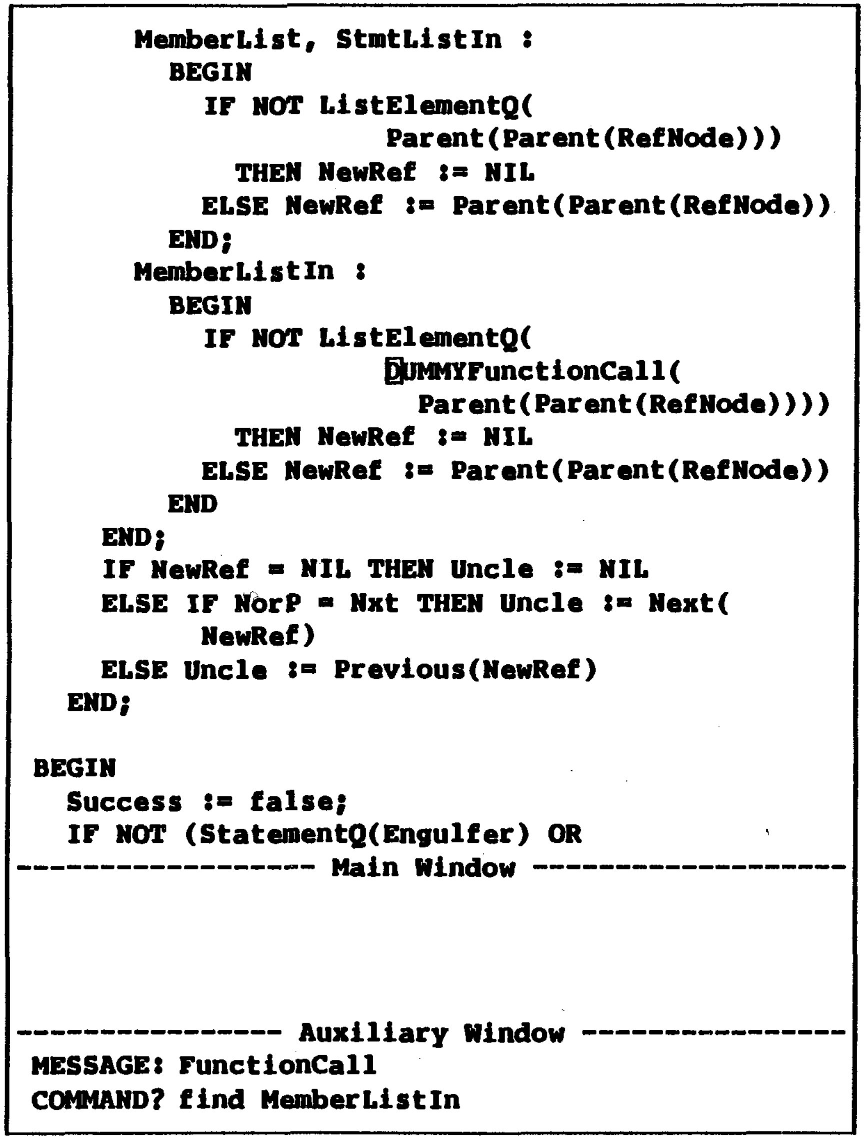
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
              StatementListQ(Engulfer) OR
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: Node Ejected "forward"
COMMAND? find MemberListIn
```



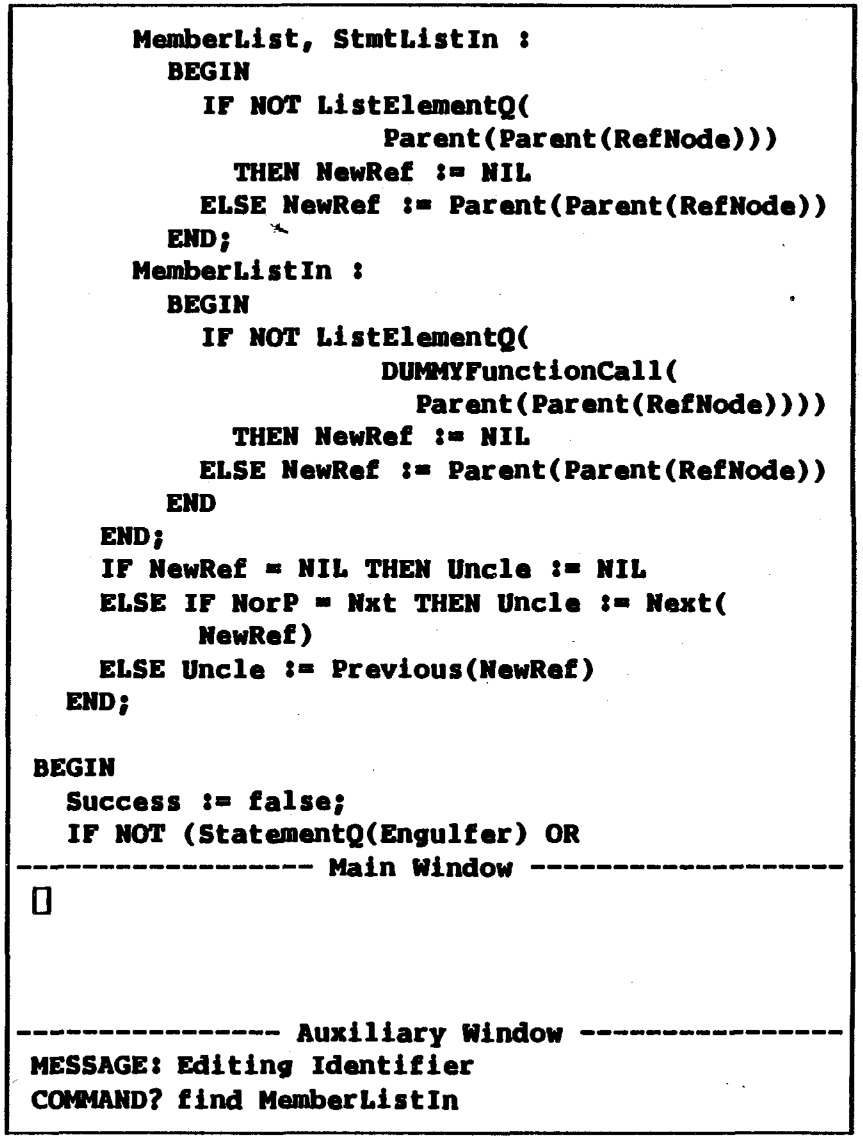
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     DUMMYFunctionCall(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window -------------------



---------------- Auxiliary Window ----------------
MESSAGE: FunctionCall
COMMAND? find MemberListIn
```

EMBED

6.3c: The function call is embedded in a function call.



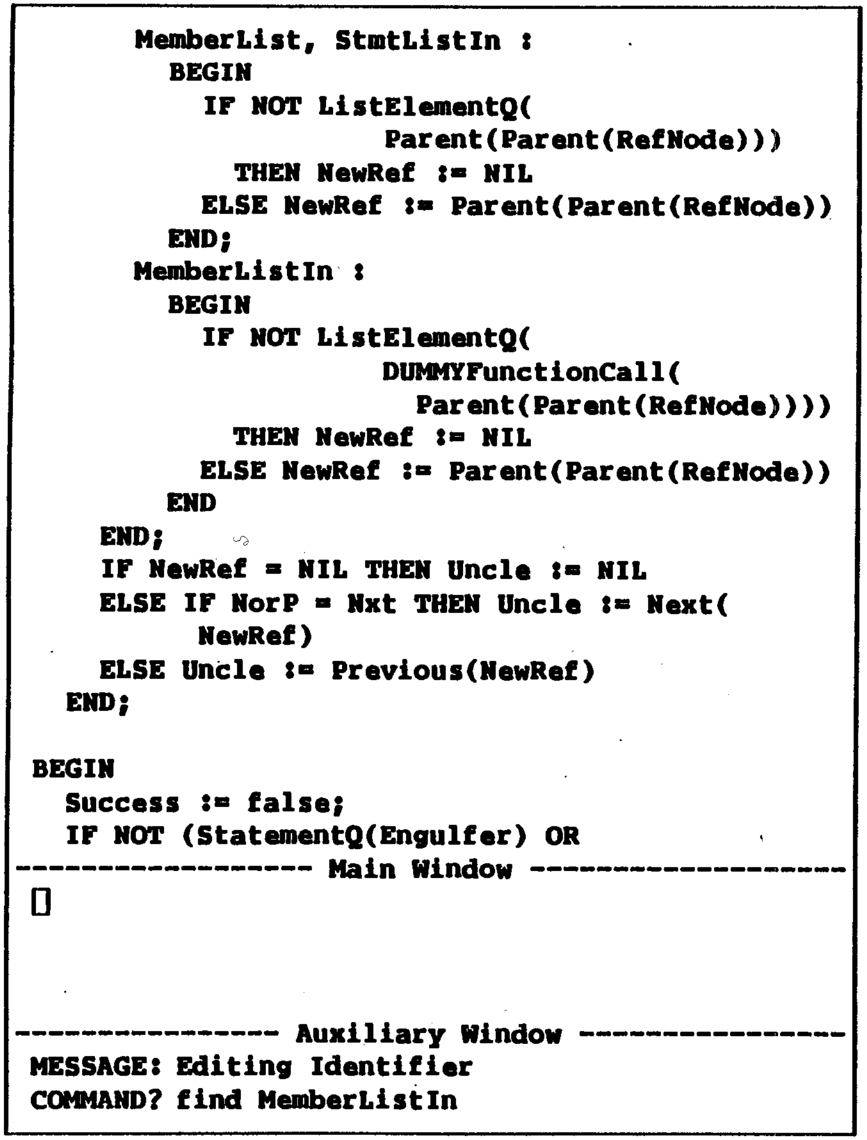
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     DUMMYFunctionCall(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
----------------- Main Window ------------------




---------------- Auxiliary Window ---------------
MESSAGE: FunctionCall
COMMAND? find MemberListIn
```



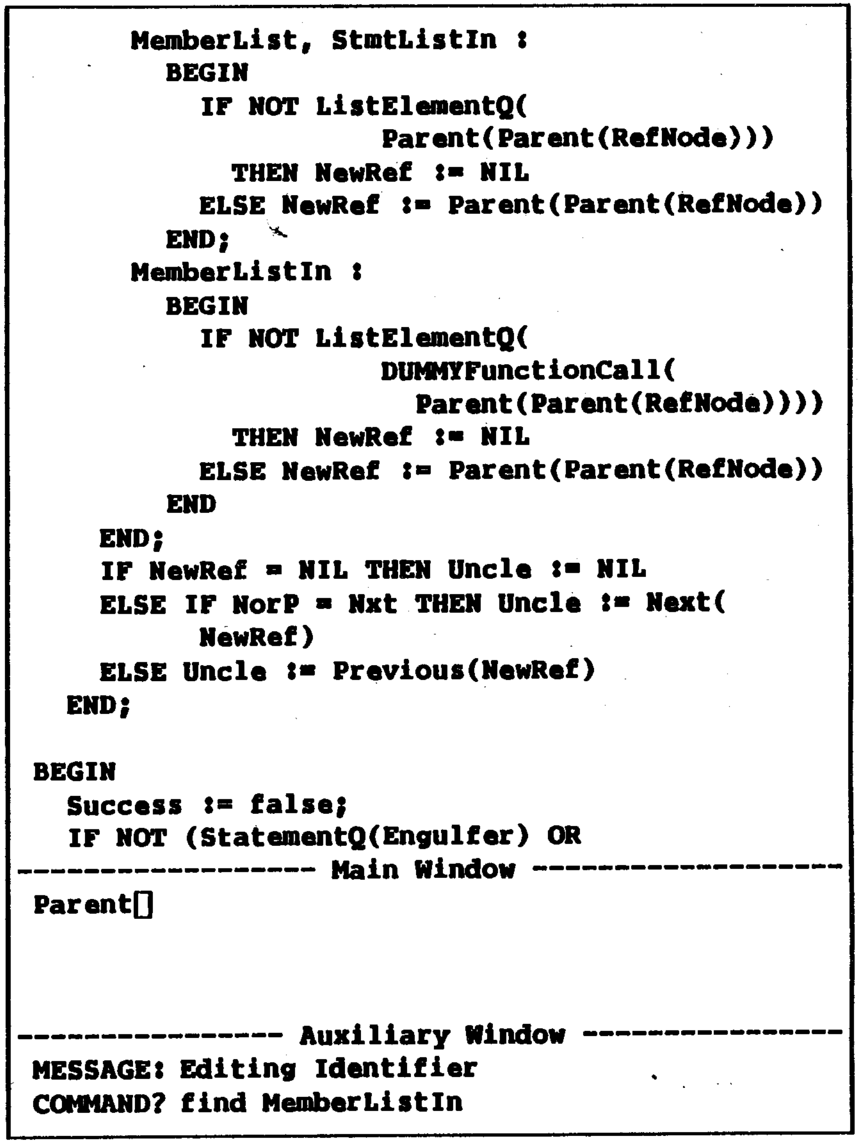
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     DUMMYFunctionCall(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
----------------- Main Window ------------------
□



---------------- Auxiliary Window ---------------
MESSAGE: Editing Identifier
COMMAND? find MemberListIn
```

MODIFY CURRENT

6.3d: The onboard editor is called.

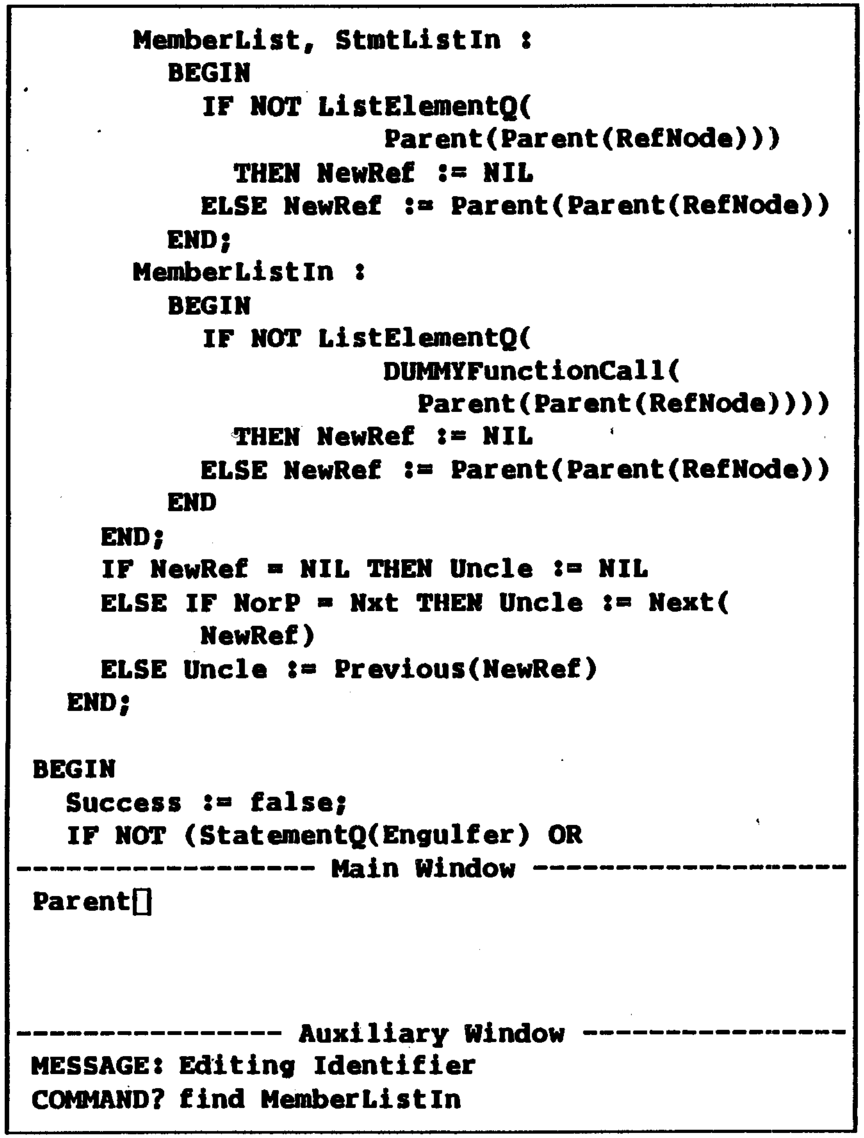


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     DUMMYFunctionCall(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
----------------- Main Window ------------------
[]



---------------- Auxiliary Window ---------------
MESSAGE: Editing Identifier
COMMAND? find MemberListIn
```

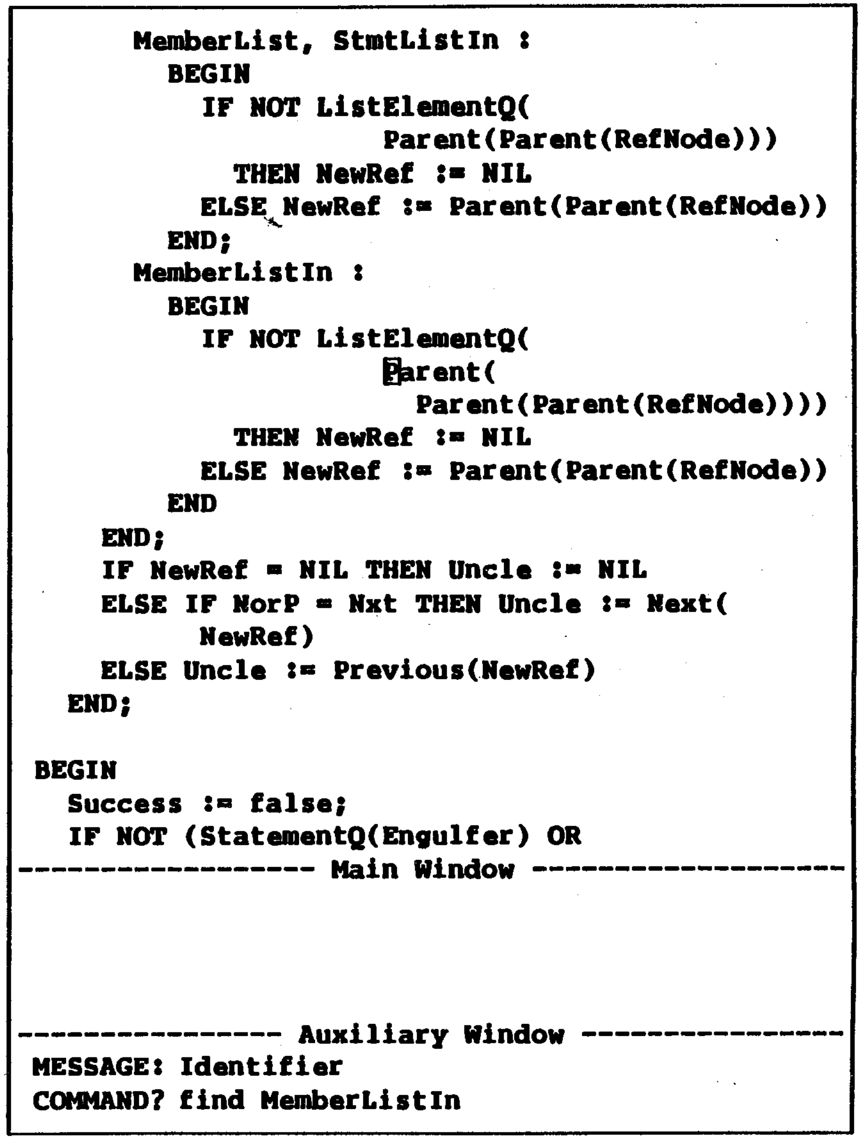


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     DUMMYFunctionCall(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
----------------- Main Window ------------------
Parent[]



---------------- Auxiliary Window ---------------
MESSAGE: Editing Identifier
COMMAND? find MemberListIn
```

6.3e: The function name is typed.



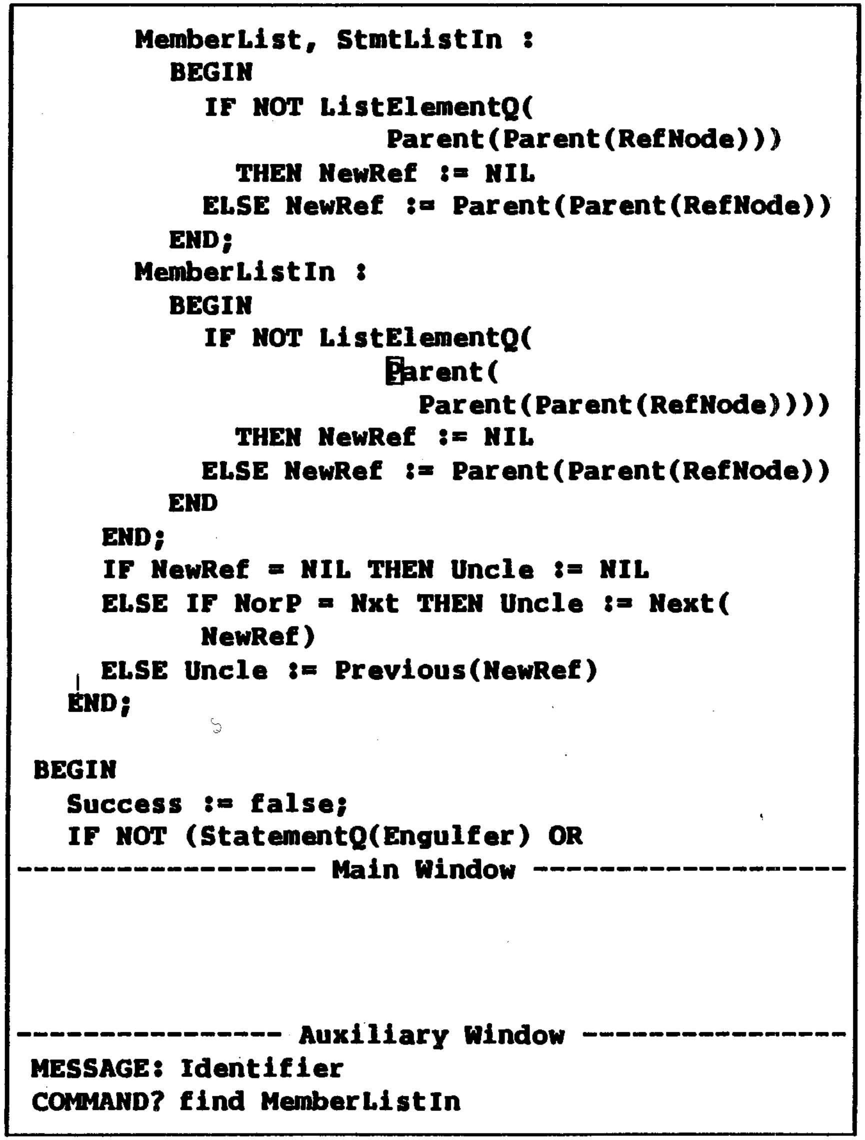
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     DUMMYFunctionCall(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window ------------------
Parent[]



---------------- Auxiliary Window ----------------
MESSAGE: Editing Identifier
COMMAND? find MemberListIn
```



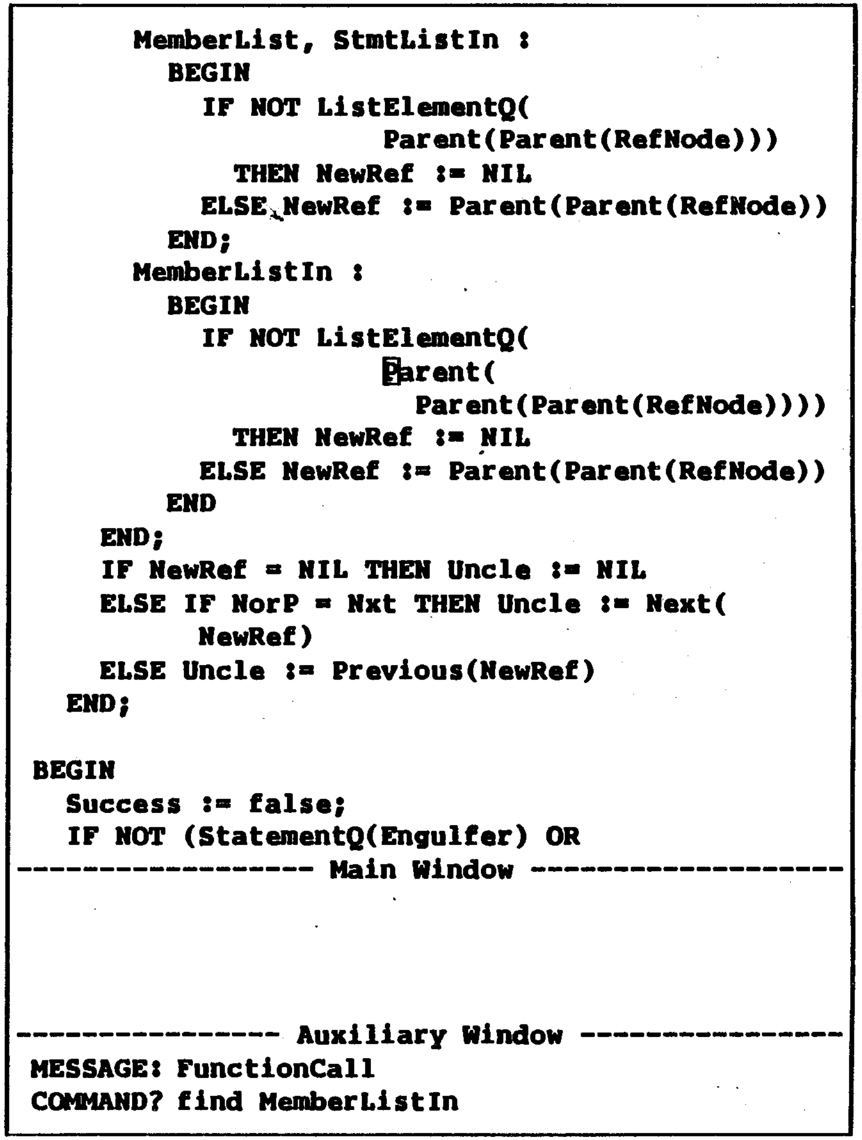
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: Identifier
COMMAND? find MemberListIn
```

EDIT COMPLETE

6.3f: The placeholder is replaced by the function name.



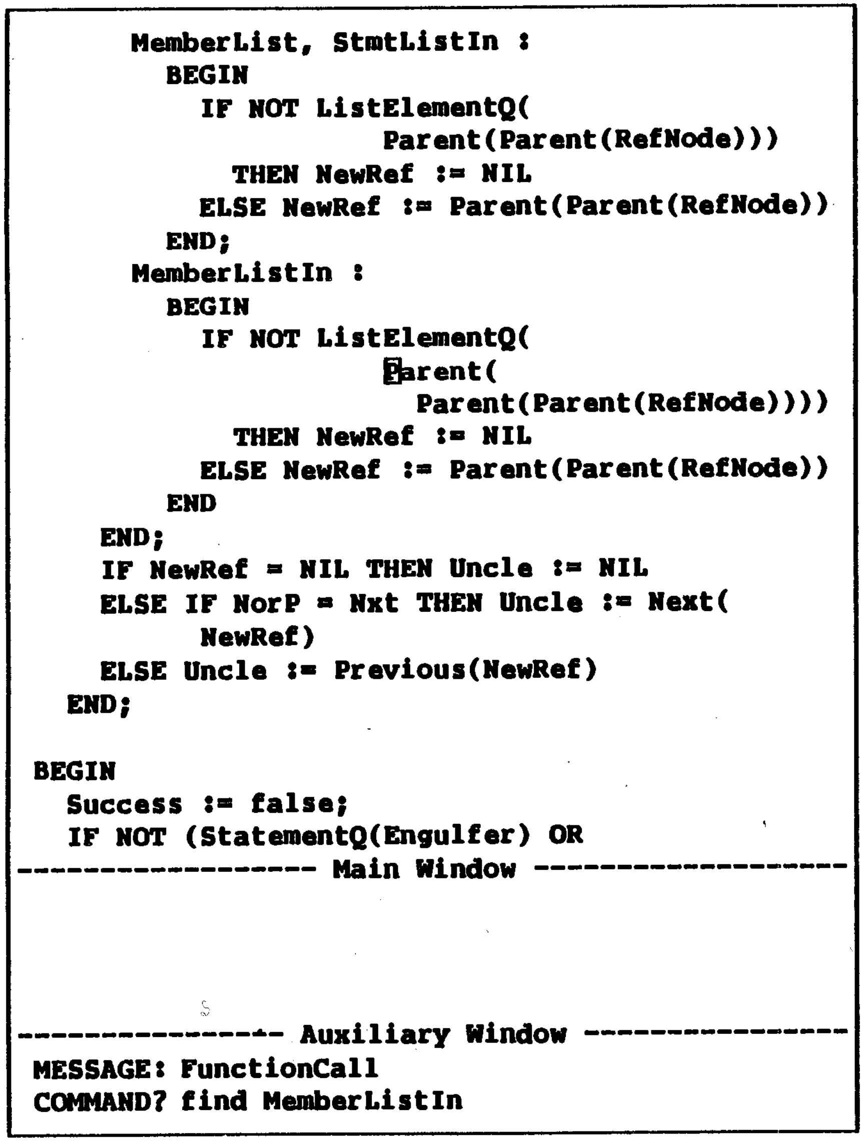
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window -------------------




---------------- Auxiliary Window ----------------
MESSAGE: Identifier
COMMAND? find MemberListIn
```



```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window -------------------




---------------- Auxiliary Window ----------------
MESSAGE: FunctionCall
COMMAND? find MemberListIn
```

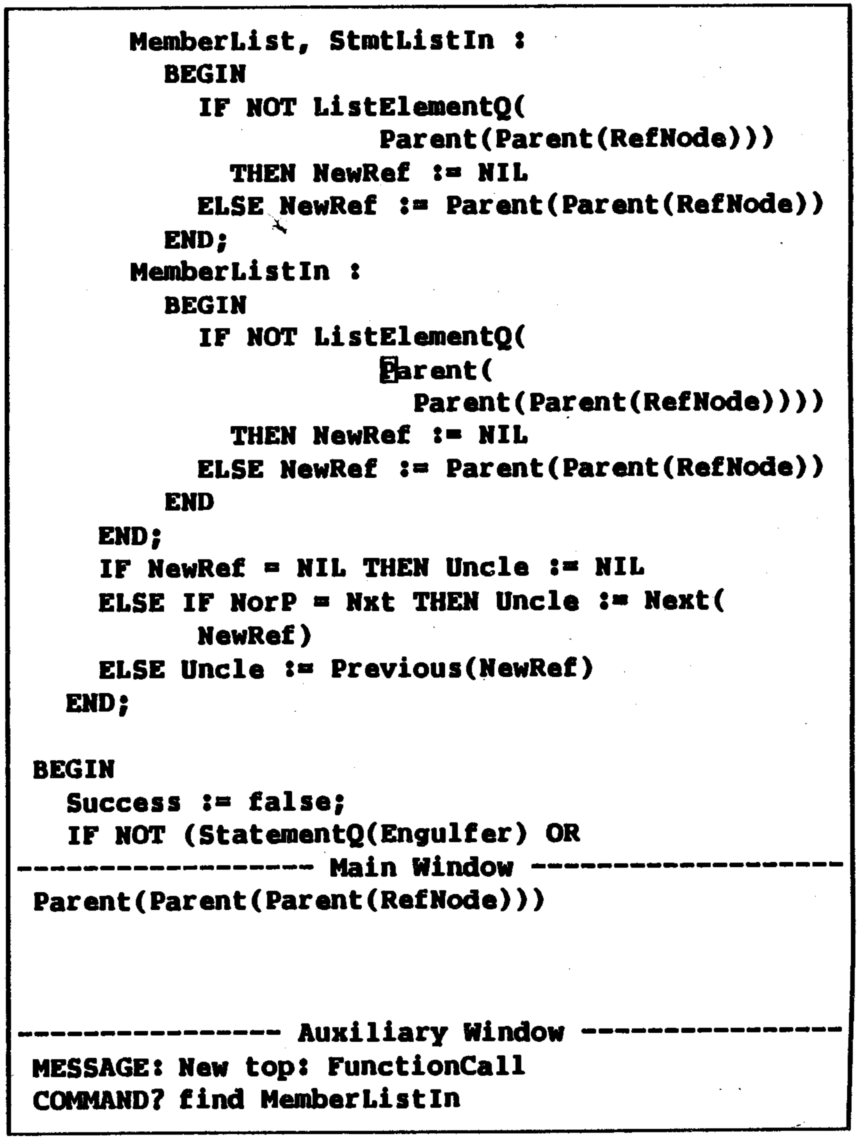
CURSOR UP

6.3g: Syntactic cursor is moved to outermost call to `Parent`.



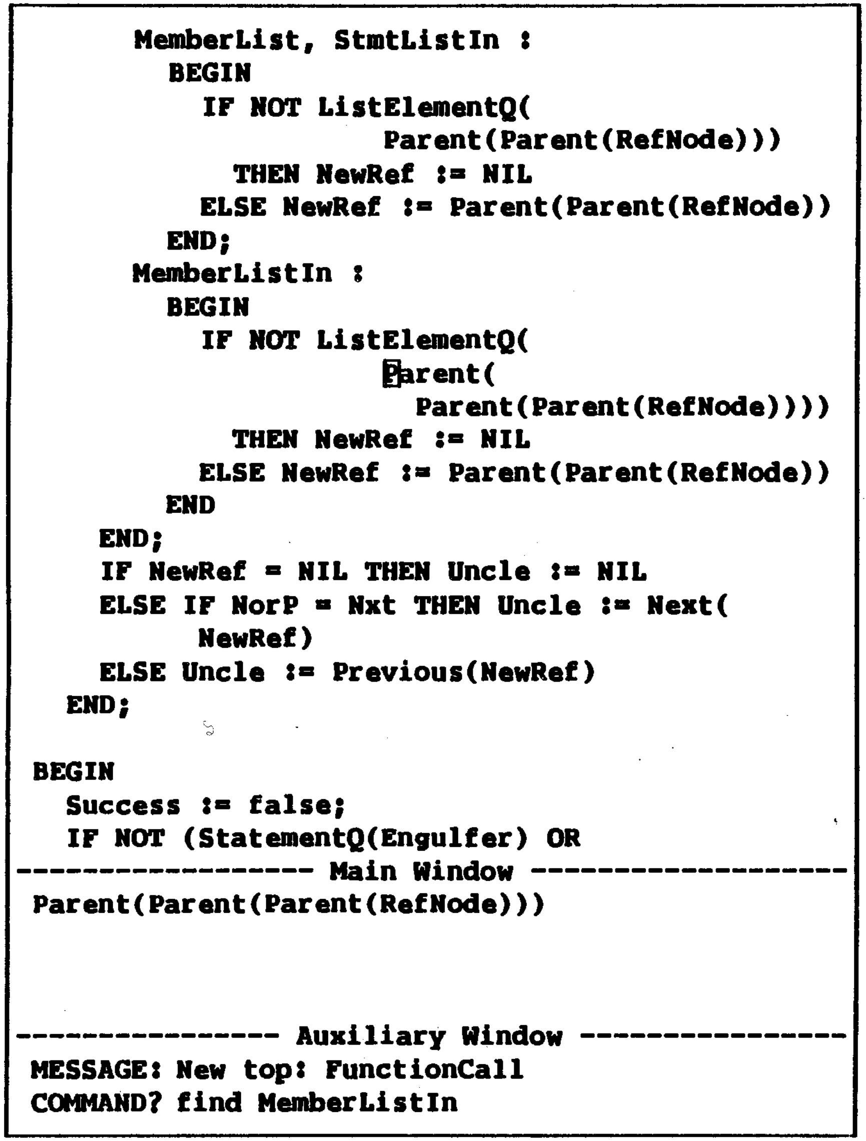
```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window ------------------




--------------- Auxiliary Window ---------------
MESSAGE: FunctionCall
COMMAND? find MemberListIn
```

SELECT

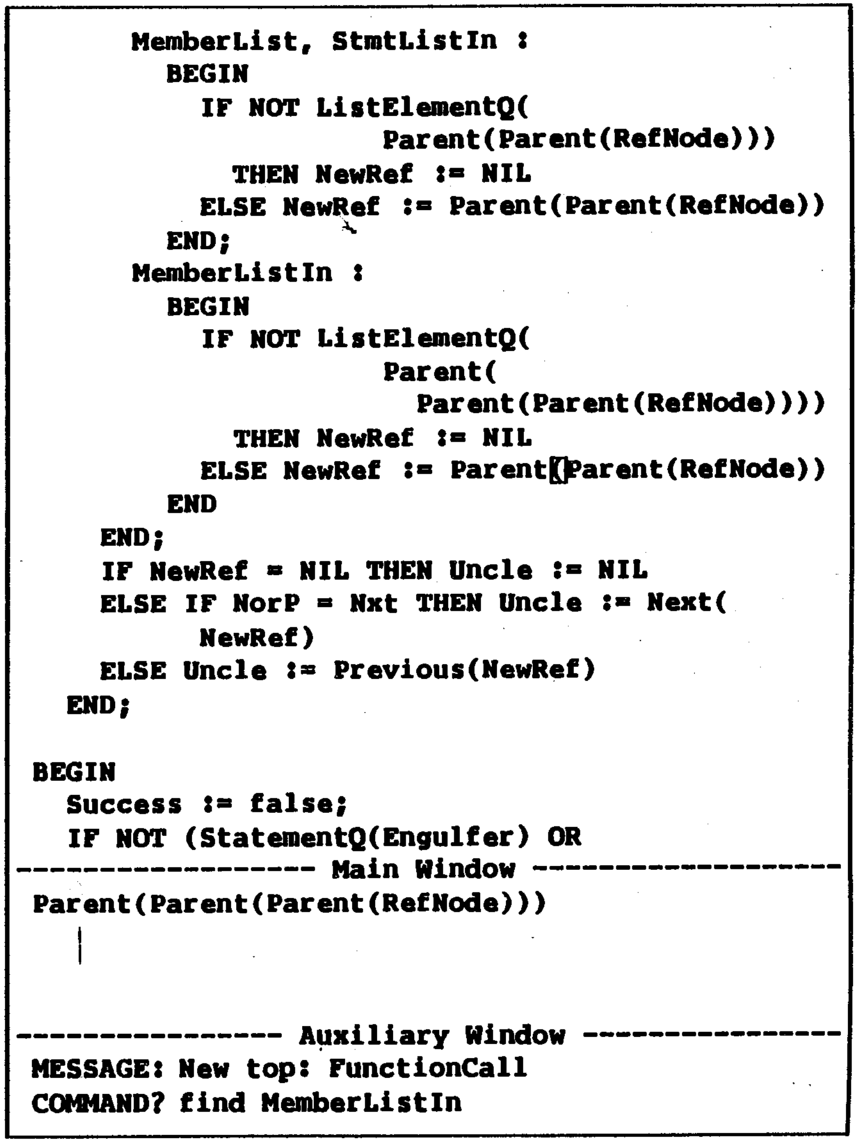


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window ------------------
Parent(Parent(Parent(RefNode)))



--------------- Auxiliary Window ---------------
MESSAGE: New top: FunctionCall
COMMAND? find MemberListIn
```

6.3h: The node is stacked.

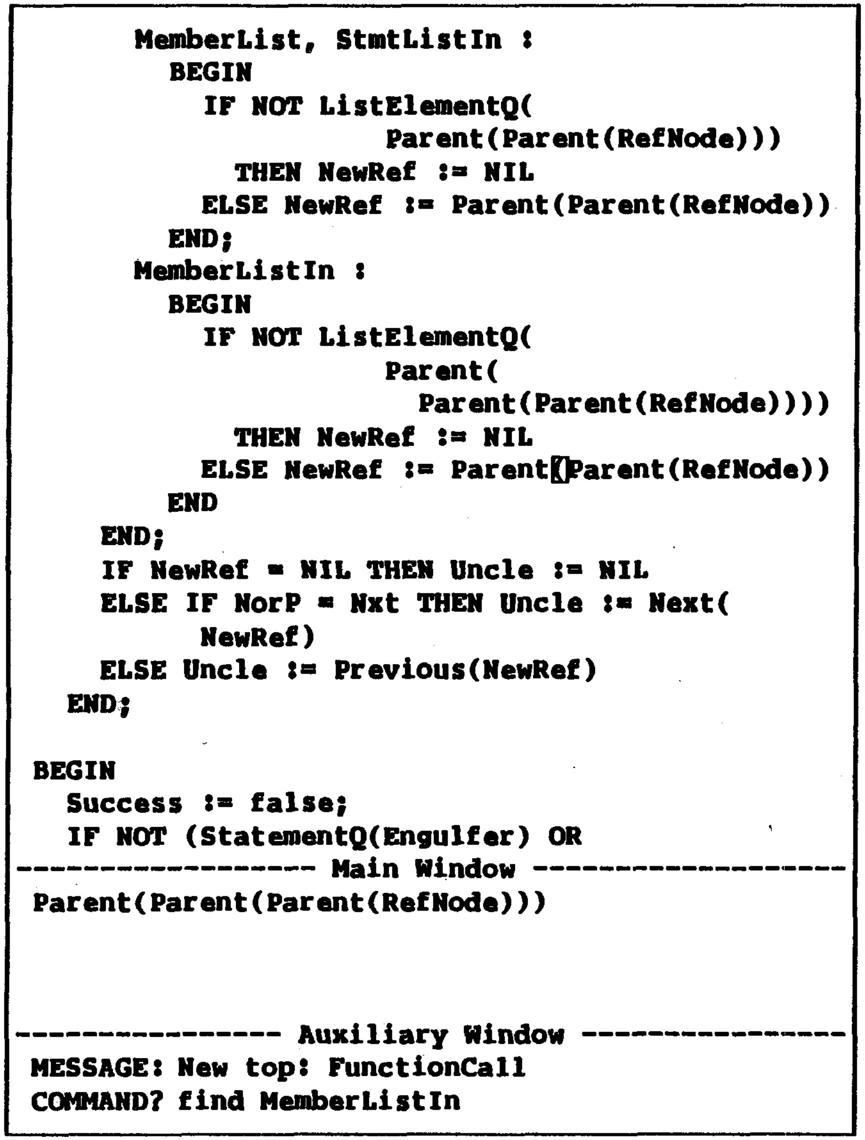


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window -------------------
Parent(Parent(Parent(RefNode)))



---------------- Auxiliary Window ----------------
MESSAGE: New top: FunctionCall
COMMAND? find MemberListIn
```

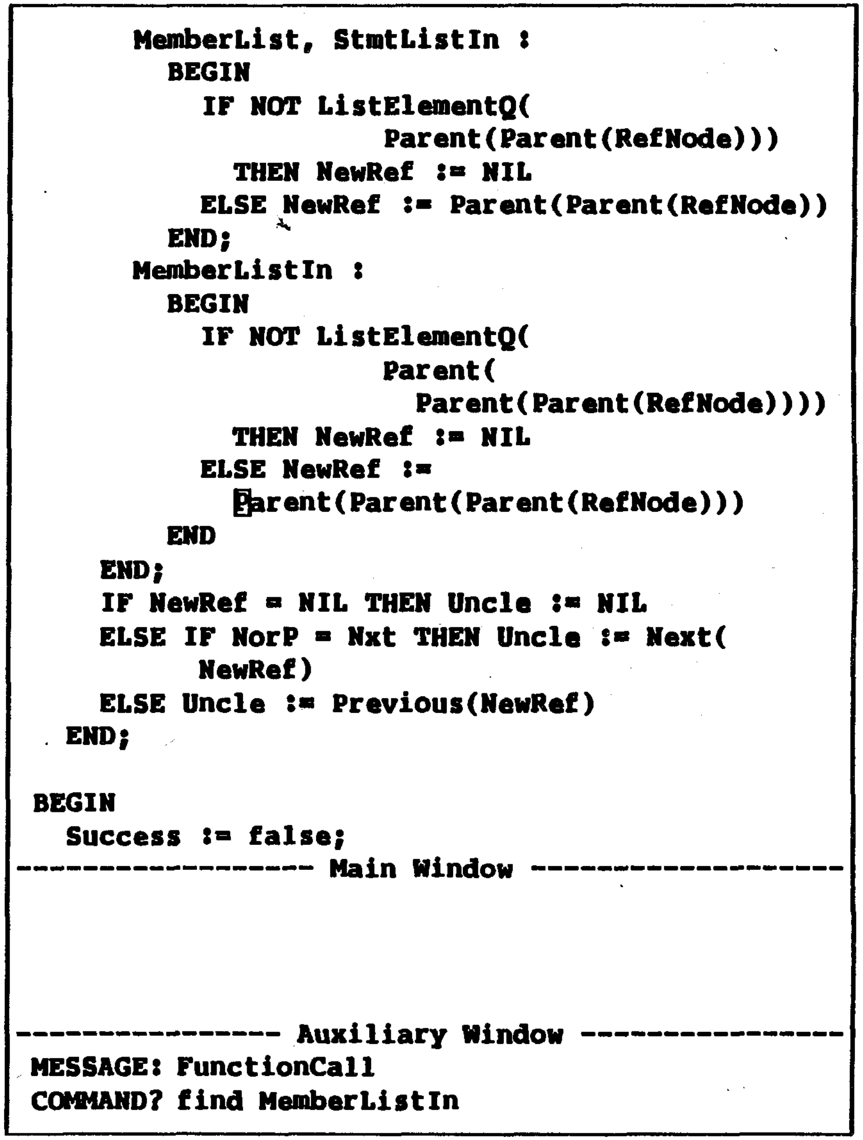


```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window -------------------
Parent(Parent(Parent(RefNode)))
   |


---------------- Auxiliary Window ----------------
MESSAGE: New top: FunctionCall
COMMAND? find MemberListIn
```

6.3i: Screen cursor is moved to the other erroneous node.



```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
  IF NOT (StatementQ(Engulfer) OR
------------------ Main Window ------------------
Parent(Parent(Parent(RefNode)))



---------------- Auxiliary Window ----------------
MESSAGE: New top: FunctionCall
COMMAND? find MemberListIn
```



```
      MemberList, StmtListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(Parent(RefNode)))
            THEN NewRef := NIL
          ELSE NewRef := Parent(Parent(RefNode))
        END;
      MemberListIn :
        BEGIN
          IF NOT ListElementQ(
                     Parent(
                       Parent(Parent(RefNode))))
            THEN NewRef := NIL
          ELSE NewRef :=
            Parent(Parent(Parent(RefNode)))
        END
    END;
    IF NewRef = NIL THEN Uncle := NIL
    ELSE IF NorP = Nxt THEN Uncle := Next(
          NewRef)
    ELSE Uncle := Previous(NewRef)
  END;

BEGIN
  Success := false;
------------------ Main Window ------------------



---------------- Auxiliary Window ----------------
MESSAGE: FunctionCall
COMMAND? find MemberListIn
```

REPLACE

6.3j: The previously selected node replaces the erroneous one.

CHAPTER VII
IN CONCLUSION

An Assessment

It is now time to assess the project, to see how far it has gone toward reaching the main goals set out in Chapter III. How good is the editor? How useful are the manipulative commands? Has a step been taken toward a more effective editing style or technique?

The editor, *qua* editor, is a somewhat unevenly developed tool. The functionality of what I have called the editing facility seems quite rudimentary when compared to that of some other implementations, particularly in the realms of code production and syntax-directed aids to inspection and selection (browsing, holophrasting, etc.). What is provided is the basic capacity to edit programs (and program fragments) with a structural or syntax-based approach. Since the real motivation for its construction was the desire to create a milieu in which to develop and examine what I have called the manipulative facility, a job for which it has proven adequate, the editor itself may be considered a qualified success.

I have to an extent already addressed the issue of the effectiveness of the manipulative facility. In addition to description, Chapter V set out some of the rationale for the commands of the facility. In the discussion accompanying the examples of Chapter VI, I pointed out some of the strong points of the facility. Here I will recapitulate the more relevant of those arguments.

All else being equal, it is desirable to reduce the number of keystrokes required for the performance of a given operation. Typing takes time and effort and invites error. Not surprisingly, the use of the commands under discussion typically results in keystrokes saved. Those commands which are implementable in terms of the basic syntax-based operations offer keystroke savings which are large in percentage terms, though generally small in numeric terms. Some operations (EMBED and the transformational family) cannot be performed using just the basic commands. They can be performed textually. Here the savings may be minimal for simple embedding followed by textual entry, but for more complex operations involving multiple commands, the savings become significant. The transformation commands tend to save a great deal of retyping.

A part of the savings in keystrokes is attributable to what might be termed the recycling of code. Often, in the course of textually editing a program, one would like to extract a significant fragment of code, which may be textually embedded in other code or whose limits are not readily ascertainable, and insert it elsewhere. In general, SBE's provide this capability. The new commands frequently enhance this capability by providing a framework for the insertion of such fragments (EMBED) and by providing more convenient methods for accomplishing such operations (ROTATE/SWAP, ENGULF/EJECT). The example cited previously of the application of INLINE to a case statement demonstrates how operations of the transformational family may assist in the reuse of existing code. In fact the transformational commands may in general be thought of as recycling code, in that they consume the old code before regurgitating it in the transformed state.

Some textual entry will always be necessary. Here the contribution of the commands is rationalization of the operation. Details, including keyword production, bracketing, punctuation and formatting, are handled by the system. Templates are provided so that the purpose of the textual entry is at all times clear. The amount of textual entry is minimized, and it takes place at particular places for particular reasons which are related to the *programming* problem at hand.

The higher level commands do seem to have succeeded, to a degree, in bringing into correspondence the statement of problems of program alteration and their solution. They address such perceived problems as:

This variable declaration deserves special attention.

This statement should be executed repeatedly until some condition arises.

The logic of this predicate is incomplete.

This expression may be simplified.

And they generally do so in a direct and reasonable manner, e.g., "This statement [place cursor] should be executed if [EMBED in the if statement], and only if, this condition [type in predicate] holds." I submit that in program editing, as in the programming process generally, it is advantageous to stay close to the problem of interest, avoiding entanglement in detail wherever possible.

All of the foregoing contribute to that elusive state or attribute: programmer satisfaction. Even skillful typists do not enjoy wasting keystrokes and should welcome any reasonable means to avoid doing so, whether it be the enhanced capability to reuse existing fragments of code, or the partial automation of code production. Moreover, as professional problem solvers, programmers would also be expected to welcome any editing aid which helps to keep their concentration focused upon their real job.

Given that the commands are effective when applied, how widely applicable are those commands? Unfortunately, I do not have an empirically based answer to this question. I can only speculate from the basis of my own experience. In the course of "bootstrapping" the editor, I have certainly had occasion to use the higher level commands of the facility. I cannot claim that they have come close to supplanting the lower level operations. It seems that everyday editing will always involve a great deal of textual entry, and that the simple syntax-based commands will be sufficient for the performance of many operations, but, as I have pointed out in Chapter VI, it is often the new commands which enable the effective use of the simpler ones.

I have personally found the facility useful, but I am one programmer, working in a single applications domain, and dealing with a program in a particular stage of its development. Will others, working in different domains under different sets of circumstances, find this to be so? There are reasons to believe that this will be the case.

To begin with, the operations have been selected and organized in accordance with a not-unreasonable classification of the types of program manipulation activities. Programmers do manifestly seek to effect changes in their code by altering the sequencing of elements and by manipulating the nesting of program structures. These operations in combination with more basic operations are often aimed at bringing about identifiable transformations. The command families which implement these operations have been developed with the explicit intent of achieving a degree of general applicability.

A second encouraging factor is the extensive functionality of the facility. Though few in number, the individual commands have been designed to be applicable to a variety of node types in varying contexts. The sequencing operations of the ROTATE/SWAP family may be applied to sequences or lists of all types, and the notion has been extended to list-like objects as well. Commands of the ENGULF/EJECT family work on all nested statements and statement lists, plus a variety of nested non-statement lists. Particularly notable is the extension to lists with associated nodes (variable declarations and their ilk). EMBED may be applied to any statement or expression, providing a full set of embeddings in the former case and an extensive set of embeddings (both logical and arithmetic) in the latter. Universal application has been proposed for EMBED/STRIP. Finally, the transformational family has been extended beyond what might be termed the classical transformations with proposals for application of INLINE to statements other than procedure calls and to data structure definitions. All of this increases the likelihood that a working set of operations will be found to suit a particular programming style, applications domain, and life-cycle situation.

Finally, a measure of functional redundancy has been built into the system. There is often more than one way to perform a task. The editing samples previously presented suggest examples. An item may be moved to the head of a list by multiple rotations of the entire list, by finding the item and then either rotating it to the head or swapping it with the head item, or, quite possibly, by other means. Sublists may frequently be handled either by means of the basic sublist commands or by repeated invocations of ENGULF. One typically has a range of options for filling EMBED-created placeholders: insertion of a previously selected (or created) node, textual entry, engulfing, and (potentially) template expansion. This flexibility admits variations in personal style and programming context.

Is there a possibility that this will lead to new and better ways of altering programs, in a manner analogous to the way in which the advanced inspection/selection capabilities of other SBE's have led away from arbitrarily linear ways of examining programs? I have found that in particular circumstances problems can be solved in ways which are not only more efficient, but, in a sense, more meaningful as well. Code is manipulated on the basis of its structural organization, and, in some cases, in ways which relate quite directly to the accompanying semantics. The generalization of this orientation is, I suspect, inhibited by the necessity for low level text entry. Although this cannot be eliminated, it could, perhaps, be rationalized by implementation of a template expansion facility, together with some advanced inspection/selection features borrowed from other syntax-based implementations.

Suggestions for future work

A successful approach to the development of interactive tools has been the iterative one [Bro77, Woo81]. A prototype is implemented. A body of users is invited to make use of it. Based upon the experiences of those users, a new version is implemented, and the process is repeated until there is a nice fit between user requirements and the functionality of the tool. I have created a facility for syntax-based editing environments which is interesting, potentially useful, and possibly influential. The next step in its development would seem to be exposure over an extended period of time to a variety of users, preferably working at a variety of programming tasks. It would be relatively easy to build into the editor means for monitoring the use of its features to determine which commands were being used and which were not and under what circumstances. It might even be possible to discover frequently used sequences of operations which would constitute candidates for inclusion in the next generation of commands. Consultation with users would help to determine why certain operations were being used and others not, and would, of course, reveal operations which should have been available but were not. Given the editor's capacity for extension, users who were familiar with the underlying MPS system, or willing to become familiar, could contribute directly to the enhancement of the editor's capabilities.

Unfortunately, the current implementation is probably not attractive enough to generate widespread use. Above and beyond the enhancements and extensions already suggested, some major implementation efforts are in order:

1. A more acceptable environment - The MTS system is no longer being heavily used by computing science researchers as a programming environment. Reimplementation of the editor on the departmental research system (for example) would be beneficial.
2. A more sophisticated interface - In a modern workstation environment it should be possible to construct a more helpful and attractive

interface.

3. More SBE features - Implementation of some of the well-researched syntax-based features of other editors (browsing and code generation, for example) would make the editor as a whole more powerful. I suspect that they would enhance the usefulness of the manipulative facility as well.

I have claimed that the new manipulative facility has the potential to make program editing easier and more enjoyable. Embedding the facility in such a supportive environment would test that claim and, perhaps, expand the potential.

APPENDIX - SUMMARY OF COMMANDS

Manipulations

Basic Operations

INSERT-BEFORE

Insert top node at the position immediately preceding that of the current node.

INSERT-AFTER

Insert top node at the position immediately following that of the current node.

REPLACE

Replace the current node in its context with the top node.

DELETE

Delete the current node.

DELETE-SUBLIST

Delete the sequence of elements bounded by the current node and the top node.

MODIFY

Call the on-board text editor for the modification of the current node.

Alterations to Sequencing

ROTATE-FORWARD

Rotate list elements forward. (Last element becomes the first.)

ROTATE-BACKWARD

Rotate list elements backward. (First element becomes the last.)

SWAP-WITH-TOP

Exchange the current node and the top node.

SWAP-NEXT

Exchange the current node and its immediate successor.

SWAP-PREVIOUS

Exchange the current node and its immediate predecessor.

Alterations to nesting

EMBED

Replace the current node in its context with a template in which the current node has been embedded as a component. NEXT key is used to view alternative embeddings.

STRIP

Strip away a level of nesting from the current node.

ENGULF-NEXT

Move the enclosing structure's sequential successor into the enclosed structure.

ENGULF-PREVIOUS

Move the enclosing structure's sequential predecessor into the enclosed structure.

EJECT-FORWARD

Move element of the enclosed structure to position immediately following the enclosing structure.

EJECT-BACKWARD

Move element of the enclosed structure to position immediately preceding the enclosing structure.

Transformations

INLINE

Replace node with a more explicit, semantics-preserving encoding.

SIMPLIFY

Replace expression or statement with a transformation which embodies a logical or arithmetic simplification.

PROPAGATE

Given an assignment, replace the next occurrence of the variable on the left hand side with an instance of the expression on the right hand side.

Supporting Commands

Traveling

SCROLL-UP

Move window up in textual representation.

SCROLL-DOWN

Move window down in textual representation.

CURSOR-UP

Move syntactic cursor up one level.

CURSOR-DOWN

Move syntactic cursor down one level. (Place on left-most node.)

CURSOR-RIGHT

Move syntactic cursor one element to the right.

CURSOR-LEFT

Move syntactic cursor one element to the left.

MOVE-TO-TOP

Move syntactic cursor to the top node on the node stack.

DEFINING-OCCURRENCE

Move syntactic cursor to the defining occurrence of the current (identifier) node.

Selection and Utilities

SELECT

Push a reference to current node onto the node stack.

SELECT-SUBLIST

Replace top node with a copy of the sublist bounded by the current node and the top node.

SHOW-TOP

Display the node stack's top.

POP

Pop the node stack.

SHOW-STRUCTURE

Display structure of the current node.

EDIT-NEW

Call the on-board text editor for the entry of a new node. (Node class must be provided.)

Typed Commands

PARSE <file> <nodetype> [<name>]

Parse contents of *file*, a syntagm of class *nodetype*, and display in main window, or, optionally, store under *name*.

PRINT < file> [<name>]

Prettyprint node in main window (or node stored under *name*) to *file*.

CHECKPOINT <file> [<name>]

Checkpoint node in main window (or node stored under *name*) to *file*.

RESTORE <file> [<name>]

Read previously checkpointed *file* into main window (or store under *name*).

FETCH <name>

Display copy of node stored under *name* in main window.

STACK <name>

Push copy of node stored under *name* onto stack.

STORE <name>

Store copy of top node under *name*.

MT

Escape to operating system. (Restart is possible.)

$ <MTS-command>

Escape to operating system, execute *MTS-command* and return.

REFERENCES

[ABL84] Alberga, C.N., Brown, A.L., Leeman, G.B., Jr., Mikelsons, M. and Wegman, M.N., "A program developement tool," *IBM Jour. of Res. and Develop.*, 28, 1 (Jan., 1984), 60-73.

[Ada84] Adams, Edward N., "Optimizing preventive service of software products," *IBM Jour. of Res. and Develop.*, 28, 1 (Jan., 1984), 2-14.

[All83] Allison, Lloyd, "Syntax directed program editing," *Software - Prac. and Exper.*, 13, (1983), 453-465.

[AMN81] Atkinson, L.V., McGregor, J.J. and North, S.D., "Context sensitive editing as an approach to incremental compilation," *Computer Journal*, 25, 3 (1981), 222-229.

[Ars79] Arsaç, Jacques J., "Syntactic source to source transforms and program manipulation," *Commun. ACM*, 22, 1 (Jan., 1979), 43-54.

[BaS86] Bahlke, Rolf and Snelting, Gregor, "The PSG System: From formal language definitions to interactive programming environments," *ACM Trans. Program. Lang. Syst.*, 8, 4 (Oct., 1986), 547-576.

[BBS85] Brun, G., Businger, A. and Schoenberger, R., "The token-oriented approach to program editing," *SIGPLAN Not.*, 20, 2 (Feb. 1985), 17-20.

[BSS84] Barstow, David R., Shrobe, Howard E. and Sandewall, Eric, Eds., *Interactive Programming Environments*, McGraw-Hill Inc., 1984.

[BuD77] Burstall, R.M., and Darlington, John, "A transformation system for developing recursive programs," *J. ACM*, 24, 1 (Jan., 1977), 44-67.

[Bro77] Brooks, Frederich P., Jr., "The computer 'scientist' as toolsmith - studies in interactive computer graphics," in *Information Processing 77*, Bruce Gilchrist, Ed., North-Holland Publishing Company (1977), 625-634.

[CaI84] Cameron, Robert D. and Ito, M. Robert, "Grammar-based definition of metaprogramming systems," *ACM Trans. Program. Lang. Syst.* , 6, 1 (Jan., 1984), 20-54.

[Cam86] Cameron, Robert D., *MPS Reference Manual* (draft), 1986

[Cam87] Cameron, Robert D, Personal communication, 1987

[Cap85] Caplinger, Michael, "Structured editor support for modularity and data abstraction," *SIGPLAN Not.*, 20, 7 (July, 1985), 140-147.

[Dar84] Darlington, John, "Program transformation in the ALICE project," in *Program Transformation and Programming Environments*, P. Pepper, Ed., Springer-Verlag, 1984, 347-353.

[DHK84] Donzeau-Gouge, Vèronique, Huet, Gérard, Kahn, Gilles, Lang, Bernard, "Programming environments based on structured editors: The MENTOR experience" in [BSS84], 128-140.

[Gla81] Glass, Robert, L., "Persistent software errors," *IEEE Trans. on Software Engineering*, SE-7, 2 (Mar., 1981), 162-168.

[GMH85] Gustafson, David A., Melton, Austin and Hsieh, Chyuan Samuel, "An analysis of software changes during maintenance and enhancement," *Conference on Software Maintenance* , IEEE Computer Soc. Press, 1985, 92-95.

[HLC80] Hammond, N., Long, J., Clark, I., Barnard, P. and Morton, J., "Documenting human-computer mismatch in interactive systems," *Proceedings of the Ninth International Symposium on Human Factors in Telecommunications*, 1980, 17-24.

[JeW85] Jensen, Kathleen, and Wirth, Niklaus, *Pascal User Manual and Report*, 3e, Springer-Verlag, 1985.

[Lev86] Levenson, Nancy G., "Software safety: Why, what and how," *ACM Comput. Surv.*, 18, 2 (June, 1986), 125-163.

[Lov77] Loveman, David B., "Program improvement by source-to-source transformation," *J. ACM*, 24, 1 (Jan., 1977), 121-145.

[PaS83] Partsch, H. and Steinbruggen, R., "Program transformation systems," *ACM Comput. Surv.*, 15, 3 (Sept., 1983), 199-136.

[Rei84] Reiss, Steven P., "PECAN: Program developement systems that support multiple views," *Proceedings - International Conference of Software Engineering, March, 1984* , IEEE Computer Soc. Press, 324-333.

[ReT84] Reps, Thomas and Teitelbaum, Tim, "The synthesizer generator," *SIGPLAN Not.*, 19, 5 (May, 1984), 42-48.

[San87] Sand, Paul A., "Three Modula-2 programming systems," *Byte*, 12, 1 (Jan., 1987), 333-336.

[Sch76] Schneiderman, Ben, "Exploratory experiments in programmer behavior," *Internat'l Jour. of Computer and Info. Sci.*, 5, 2 (1976), 123-143.

[Sch80] Schneiderman, Ben, *Software Psychology - Human Factors in Computer and Information Systems*, Winthrop Publishers, Inc., 1980, 46-54.

[SGW81] Spier, Michael J., Gutz, Steve and Wasserman, Anthony I., "The ergonomics of software engineering - description of the problem space," *in Software Engineering Environments*, H. Hünke, Ed., North-Holland Publishing Company, 1981, 223-234.

[Sol86] Soloway, Elliot, "Learning to program = learning to construct mechanisms and explanations," *Commun. ACM*, 29, 9 (Sept., 1986), 850-859.

[SSS86] Schneiderman, Ben, Shafer, Philip, Simon, Roland and Weldon, Linda, "Display strategies for program browsing: Concepts and experiments," *IEEE Software*, 3, 3 (May, 1986), 7-14.

[Sta84] Stallman, Richard, "EMACS: The extensible, customizable, self-documenting display editor," in [BSS84], 300-324.

[Stu84] Stucki, Leon G., "What about CAD/CAM for software? The ARGUS concept," in *Software Validation*, H.L. Hausen, Ed., Elsevier Science Publishers B. V. (North-Holland), 1984, 311-320.

[Tei84] Teitelman, Warren, "Automated programmering: The programmer's assistant," in [BSS84], 232-239.

[TeR81] Teitelbaum, Tim and Reps, Thomas, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Commun. ACM*, 24, 9 (Sept.,1981), 563-573.

[Toy84] Toy, W.N., "Hardware/software tradeoffs" in *Handbook of Software Engineering*, Charles V.Vick and C.V. Ramamoorthy, Eds., VanNostrand Reinhold Company Inc., 1984, 149-183.

[Wil84] Wilander, Jerker, "An interactive programming system for Pascal," in [BSS84], 117-127.

[Woo81] Wood, Steven R., "Z - the 95% program editor," *SIGPLAN Not.*, 16, 6 (June, 1981), 1-7.

[Zel84] Zelkowitz, Marvin V., "A small contribution to editing with a syntax directed editor," *SIGPLAN Not.*, 19, 5 (May, 1984), 1-6.