# Graphical Program Development with PECAN Program Development Systems†

Steven P. Reiss

Department of Computer Science
Brown University
Providence, RI 02912

## ABSTRACT

This paper describes the user's view of the PECAN family of program development systems. PECAN is a program development system generator for algebraic programming languages. The program development systems it produces support multiple views of the user's program, its semantics, and its execution. The program views include a syntax-directed editor, a declaration editor, and a structured flow graph editor. The semantic views include expression trees, data type diagrams, flow graphs, and the symbol table. Execution views show the program in action and the stack contents as the program executes. PECAN is designed to make effective use of powerful personal machines with high-resolution graphics displays and is currently implemented on APOLLO workstations.

## 1. Introduction

The availability of powerful personal computers and the desire for increased programmer productivity have led to the recent development of interactive programming environments for algebraic languages. In this paper we describe the user's view of the PECAN family of environments developed at Brown University. PECAN is a generator of program development systems for algebraic programming languages.[1, 2] It differs from other systems by generating complete environments from simple specifications. The generated environments differ in their use of the graphical facilities of personal workstations and in their support for multiple concurrent views. PECAN environments provide views of the program, its semantics, and its execution.

### 1.1. Objectives

One objective in developing PECAN is to investigate ways of making full use of the computing power and graphics available on the new generation of personal machines for program development. PECAN is a prototype system that illustrates some of these ways. PECAN environments are designed for both the experienced and the novice programmer. They are easy and fast to use, offer immediate feedback to the user, and allow the user to visualize his program. A simple example of the use of PECAN for program development is presented in section 3 of this paper.

PECAN environments combine many of the best features of similar program development systems. These include:

* Immediate feedback of semantic and syntactic errors while the user is editing.
* An undo facility whereby the user can undo (and redo) any action back to the beginning of his session.
* Structured templates for building the program that are available as commands.
* The flexibility to type text at any time instead of using templates.

† This research was supported in part by National Science Foundation grants MCS-7905992, MCS-8200670, SER-8004974 and MCS-8121806, by the Office of Naval Research under contract nos. N00014-78-C-0396 and N00014-83-K-0146 and by DARPA order n. 4786.

* The use of menus as alternatives to typing for most commands.
* A multiple window display to make effective use of the screen.
* Incremental semantics that allow the program to be compiled as it is edited.
* A framework that handles a variety of (algebraic) programming languages with the same commands.
* Forward and backward execution, with user control of speed and the ability to step one statement at a time.
* The ability to edit during execution.

### 1.2. Views in PECAN

PECAN environments differ from other program development systems in their use of multiple views of shared data structures. The program is represented internally as an abstract syntax tree. The user does not see this tree directly, but instead sees views or concrete representations of it. One such view is a syntax-directed editor. Another view of the program is a Nassi-Schneiderman structured flowchart. A third view is a declaration editor. A fourth view would be a module interconnection diagram showing how the program is organized. Each of these views may be read-only or editing. They each display the abstract syntax tree. This display is updated automatically as the tree changes. Editing views support modifications to this tree using the displayed representation. The current views are discussed in section 4.

PECAN provides more than program views. In addition to the data structure representing the syntax of programs, the incremental compiler supports semantic data structures for the symbol table, the set of data types, expression trees, and control-flow graphs. PECAN supports incremental views of these data structures. These are described in section 5.

PECAN supports interpretive program execution. It provides a controlled environment that lets the user execute his program forwards and backwards, set breakpoints, and single step. It uses the program and semantic views to show where the program is executing at a given time. It provides additional views to display the program's data. This currently includes a stack and data view that shows the values on the program stack as they are allocated, freed and as they change. The data part of this view is used for displaying structures, arrays, and the values pointed to by pointers. Future displays will be obtained by merging PECAN with the BALSA system.[3,4] BALSA provides multiple dynamic views of a fixed program's data structures while the program is executing. The current execution views are discussed in section 6.

PECAN supports multiple views concurrently with automatic updating. There can be several views of the same abstract syntax tree. If any one of them modifies the tree, PECAN sends messages to inform each view that it should be updated. The incremental compiler is treated as an invisible view and is called whenever a tree changes. In turn, as the compiler updates the semantic representation, views of the semantics are notified of the change so that they can update their displays. The same philosophy is used for execution views. As a variable or the stack or the location counter changes, any views that are displaying these items are notified to update their display.

PECAN is currently being developed at Brown University on APOLLO workstations. The first version is designed with experimentation in mind and with the goal of supporting student programming. The system design is flexible so that new views may be easily tried and so that a reasonable combination of views can be found. The system design is highly modular. Components such as the incremental compiler can be changed without affecting other views or modules.

## 2. Related Work

There are several efforts aimed at building interactive programming environments for personal machines. While PECAN borrows from many of these systems, it is differentiated by its use of multiple views, its retargetability, and its reliance on graphics. In this section we briefly relate the more well-known systems.

The Cornell Program Synthesizer[5] provides a full system for PL/C including a syntax-directed editor and an interpreter. Its editor is template-based, but provides text editing for fixed constructs such as expressions. It has recently been implemented as a generator so that it is possible to create synthesizers for different languages using attribute grammers to describe the output and semantics for each production of the abstract syntax. The COPE system[6] also developed at Cornell provides another approach based on an

intelligent parser. The editor in this system is a text editor tied to an error correcting parser. The parser is able to insert missing keywords and tokens to get about the same effect as templates do in the Synthesizer. This scheme has an advantage in that the user can type his program at any time. It has the disadvantage that the user is not shown what templates are currently valid. The current PECAN editor is a compromise between these two schemes. It provides templates at every position, but allows the user to type at any point. Moreover, the parser has some error correcting capabilities. COPE also includes the important concept of being able to undo and redo both editing and execution. PECAN provides a similar ability.

The GANDALF effort at Carnegie-Mellon University is an incremental programming environment generator currently working for several algebraic languages.[7] It includes the ALOE syntax-directed editor generator.[8] The abstract syntax descriptions and print specifications used by PECAN are derived from those defined for ALOE. ALOE produces general structure editors not geared toward programming languages. Thus, the semantics for the language in question have to be described procedurally. ALOE also is a template based editor in that it provides templates for all productions in the tree and does not automatically allow parsing. It is possible, however, for the user to write his own parser and to use it for limited text editing. Much effort in GANDALF has been aimed toward programming in the large, an issue not addressed by PECAN. GANDALF until recently has not been targeted for personal machines with graphics capabilities.

Several programming environments have been developed at the Xerox Palo Alto Research Center. The Smalltalk system[9] is an interpreter for an object-oriented language based on message passing. It makes heavy use of windows and the capabilities of the graphical display, but is language-dependent. The Interlisp environment[10] provides a good example of what can be done with an interpreted language and a high-resolution display. The Mesa[11] environment, and more recently CEDAR,[12] apply many of the ideas from Smalltalk and Interlisp to an algebraic, compiled language. Many of the ideas proposed for CEDAR can be found in PECAN. CEDAR is designed as a production programming environment. PECAN differs in its emphasis on programming in the small, interaction, the use of graphics, and on showing the user multiple views of his program. PECAN is designed to be interactive and to eventually support graphical programming.

## 3. An Example

In this section we examine a session with a PECAN environment in which we create and execute a simple program. The language used here is an extended version of Pascal.

The system starts with an initial display that can be specified by the user. For the purposes of demonstration and debugging we use the display shown in figure 1. This consists of a large number of the possible views all displayed at once. This would be much too cluttered for the average user of the system, and a typical user might restrict himself to an editor, the execution monitor, and a data view. The large view in the upper right is a syntax-directed editor for the syntax tree labeled *example*. Below it are semantic views of the flow graph and the current data type. Below the data type window is a transcript window where the system shows the user all of his commands. This window can be used for controlling the undo and redo capabilities of the system. The column of windows on the right contains the control view from the interpreter and semantic views of the current expression and the symbol table. Finally, the two views in the lower-left show what the four buttons on the locator device do and the time respectively.

PECAN systems are built on top of the Brown workstation environment.[13] This environment contains a variety of tools. Four of these tools are used extensively by PECAN. MAPLE is the input management package. It is used to provide the various menus within windows and to direct inputs to particular parts of the program depending on which window the user is currently in. ASH is the graphics package that is used throughout. It provides a variety of raster graphics operations through the abstraction of a window and its view on the screen. ASH provides a hierarchy of windows. It allows windows to overlap and to be made invisible. It takes care of maintaining and displaying all the windows, including those that are partially or totally obscured. VT is a package that sits on top of ASH and provides extensive text support in an ASH window. It allows multiple fonts along with editing operations like those provided by an intelligent terminal. It stores a semi-infinite pad and manages the display for any particular part of the pad. Finally, WILLOW provides a user-

configurable window manager. WILLOW provides the utility buttons at the bottom of the screen and the icons above them that are used for creating new views. It also provides the move and resize buttons in the upper-left and upper-right of each of the windows. Other willow buttons are those in the interpretor control view, including the scroll bar for controlling the speed of execution, and buttons for pushing, popping and removing windows that are hidden under the title banners of each of the windows. WILLOW allows the user to configure the system to his liking. It lets him choose the initial screen layout. It lets him choose the fonts to be used in each of the windows. It even lets him choose the user-interface to be used for manipulating windows.

In figure 2 we have started the editing session. We have moved the cursor into the syntax-directed editor window and have indicated that we want to create a new PROGRAM. The system filled in the name of the program from the tree name. Then we moved to the declaration and, using the menu on the right of the editor view as a guide, indicated that we wanted a VAR declaration. This could have been done by using the locator device to pick the VAR button on the menu. Instead, we chose to type in the word *var* and have the system realize that this meant to use the button. Finally, we entered the variable definition by typing it all in instead of selecting off the menu. These actions illustrate the three means that the editor provides for program entry. The user can either select templates using the menu, can select templates by typing in a prefix of the corresponding button name on the menu, or can type in the full text for the current construct.

Figure 2 also shows that PECAN compiles as it goes. This is seen in the semantic views that are active. The symbol table view in the lower-left shows the symbol table as it exists up to this point. Similarly, the data flow view in the lower-right shows the complete flow graph. The hexagonal flow nodes indicate that the corresponding variable should be allocated at this point. Both of these views are concrete representations of the intermediate representation that is generated by the incremental compiler.

The various semantic views are more fully illustrated in figure 3. Here we have continued to use the editor to enter the function *gcd*. The system augments our editing with automatic comments such as that on the BEGIN statements and the names of the parameters on the function call. After entering the body of the function we used the locator to point and move to the parameter *b* in the recursive call. The semantic views are keyed on the current editing location. Thus the expression view contains the expression tree corresponding to the call, with the node corresponding to the current parameter highlighted. The symbol table view similarly highlights the current variable. The data type view shows the data type of this variable. Finally, the flow graph viewhighlight sthe corresponding node of the flow graph, although this node is not currently visible.

We next finish typing in the program that reads two values, calls the function *gcd* with them, and then prints the result. Then we are ready to try out the program. We first rearrange the screen, eliminating most of the semantic views, enlarging the flow graph view so that it shows the whole program, and creating a stack view to show the data. We use the GO button in the interpreter control view to start execution, enter two values when requested, and then stop the program during execution by clicking on the locator. The result is shown in figure 4.

Several elements of PECAN can be seen in figure 4. The current statement being executed is highlighted in both the flow graph view and in the syntax-directed editor. When these views are displayed, this highlighting is done automatically when execution stops. The MONITOR button in the interpreter control window causes the highlighting to be done continuously as the program executes. The interpreter control view shows the user the state of execution as well as any input and output of the executing program. The stack display shows the current state of all the program's variables. It shows the recursion inherent in the program and the different variable values at each level.

In the next sections we consider the current and planned views. Section 4 looks at the program views. Section 5 looks at the semantic views. Section 6 considers the execution views.

## 4. Program Views

Program views are visual representations of abstract syntax trees. They can be either read-only views or they can be editing views. In either case they are automatically updated as the trees they represent are changed, whether they initiated the change or not.

### 4.1. The Syntax-Directed Editor

The syntax-directed editor view provides a textual representation of the program. It offers a cross between a traditional text editor and a structure editor. It allows the user to move to any point in the program and do whatever text editing operations are appropriate. The editor parses the result, puts it into its internal representation, and reformats the display. The freedom to treat a textual representation of the program as text is important since many of the changes a programmer makes are textual ones, such as correcting typing errors. Moreover, when there are many corrections to a program, the programmer is likely to want to treat the program as a large textual object, make the corrections, and not want to worry about its structure. Several systems, including COPE[6] and POE[14] use a textual approach to syntax-directed editing.

At the same time the syntax-directed editor view provides a complete set of templates that allow the programmer to make full use of the structure of the underlying syntax tree. The current location is not a single character, but is a tree node, and the text for this node is placed in a box on the display. The user has available simple commands to climb around the syntax tree and to pick and put nodes of the tree. Whenever the current node is a meta variable or a leaf node, the editor provides a special menu that lists the possible expansions or contents. For meta nodes these are the templates associated with such syntax-directed editors as ALOE[8] or the Cornell Program Synthesizer.[5] For leaf nodes, they are a list of the appropriate candidate names from the symbol table. Editing with templates is often easier for novice programmers and can save considerable typing on program entry.

While trying to compromise between structured editing and text editing, the syntax-directed editor also makes heavy use of the facilities of the graphical display and the pointing device. The editor uses multiple fonts to distinguish between keywords, meta symbols that have yet to be expanded, text for identifiers, and text that contains semantic errors. The user is free to choose any of the available fonts for these purposes. The editor uses line drawing to place a box of whatever shape is necessary around the text corresponding to the current node.

The interface to the editor uses the pointing device and the keyboard. The pointing device can select any of the menu buttons that are seen in the example. These provide the basic editing commands, and, for meta nodes and appropriate leaf nodes, the relevant templates and name alternatives. The pointing device is also used with the displayed tree. The user can change the current node by pointing at the desired location. Other buttons on the puck allow the user to pick, put, delete and insert into the tree. Multiple clicks at the same location allow the user to move about the tree structure.

The keyboard is used primarily for text editing, but also can be used for many of the editor commands. The user is free to type text or text-editing commands. When the user is done text editing, he issues a command not involving text-editing or an explicit done command, the edited text is parsed, and the result is put into the tree. Control and function keys on the keyboard provide most of the tree-editing commands (delete, insert, pick, put), as well as tree movement and view scrolling commands. This makes it possible for the user to edit without continually having to move his hands from the keyboard to the pointing device and back again.

### 4.2. Nassi-Schneiderman View

A second program view is a graphical one based on Nassi-Schneiderman flow charts.[15] These are a form of structured flow charts that use different types of blocks for different program constructs and use the nesting of blocks to represent the nesting of the program. With this view, the user sees the complete structured form of his program drawn as a structured flow graph.

The view is based on a pretty-print notation that describes how each abstract syntax construct should be represented graphically using a simple print rule. The rule shows the nesting and the creation of new blocks in the flow graph. These blocks are either rectangular blocks that indicate simple processing, decision blocks with a test on top and the alternative paths in sub-boxes beneath, iteration boxes that contain the loop body and the tests on the outside, and a separate box type to indicate scoping.

The view is tied into the system and is automatically updated as the user changes his program. We are currently working on adding basic editing operations to this view to allow the user to edit his program graphically.

### 4.3. Declaration View

Another editing view is provided exclusively for declarations. The user can select a particular

declaration either by pointing to the declaration or by pointing to the name of the declaration somewhere in the program text. If no declaration already exists for the name, then a new one is created automatically in the current scope. Once a declaration is chosen, the user can use this view to add, delete or change the associated names, and to locally edit fields of the declaration such as the data type of a variable. He can change the class of the declaration, such as local variable to global variable or variable to constant. He can move the declaration to a different scope by pointing to the new scope and selecting the SCOPE button.

This view serves two purposes. First, it allows the user to create programs interactively, declaring variables as he uses them. Second, it supplements the Nassi-Schneiderman and future graphical program views that do not directly support declarations.

### 4.4. Other program views

There are other program views that could be supported by PECAN and that we hope to see implemented some time in the future. In the short term, we are planning to do a general-purpose structured flowgraph editing view. The initial implementation of this will support Rothon diagrams and Nassi-Schneiderman diagrams. Future program views will support programming in the large through a procedure-level connection diagram and through pictures of the top-level data flow. Other possible views include one that supports data-flow programming and a verification view that lists the predicates known at each point in the program and allows the user to prove correctness while programming.

## 5. Semantic Views

While abstract syntax trees representing user programs are the most logical data structure to view, it is often useful to provide the user with specialized views of the internal forms supported by PECAN. During program editing, the relevant forms are the semantic representations of the program: the symbol table, data type definitions, expression trees, and control flow graphs.

### 5.1. The Symbol Table View

The compiler builds and incrementally maintains a scoped symbol table of all built-in and user-defined symbols. The corresponding view displays the scopes and symbols that are defined for a particular syntax tree. The boxes display the nesting of the various scopes. Each box contains a brief description of the scope and a list of all the names defined by the user program in that scope. Each name is displayed with its class, such as variable, type, or label, and other pertinent information such as the data type of a variable. If the current node in a program view is a variable, then this variable is highlighted in the symbol display. Pointing at a name or scope in the symbol display causes the appropriate program view to make the definition node for that scope or name the current location.

### 5.2. The Data Type View

The compiler maintains information on built-in and user-defined data types. A semantic view displays the current data type based on this information. If the user is editing a type definition, then the type being edited is displayed. If the current editing node is a variable, then the type of the variable is displayed. If the current node is an expression, then the type of the expression is displayed. The top line of a type display contains the class of the type such as RECORD, POINTER, ARRAY. The following lines show the parameter values for this class. Where these parameters are also types, the display is recursive. To avoid infinite displays, the recursion typically stops after one level. However, the user is free to choose an unexpanded type and ask that it be expanded, either within the display or by making it the only type displayed. Pointing at a type definition causes the appropriate editor to make that definition its current location.

### 5.3. The Expression View

The third component of the compiler maintains expression trees for each expression in the program. The corresponding view draws these trees, affording the user a different perspective of the expressions. The tree that is drawn is the one that is the current focus for editing operations. The node being edited is highlighted on the display. The user may pan over the expression tree when the tree is too large to fit in the window assigned for expression display. He can also point at the tree to make the corresponding node be the current location for editing.

### 5.4. The Flow View

The final component of the incremental compiler builds and maintains graphs that describe the flow of control for an abstract syntax

tree. The corresponding view provides a flowchart of the program that is dynamically updated as the program changes. Nodes of the flowchart represent expression evaluations, conditional branches, contour entrys and exits, variable allocations, gotos and labels. The user can pan over a complex flowchart to view the currently relevant portion. He can also point at a given block and cause the corresponding code to become the focus for editing operations.

## 6. Execution Views

PECAN supports views of program execution. These views are of three different types, control, program, and data.

The execution control view is provided by the interpreter. This view contains messages that indicate the current execution state of the program. It is used to display error messages incurred during execution. It also displays the program input and output using different fonts to provide a full transcript of the user's debugging. The control view provides user debugging facilities. These include the ability to reverse program execution, to step through the program one statement at a time, either forward or backward, and to insert breakpoints by pointing at the place to stop and then hitting the BREAK button. It also provides a speed control so that the user can slow down the program execution.

Execution can be monitered in the program views and in the semantic flow graph view. If monitoring is turned on in the control view, then each of the program views currently active will be instructed to highlight the current statement as it is executed. If the flow graph view is active, then it will be used to show the step-by-step execution of the program.

The user is provided with views of his data as the program executes. The stack view is the only current data view, although more views are planned. The stack view divides its window into two portions. The left side provides a display of the current execution stack. It shows where each block on that stack begins. It shows each variable in the block and displays its current value. Undefined values are labeled as such. Simple values such as integers, enumeration constants, reals, and characters are displayed directly. Complex values such as arrays, records, and pointers are not displayed, but have an appropriate token in their place. The right side of the view is used to display these complex values. The user points at the token for an array or record or pointer and the corresponding value will be displayed on the right. This technique can be used recursively to display chains of pointers or nested structures.

A future goal of PECAN is to provide a wide variety of execution views based on experiences with the BALSA system.[3] These will include data views that render a graphical representation of a user's data structure such as a tree that is automatically updated as the program executes. This work will be based on an extensive library of possible graphical representations and is the subject of current research. Other data views would provide the user with more classical representations of the stack or of a selected set of variables. These will again be updated as the program executes. Execution views will also show the program in action. Simple program views will highlight each statement as it is executed. Other views will provide performance information such as execution counts in various graphical forms.

## 7. Acknowledgements

The effort that has gone into PECAN has been supported by much of the environment activity at Brown. Marc Brown is largely responsible for the MAPLE menuing package. Joe Pato and Mark Vickers developed the virtual device interface for the Apollos. Marc Brown and Mike Strickman did the core of the BALSA implementation. Rob Rubin and Jeanette Hung did the Nassi-Schneiderman view.

## 8. References

1. Steven P. Reiss, "PECAN: Program development systems that support multiple views," *Proceedings of the Seventh International Conference on Software Engineering*, (March 1984).
2. Steven P. Reiss, "An approach to incremental compilation," *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, (June 1984).
3. Marc H. Brown, Norman Meyrowitz, and Andries van Dam, "Personal computer networks and graphical animation: rationale and practice for education," *ACM SIGCSE Bulletin* **15**(1) pp. 296-307 (February 1983).
4. Marc H. Brown and Steven P. Reiss, "Toward a Computer Science Environment for Powerful Personal Machines," *Proceedings of the 17th Hawaii System Sciences Conference*, (January 1984).

5. Tim Teitelbaum and Thomas Reps, "The Cornell program synthesizer: a syntax-directed programming environment," Cornell University Technical Report TR 80-421 (May 1980).
6. James Archer, Jr. and Richard Conway, "COPE: A cooperative programming environment," Cornell TR81-459 (June 1981).
7. A. N. Habermann, "The Gandalf Research Project," Computer Science Research Review, Carnegie-Mellon University 1979 (1979).
8. Raul Medina-Mora and David S. Notkin, "ALOE users' and implementors' guide," Carnegie-Mellon Computer Science Department Research Report CS-81-145 (November 1981).
9. Adele Goldberg, "The influence of an object-oriented language on the programming environment," *Proc. ACM Computer Science Conference*, (February 1983).
10. Warren Teitelman, *Interlisp Reference Manual*, XEROX (1974).
11. James G. Mitchell, William Maybury, and Richard Sweet, "Mesa language manual," *Xerox CSL-79-3*, (April 1979).
12. L. Peter Deutsch and Edward A. Taft, "Requirements for an experimental programming environment," Xerox CSL-80-10 (June 1980).
13. Joseph N. Pato, Steven P. Reiss, and Marc H. Brown, "The Brown workstation environment," Brown University CS-84-03 (October 1983).
14. Charles N. Fischer, Greg Johnson, and Jon Mauney, "An Introduction to Editor Allen Poe," Univ Wisconsin-Madison TR 451 (October 1981).
15. I. Nassi and B. Schneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices* **8**(8)(August 1973).

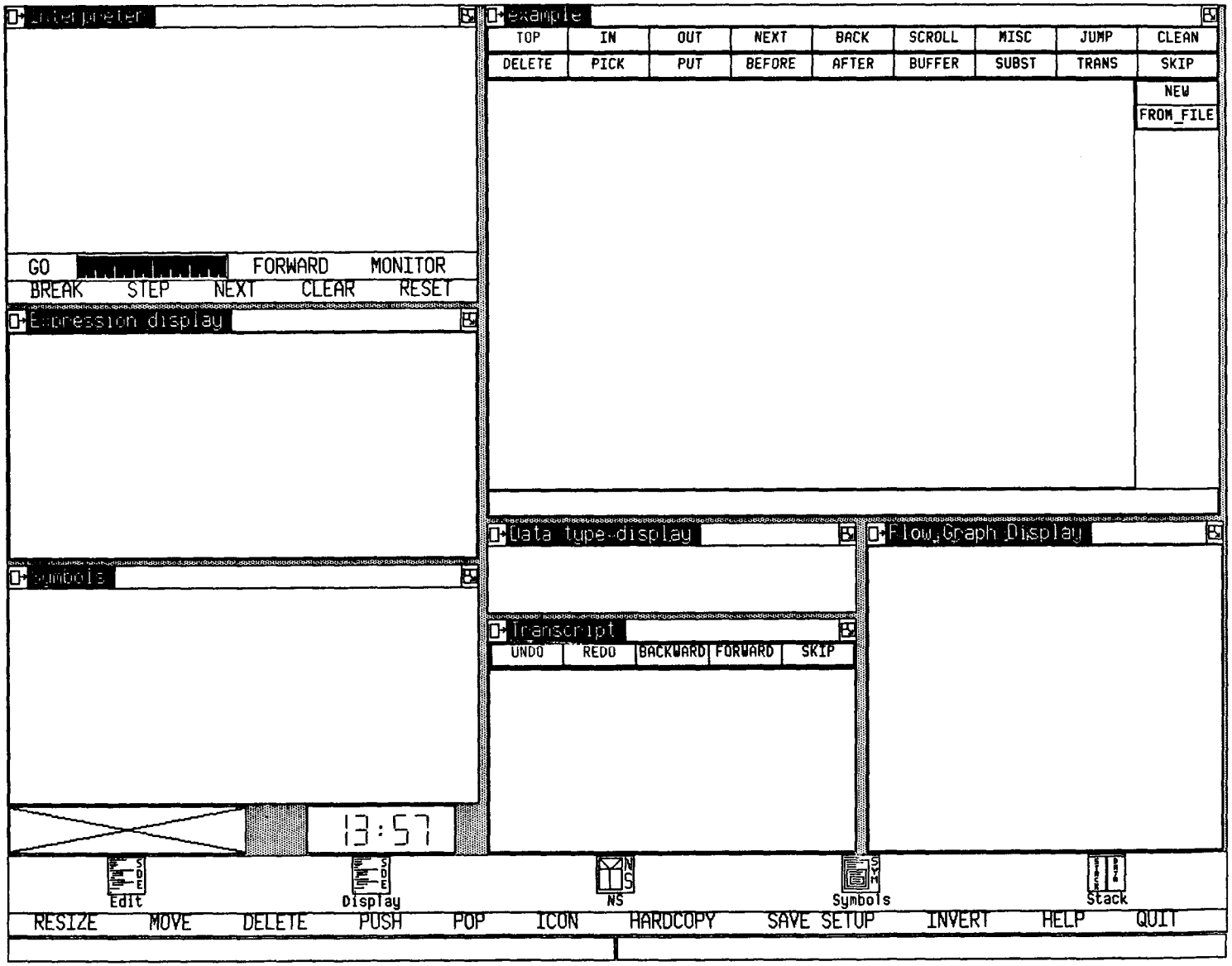


Figure 1: Initial PECAN Display

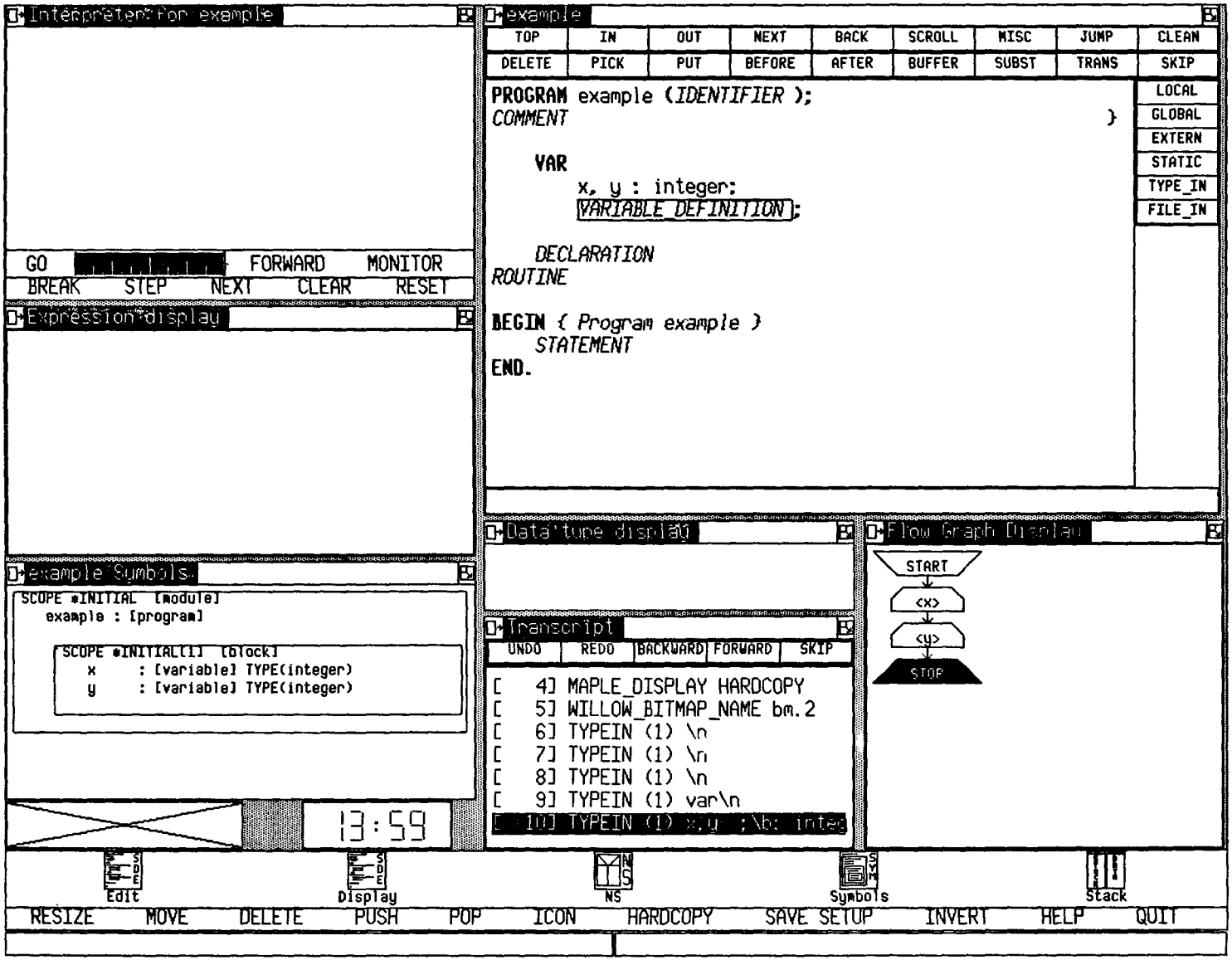


Figure 2: PECAN Display Showing Editing

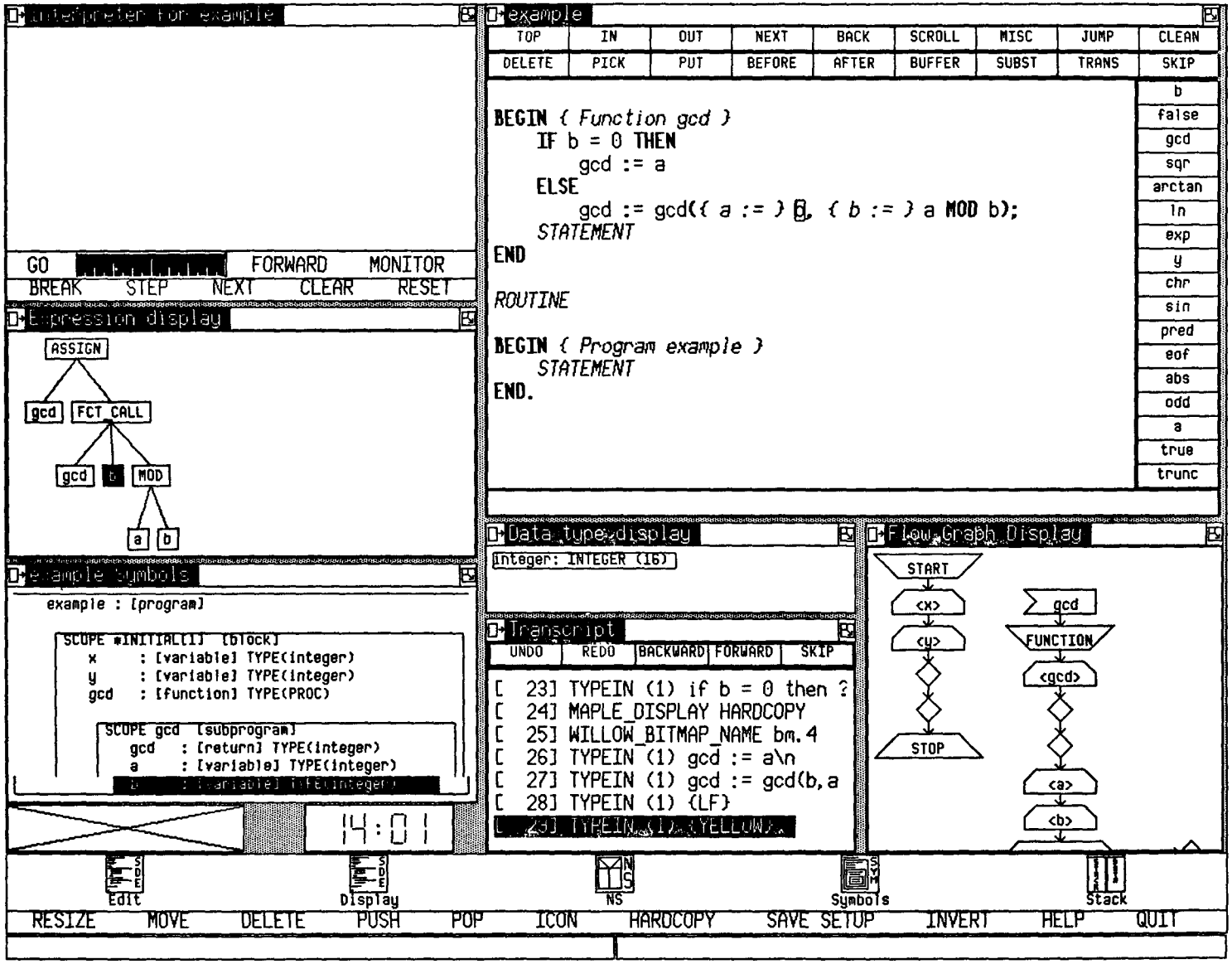


Figure 3: PECAN Display Showing Semantic Views

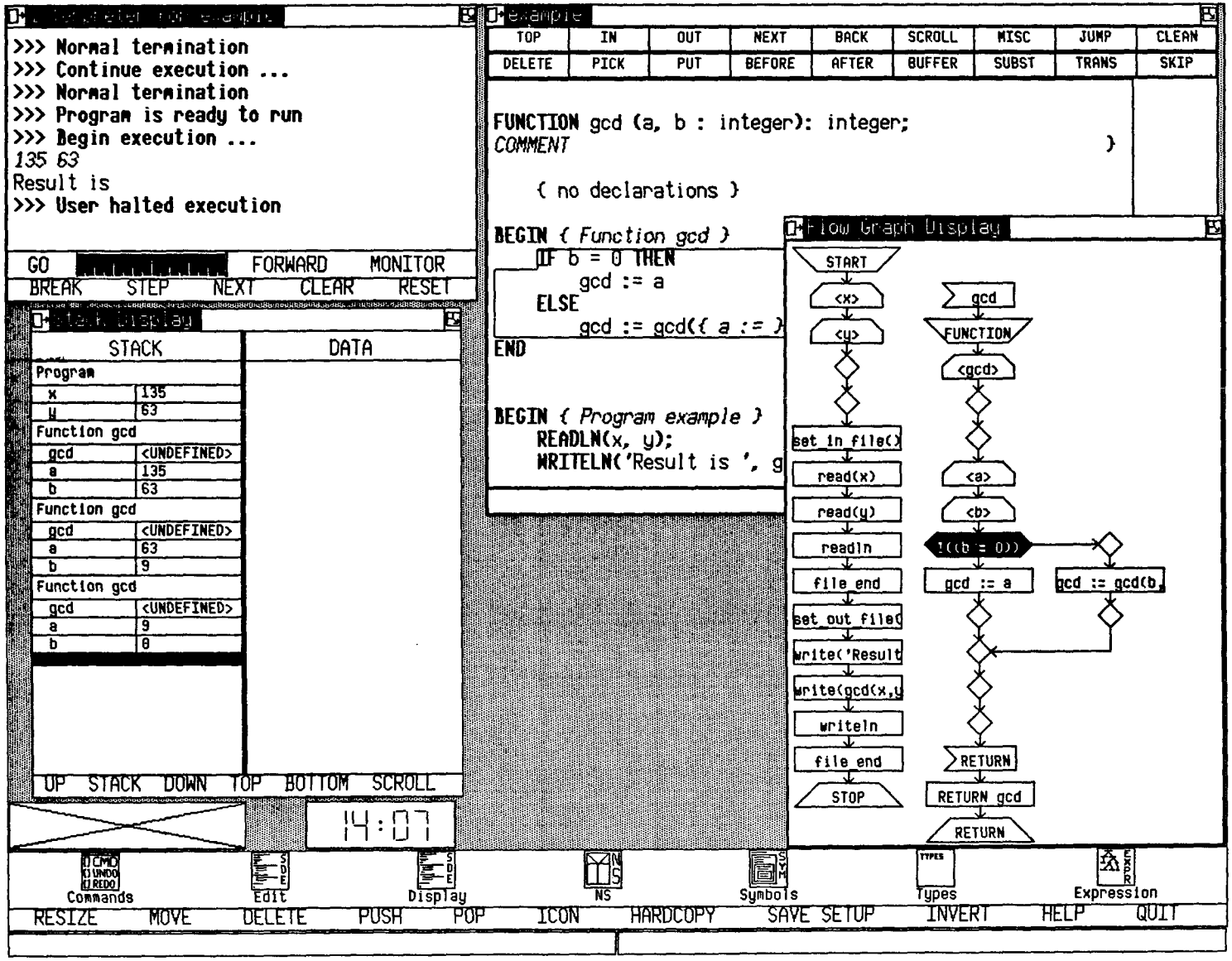


Figure 4: PECAN Display Showing Execution