Programming environments based on structure editing: The GNOME approach

by RAVINDER CHANDHOK, DAVID GARLAN, DENNIS GOLDENSON, PHILIP MILLER and MARK TUCKER
Carnegie-Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

The use of integrated programming environments based on structure editing is an emerging technology that has now reached the stage of being both demonstrably useful and readily implementable. We have outlined some of the salient aspects of our work in developing the GNOME and MacGNOME programming environments and suggested paths of implementation that seem to be worth traveling. A predominant theme in all of this has been the need to separate policy from mechanism. While the choice of user interface policies will probably differ widely from those we have made here, the mechanisms that we have sketched will nonetheless be applicable to future environments.
INTRODUCTION

Traditionally, program development has taken place in what remains a very poorly integrated environment. A program typically is constructed using a text editor, then translated in a stand-alone compiler, while error correcting is done with the aid of separate debugging tools. Each piece of software, and the host operating system, has its own unique command language. Hence, a good deal of expert knowledge is required to use the tools effectively. This leaves much to be desired from the point of view of both expert and novice programmers. The novice must focus early attention on extraneous details of tool invocation at the expense of concentration on abstraction and problem solving. The expert probably knows a good deal about such details, but these details still can be time consuming and error prone.

There has been considerable discussion recently about the development and potential of structure editing to alleviate the problem by supporting integrated programming environments. These environments can be viewed as a collection of cooperating tools that communicate to the user through a single uniform interface combining the functions of a text editor, interpreter, compiler, and debugger. Our own work has focused on the development of integrated programming environments designed for the teaching of block structured, high level languages to novice users.

Many computer science educators agree that it is important that the beginning programmer meet with early, easy, and immediate success. Simple programs should be simple to write. They should execute rather than respond with cryptic messages about parsing failures or declaration shortcomings. Additionally, the first few days and weeks of programming are times when programming habits are formed. If the student’s attention is focused on the location of semicolons and the spelling of predefined types, it isn’t focused on problem-solving, top-down design, modularization, and abstraction.

When constructing a program in a traditional text editor, one enters a program as if it were no different from any other textual document. The editor edits characters that just as easily could be part of a business letter or poem. Hence, it is easy to make punctuation and other syntax errors, errors which often are difficult for a novice to detect. Structure editors, on the other hand, are syntax-directed. They are knowledgeable about the programming language being used. Rather than single characters, their basic elements are syntactically meaningful entities. A syntax-directed editor for a procedural programming language, for example, operates on data declarations, subprogramming units, statements, expressions, and so on. The editor allows no constructions which are not syntactically valid; since it is then impossible to make a syntax error, the novice programmer is freed to concentrate on higher levels of program abstraction.

While the structure editor prevents syntax errors, other tools in the environment combine to support the various activities concerned with producing a correct and runnable program. In particular, an incremental semantic analyzer can check for “compile-time” correctness as the program is entered; some semantic errors can be prevented altogether, others can be reported to the programmer in the immediate context in which they are made. Run-time tools can allow a mix of editing, compiling, interpreting, and running supporting various active views of the program. Other tools provide an interactive on-line help system, smooth interface to the file system, and documentation aids.

The remainder of this paper describes some of the work we have done in developing structure editing environments at Carnegie-Mellon University. It highlights several aspects of the work that we hope will be of interest from the point of view of both users and implementors, computer science educators, and programming systems professionals. The first two sections give some background about the GNOME environments on which we have been working, what they look like to a user, how they work, and why we think they provide a useful alternative to traditional approaches to programming.

When designing software, it is useful to separate design issues involving system functionality from those which are strictly a matter of user interface. We characterize this distinction as one between mechanism and policy. In describing some of the important lessons we have learned in implementing the GNOME environments, the predominant theme is that by clearly distinguishing between mechanism and policy, an implementor can provide a wide variety of policies using relatively easily implemented mechanisms. This is done by shifting much of the burden of maintaining an incremental environment off the shoulders of elaborate data structures and onto the shoulders of efficient procedures that can regenerate information as needed.

GNOME PROGRAMMING ENVIRONMENTS

Many syntax-directed environments work by presenting the programmer with a program template directing attention to points where further program construction is permissible. For example, in the GNOME environment, a program initially looks as depicted in Figure 1. The words prefixed with a $ symbol are at those points in the program where further construction is valid. They are called metanodes, and correspond to the non-terminals of a formal specification of the pro-
programming language.* The editor automatically displays all syntactic “sugar” (keywords, punctuation, etc.) and formats the program. The current focus of attention, or cursor, is indicated in GNOME environments by a boxed area that is displayed with inverse video. Allowable choices for construction at any metanode are displayed in a separate help window. The programmer need only uniquely identify the desired construction and it is inserted into the program tree.

GNOME maintains the program internally as a tree. Rather than parsing text into a tree that is then compiled into machine code, GNOME grows and shrinks the program tree, unparsing the tree representation into a text representation that the programmer sees at the terminal.

Until recently, structure editors have not been widely available for the broad spectrum of classrooms in which programming is taught, nor for serious programming outside of a few well-supported universities and research institutes. Some existing structure editor environments work only on subsets of languages, and handle only small programs. Others require expensive time-shared systems run on exotic operating systems, or depend on expensive terminals. Still others use prohibitively costly personal computers.

MacGNOME is an implementation of a GNOME environment targeted for the Apple Macintosh computer. Through MacGNOME, we intend to make structure editor environments widely available, taking advantage of the high resolution graphics capability and 32-bit architecture found in the emerging crop of affordable personal computers. Initially, we are working on two language environments, Pascal 7 and Rich Pattis’ Karel the Robot. Use of the Macintosh ensures that the environment will be readily available, and of sufficiently low cost to allow its use by secondary schools as well as universities. The environment supports full ISO Standard Pascal programs of moderate size and provides all of the kinds of integrated program development support outlined above.

MacGNOME ENVIRONMENTS: THE USER’S PERSPECTIVE

In this section, we sketch how the user interacts with a MacGNOME environment by focussing on four important aspects of the system, the views that the user has of a developing program, the way the user navigates around a program, how he constructs a program, and how errors are reported to him.

Program Views

Both text and structure editors tend to impose a single view of the program. Text editors present a program as a linear stream of characters, in full detail. In the style of Pecan and Gandalf, MacGNOME allows multiple “views” of a program. These provide alternative visualizations of a program, appearing in multiple windows on the screen.

MacGNOME supports both a linear text and a “scoped” view of the program. In the former instance, the bodies of subprograms or other large blocks of code can be collapsed into icons and hidden from view, or left expanded in place. The latter view displays the program using separate windows for each subprogram. This means that only the subprogram headers are visible in the blocks where they are declared, but both the header and its contents are visible in a separate window when the programmer wishes to access them.

We are also implementing an outline view of the program. It focuses the programmer’s attention on scoping and nesting issues. This view is particularly helpful for organizing a program at high levels of abstraction, and for navigating through the program when it gets large and detailed. Indeed, some prefer to construct the program initially using this view, then moving into a scoped view to fill out the details.

Within a particular view of a program, it is also possible to affect the display of various components of the grammar by not displaying certain constructs of a language in the default view.

MacGNOME supports all four types of views, including a linear text view, a “scoped” view, an outline view, and a linear text view with certain components of the grammar removed.

Program Views

Both text and structure editors tend to impose a single view of the program. Text editors present a program as a linear stream of characters, in full detail. In the style of Pecan and Gandalf, MacGNOME allows multiple “views” of a program. These provide alternative visualizations of a program, appearing in multiple windows on the screen.

MacGNOME supports both a linear text and a “scoped” view of the program. In the former instance, the bodies of subprograms or other large blocks of code can be collapsed into icons and hidden from view, or left expanded in place. The latter view displays the program using separate windows for each subprogram. This means that only the subprogram headers are visible in the blocks where they are declared, but both the header and its contents are visible in a separate window when the programmer wishes to access them.

We are also implementing an outline view of the program. It focuses the programmer’s attention on scoping and nesting issues. This view is particularly helpful for organizing a program at high levels of abstraction, and for navigating through the program when it gets large and detailed. Indeed, some prefer to construct the program initially using this view, then moving into a scoped view to fill out the details.

Within a particular view of a program, it is also possible to affect the display of various components of the grammar by not displaying certain constructs of a language in the default view.

MacGNOME supports all four types of views, including a linear text view, a “scoped” view, an outline view, and a linear text view with certain components of the grammar removed.

MacGNOME ENVIRONMENTS: THE USER’S PERSPECTIVE

In this section, we sketch how the user interacts with a MacGNOME environment by focussing on four important aspects of the system, the views that the user has of a developing program, the way the user navigates around a program, how he constructs a program, and how errors are reported to him.

Program Views

Both text and structure editors tend to impose a single view of the program. Text editors present a program as a linear stream of characters, in full detail. In the style of Pecan and Gandalf, MacGNOME allows multiple “views” of a program. These provide alternative visualizations of a program, appearing in multiple windows on the screen.

MacGNOME supports both a linear text and a “scoped” view of the program. In the former instance, the bodies of subprograms or other large blocks of code can be collapsed into icons and hidden from view, or left expanded in place. The latter view displays the program using separate windows for each subprogram. This means that only the subprogram headers are visible in the blocks where they are declared, but both the header and its contents are visible in a separate window when the programmer wishes to access them.

We are also implementing an outline view of the program. It focuses the programmer’s attention on scoping and nesting issues. This view is particularly helpful for organizing a program at high levels of abstraction, and for navigating through the program when it gets large and detailed. Indeed, some prefer to construct the program initially using this view, then moving into a scoped view to fill out the details.

Within a particular view of a program, it is also possible to affect the display of various components of the grammar by not displaying certain constructs of a language in the default view.

MacGNOME supports all four types of views, including a linear text view, a “scoped” view, an outline view, and a linear text view with certain components of the grammar removed.

MacGNOME ENVIRONMENTS: THE USER’S PERSPECTIVE

In this section, we sketch how the user interacts with a MacGNOME environment by focussing on four important aspects of the system, the views that the user has of a developing program, the way the user navigates around a program, how he constructs a program, and how errors are reported to him.

Program Views

Both text and structure editors tend to impose a single view of the program. Text editors present a program as a linear stream of characters, in full detail. In the style of Pecan and Gandalf, MacGNOME allows multiple “views” of a program. These provide alternative visualizations of a program, appearing in multiple windows on the screen.

MacGNOME supports both a linear text and a “scoped” view of the program. In the former instance, the bodies of subprograms or other large blocks of code can be collapsed into icons and hidden from view, or left expanded in place. The latter view displays the program using separate windows for each subprogram. This means that only the subprogram headers are visible in the blocks where they are declared, but both the header and its contents are visible in a separate window when the programmer wishes to access them.

We are also implementing an outline view of the program. It focuses the programmer’s attention on scoping and nesting issues. This view is particularly helpful for organizing a program at high levels of abstraction, and for navigating through the program when it gets large and detailed. Indeed, some prefer to construct the program initially using this view, then moving into a scoped view to fill out the details.

Within a particular view of a program, it is also possible to affect the display of various components of the grammar by not displaying certain constructs of a language in the default view.

MacGNOME supports all four types of views, including a linear text view, a “scoped” view, an outline view, and a linear text view with certain components of the grammar removed.

MacGNOME ENVIRONMENTS: THE USER’S PERSPECTIVE

In this section, we sketch how the user interacts with a MacGNOME environment by focussing on four important aspects of the system, the views that the user has of a developing program, the way the user navigates around a program, how he constructs a program, and how errors are reported to him.

Program Views

Both text and structure editors tend to impose a single view of the program. Text editors present a program as a linear stream of characters, in full detail. In the style of Pecan and Gandalf, MacGNOME allows multiple “views” of a program. These provide alternative visualizations of a program, appearing in multiple windows on the screen.

MacGNOME supports both a linear text and a “scoped” view of the program. In the former instance, the bodies of subprograms or other large blocks of code can be collapsed into icons and hidden from view, or left expanded in place. The latter view displays the program using separate windows for each subprogram. This means that only the subprogram headers are visible in the blocks where they are declared, but both the header and its contents are visible in a separate window when the programmer wishes to access them.

We are also implementing an outline view of the program. It focuses the programmer’s attention on scoping and nesting issues. This view is particularly helpful for organizing a program at high levels of abstraction, and for navigating through the program when it gets large and detailed. Indeed, some prefer to construct the program initially using this view, then moving into a scoped view to fill out the details.

Within a particular view of a program, it is also possible to affect the display of various components of the grammar by not displaying certain constructs of a language in the default view.

MacGNOME supports all four types of views, including a linear text view, a “scoped” view, an outline view, and a linear text view with certain components of the grammar removed.

MacGNOME ENVIRONMENTS: THE USER’S PERSPECTIVE

In this section, we sketch how the user interacts with a MacGNOME environment by focussing on four important aspects of the system, the views that the user has of a developing program, the way the user navigates around a program, how he constructs a program, and how errors are reported to him.

Program Views

Both text and structure editors tend to impose a single view of the program. Text editors present a program as a linear stream of characters, in full detail. In the style of Pecan and Gandalf, MacGNOME allows multiple “views” of a program. These provide alternative visualizations of a program, appearing in multiple windows on the screen.

MacGNOME supports both a linear text and a “scoped” view of the program. In the former instance, the bodies of subprograms or other large blocks of code can be collapsed into icons and hidden from view, or left expanded in place. The latter view displays the program using separate windows for each subprogram. This means that only the subprogram headers are visible in the blocks where they are declared, but both the header and its contents are visible in a separate window when the programmer wishes to access them.

We are also implementing an outline view of the program. It focuses the programmer’s attention on scoping and nesting issues. This view is particularly helpful for organizing a program at high levels of abstraction, and for navigating through the program when it gets large and detailed. Indeed, some prefer to construct the program initially using this view, then moving into a scoped view to fill out the details.

Within a particular view of a program, it is also possible to affect the display of various components of the grammar by not displaying certain constructs of a language in the default view.
presentation. This has been particularly helpful in educational settings where it is used to hide constructs of the Pascal grammar with which students are not yet familiar.

Moving Through the Program

With a traditional text editor, default cursor movement is not an issue; one types text sequentially until it is time to change what was initially written. However, with a structure editor, the basic entities are meaningful in terms of the language being edited. So, decisions of style and method must be made about the order in which information ought to be entered. Our preference for novice users is to follow a top-down design philosophy where users first complete the statement part at any given Pascal scope, before dealing with the "details" of the declaration part. Variables generally are used, procedures and functions are called, when calls are first made. MacGNOME environments, however, do not restrict users to any fixed form of entry, allowing completely random positioning of the cursor with a pointing device. The user is also able to customize default cursor movement (and many other aspects of the user interface) to suit his own preferences.

Program Construction

The user must have some way of communicating the choices of construction to fill in the templates provided by the structure editor. Menus and/or other windows are an obvious place to enumerate valid choices at any particular stage in the program construction process. They also provide a mechanism for data entry that need not require active textual input by the user. In MacGNOME, for example, a mouse click on the appropriate option either expands the user's choice further, and/or enters the appropriate code. In addition to providing for the addition of new statements, expressions, and so on, the MacGNOME's internal program representation allows a "list" menu which can be used to remind the programmer of user-defined names by scope, and to enter them where appropriate. We also allow the user to modify any logical entity in a program using a text editor.*

Error Reporting

The manner in which errors are reported often proves to be a great source of confusion for non-expert programmers. We have found that it is useful to report errors at several distinct phases in the programming process. An editor ought to prevent the user from making certain errors in the first instance. For example, if the user tries to declare the "same" identifier twice in the same scope, MacGNOME will refuse to accept the declaration. Some illegal assignments are also spotted at this stage. Other errors, and style warnings are reported immediately on exit from the scope of a procedure/function. Run-time errors must also be reported when they occur in an executing program.

Typically, error messages are cryptic at best. When an error occurs, the system must inform the programmer, but it ought to be done with a clear one or two sentence error message, such as "The variable 'SuchAndSuch' is used but never declared in PROCEDURE 'SomeName'." In our work with GNOME, we have catalogued the errors most commonly made by novice programmers. In MacGNOME, we are providing further suggestions about the most likely causes of the error in question. Associated with the most common errors, a few short paragraphs of text are available about what probably went wrong.

IMPLEMENTATION OF STRUCTURE EDITING ENVIRONMENTS

The wide variety of structure editing environments built over the past few years differ widely in the way they may look to the user, but internally they must all provide certain basic mechanisms to support an integrated environment for programming. In the next several sections, we outline some of the important design issues in implementing those mechanisms and sketch what we believe to be reasonable approaches to the problems they present. The primary consideration is to derive solutions which represent an acceptable compromise between functionality, generality, and ease of implementation.

The basic mechanisms can be generally classified by the need to support certain fundamental requirements of structure editing. These are:

1. The need to represent trees and tools. We must decide on an appropriate data representation for programs and provide some way to associate tool operations with various events in the system.
2. The need to support checking and running of programs. We must decide how frequently errors can be checked and reported. We must give some way to run and debug a program or some part of it.
3. The need to display trees, program state, etc. We must decide how the abstract representation of the program is to be mapped to concrete text, and how to make this run quickly.

We now consider each of these areas in turn.

THE KERNEL

At the center of every structure editor is a tree-oriented database. Because this tree database is the primary static data structure of the system, care must be taken in its design so that it may be useful to all the tools in the environment. Moreover, dynamically, it must facilitate the integration of many tools so that all the right routines are invoked at the right time.
Tree Representation

Because the entire system operates on syntax trees rather than on text, it is crucial that the representation chosen for this database be easy for an implementor to understand and use when building the various tools that make up the entire environment. An implementation goal for us has been to keep the static data structures as simple as possible, and to compute dynamic properties of programs as needed rather than incrementally. Our emphasis on "light weight" trees is in contrast to other structure editors that incrementally maintain semantic information threaded through the tree. An example of this approach is our implementation of identifier definition site binding. (This will be discussed in a later section.)

Thus, the database elements, called TreeNode s, are simply linked tree nodes with a tag field that points to the appropriate grammar production description. While the basic structure of the database remains simple, our implementation also provides flexible interaction with a multitude of tools. This flexibility is supported by a LISP-like property list field in each TreeNode into which tools can store specialized information.

Integration and Object Oriented Philosophy

The static representation of the database must not only be clean, but the kernel is required to orchestrate the random and frequent incremental processing that is characteristic of interactive environments. To foster tool integration we have adopted an object-oriented perspective. By object oriented, we mean that we view each node in the tree as an instance of a class. The class plays the role of a type in a standard high level language, determining what static fields are available in each node, and what functions are defined over that type. In addition, classes are arranged in a hierarchy so that behavior specified once can be shared by each class that is declared to be a subclass. Thus, a subclass refines the behavior associated with its parent class.

The most general class in our system is the TreeNode. Any behavior that the implementor needs to apply to every node in the system can be specified by including it in the class specification of TreeNode. At the next more specific level, we distinguish between NonTerminals, Terminals and Metanodes. Finally, refining NonTerminals, we get to classes that represent individual productions of the language. Thus, a particular IF node is an instance of the class IF. Via inheritance it behaves like any other NonTerminal, as well as a TreeNode. (See Figure 2 for a graphic description of this hierarchy.)

While the class hierarchy groups together node types that behave similarly, the actual internal events that occur on nodes are defined around the tool-independent events CREATE, DELETE, EXPUNGE, and MODIFY. Behavior can be associated with the tuple [EVENT, CLASS]. For example, when the identifier of a type declaration (TypeId) is deleted, the code associated with [DELETE, TypeId] is executed. In addition, code associated with its superclass (IdDef) is invoked through the tuple [DELETE, IdDef].

The Benefits of This Approach

The object-oriented programming style seems to be a particularly effective integration technique for interactive programming environments. An interactive system is characterized by multiple subsystems, each contributing conflicting requirements, and unpredictable sequences of state changes. Ad hoc integration techniques usually fail to produce flexible and robust interactive systems.

We have found the class hierarchy to be an effective way to organize these requirements, allowing sharing where appropriate. Each tool or subsystem can group together classes of nodes that behave similarly, allowing for a compact description of the behavior of any node.

The availability of the class hierarchy and the identification of the basic internal transition events has led to a disciplined approach to new tool integration. To add a new tool, the implementor must decide how his tool is to act in response to each of the basic transition events. The system guarantees that if his tool can handle any transition, it will work in the face of unpredictable events triggered by user editing commands and the integration of other tools. The class hierarchy gives the implementor the ability to share specification of behavior among nodes, thereby simplifying his problem of specifying the behavior of every node.

---

1 In object oriented terminology, the "messages sent to the instances."

---

![Figure 2—The class hierarchy](from the collection of the Computer History Museum (www.computerhistory.org))
Programming Environments Based on Structure Editing

An Implementation: Getting Pascal To Do It

Fortunately, we can reap the benefits of object-oriented programming without the cost of implementing a full-blown object-based system. This is done by exploiting our stylized and well-defined use of objects. Rather than treating every data structure as an object, only TreeNodes are treated as objects. Because the class hierarchy is known in advance, we can coalesce the memory requirement for a TreeNode into a simple Pascal variant record (see Figure 3). Finally, since all message names are known, we can simulate "method lookup" by using case statements.

For example, in Figure 4, sending the message DELETE to an arbitrary object is handled by a routine XXDelete that glues together all the code pieces specified for each tuple matching [CLASS, DELETE]. In this case, deletion of a type definition causes its name to be removed from the menu of available types in addition to any other action specified for TreeNodes in general.

The real benefit of the object-oriented approach has been the order it imposes on the complexity of an interactive system consisting of a large variety of node types. It has enabled us as implementors to formalize our task: for each subsystem, and each node type, completely specify its behavior for each internal transition event, and to simplify its solution by grouping common behavior into a single piece of code, and knowing that the Kernel guarantees to invoke the code appropriately.

DISPLAY

While the internal representation of a program is a tree, the tree must be displayed as text and graphics on a screen in order for a programmer to view it. The process of mapping the abstract internal structure to concrete, visible forms is often called unparsing, and the part of the kernel devoted to this operation is called the unparsers. The quality of the interaction between a user and the programming environment will depend largely on the ability of the unparsers to present flexible, powerful, and insightful projections of the underlying structure. As such, the unparsers is one of the critical components of design.

The User's View

There are four important display issues for the user:

1. Appearance. Programs should make appropriate use of fonts and font styles, indentation, and graphics. Formatting should be sensitive to the amount of space available; it should be possible to elide* parts of the program, and the representation of nodes should conform to window widths.

2. Response. Small changes must result in quick updates. While a user may be willing to wait for the system to respond to major global changes, small local changes need to be reflected immediately.

3. Functionality. The display must support the usual window operations: scrolling, paging, mouse selection, window dividing, etc. It is also extremely valuable for the user to be able to edit textual items "in-place."

---

```pascal
TYPE TreeNode = RECORD
    classTag : ClassKinds;
    father, sibling : TreeNode;
    propertyList : AttributeValuePair;
    CASE Shape : ImplementationKind OF
    NonTerminal : (children : TreeNode);
    Terminal : (value : string);
    END: { TreeNode }
END; { TreeNode }

PROCEDURE XXDelete(aNode : pTreeNode; aTag : ClassKinds);
    BEGIN
    {First, handle the most specific CLASS }
    CASE aTag OF:
    TypeId: {Code for [DELETE,TypeId]}
        RemoveNameFromTypeMenu(NameOf(aNode));
    IdDef:
        {Code for all Identifier Definitions on DELETE.}
    TreeNode:
        {Behavior common to all TreeNodes.}
    END;{CASE}
    { Now, let the SuperClass also be notified }
    IF IsDefined(SuperClassOf(aTag)) THEN
        XXDelete(aNode.SuperClassOf(aTag));
    END;

Figure 3—The TreeNode record type

Figure 4—The DELETE event dispatcher
```

---

*By *elide* we mean the ability to suppress visual details. The elided form of "IF a < b THEN x = 2;" might be "IF a < b THEN .... "

From the collection of the Computer History Museum (www.computerhistory.org)
example, to change the phrasing of a comment, the user should be able to point to the comment and simply edit it as he would edit any piece of text in a text editor. Finally, it should be possible to tailor the details of appearance to personal taste. For example, one user might prefer Pascal keywords to be italicized while another might prefer bold face.

4. Views. It must be possible to view the program from different perspectives. For example, a *documentation view* might suppress details of the code; a *code view* might suppress details of documentation.

The Implementor View

For the implementor, the requirements of flexibility and efficiency itemized above present a significant challenge. In particular, there are two design issues to resolve: the kind of language that the implementor will use to express the desired formatting and conditionality, and a way to make it run fast. We now sketch the approach we have taken to these two issues. (For a more thorough treatment see Garlan.9,11)

The parse specification language

The language for specifying the representation of a node must allow the implementor to attach to each node a *collection* of representations that represent the way the node will be formatted in its various contexts. Multiple views and conditional formatting imply the need for multiple representations. This in turn requires some way to specify how the format for a node is to be chosen from the many alternatives. The problem of control of unparsing presents two conflicting goals, first because there is a hierarchical relationship between nodes, a node must be able to influence the choice of representation of nodes in its subtree. For example, if the user indicates that a procedure is to be elided, all the nodes in the procedure subtree must be informed of the fact, and second because the display of nodes is quite idiosyncratic, the node itself must be responsible for determining how it should look. For example, display of an IF-THEN-ELSE is quite different from the display of a WHILE-DO. Somehow, an unparse scheme language must allow both kinds of control to take place.

Our solution is illustrated in Figure 5, which shows the unparse specification for the IF-THEN node in the *code view*. The important points to note are as follows:

1. Multiple views allow different such specifications. Thus a *documentation view* of this node would have a completely different set of formatters.
2. For a given view, the representation of the node is determined by a set of condition-action pairs. When the node is to be unparsed, each *condition part* is evaluated. The first condition to succeed triggers the corresponding action, or scheme part of the specification. (Since the last condition is always "TRUE," one scheme will always succeed.)
3. The scheme part of the specification itself allows space-dependent conditionality. In the example, the "Horiz-Vert" notation indicates that the IF will be formatted horizontally if there is room; vertically otherwise.
4. The condition part of the scheme can make use of perspectives such as *ellipsis*. These perspectives are passed down from the parent node, thus allowing one node to influence the behavior of its subtree while maintaining autonomy at the node itself. In the example, the *ellipsis* perspective causes the IF node to be elided.
5. The values of attributes of a node can influence how it is displayed. In the example, the *err* attribute of the node causes a special kind of highlighting (defined by the implementor). Because comments in our system are treated as attributes of a node, the existence of a comment on the IF-THEN node causes the comment to be displayed in the documentation view of the program.

The behavior of a node embodied in this style of unparsing is very much in keeping with the object-oriented approach discussed above. Nodes are viewed as individual and idiosyncratic objects which have considerable freedom in determining their individual behavior.

```plaintext
Node IF-THEN ::= cond: bool-exp then-part: statement

Default View code

Schema: ellipsis ==> "If <cond> Then ..."
       .err   ==> "@myhighlight(If <cond> Then <then-part>)"
       TRUE  ==> "@horiz-vert((If <cond>)(Then <then-part>))"

View documentation

Schema: .comment ==> "<.comment>"
       TRUE   ==> ""
```

Figure 5—Unparse specifications for the IF-THEN node
Efficient implementation

Implementations of unparsing can be characterized by the degree of incrementality that they allow. Early unparsers redisplayed the entire tree after every change to the tree. Typically, they would start at the root, interpreting the scheme associated with that node, and recursively each subcomponent of the node. While this has the advantage of being easy to understand and implement, it is unforgivably slow on large trees.

A second generation of unparsers sought to improve performance by unparsing only a portion of the tree. Ideally, only the nodes that would actually be visible would be unparsed after each change. This gives acceptable performance to unparsing, but requires that the unparsers go through some curious maneuvers to figure out where to start unparsing. The problem with this approach is that unparsing schemes had to be relatively simple in order for this calculation to be reasonable. In particular, the kind of conditionality that we outlined above is quite beyond the abilities of this strategy.

A third technique, the one we have adopted, only updates the screen for nodes that have actually changed. In order to make this extreme form of incrementality possible, we cache the information from previous unparsing in an unparsing tree (U-tree). The U-tree encodes all the concrete syntactic information associated with the current view(s) of the abstract tree. When changes occur in the abstract tree, the corresponding subtree of the U-tree is recalculated. Further dependencies within the U-tree are then allowed to propagate. Finally, some portion of the U-tree is written to a buffer for display to the screen. The calculation of the amount to be written to the buffer is now easy because all the size information has been computed in the U-tree.

SEMANTICS

While the syntactic and lexical guidance provided by a simple structure editor is extremely helpful, it still does not aid in detection and correction of semantic errors. There are three classes of semantics that require attention:

1. Static semantics—semantics that can be done at compile-time; for example, type checking of expressions, and identifier resolution.
2. Runtime semantics—semantics that depend upon the running state of the program tree; for example, checking that array index references are within bounds.
3. Environmental semantics—semantics that characterize the relationship between the interaction between the program editor and the editor environment; for example, user-settable key bindings, positioning of windows, etc.

Static Semantics

The choice of mechanisms to support static semantic checking is closely tied to the policy issue: What grain of incremental checking should be provided? There is a wide spectrum of choices. At one extreme, the editor could provide semantic feedback batch style as in a traditional compiler environment. Here the entire tree is reprocessed when checking takes place. Consequently, checking is usually done for large grain sizes.

At the other extreme, every change to the program tree could cause incremental rechecking of the changed pieces. For the implementor, the former choice may appear attractive because it allows him to integrate an existing stand-alone compiler directly into the editing environment. By contrast, the latter choice requires close cooperation between the editor and the semantic analyzer, and so a special-purpose semantic tool must be integrated into the system. However, for the user, the batch method can be awkward and even painful; there are frequently long delays between any change and re-checking of the program, and lists of errors can become quite long and may have little relation to the logical structure of the program since they are frequently organized by line number.

On the other hand, incremental error reporting at the finest grain can be intrusive and misleading. For example, if the user programs by first using procedures and then defining them, it can be quite annoying if the system keeps reminding him of the undeclared items.

An appropriate solution is one which provides a flexible enough underlying mechanism such that policy can be tuned for maximum benefit and minimum frustration. In doing this, the mechanism may not support either extreme, but will provide an efficient implementation for a wide range of grain sizes of incremental semantic interaction.

In implementing incremental semantic processing, there is an important tradeoff between the amount of incremental semantic information, state, that is stored and the amount of recalculation that is done when the program is to be checked. In the previous two examples, the batch method requires no saved state and does all checking from scratch each time while the highly interactive method benefits from as much semantic state as can be made available. Unfortunately, the more state that is stored, the more computational overhead is required to maintain it during editing. For example, if the definition site of an identifier is kept with every one of its use sites, any change to the definitions in an enclosing scope requires updating all use sites.

We have found that by making a judicious choice of the information stored after each semantic check, it is possible to provide highly incremental semantics with a minimal amount of incremental overhead. We now illustrate how this is done with respect to identifier resolution for a Pascal environment.

In order to perform most of the static semantic checking for Pascal, it is first necessary to associate uses of an identifier with their definition site. This might seem relatively easy to do; after all, Pascal was designed to be compiled in one pass. But in an interactive environment, we cannot afford to process the complete tree (batch style) every time any semantic checking takes place. One solution to this problem is to calculate the definition site incrementally with every use and store the value with every use site. But this has the disadvantage, mentioned above, that any change to variable declarations requires a complete reprocessing of the affected scope.

An alternative solution that we have adopted keeps no use-definition associations with definition sites, but provides a
simple mechanism for a quick mapping from an identifier use to its definition site. The mechanism is based on the fact that at any point $P$ in the tree, the naming environment at $P$ is completely determined by the set of modifier nodes on the path between $P$ and the root of the tree. The modifier nodes include the "." (field selection) operator, the WITH statement, and any declaration lists. The naming environment of $P$ can thus be taken to be a stack of modifiers. Consequently, to resolve a use site of an identifier, we initialize a stack of modifiers by walking up the chain of fathers of the use site, pushing an appropriate modifier representation as we encounter it. Once this is done, the use site can be resolved by scanning through the stack until a modifier is found that defines the current use. Subsequent identifier resolutions need not reinitialize the stack each time, but the modifier stack is kept current as the semantic analyzer moves from one node to another.

The success of this technique depends on the fact that Pascal program trees are not very deeply nested, that only a few nodes in the tree can modify the naming environment, and that the information stored at each modifier is in an efficiently usable form. The last criterion is met by allowing each modifier to be, in effect, a symbol table. For example, the modifier for a WITH statement would be a symbol table containing the fields of the record type specified in the WITH statement. In this way, pushing a modifier on the stack involves merely copying a pointer, and resolving an identifier requires looking up the name in each modifier symbol table until one is found that defines it.

In a sense, we have a semi-batch solution to the problem. Because name look-up is cheap through the calculation just outlined, we can avoid the overhead of fully incremental semantic state and still achieve a wide range of policies. Experience has shown that the efficiency of this technique is more than adequate for all but the smallest grain sizes of incremental checking, and for all but large, heavily nested programs. The important general lesson to be learned from this is that it is not worthwhile to incrementally maintain information that is easy to regenerate on demand.

**Runtime Semantics**

For the user, the following are legitimate expectations of a runtime mechanism in an integrated system:

1. It should provide reasonable execution speed for debugged code.
2. It should support source level debugging.
3. It should give the user the ability to edit source code during the execution of a program.

The existence of the syntax tree as an underlying representation of a program makes it possible for an implementor to provide these facilities in an integrated system. We illustrate how these goals can be met by mentioning three details of implementation:

1. We allow a mix of compiled and interpreted code. To make this possible, compilation takes place at the grain size of procedure. Procedure calls are implemented by an indirect call through a jump table. Each entry in the table indicates whether the procedure is to be interpreted or executed as compiled code. Compiled procedures can then be replaced by their "interpreted" counterparts, or by new compiled versions simply by modifying the procedure jump table.
2. Interactive editing during execution takes place on the syntax tree itself. For example, breakpoints are set by adding a special BREAK statement to the source. (See Feiler and Medina-Mora 12 for more details on how this is done.) Modified procedures can then be interpreted or recompiled as desired.
3. To limit the amount of rechecking needed after a user makes a change to the source, we maintain with each procedure or function a list of non-local identifiers used. Thus, when a variable's declaration in a procedure is modified, we need only recheck nested procedures that actually use the affected variable, not every nested procedure.

The combination of these features can provide a powerful working environment for the user. A typical scenario might proceed as follows: The user starts out by building a simple program, one procedure at a time. After each procedure is built, it is tested by running the program with a testing call to the new procedure. At this point, the user would "run" his code using the interpreter in order to provide a quick run-edit-run cycle. Upon verifying these first few procedures, the user might tell the system to compile them, and move on to other tasks. Then, the program would run in a mixed mode, executing directly-compiled procedures while interpreting the newer code. This process would continue until the entire program is debugged and ready, and would result in a totally compiled program. However, a procedure may be edited and interpreted at any point in order to use the breakpoint facility or the source level debugger (the debugger would also work on the compiled code, but with a limited functionality).

**Environmental Semantics**

The interaction between the user and the program can be seen as another set of semantic interactions within the program. This notion is quite powerful in that it abstracts the real-time interaction away from the core functionality of the program. This enables the internal mechanisms to function solely based on state and goal information.

For the implementor, the operative words are extensibility, predictability, and flexibility. We have found that the most useful implementation is one whereby all user interactions flow through a dispatcher. This clearing house for commands keeps the external interactions well removed from the internal operation, and allows the user interface to be incrementally adjusted as development of the environment progresses. In our system, all internal operations are defined independently of the way they are invoked in the user interface; for example, it could be from the keyboard, a file, the mouse, or any input device. The user is allowed to define the mapping from real-world events to internal events. Of course, it is necessary to
establish defaults of various kinds, particularly in an environment meant for novice users. The novice need not even be aware of the features that can be changed, but we believe it is important to allow considerable flexibility for those who wish it. It is possible to anticipate and accommodate all sorts of preferences ranging from fonts and highlighting to cursor movement. Menus and interactive windowing, among other methods, make such changes easy, and can be saved in a user's "profile" for future use.

CONCLUSIONS

The use of integrated programming environments based on structure editing is an emerging technology that has reached the stage of being both demonstrably useful and readily implementable. We have outlined some of the salient aspects of our work in developing such environments, and suggested paths of implementation that seem to be worth traveling. A predominant theme has been the need to separate policy from mechanism. While the choice of user interface policies will probably differ widely from those presented here, the mechanisms that we have sketched nonetheless will be applicable to future environments.

One of our primary motivations in developing GNOME and Mac GNOME has been the desire to improve the learning environment for novice programmers. GNOME has already had a marked impact on our students. They are learning more, in less time, than they did a few years ago. Moreover, we think that they are approaching programming in a different manner following the deemphasis on details and emphasis on modular problem solving that the programming environment encourages.

Syntactic errors are not possible. Almost all semantic errors are either prevented from being made, or are caught in an incremental manner, immediately or on exit from scope when constructing the program. Students thus are unlikely to accumulate huge numbers of seemingly unrelated errors. Errors are reported in context by scope, with the offending code highlighted in inverse video. Messages are less cryptic than one sometimes sees. Particularly useful editor commands in GNOME include those that allow for easy checking for semantic errors by scope at any time, and for consistency between actual and formal parameter specifications.

Because we have been developing these environments incrementally at Carnegie-Mellon, we have not yet made a formal assessment of their impact. We are planning to test the GNOME environment elsewhere, where we can take advantage of better quasi- and field-experimental control conditions. Much of our evidence about GNOME's effectiveness is based on our informal impressions. Admittedly, other factors have also changed over the period during which we have been developing these environments. Still, that period has been concurrent with a marked rise in student performance on their mastery examinations. That is compelling evidence to us, particularly since it has occurred in spite of the fact that our student body has increased in size and heterogeneity.

ACKNOWLEDGMENTS

Thanks are due particularly to Lee Miller and Robert White for their many contributions to the ideas developed in this paper.

REFERENCES
