Evolution of Novice Programming Environments:

the Structure Editors of Carnegie Mellon University

Philip Miller

plm+@cs.cmu.edu

John Pane

pane+ile@cs.cmu.edu

Glenn Meter

gcm+@cs.cmu.edu

Scott Vorthmann

vorth+@cs.cmu.edu

Computer Science Department

Carnegie Mellon University

Pittsburgh, PA 15213-3890

Abstract

Beginning in the early 1980's, the Computer Science Department at Carnegie Mellon University developed and used three generations of novice programming environments. The focus of these systems was to apply, advance and tune structure editor technology in support of the teaching and learning of computer programming. The use of these pedagogical systems in Carnegie Mellon's introductory programming courses provided feedback and inspiration to guide the projects. This paper tracks the evolution of the programming environments and courses, documenting important lessons and discoveries about novice programmers and the environments that support them.

Introduction

This paper describes three projects in the Computer Science Department at Carnegie Mellon University that, beginning in the early 1980's, developed a series of novice programming environments based on structure editors. Through a chronology we present ways these projects refined and advanced structure editor technology in support of CMU's introductory programming courses. In the context of this evolution of technology, we track a corresponding evolution in the emphasis and philosophy of introductory programming instruction, noting lessons learned along the way.

There were three distinct novice programming environment projects; GNOME, MacGnome, and ACSE. The first project, GNOME, used structure editor technology to assist novices in the creation of programs that are free from syntax errors. The second project, MacGnome, attended to system usability and the higher-level skills of planning, abstraction and program visualization.

Finally, ACSE marked an educational shift away from the mechanics of programming, toward the use of programming as an enabling tool for modelling; thereby leveraging understanding in other domains. The ACSE shift was supported by a transition to object-oriented programming, multimedia, literate programming and semantic tools.

Basics of Structure Editors

To understand the CMU programming environments it is important to contrast them with the widely used environments that are, as a rule, not based on structure editors. These traditional environments offer a text editor, compiler, debugger, and run-time environment for program construction, compilation, debugging, and execution, respectively. The simplest text editors operate on an internal program representation that is a stream of characters in a file, perhaps separated into lines of text. More powerful text editors assist the user by detecting keywords of the language, by indenting, or by matching the boundaries of comments and other structures. When it is time to run the program, the whole file of text is fed to the compiler. The compiler is integrated with the editor through line numbers, which are used to refer errors to particular lines of text in the editor. This is true even though most text editors do not display the line numbers. When a compiler begins to analyze a text program, it starts with a syntactic operation called *parsing*. In parsing the compiler constructs a *abstract syntax tree* representation of the program and proceeds to analyze and compile executable code.

Structure editors, in contrast, operate directly on an abstract syntax tree. As a program is created, the structure editor builds up the abstract syntax tree, with each edit consisting of a transformation from one syntactically correct program to another. Each legal transformation is determined by a formal specification, known as the *grammar*, of the programming language. The grammar provides the structure editor with the information needed to constrain edits to those that yield syntactically legal programs. For any location in a partially complete program, the structure editor typically provides a menu of the syntactically correct transformations.

The abstract syntax tree is a concise, but not human-readable, representation of the program. Because of this, structure editors display the program textually, through a mechanism known as *unparsing*.[1] Any required parts of the program that have not yet been specified by the programmer are unfilled nodes in the abstract syntax tree and are represented in the textual output by placeholders. The process of writing a program using a structure editor has the programmer repeatedly choosing from menus of legal substitutes to fill these placeholders. The editor automatically embellishes each edit with the required syntax, such as keywords and punctuation.

The first novice programming environment project at CMU set out to bring three perceived advantages of the structure editor to student programmers:

- generation of programs should be easier because the student is guided by menus of legal operations;
- long sessions of searching for and correcting syntax errors should be eliminated, because the structure editor keeps the program in a syntactically valid state at all times; and,
- the editor's partially-compiled knowledge of the program should facilitate new ways of viewing and operating on the program that are difficult or impossible with unstructured text files.

This application of structure editing to novice programming was inspired by projects at Cornell and Carnegie Mellon, described in the next section.

1. As discussed below, the unparser is also used to generate graphical visual representations of the abstract syntax tree.

Ancestors

Cornell Program Synthesizer

In the spring of 1978, Tim Teitelbaum demonstrated the first Cornell Program Synthesizer at the ACM Computer Science Conference: Special Interest Group on Computer Science Education (Reps & Teitelbaum, 1989). That programming environment implemented a subset of PL/1. Student programs were limited to a maximum of 24 lines of code and no sub-programs were allowed. Despite these obvious limitations, the Cornell Program Synthesizer was, from the point of view of the teacher of introductory programming, a thing of beauty. It eliminated an intellectual barrier that was preventing students from learning the principles of programming: the syntax error. The synthesizer always presented the allowed set of commands and never missed an *END* or a semi-colon. Just as importantly, it provided an integrated environment that displayed the program's statements and visually traced program code one statement at a time as it ran. The output from program execution and the program code which produced that output, could be clearly seen alongside user input. This made it easy for students to understand the nature of the von Neumann machine: sequential execution of instructions causing changes to program state. After surveying the state of introductory programming courses at a number of leading universities in the country, one of the authors, who is the Director of Introductory Programming at CMU, was convinced that structure editing had a place in teaching novices to program.[2]

At that time, Carnegie Mellon was teaching Pascal in its introductory programming courses. The curriculum focused on teaching problem representation in elementary and intermediate data structures, problem decomposition through procedural abstraction, and elementary reasoning

2. Philip Miller has directed introductory programming at Carnegie Mellon University since 1979, and has been Principal Investigator of all of the CMU novice programming environment projects described in this paper.

about program complexity and correctness. An editor constrained to 24 lines of a subset of PL/1 that did not include sub-programs was not acceptable. The development of a full Pascal programming environment was not a part of the agenda for Teitelbaum's group at Cornell, so another solution had to be found.

Gandalf

In addition to the effort at Cornell, major projects were underway at Bell Labs, The University of Wisconsin, Carnegie Mellon, and a number of other places. The project at Carnegie Mellon was the Gandalf project (Habermann, Garlan & Notkin, 1991; Habermann & Notkin, 1986). At the inception of our work, Gandalf had already built a system that allowed structure editors to be automatically generated from language descriptions. The Gandalf structure editors were called ALOEs, for **A L**anguage **O**riented **E**ditor. The tool that generated an ALOE was written by Raul Medina-Mora (Medina-Mora, 1982). This tool made it relatively easy to build structure editors. Gandalf also provided a language for describing program *views*. Views are alternative mappings of the abstract syntax tree into visual representations that are useful for the programmer. In Gandalf views were implemented by unparsing the abstract syntax tree (Garlan, 1987). We adopted these Gandalf technologies as the basis for our first novice programming environments, and named the project GNOME.

The GNOME Environments

With the philosophy that different ways of looking at programs are useful in different contexts, GNOME used the unparsing facilities to provide several views (Garlan & Miller, 1984). Since problem solving through decomposition into subprograms is a fundamental principle that has been taught in introductory programming since the 1960s, the dominant program view of

GNOME displayed the program in a hierarchy, one procedure at a time, with only the headers of nested subprograms shown. This guided students to think about programs as collections of high-level procedural abstractions, rather than as an unstructured sequence of hundreds of lines of code.

The GNOME environments took advantage of the abstract syntax tree representation of programs to include some incremental checking of static semantics. This facility ensured that variables were declared, that types matched, and that parameters were bound correctly. This semantic analysis was provided in two ways:

- the programmer could request an analysis at any time; or,
- if semantic errors remained when the programmer attempted to move the edit cursor outside the current lexical scope, they were reported at that time.

This was a first attempt at reporting errors in context, balancing the intrusiveness of immediate error reporting with the desire to provide useful information to the programmer while the ideas surrounding the code in question are still fresh in mind.

The GNOME family of environments ran on the Unix operating system on two DEC Vax 11-780s that were dedicated to the course.[3] To integrate a runtime environment, the entire program was unparsed into a text file and sent to *pi*, the Berkeley Unix Pascal interpreter. The interpreter performed its own syntactic and semantic analyses (detecting no errors when our semantics were fully and correctly implemented), and then generated code that was finally interpreted.

We created GNOME environments for Karel the Robot (Pattis, Roberts & Stehlik, 1995), Pascal, FORTRAN, and LISP. The LISP system never competed successfully with the available LISP environments. The FORTRAN environment allowed us to successfully teach this language

3. It is hard to believe it today but that pair of computers and the 45 video terminals cost nearly one million dollars.

as a 2-week unit inside the Pascal course, and was used until we no longer taught FORTRAN. The Karel and Pascal environments were heavily used for five years. Evaluation clearly suggested that the environments had a positive educational impact (Scheftic, 1985; Scheftic, 1986; Scheftic & Goldenson, 1986), but there were obvious problems with these systems.

Syntax and Usability in GNOME

It may be somewhat surprising that instead of banishing syntax as an issue in novice programming, the GNOME environments actually elevated it. While it is true, for example, that keywords were always spelled correctly and every *begin* had a matching *end*, there were three serious problems with the GNOME implementation:

- navigation required that the student build a mental model of the abstract syntax tree;
- the abstract syntax representation of a program, in some cases, differed from the textual representation; and
- the structure editor's rigid requirement of syntactic correctness inhibited the user from making some desired transformations of the program.

The first problem arises from the fact that the structure editor maintains a *current selection*, which is a node in the abstract syntax tree. At all times, the current selection is mapped by the unparser to the corresponding text on the user's screen. Ordinary text editors usually provided (in those days before the proliferation of the mouse) an insertion point that could be moved to an arbitrary place in the text file by moving forward and backward by characters or sometimes by words, and up and down by lines. In structure editors, the current selection is almost always a range of text, and it is meaningless to move by characters or lines of text because those units do not map uniquely to a node in the abstract syntax tree. The GNOME environment used arrow keys (up, down, left, right), which corresponded to movements in the abstract syntax tree (parent, first child, previous sibling, next sibling). But, mapping from a desired movement in the visible text representation into a correct sequence of navigations through the abstract syntax tree was often maddeningly counter-intuitive. Mastery of navigation in this system required the user to build a mental model of the abstract syntax tree. While this is pedagogically useful in some cases, like understanding the block structure of the language, it is meaningless detail in most other parts of the language.

As an example of the second problem, consider the dangling-else problem in Figure 1. In this ambiguous situation, the Pascal language specifies that the *else* clause belongs to the inner *if* statement (b2). In order to attach the *else* clause to the outer *if* statement, a *begin-end* block is required around the inner *if* statement. In this situation, GNOME had two distinct locations in the abstract syntax tree for *else* clauses, one for the outer *if* and one for the inner *if*. Ideally it would have prohibited the filling of the outer *else* clause in situations like this, but there was no way to specify this rule in the grammar from which the editor was generated.

```
if b1 then
    if b2 then
        s1
else
    s2;
```

Figure 1: Dangling-Else Problem

This led to a very misleading bug: the GNOME system permitted a student programmer to build this structure, and then, through indentation and the navigation mechanisms, supported the student's mistaken belief that the *else* really was attached to the outer *if* statement. Since this situation is actually not ambiguous to the structure editor, perhaps the dangling-else rule could have been eliminated in our system, as a deviation from the Pascal standard. The user could then trust the screen representation. Unfortunately, we were not able to do this because the GNOME system communicated with the Pascal interpreter through a text file, re-introducing the ambiguity.

The third problem arises from the structure editor's strict invariant that the program is always syntactically correct. In practice, this places an unnecessary burden on the programmer, because sometimes the most direct or obvious way to transform one legal piece of structure into another requires temporary violation of syntactic correctness. Without this flexibility, the user is required to jump through hoops to make these changes, often retyping, rather than rearranging, the code. The ALOE was able to solve only a subset of these cases with an operation called "transform". This operation was useful in cases where a nonterminal node in the abstract syntax tree is being mapped to another nonterminal node with identical children. It could be used, for example, to change an *If-Then* statement (but not an *If-Then-Else* statement) into a *While* statement. The effect in this case is to replace the control structure without disturbing the conditional or controlled statements: *if b then s;* becomes *while b do s;*. There are edits, however, where transform is not available. Consider the expression *a = (b or (c = 0)),* where the correct parenthesization is *(a = b) or (c = 0).* In GNOME this transformation, while textually quite simple, required a complex set of actions at least as difficult as re-typing the entire expression.

In addition to these failures to eliminate syntax as an issue, there were other shortcomings with the GNOME environments. First, our integration with the existing Unix tools saved a lot of tool-building work, but inhibited our ability to assist students in understanding and debugging their running programs. Second, we were using character-based terminals on a time-sharing system. The machines were slow, and the communication channels between the user and the system were not rich enough.

Lessons Learned in GNOME

Some kinds of syntax problems were banished by the GNOME environment, such as those involving punctuation or balancing of begin-end keywords. But in many other cases, issues of syntax were still prominent. Even so, the GNOME environment did have a positive impact on our students. There are three features that may have contributed to this advantage:

- the system presented the program in a hierarchical form that supported abstraction;
- the system offered non-intrusive incremental semantic checking;
- the syntax issues were presented incrementally, in context, rather than in batch mode at compilation time.

It was clear, however, that the system suffered from usability problems. We addressed many of these in our next system.

First Generation Genies: the Macintosh, MacGnome, and External Users

The advent of the Macintosh personal computer in the early 1980s presented an opportunity to address many of the problems we had identified in the GNOME environments. With support from Apple Computer, we began the MacGnome project. The novice programming environments produced by this project were called Genies.

Solving GNOME’s User Interface Problems

Among the biggest contributions of the MacGnome project were the careful incremental refinements we made to the user interface of the structure editor. Primarily, these refinements endeavored to provide an intuitive integration of structure editing with the text-editing model that is ubiquitous in word processors and traditional programming editors.

The Genie environments, built on the Macintosh graphical user interface, solved the problem of navigation by using the mouse for selection and traversal. The focus can be moved to any node by simply pointing and clicking. The Genie always navigates to the smallest piece of structure that contains the mouse click. If a larger piece of structure is desired, holding the mouse down while dragging interactively selects increasingly larger pieces of structure as the mouse gets further from the location of the initial click. Some basic manual dexterity is required of the user, but the code navigation problems of GNOME were effectively eliminated by incorporating this modern user interface.

In the Genie environments, the approach to the problem of program transformation is to allow textual editing. When the user enters this mode, the selection is converted into text, where it can be edited without syntactical constraints. When the user is ready, the text is changed back into structure using an incremental parser.

Textual editing also solves another problem that existed in the GNOME environments. Those systems required the structure to be built using prefix notation. Thus, to build the expression *(a = b) Or (c = 0)*, the following sequence of commands had to be entered: *or = a b = c 0*. Until the user built up a correct mental model of the structure being built, this was difficult. With textual editing, this expression can be typed exactly as it will appear in the program.

Both of these text editing operations in the Genie temporarily subvert the benefits as well as the inconveniences of the structure editor. As soon as the user is in text-editing mode, syntax errors are possible. When the text-editing operation is finished and the parser is invoked to convert the text into structure, a parser error may be reported. Fortunately, the error is reported in context, shortly after it was typed by the user. But even in the absence of a parser error, sometimes text is successfully parsed, but the resulting structure is not what the user expected.

In order to discourage the user from generating large changes to the program while in text-editing mode, and thus risking numerous parsing errors, the Genie automatically invokes the parser at certain sensible times. In Pascal, this is done each time the user types a semicolon (except as part of a literal string or a comment). This keystroke was chosen because it is a list separator for most lists, such as statement and declaration lists. Parsing also happens when the programmer clicks outside of the text currently being edited, to automatically prepare for the next edit. This automatic parsing forces the user to deal with syntax errors soon after they are introduced, while gracefully allowing continued typing if no errors are found.

The incremental parser is also a convenient way to provide macros for building structure, because it accepts partial statements. For example, the parser can build the complete structure for a For loop when the user simply types *for <enter>*. Figure 2 shows the result. Placeholders are generated where additional code is syntactically required. For example, *$control-variable$* is a placeholder for the loop control variable. The user completes the For loop by selecting from construct menus which popup from the right-pointing triangles, by typing code, or some mixture of the two. Of course, the popup construct menus contain only the syntactically legal constructs. For example, the construct menu for the *$statement$* placeholder only lists valid statements.

```
For $control-variable$ := $start-value$ $direction$ $end-value$ Do
    Begin
        $statement$
    End
```

Figure 2: For Loop With Place-Holders

In structure editors, the program is built by filling placeholders. One of the problems of structure editors is the management of placeholders: inserting new ones when you want to add more structure, and discarding the ones that you do not need. The Genies use the following heuristic to add new placeholders: add a new placeholder whenever an existing placeholder is filled in, but do not add one when existing structure is replaced with another structure. This corresponds to the difference between *constructing* and *editing* code. In practice, we found that this heuristic works fairly well.

Often a new placeholder is needed between two adjacent list elements. This functionality is provided by the *extend* functions, which allow creation of a new placeholder before or after a selected list element. This is another area where GNOME was inconvenient, because it required precise selection of the correct list node. For example, consider the two-statement list in Figure 3. Suppose the user desires a placeholder after the *For* loop. In GNOME, the extend-after command would work only when the entire *For* loop is selected. If the user selected only the assignment statement *a[i] := 0;* or the *0* in that assignment statement, the extend operation was not available. The Genie attempts to provide a more intelligent extend, which allows extend-after to be used from any of those places to create the desired placeholder.

```
Begin
    For i := 1 To 10 Do
        a[i] := 0;
    sum := 0;
End;
```

Figure 3: Adding Placeholders with Extend

Now consider adding a second statement inside of the *For* loop of Figure 3. This requires the statements inside the loop to be nested within a new *Begin-End* block. After user testing, we found that novice users had a easier time in situations like this if we always build *Begin-End* in control structures (as shown in Figure 2). While these blocks are sometimes extraneous and take up extra screen space, novices did not care about this detail and liked the convenience of the new behavior. Of course the programmer is free to remove the extraneous *Begin-End* blocks if they are not desired.

The automatic creation of these *Begin-End* blocks also addresses the dangling-else problem, which was described earlier. However, it is interesting to note that because the Genie has an integrated compiler, it doesn't have to parse the program. Instead, it simply walks the abstract syntax tree to generate code. For this reason, the execution of the dangling-else is consistent with its visual display (indentation, etc.).

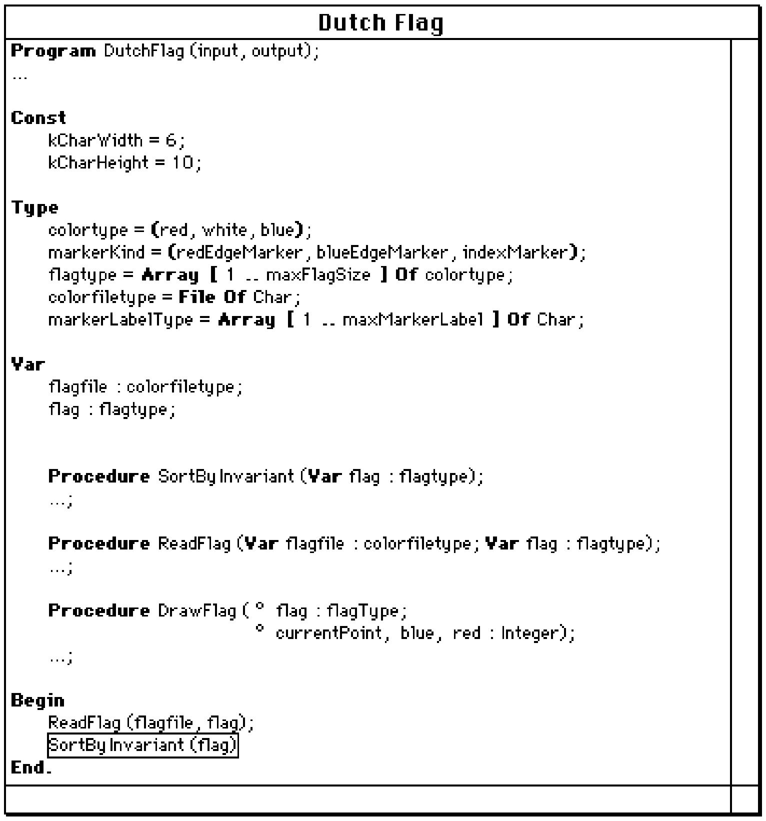
Another problem with the management of placeholders occurs with deletion of structure. Because the editor automatically provides additional placeholders as existing ones are filled in, eventually the user must delete the final placeholder(s). Consider the variable declarations in Figure 4. The GNOME environments were inflexible in deleting the remaining placeholders. They required that the entire line be selected. The Genie relaxed this restriction: as long as neither placeholder has been filled in, deleting either one deletes the entire line.

```
Var
   a : Integer;
   b : Real;
   $variable-name$: $type$;
```

Figure 4: Removing Placeholders

Advances of the First Genies

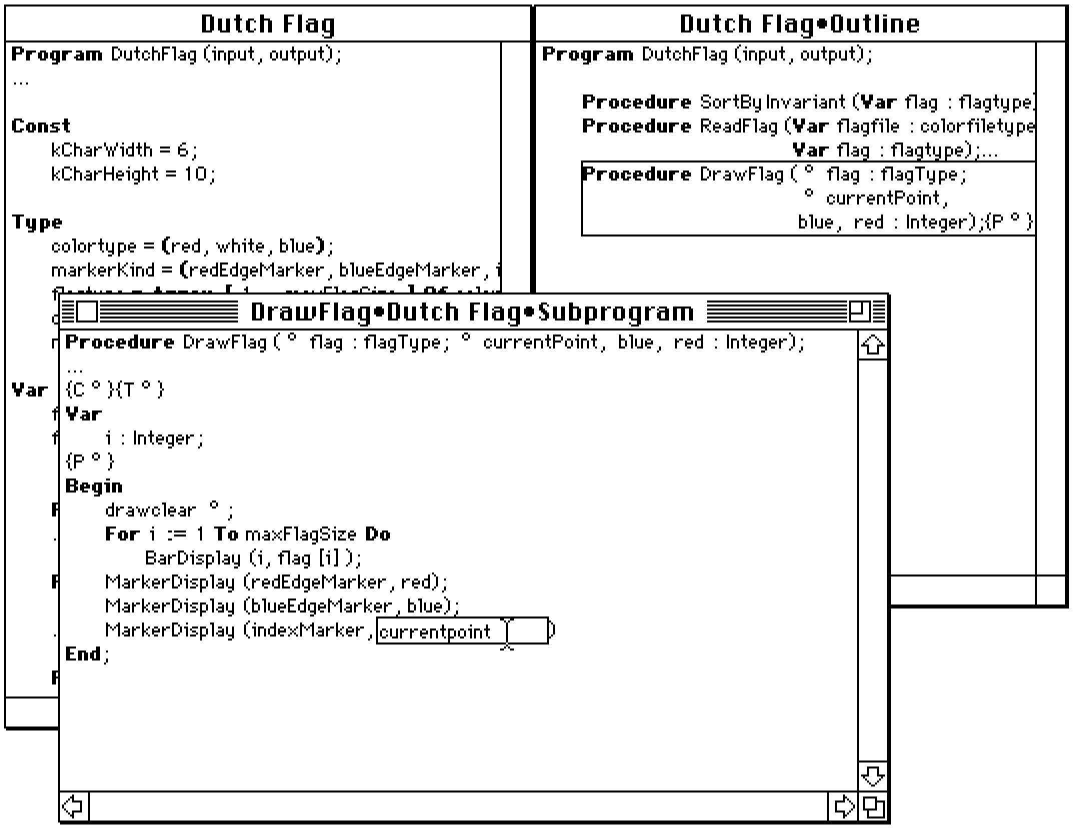
The graphical, window-oriented nature of the Macintosh user interface, together with a more advanced unparser, allows the Genie to provide multiple simultaneous views of the program. Each view is a projection of the abstract syntax tree, and provides tools for editing. Any change made in a particular view is mapped into a modification of the central abstract syntax tree database. Any view that displays that section of the database is then incrementally updated to reflect the change.


Dutch Flag

```
Program DutchFlag (input, output);
...

Const
     kCharWidth = 6;
     kCharHeight = 10;

Type
     colortype = (red, white, blue);
     markerKind = (redEdgeMarker, blueEdgeMarker, indexMarker);
     flagtype = Array [ 1 .. maxFlagSize ] Of colortype;
     colorfiletype = File Of Char;
     markerLabelType = Array [ 1 .. maxMarkerLabel ] Of Char;

Var
     flagfile : colorfiletype;
     flag : flagtype;


     Procedure SortByInvariant (Var flag : flagtype);
     ...;

     Procedure ReadFlag (Var flagfile : colorfiletype; Var flag : flagtype);
     ...;

     Procedure DrawFlag ( °  flag : flagType;
                          °  currentPoint, blue, red : Integer);
     ...;

Begin
     ReadFlag (flagfile, flag);
     SortByInvariant (flag)
End.
```

Figure 5: Pascal Genie 1.0 Code View

In the first generation Genies, the main view of a program is the code view, as seen in Figure 5. In this figure, the bodies of subprograms are elided (that is, replaced with “...”). In contrast with the GNOME environment, however, this elision is not automatic. In this case it was done manually by the programmer, using a operation that is available on any piece of structure.



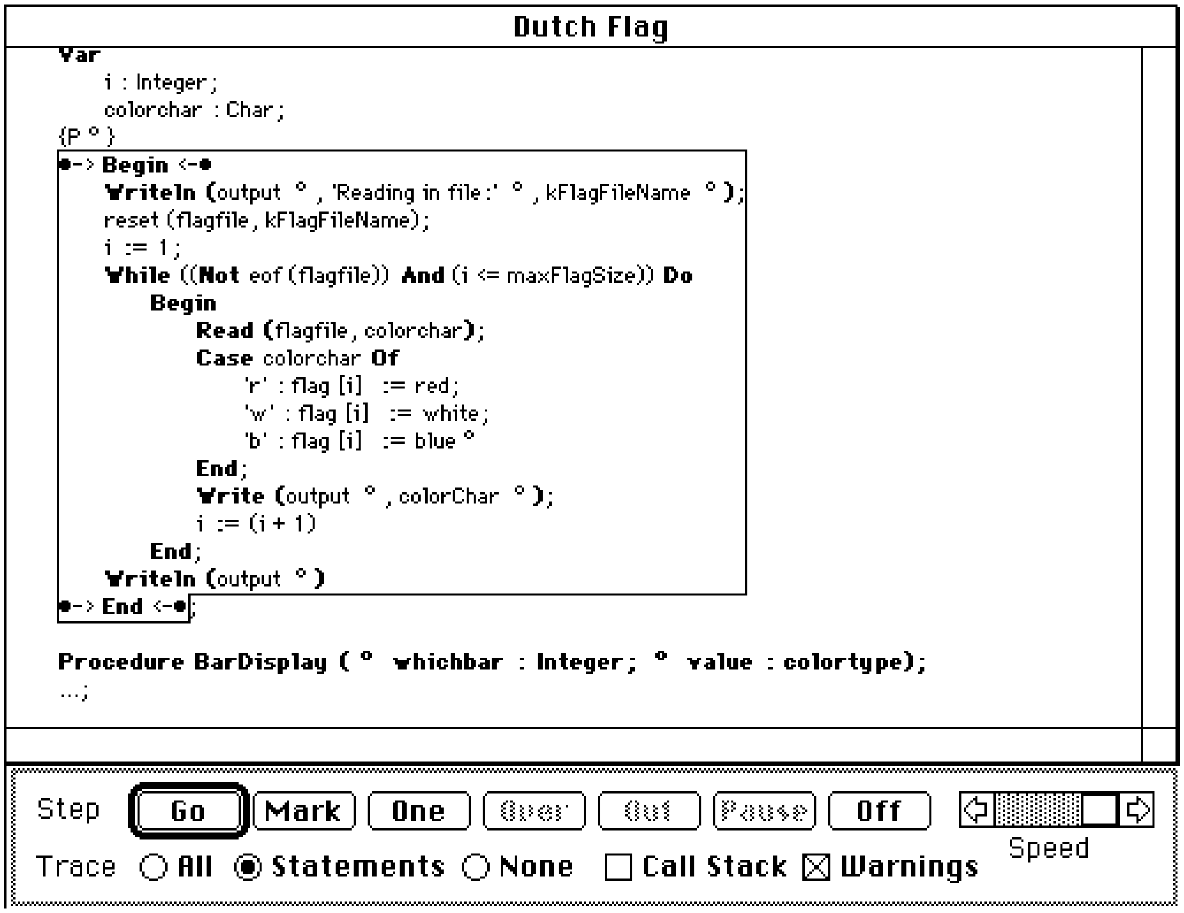
```
Program DutchFlag (input, output);
...

Const
     kCharWidth = 6;
     kCharHeight = 10;

Type
     colortype = (red, white, blue);
     markerKind = (redEdgeMarker, blueEdgeMarker,
```

```
Program DutchFlag (input, output);

     Procedure SortByInvariant (Var flag : flagtype
     Procedure ReadFlag (Var flagfile : colorfiletype
                         Var flag : flagtype);...
     Procedure DrawFlag ( ° flag : flagType;
                          ° currentPoint,
                         blue, red : Integer);{P ° }
```

```
Procedure DrawFlag ( ° flag : flagType; ° currentPoint, blue, red : Integer);
...
{C ° }{T ° }
Var
     i : Integer;
{P ° }
Begin
     drawclear ° ;
     For i := 1 To maxFlagSize Do
          BarDisplay (i, flag [i] );
     MarkerDisplay (redEdgeMarker, red);
     MarkerDisplay (blueEdgeMarker, blue);
     MarkerDisplay (indexMarker, currentpoint )
End;
```

Figure 6: Pascal Genie 1.0 Outline and Subprogram Views

Figure 5 illustrates some other views that are available. An outline view does elide all implementation details, showing only the headers of subprograms. The user can refer to subroutine headers through the outline view, and also navigate to a particular routine in the code view by double-clicking on its header in the outline view. Another view, the subprogram view, allows users to open up a window containing a single procedure or function. This makes it easy to work on a procedure while referring to other sections of code.



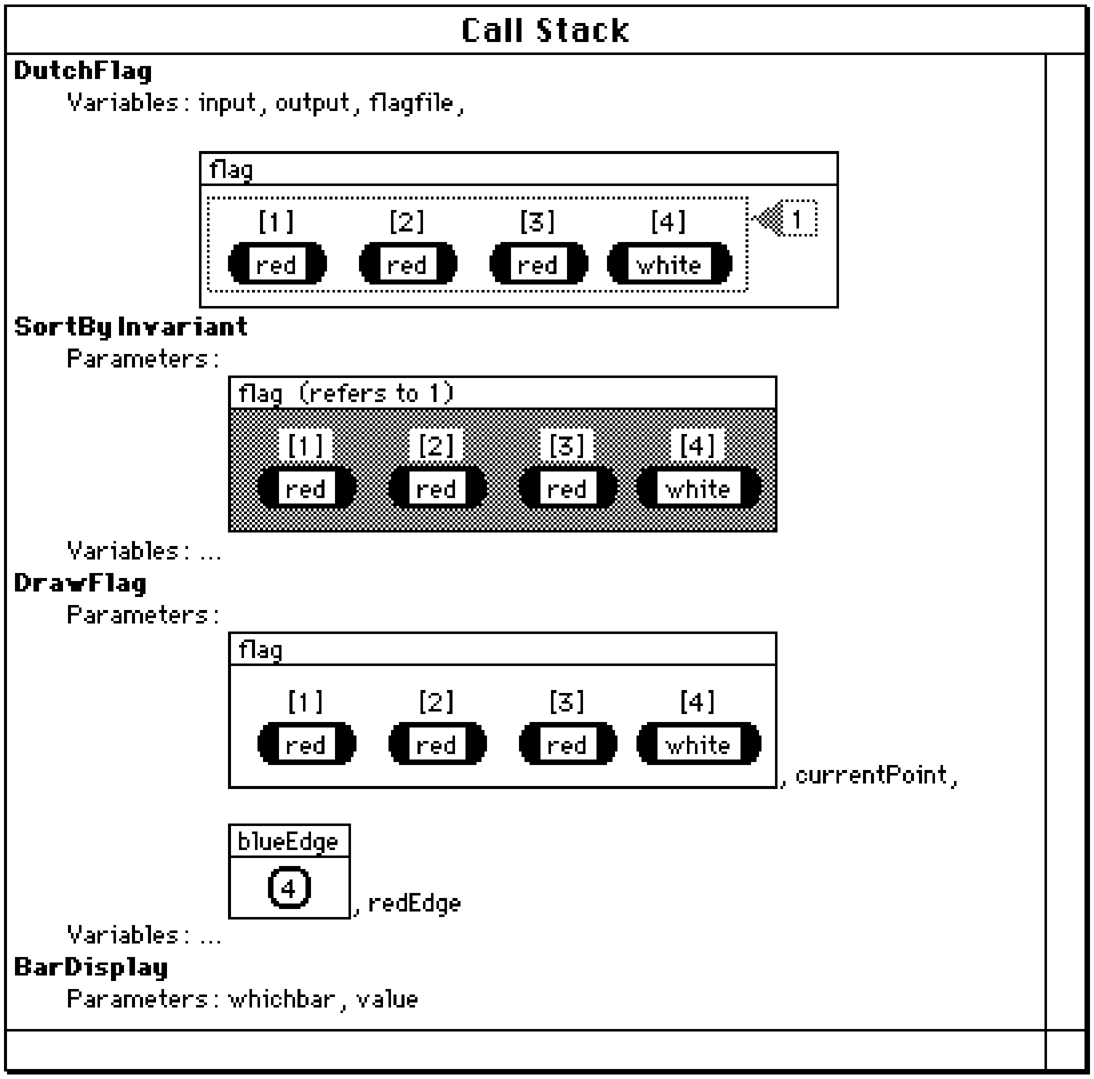
```
Var
    i : Integer;
    colorchar : Char;
{P ° }
•-> Begin <-•
    Writeln (output ° , 'Reading in file:' ° , kFlagFileName ° );
    reset (flagfile, kFlagFileName);
    i := 1;
    While ((Not eof (flagfile)) And (i <= maxFlagSize)) Do
        Begin
            Read (flagfile, colorchar);
            Case colorchar Of
                'r' : flag [i] := red;
                'w' : flag [i] := white;
                'b' : flag [i] := blue °
            End;
            Write (output ° , colorChar ° );
            i := (i + 1)
        End;
    Writeln (output ° )
•-> End <-•;

Procedure BarDisplay ( ° whichbar : Integer; ° value : colortype);
...;
```

Trace ( ) All (●) Statements ( ) None ☐ Call Stack ☒ Warnings

Figure 7: Pascal Genie 1.0 Tracing Program Execution

Runtime Tools

Another contribution of the Genie is the tight integration of tools, including the runtime system. The runtime system provides a collection of tools for program tracing and debugging. Figure 5 shows the tracing of an executing program. The runtime control panel provides flow-control commands like single-stepping and stepping to marks (breakpoints). Here the student is stepping through the Dutch Flag program. The program has run to the Mark indicated by the •-> <-• pairs around the *Begin-End* subprogram block.



```
Call Stack
DutchFlag
    Variables: input, output, flagfile,
        flag
        [1]   [2]   [3]   [4]
        red   red   red   white      1
SortByInvariant
    Parameters:
        flag (refers to 1)
        [1]   [2]   [3]   [4]
        red   red   red   white
    Variables: ...
DrawFlag
    Parameters:
        flag
        [1]   [2]   [3]   [4]
        red   red   red   white   , currentPoint,
        blueEdge
        4   , redEdge
    Variables: ...
BarDisplay
    Parameters: whichbar, value
```

Figure 8: Pascal Genie 1.0 Call Stack With Visualizations

The final component of the runtime system is the Call Stack view of an executing program, shown in Figure 8. This view displays the stack of currently active sub-programs. Unlike most debuggers, this is not just a static view of the execution state while the program is paused – it is updated while the program runs. This has proven to be extremely useful in teaching students about sub-program invocation and recursion.

In addition to animating the dynamics of procedure calls, this view is a framework for inspecting and monitoring data. The names of parameters and variables are shown next to their subprograms. These names can be clicked to expand them into automatically generated pictures of the data. As the program executes, the pictures are continually updated. They are available for every data structure, including pointer structures (Myers, Chandhok & Sareen, 1988). For example, simple linked lists and trees are automatically displayed as they might be drawn on a blackboard by an instructor.

Figure 8 shows some of the Call Stack visualizations. Each basic data type is associated with a particular shape. Displays for user-defined types are composed from the basic types into pictures that are distinct and readable. Care was taken to ensure that places where students become confused are clarified by the data visualizations. For example, a common problem is the distinction between call-by-value and call-by-reference parameter binding. As can be seen in Figure 8, *flag* is visible in three different places. This is because it is declared in three scopes: DutchFlag (the main program), SortByInvariant, and DrawFlag. In the main program *flag* is a variable. It is tagged with a label “1” because it is bound by reference to a parameter in SortByInvariant, where the picture is labelled “refers to 1”. This copy of the data is shown on a gray background, to indicate visually that the name is local, but the data is shared. Finally, the parameter *flag* in DrawFlag is on a clear background, indicating that it is a call-by-value parameter, and therefore has its own memory allocated. These data visualizations were extremely helpful in teaching parameter bindings, recursion, user-defined static data structures, and linked data structures. The visualizations have proven to be the single most popular feature of the Genie environments.

Support for High Level Program Design

Another program view provided by the Genie environments is a call graph representation known as the design view (Roberts, Pane, Stehlik & Carrasquel, 1988). This view grew out of a chalkboard technique developed by an instructor to support the planning of program design. The design view, like all program views, operates directly on the underlying program representation. The user can add procedure calls by drawing new edges in the tree, and then entering a name. As always, these changes are immediately reflected in every view. In addition to calls, the design view shows variable declarations and formal parameters.

In Figure 9, call-by-reference parameters are identified by up arrows. This indicates that information flows out of the subprogram, into the calling scope. A down arrow indicates a call-by-value parameter. Local variables are displayed as well, but parameter bindings are not shown. The student can move from the design view to the main code view by double-clicking on a name. It is interesting to note that this view was conceived, designed, and largely implemented by full-time teaching staff at Carnegie Mellon, with technical support from the MacGnome research staff (Miller, 1989).

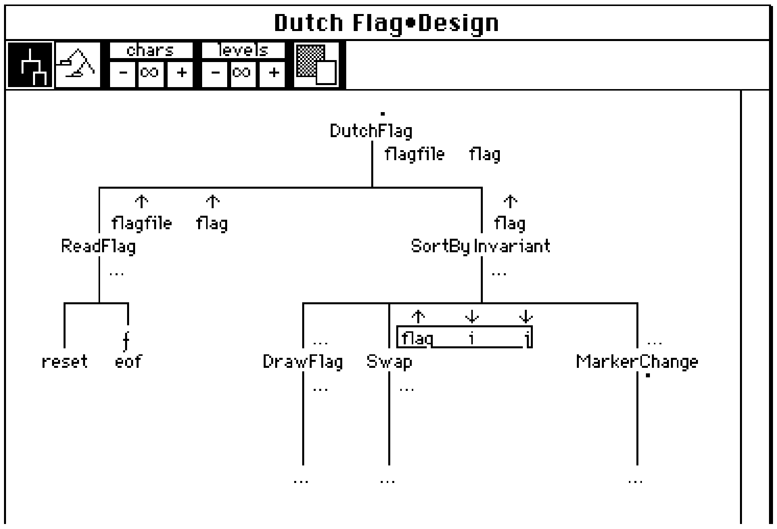


Figure 9: Pascal Genie 1.0 Design View

Partially-Completed Programs

Two features of the Genie make it easier for students and teachers to plan and share partially-completed programs. First, when a procedure or function is called which has not yet been declared, the Genie will, if the user approves, automatically add a template for the subprogram. This is especially useful in conjunction with the design view, when planning the top-down decomposition of a program. As new subprograms are drawn into the tree, templates for the routines are generated to be filled in later.

```
Procedure InsertItemAtEnd (
                Var theList: LinkedList;
                newItem: Element);
Begin
    $statement$;
    While $test$ Do
        Begin
            $statement$
        End;
    $statement$
End;
```

```
Procedure InsertItemAtEnd(
                Var theList: LinkedList;
                newItem: Element);
Begin
    $set up the variables that you will need$;
    While $you are not at end of list$ Do
        Begin
            $make progress towards end of list$
        End;
    $reached the end of the list...insert the item here$
End;
```

Figure 10: Use of Named Placeholders to Plan a Program

Secondly, it is possible to use arbitrary text strings as placeholders. These named placeholders can be used by teachers to provide pseudocode to students when passing out assignment templates. An example is shown in Figure 10. In addition, named placeholders are automatically generated by the system for subroutine calls, in order to help the user with the order, name and type of parameters. When a new subroutine call is entered into the program, the Genie generates a placeholder for each parameter, and names each placeholder with the name and type of the corresponding formal parameter. For example, when the student types *InsertItemAtEnd <enter>*, the Genie generates the code shown in Figure 11. Thus, the student immediately knows that this function takes two parameters; the first parameter is a variable parameter of type *LinkedList*, named *theList*; and the second parameter is a value parameter of type *Element*, named *newItem*.

```
InsertItemAtEnd ($LinkedList VAR theList$, $Element newItem$);
```

Figure 11: Automatic Generation of Named Placeholders

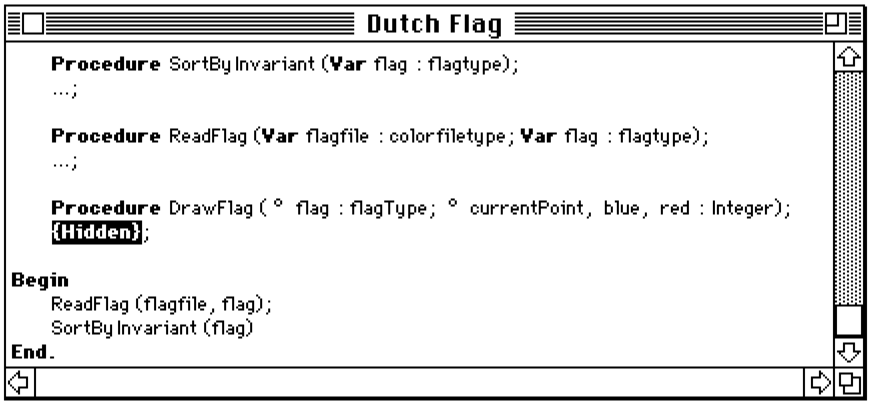
Encapsulated Documents

Another feature of the MacGnome Genies was the ability to attach text documents to a program. This provided a convenient way for teachers to pass out assignment information and sample data sets with program templates, as well as a way for the student to communicate back to the teachers. The popularity of this communication path between students and teachers led to further developments in the next-generation system, described later in this document.

Hiding Code

While we were building the Genies, instructors began using a case study approach to teaching CS1 (Stehlik, 1993). In a case study, students learn by modifying an existing large program. Students come to grips with good code examples created by the instructor. The programs also tend to be more engaging, since they are fully working systems from the start of the assignment. Yet often there are parts of a case study that the teachers want to hide from the students, perhaps because they include code that the students will later be asked write themselves. To support this, the Genie provides the ability to permanently hide code.[4] In Figure 12, the body DrawFlag cannot be viewed in any way. This is a nice way to enforce the distinction between specification and implementation.

4. Permanently hiding code was easier than implementing password locking. While instructors had to make sure that they had an original, unmodified copy, we didn't have to worry about students breaking or stealing passwords.



```
    Procedure SortByInvariant (Var flag : flagtype);
    ...;

    Procedure ReadFlag (Var flagfile : colorfiletype; Var flag : flagtype);
    ...;

    Procedure DrawFlag ( ° flag : flagType; ° currentPoint, blue, red : Integer);
    {Hidden};

Begin
    ReadFlag (flagfile, flag);
    SortByInvariant (flag)
End.
```

Figure 12: The Body of DrawFlag is Permanently Hidden

The Influence of Pedagogy

The Genie represents a serious attempt to tailor a programming environment to support teaching the fundamental concepts of the introductory computer science courses at CMU. In (Miller & Chandhok, 1989) we said, “It was our belief that traditional programming environments did little to facilitate teaching of [fundamental computer science and software engineering] concepts. In particular we wanted tools that allowed us to concentrate on procedural abstraction, data abstraction, and reasoning about programs... We also wanted to support the process of teaching.” We went on to argue in that paper that the various tools described above achieved those objectives. One measure of the success of the Genie environments is the number and quality of the schools who chose to use the environments in their introductory computer science courses. The list numbers over 40 institutions, including Harvard, Stanford, New York University, Swarthmore, Fordham, and Haverford.

Evaluation

Early evaluation was supportive of our work. Comparisons against other Pascal programming environments showed that students using the Genies performed a letter grade better on their final exam, and enjoyed the environments more throughout the term (Scheftic, 1985; Scheftic, 1986; Scheftic & Goldenson, 1986). Performance differences were most pronounced when the courses assigned larger programming tasks.[5] We conclude that the Genie's support for procedural abstraction paid off.

A later evaluation attempt was based on recording all of the student's actions within the system for later playback and analysis. This proved to be extremely difficult because of the continued evolution of the system, and the huge volume of data to be analyzed. Nonetheless, we did glean some valuable insights from this data (Goldenson & Wang, 1991):

- Even advanced users use the structure editor.

  We investigated the students' usage of the construct menus versus their use of the parser as a measure of their utilization of the basic structure editing features of the Genie. As expected, the menus were most heavily used early during the semester, and tapered off as the students learned the language. But even the most advanced students still used the construct menu frequently, indicating that they found this feature to be valuable.

- Many features were never used by students.

  Many powerful features went unused by the students. We found that the subsets of features that were utilized by the students was largely determined by which ones

5. In fact, in courses which taught just language constructs and syntax, the Genie students performed no better on average than students using more traditional environments.

their teachers used during class. While this has interesting pedagogical implications, some blame can also be assigned to a lack of attention to human-interface issues – as we incrementally added features to the system, we often took an ad hoc approach to adding the commands to the menus. By the end of the project, the menus were an embarrassing hodgepodge of commands, with little to distinguish important features. Other times, feature usage may have been deterred by the lack of screen space and slow performance of the Macintosh Plus computers the students used.

- Low utilization of the help system.

  The on-line help system was spectacularly underutilized. Even students working in the dormitories, without immediate access to a teacher or tutor, never used it. Our best guess is that the system has to be proactive in discovering problems and offering solutions.

- Supporting modular programming improves performance.

  The students who tested their programs in a modular fashion (i.e. frequently ran a semantically correct program) achieved higher grades. In addition, there was a positive correlation between use of the scope views and performance. We conclude that environments should support procedural abstraction through tools that facilitate modular viewing and testing.

- Skilled use of the debugger.

  We discovered that error frequency declined among students who made most frequent use of the Call Stack debugger. Even the “better” students were assisted by seeing a complete, dynamic picture of their program execution.

- Observation of users.

  The advantages of user testing have been widely touted. We learned these advantages ourselves while playing back a session where the student was thrashing amidst parser errors. This short session finally spurred our researchers (who thought the system was already better than most structure editors and who rarely went into the classrooms to watch students firsthand) to make the error dialog more usable.

As for comparisons with currently available commercial programming environments, we have not recently done any head-to-head evaluations. However, while our interpreted system does not have the speed of commercial compilers, and does not include all of the language features (such as software libraries), we found that teachers who use the system do not want to move to other environments.

Lessons Learned in MacGnome

The MacGnome environments made the following contributions to the state of the art in structure editors:

- many of the usability problems of earlier structure editors were overcome by taking advantage of a graphical user interface, and through careful incremental refinement;
- a rich set of tools were created for viewing and editing programs, resulting in improved support for abstraction; and,
- a sophisticated program tracing and data visualization system was integrated into the system, assisting in debugging and allowing students to more easily learn difficult topics like recursion.

The end result is a system that demonstrated measurable improvement in student learning, and was widely adopted by universities and high schools for introductory programming courses. However, several drawbacks of the system, such as slow execution and lack of external libraries precluded its use in more advanced education or in the production of commercial software. Furthermore, we did not achieve the goal of a generator that could rapidly create environments for new languages, based on a formal specification. During the process of addressing usability issues, we discovered that more and more of the system had to be hand-coded and hand-tuned based on user observation. These user observations also pointed out that many powerful features of the system go unused unless students are instructed in their effective use. We hypothesize that a holistic reevaluation and redesign of the user interface may also yield more effective student use. Finally, a popular feature of the GNOME environments was not implemented in the Genie: the incremental checking of static semantics. This feature is an important technology that is enabled by structure editors, and should not be neglected.

ACSE: Multimedia, Situated Programming, and Generated Semantics

Several things came together to redirect the course of our work:

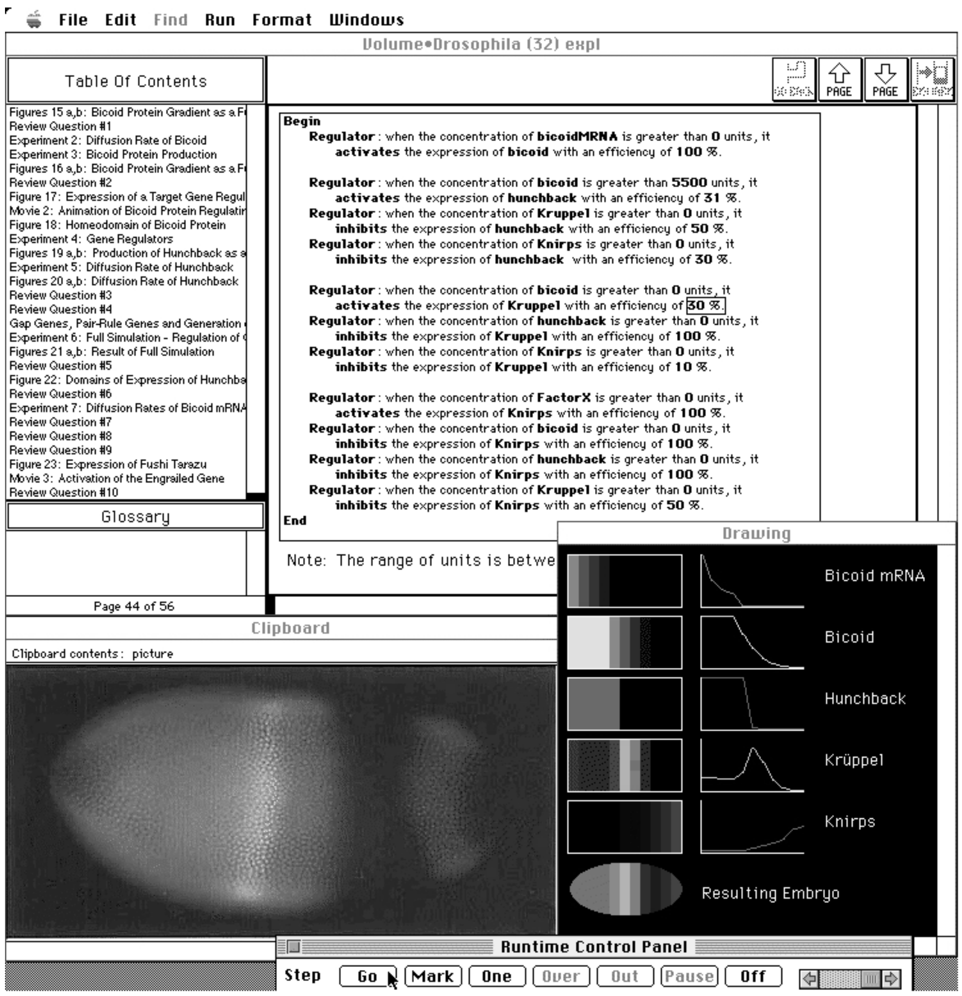
- there was a desire to bring object-oriented programming into the introductory computer science courses;
- we wanted to bring back the incremental semantic analysis that was present in the GNOME systems; and,
- the technology to support high quality multimedia was emerging on personal computers, leading us to a collaboration with a developmental biologist to experiment with this technology to build a science learning environment.

These considerations led to a project named ACSE, which produced the next generation of our structure editor based programming environments.

The ACSE Multimedia Science Learning Environment

Certain aspects of scientific reasoning that are difficult for students to learn with traditional lecture and textbook materials may be facilitated by computer-based learning environments. Two examples are understanding complex time-varying processes and planning an investigation to discover the mechanisms of such processes. The Advanced Computing for Science Education (ACSE) project set out to build a simulation-based multimedia science learning environment to address the teaching of these skills (Pane & Miller, 1993).

The ACSE software system was built on top of the Genie technology. A new program view, called the Volume view (see Figure 13), provides a multimedia document that can contain text, still graphics, movies, and simulations, and provides tools for navigating, viewing, and manipulating these components. We define movies as dynamic presentations that are the same each time they are viewed, e.g. animations, videos, and time-lapse photography, while we define simulations as any graphics, tables or other output that are generated by a program. Simulation outcomes are subject to changes that the student makes, and thus may be different each time they are viewed. The ACSE project hypothesized that movies, in particular, should improve students' understanding of time-varying processes, while simulations should exercise and improve students' skills in planning and carrying out investigations.



Figures 15 a,b: Bicoid Protein Gradient as a F
Review Question #1
Experiment 2: Diffusion Rate of Bicoid
Experiment 3: Bicoid Protein Production
Figures 16 a,b: Bicoid Protein Gradient as a F
Review Question #2
Figure 17: Expression of a Target Gene Regul
Movie 2: Animation of Bicoid Protein Regulatir
Figure 18: Homeodomain of Bicoid Protein
Experiment 4: Gene Regulators
Figures 19 a,b: Production of Hunchback as a
Experiment 5: Diffusion Rate of Hunchback
Figures 20 a,b: Diffusion Rate of Hunchback
Review Question #3
Review Question #4
Gap Genes, Pair-Rule Genes and Generation
Experiment 6: Full Simulation - Regulation of
Figures 21 a,b: Result of Full Simulation
Review Question #5
Figure 22: Domains of Expression of Hunchba
Review Question #6
Experiment 7: Diffusion Rates of Bicoid mRNA
Review Question #7
Review Question #8
Review Question #9
Figure 23: Expression of Fushi Tarazu
Movie 3: Activation of the Engrailed Gene
Review Question #10

```
Begin
    Regulator : when the concentration of bicoidMRNA is greater than 0 units, it
        activates the expression of bicoid with an efficiency of 100 %.

    Regulator : when the concentration of bicoid is greater than 5500 units, it
        activates the expression of hunchback with an efficiency of 31 %.
    Regulator : when the concentration of Kruppel is greater than 0 units, it
        inhibits the expression of hunchback with an efficiency of 50 %.
    Regulator : when the concentration of Knirps is greater than 0 units, it
        inhibits the expression of hunchback  with an efficiency of 30 %.

    Regulator : when the concentration of bicoid is greater than 0 units, it
        activates the expression of Kruppel with an efficiency of 30 %.
    Regulator : when the concentration of hunchback is greater than 0 units, it
        inhibits the expression of Kruppel with an efficiency of 100 %.
    Regulator : when the concentration of Knirps is greater than 0 units, it
        inhibits the expression of Kruppel with an efficiency of 10 %.

    Regulator : when the concentration of FactorX is greater than 0 units, it
        activates the expression of Knirps with an efficiency of 100 %.
    Regulator : when the concentration of bicoid is greater than 0 units, it
        inhibits the expression of Knirps with an efficiency of 100 %.
    Regulator : when the concentration of hunchback is greater than 0 units, it
        inhibits the expression of Knirps with an efficiency of 100 %.
    Regulator : when the concentration of Kruppel is greater than 0 units, it
        inhibits the expression of Knirps with an efficiency of 50 %.
End
```

Note: The range of units is betwe

Figure 13: The ACSE Volume View

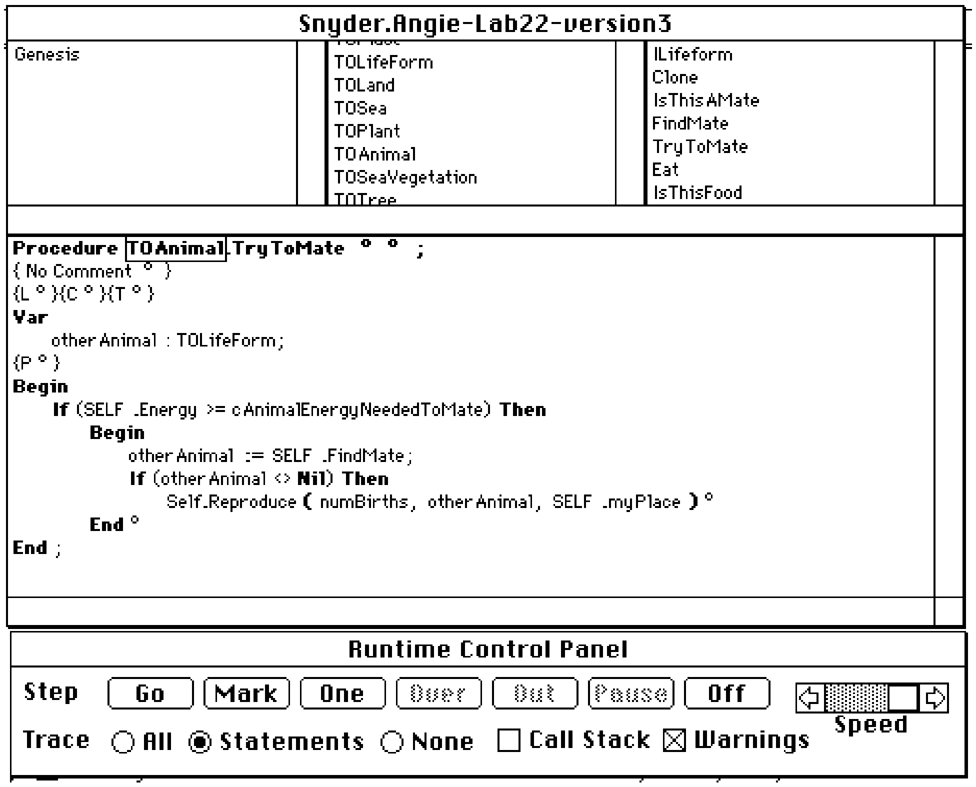
In typical simulation-based learning systems, the program itself is hidden from the user. This is a natural choice, because the science that is embedded in that simulation is a very small portion of the overall program, and it is not organized in a manner that would be readily understood by the science student. A set of controls are provided that permit the student to manipulate certain carefully chosen parameters and see the results. ACSE is distinguished from these typical systems in the way the simulation is integrated with other lesson contents. Selected pieces of the simulation are interspersed throughout the lesson in a manner that is not unlike the use of mathematical formulas in a textbook. These small pieces of the simulation are chosen because they describe the essential science that is embedded within the much larger program. The program elements are provided in the context of explanatory materials, and irrelevant details are abstracted away.

The student can manipulate the simulation by directly modifying these program elements. The full-featured novice programming environment is available to support this activity. Thus the expressive power of a programming language is available as the student investigates the simulation. This enables structural changes – those that involve modifying or replacing algorithms – in addition to the kinds of parametric changes that are permitted in traditional simulation systems.

Objects, Navigation, and a Browser

The first step toward supporting object-oriented programming was to add the features of Object Pascal to the Genie's Pascal. This involved additional hand tuning of almost every aspect of the system (grammar, unparsing schema, parsing, compiler and runtime system), but because the changes were not sweeping, this effort was much smaller than going to a completely new language.

The next step was to make it easier to navigate through large programs. To do this, we returned to the GNOME idea of focusing the user on one subroutine at a time in the main view of the program. But we supplemented this view with a browser, in order to give the programmer a context and an easy way to change to another subroutine. Figure 14 shows the result.



```
Procedure TOAnimal.TryToMate  °  °  ;
{ No Comment  °  }
{L ° }{C ° }{T ° }
Var
    otherAnimal : TOLifeForm;
{P ° }
Begin
    If (SELF .Energy >= cAnimalEnergyNeededToMate) Then
        Begin
            otherAnimal := SELF .FindMate;
            If (otherAnimal <> Nil) Then
                Self.Reproduce ( numBirths, otherAnimal, SELF .myPlace ) °
        End °
End ;
```

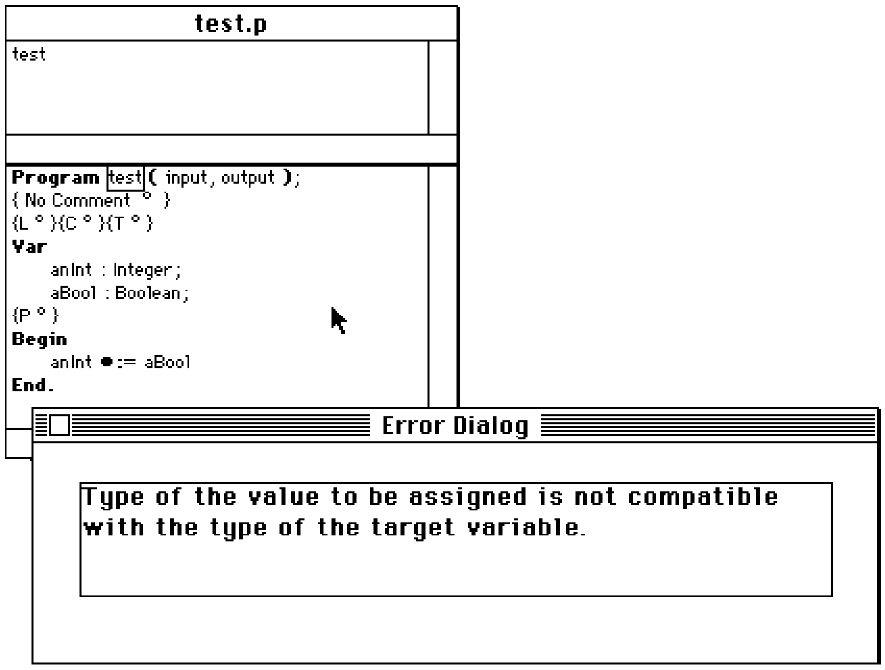
Trace ○ All [✓ Statements] ○ None ☐ Call Stack [✓ Warnings]

Figure 14: Object Pascal Genie Browser and Runtime Control Panel

Semantics

The original GNOME environments demonstrated the effectiveness of incremental semantic analysis: errors reported in context, rather than in bulk at compilation time, were easier for novices to understand and correct. The Genie lost this advantage, once again relying on the compiler to provide semantic checking. In ACSE we reintroduced semantic analysis, in a more incremental flavor. The analysis is performed after each edit, and errors are immediately tagged in a noticeable yet unobtrusive manner. This permits the student to defer attention to the problem. The error tags persist as long as the errors remain uncorrected.

Errors are not cascaded, focusing attention on the source of the problem, rather than extraneous symptoms. The error tags give a visual a hint about the type of the problem, for example undeclared identifier references are visually distinguishable from type clashes. More information about an error is available by clicking on the error tag. The student can navigate through the existing errors using the “Find Next Error” command. Figure 15 shows a semantic error, with the error dialog opened to show the error message.



```
Program test ( input, output );
{ No Comment ° }
{L ° }{C ° }{T ° }
Var
     anInt : Integer;
     aBool : Boolean;
{P ° }
Begin
     anInt ● := aBool
End.
```

Type of the value to be assigned is not compatible with the type of the target variable.

Figure 15: Error Reporting

The advantages of incremental semantic analysis go well beyond simple error checking and reporting. The system is maintaining complete information about the bindings of all identifier references to declarations. A simple, but very frequent, use of this information is the “Go To Declaration” command, available at any name reference. More interestingly, this information could be used to support a variety of graphical program views, such as a Class Hierarchy diagram, the Design view, a dependency/module interconnection analysis tool, etc. Moreover, each of these views can be incrementally updated as the semantic information changes after each edit.

Situated Programming: Object-Oriented Programming in CS1

While the ACSE Volume view was initially conceived as primarily a science learning environment, the CMU introductory programming courses adopted it as the primary communication channel between the teacher and students (Meter & Miller, 1994). In the first class that taught object-oriented programming, students were given a small artificial life simulation and, through the Volume view, were given access to the program’s parameters. All of the other program code was hidden away and not accessible. The first assignment was to run the program using a variety of values for the constants and then to write an essay about the program. The students discussed what they liked and what they didn’t like. They talked about how it should have worked if it were to mimic real life. The rest of the assignments were drawn from these commentaries. Students implemented birds that returned to their nests and fish that returned to their spawning grounds. They introduced disease, and they made plants that budded little plants or plants that produced seeds which sprouted at later times. They made sunlight shine differentially at the equator and at the poles. They made fish swim in schools and they distinguished between sea life in the shallows and the deep. They dumped data to spreadsheets and graphing packages, tracked the interactive effects of variables, and reported results through the Volume view. When their changes were done, they always commented their programs by placing important changes into the Volume and discussed the changes through text and pictures. So, when a program was handed in it was very easy to see what they had changed, and what the results were. This had been one of the difficulties in the typical case studies we had been using. This new way of building and communicating about the program and was an exciting revolution in the course at CMU. While there are no quantitative scientific results to report, from a qualitative point of view we consider it a great success.

One interesting note about usability relates to the Volume view. The design of this view was separated into a “builder” mode and an “explorer” mode, because we first envisioned that the multimedia documents would be built by an instructor and explored by the students. Careful attention was paid to the usability of explorer mode, but less attention was given to builder mode. In retrospect, this was a mistake, because students made heavy use of builder mode as described above.

Lessons Learned in ACSE

The ACSE environment modernized our structure editor technology to include multimedia, literate object-oriented programming, and state-of-the-art incremental semantic analysis. This enabled the introductory computer science courses to teach exciting case studies, and opens the possibility of using programming as a learning tool in other domains. These positive developments are tempered by our empirical evaluation of science lessons (Pane, 1994; Pane, Corbett & John, 1995). These studies show that careful instructional design is necessary if statistically significant improvements in learning will be found in multimedia systems such as ACSE. A final lesson is found in our ad hoc user-interface design for the Volume view. We did not expect students to use builder mode very much, yet it became one of the most heavily used aspects of the system. We conclude that no aspect of the user interface should be exempt from careful design.

Conclusions

This retrospective paper chronicles a ten-year evolution in the technology of structure editors. Guided by heavy classroom use and user evaluation, we implemented many powerful tools and features, but were unable to deliver on a technology that generated complete new environments from formal specifications of a language – in actuality, much hand-coding is necessary to build a full system. One result of this problem is that we are not able to quickly port our environment to new and modern programming languages. Perhaps this, in combination with our novice programmer focus, can explain why we haven't yet been able to convince any companies to commercialize our technology.

Our approach to novice programming has received some criticism from the program-correctness crowd. Professor Edsger Dijkstra witnessed a presentation of the MacGnome environments at a seminar at the University of Texas (Miller, 1988). Later, he published the following (Dijkstra, 1989):

> "I was recently exposed to a demonstration of what was pretended to be educational software for an introductory programming course. With its "visualizations" on the screen it was such an obvious case of curriculum infantilization that its author should be cited for "contempt of the student body", but this was only a minor offense compared with what the visualizations were used for: they were used to display all sorts of features of computations evolving under control of the student's program! The system highlighted precisely what the student has to learn to ignore, it reinforced precisely what the student has to unlearn. Since breaking out of bad habits, rather than acquiring new ones, is the toughest part of learning, we must expect from the system permanent mental damage for most students exposed to it.

He goes on to say that programming is not complete until theorems about the program's correctness are proven. There is an important element of truth to Dijkstra's assertions. But his point of view neither describes programming as it is known today, nor provides tools that are capable of supporting his methods in large real-world systems. Indeed it is not possible to completely apply axiomatic semantics to large programs in the languages we use today. Whatever the merits of his approach, current practice requires the programmer to have an understanding of details as well as an ability to use abstraction. As such, our systems support the current state of the art in software engineering. Furthermore, we feel that when program correctness tools are developed for large-scale programming, structure editors will be an ideal environment in which to provide them and teach their use.

Acknowledgments

This research was supported by the National Science Foundation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. Government

In addition to the authors, the following people worked on, or supported, the GNOME, MacGnome, and ACSE environments: Lynn Baumeister, Jeffrey G. Bonar, Claire Bono, Nahid Capell, Peter Capell, Jacobo Carrasquel, Paul Chan, Kimberly Chandhok, Ravinder P. Chandhok, Rebecca Clark, Stephanie Claudy, Michael Cleron, Pat Crecine, Laurie Damianos, Charles A. Ettensohn, Christopher Fedor, Steven Fisher, Susan Gallagher, David Garlan, Terry A. Gill, Dennis R. Goldenson, Vernon Harmon, Lily Hou, David H. Kaufer, Mark Lively, John Mayer, Paul McCartney, Trina Punders, Marjorie L. Rice, James A. Roberts, Atul Sareen, Mark D. Shermis, Edward J. Skwarecki, Nicholas Spies, Mark Stehlik, Natalie Svaan, Ward Travis, Mark J. Tucker, and Bing Jyun Wang. In addition to the above, the authors acknowledge the anonymous reviewers who provided valuable comments on an earlier draft of this paper.

Finally, a special acknowledgment in memory of A. Nico Habermann. As the head of the Computer Science Department at Carnegie Mellon, and Principal Investigator of the Gandalf Project, Nico gave GNOME its start and gave MacGnome and ACSE his unequivocal support.

References

Dijkstra, E. W. (1989). On the Cruelty of Really Teaching Computing Science, SIGCSE Bulletin: Twentieth SIGCSE Technical Symposium on Computer Science Education, (Vol. 21, pp. xxv-xxix). Louisville, KY.

Garlan, D. B. (1987). Views for Tools in Integrated Environments (PhD Thesis, Department of Computer Science Technical Report CMU-CS-87-147). Pittsburgh, PA: Carnegie Mellon University.

Garlan, D. B., & Miller, P. L. (1984). GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

Goldenson, D., & Wang, B. J. (1991). Use of Structure Editing Tools by Novice Programmers. Proceedings of Empirical Studies of Programmers: Fourth Workshop.

Habermann, A. N., Garlan, D., & Notkin, D. (1991). Generation of Integrated Task-Specific Software Environments. In R. F. Rashid (Ed.), CMU Computer Science: A 25th Anniversary Commemorative, (pp. 69-98). New York: ACM Press.

Habermann, A. N., & Notkin, D. S. (1986). Gandalf: Software Development Environments. IEEE Transactions on Software Engineering, SE-12(12), 1117-1127.

Medina-Mora, R. (1982). Syntax-Directed Editing: Towards Integrated Programming Environments (PhD Thesis, Department of Computer Science). Pittsburgh, PA: Carnegie Mellon University.

Meter, G., & Miller, P. (1994). Engaging Students and Teaching Modern Concepts: Literate, Situated, Object-Oriented Programming. Proceedings of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education.

Miller, P. (1988). The MacGnome Structure Based Programming Environments and Computer Science Education. Invited presentation, Learning Resources Center and Department of Computer Sciences Seminar Series, University of Texas, Austin.

Miller, P. (1989). A Software Rotation for Professional Teachers. Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education.

Miller, P., & Chandhok, R. (1989). The Design and Implementation of the Pascal Genie. Proceedings of the 1989 ACM Computer Science Conference.

Myers, B. A., Chandhok, R., & Sareen, A. (1988). Automatic Data Visualizations for Novice Pascal Programmers, Proceedings of the IEEE 1988 Workshop on Visual Languages, (pp. 192-198). Pittsburgh, PA.

Pane, J. F. (1994). Assessment of the ACSE Science Learning Environment and the Impact of Movies and Simulations (School of Computer Science Technical Report CMU-CS-94-162). Pittsburgh, PA: Carnegie Mellon University.

Pane, J. F., Corbett, A. T., & John, B. E. (1995). Assessing Dynamics in Computer-Based Instruction (in preparation). Pittsburgh, PA: Carnegie Mellon University.

Pane, J. F., & Miller, P. L. (1993). The ACSE Multimedia Science Learning Environment. In T.-W. Chan (Ed.), Proceedings of the 1993 International Conference on Computers in Education, (pp. 168-173). Taipei, Taiwan.

Pattis, R. E., Roberts, J., & Stehlik, M. (1995). Karel the Robot: A Gentle Introduction to the Art of Programming. (Second ed.). New York: Wiley.

Reps, T. W., & Teitelbaum, T. (1989). The Synthsesizer Generator: A System for Constructing Language-Based Editors. New York: Springer-Verlag.

Roberts, J., Pane, J., Stehlik, M., & Carrasquel, J. (1988). The Design View: A Design Oriented, High-Level Visual Programming Environment, Proceedings of the 1988 IEEE Workshop on Visual Languages, (pp. 213-220). Pittsburgh, PA.

Scheftic, C. (1985). Introductory Programming Environments: A Comparison of Students' Learning Using Structure or Text Editors (PhD Thesis). Pittsburgh, PA: University of Pittsburgh.

Scheftic, C. (1986). Structure Editors: What Are Those Tools for Experts Doing in a Class for Novices? Proceedings of the 1986 Association for Educational Communications and Technology Annual Conference.

Scheftic, C., & Goldenson, D. R. (1986). Testing Programming Method and Problem Solving: The Role of Programming Environments Based on Structure Editors. Proceedings of the 1986 National Educational Computing Conference.

Stehlik, M. (1993). Approaches to Programming Assignments in CS1 and CS2. Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education.