The Poe Language-Based Editor Project

C. N. Fischer
Computer Sciences Department
University of Wisconsin - Madison, 1210 W. Dayton, Madison, WI

Gregory F. Johnson
Computer Science Department
Cornell University, Upson Hall, Ithaca, NY

Jon Mauney
Computer Science Department
North Carolina State University, Raleigh, NC

Anil Pal
Computer Sciences Department
University of Wisconsin - Madison, 1210 W. Dayton, Madison, WI

Daniel L. Stock
Computer Sciences Department
University of Wisconsin - Madison, 1210 W. Dayton, Madison, WI

Overview

Editor Allan Poe (Pascal Oriented Editor) is a full-screen language-based editor (LBE) that knows the syntactic and semantic rules of Pascal. It is the first step in development of a comprehensive Pascal program development environment.

Poe's design began in 1979; version 1 is currently operational on Vax 11s under Berkeley Unix and on HP 9800-series personal workstations. Poe is written in Pascal, and is designed to be readily transportable to new machines. An editor-generating system called Poegen is operational, and much of the language-specific character of Poe is table-driven and retargetable.

Poe was inspired in large measure by the Cornell Program Synthesizer [TR81], although it is philosophically more akin to the COPE system [AC81] (which was developed independently and contemporaneously).

As a program is entered or modified, Poe automatically structures ("prettyprints") the program and checks it for correctness. Semantic errors are noted and incorrect usages are marked. Like the Synthesizer, Poe is structure-oriented and models program development as the repeated expansion of language prototypes (termed prompts). However, Poe attempts to provide a particularly simple user interface (especially to novices). As a result, Poe uses *no templates*. Rather, it presents an interface in which the user moves the cursor to a prompt symbol and types text corresponding to the prompt. Typing a single-token prefix of a particular expansion is sufficient; an automatic syntactic error corrector will provide any added tokens which are necessary to expand the user's input and make it syntactically valid.

In the first section following this introduction, the fundamental design philosophy and user interface of Poe will be discussed. In the second section, strategies employed for parsing, semantic checking, the "undo" facility, and automatic program formatting will be presented.

This research was supported in part by NSF Grant MCS 82-02444.

The User Interface

When development of a new program is begun, the program prototype shown in Figure 1 is displayed. Poe displays three kinds of symbols:

(1) Required prompts
These are delimited by "<" and ">" (e.g., <ID>, <FILE ID LIST>). Required prompt symbols are placeholders that *must* be expanded to obtain a valid Pascal program. The expansion expected by such a placeholder is suggested by its name. Thus <ID> should be expanded into an identifier and <FILE ID LIST> should be expanded into a list of file identifiers.

(2) Optional prompts
These are delimited by "{" and "}" (e.g., {VARIABLES}, {STMT LIST}). Optional prompt symbols are placeholders that *may* be expanded to produce a Pascal construct. If an optional prompt is not expanded, it is "erased", indicating that the suggested construct is not needed in this particular program.
For example, {VARIABLES} marks the place at which program variables can be declared. However, since a Pascal program may use no variables, it is legal to ignore this symbol in developing a program.

(3) Pascal symbols
These are the ordinary symbols found in Pascal (identifiers, numbers, reserved words, etc.). For emphasis, reserved words are shown in upper case. When a complete Pascal program has been created, only Pascal symbols remain.

To create a Pascal program in Poe, the user expands optional and required prompts. To do this, the user merely moves the cursor to the prompt and types any expansion that agrees with the prompt. Thus if the cursor is at an <ID> prompt, one can type abc or xxxx or any other valid identifier.[1]

Cursor movement is controlled using the usual cursor control keys:[2] The space bar moves the cursor one symbol right; the backspace key moves the cursor one symbol left. The return key moves the cursor to the leftmost symbol of the next line. The "\" key[3] moves the cursor to the leftmost symbol of the previous line.

As a Pascal symbol is entered, a prompt symbol may be replaced by new symbols, representing the detailed structure of a construct. Thus if in the above example, the cursor were to be moved to the {STMT LIST} prompt, and "if " is typed[4] the structure shown in Figure 2 results.

Since a THEN is always created when an IF is recognized, it is impossible to create ill-formed IF statements. In fact, in Poe syntactically incorrect program structures (of any kind) *can never* be created.

But what if the user were to type something that is illegal at the point at which the cursor is positioned? For example, a THEN (which cannot begin a statement) might be entered at a {STMT} prompt, or BEGIN might be typed at the very top of a program. Rather than considering these errors, Poe uses an automatic *error-repair* algorithm to place all symbols, as they are entered, in their "most reasonable" program context. Thus typing a THEN with the cursor at a {STMT} prompt will expand the prompt into an IF-THEN construct, with the cursor immediately following the THEN. Similarly, entering a BEGIN at the top of a

```
PROGRAM <ID> ( <FILE ID LIST> ) ;
{LABELS}
{CONSTANTS}
{TYPES}
{VARIABLES}
{PROCEDURES}
BEGIN
   {STMT LIST}
END .
```

Figure 1. The initial Pascal prototype used by Poe.

[1]Poe will allow an undeclared identifier to be entered, but will highlight it until it is properly declared.

[2]"Arrow" keys found on many terminals are *not* used because they are not standard and not always available. Particular implementations (such as HP 9800 Poe) make use of such keys as an extension. Analog input devices such as mice and touch-sensitive screens can also be used when they are available.

[3]This choice is arbitrary; "reverse linefeed" is non-standard on ordinary keyboards.

[4]The blank after the "if" is needed so that the editor can distinguish between the symbol "if" and a pause in the entry of, e.g., "iff".

```
PROGRAM <ID> ( <FILE ID LIST> ) ;
BEGIN
   IF <EXPR>
   THEN {STMT}
   {ELSE CLAUSE} ;
   {MORE STMTs}
END .
```

Figure 2. The structure created in response to input of "if ".

program will move the cursor just beyond the nearest BEGIN.

This approach makes Poe fairly forgiving in the entry of program text. But what if the repair chosen by Poe *is not* what the user wants? To minimize the effect of user errors, a very general "undo" command is provided. This allows considerable experimentation without the danger of irretrievable errors. If the repair elicited by an input symbol is unwanted, the repair can be undone and an alternate input sequence can be tried.[5]

Full static semantic checking of program text is also provided. Whenever a semantically incorrect symbol or construct is entered, it is immediately highlighted. Highlighting remains until the associated semantic errors are repaired. An error message detailing a particular semantic error can be obtained by moving the cursor to a highlighted symbol.

An important difference between Poe and some other LBE's is that Poe represents an "open environment". That is, Poe can read text files created by any program or utility and can output text files usable by other programs and utilities. This follows the Unix model of allowing the output of one program to be the input of another. Some LBE's store programs (internally and externally) in a tree-structured form. In such a system it is not easy to "read" a program created on a conventional text editor or to apply other text-oriented processors such as cross-reference generators or optimizing compilers to its output. Poe, of course, can easily read programs created on other text-oriented systems. Further, it is easy (though not necessarily efficient) to augment Poe by operating on the text files it creates (e.g., to implement a cross-reference mechanism or a program compaction algorithm).

The price paid for a textual representation of a program in a structured editor is a substantial amount of processing when editing begins. It appears that some of this overhead can be avoided by maintaining associated "environment" files (as UW-Pascal does [LF79])) or by doing background processing during editing (since "think time" normally goes unused). Systems such as Gandalf [Hab79] and the Ada APSE [SFGT81] anticipate a variety of utilities all sharing a common tree-structured program representation. This allows redundant processing (such as rescanning) to be avoided and allows a much richer representation (optimization, debugging, compilation information, etc. can be included in the tree).

The question of whether a textual or tree-structured program representation (or even both, maintained in parallel!) is preferable remains open. Both have significant advantages and disadvantages.

Poe is a *modeless* editor in that all keystrokes are (by default) assumed to be input text. Cursor control and text manipulation are tied (where possible) to special-purpose keys (e.g., deletion is associated with the "del" or "rubout" key). Other commands are prefixed with an escape character ("!").

To preserve syntactic correctness, all text-manipulation is structure-oriented. That is, the only text segments that can be deleted, copied or inserted are those that correspond to valid syntactic structure (as defined by the underlying grammar).

Poe uses a top-down parsing approach, so an incomplete program is viewed internally as a parse tree with some unexpanded non-terminals as leaves. What the user thinks of as prompt symbols are actually unexpanded non-terminals. This leads to an interesting prompting feature. Whenever the

[5]A prompting facility is also provided.

cursor is on a prompt symbol, the user can ask to see a possible expansion of the prompt (without actually doing the expansion). Repeated requests will cycle through all possible expansions. The user is actually just cycling through all the (non-trivial) productions that have the "prompt" as the left-hand side symbol. Another command is provided to actually choose a suggested expansion. With this mechanism, it is possible to "explore" the structure of a language. This appears to be particularly useful when a user has an idea of what he wants, but is unsure of the exact details.

Poe takes a highly general approach to structure elision. With the cursor on any symbol, an "elide" command will replace the smallest structure containing the symbol with an "elision marker". This marker is essentially a prompt symbol with ellipsis added. Thus a BEGIN-END block can be elided to <BEGIN-END...> and an IF-THEN-ELSE can be elided to <IF-THEN-{ELSE}...>.

Poe allows a complete program to be executed, with an automatic return after normal or abnormal termination. At present, this is performed by invoking (invisibly) the standard Pascal interpreter or compiler. A built-in interpreter that allows execution of program fragments (and editing of data as well as program text) is under development.

Technical Issues

Lexical Scanning

Scanning is fairly routine except for the fact that editor commands can be intermixed with user input. The scanner is therefore actually a combined scanner/command interpreter. Further, after the user has started to enter a token he can delete characters by moving the cursor to the left. Thus the scanner must fully buffer the set of finite automaton states that it passes through. As the user hits the back-space key, the scanner must back up through its previous states in order to be able to continue scanning when the user finally starts to enter text again.

Parsing Issues

The parser uses a table-driven LL(1) technique with special provision made for lists. Because of its table-driven flavor, it is fairly easy to implement syntax-directed editors based on Poe for languages other than Pascal. The editor generator creates FMQ [FMQ80] error correction tables in addition to the parse tables, so the user-friendly "feel" of Poe is automatically preserved in transitions to other target languages.

Because LL(1) parsers abhor left recursion, lists are usually generated by productions of the form[6]:

<list> → item {more list}
{more list} → delimiter item {more list}
{more list} → ϵ

This method of generating lists is undesirable in an interactive editing environment. First, it may be necessary to add or delete items *anywhere* in a list; the above grammar form does not support this, since all editing operations in Poe take the form of sub-tree insertions and deletions. Second, lists generated by left or right recursive productions are "skewed" and "deep". To facilitate manipulation of lists as complete structures, and to make editing of individual list elements possible, "flat" and "shallow" list structures are preferred.

Poe's parser therefore recognizes special "list" structures. A list may generate ϵ, or at least one item may be required. Whenever an item is generated in a list, "more list" non-terminals are included on *both sides* of the item. For example, we start with

<list> → item {more list}

When this production is applied, we actually use a right hand side of the form

{list head} item {more list}

We include productions of the form

{list head} → item delimiter {more list}
{list head} → ϵ
{more list} → delimiter item {more list}
{more list} → ϵ

although whenever either of the non-ϵ productions is applied, we actually substitute

{list head} item delimiter {more list}

or

{more list} delimiter item {more list}

The net effect is to allow insertions or deletions of list items at any position in a list. Note too that all {list head} and {more list} subtrees are made *immediate descendents* of the <list> non-terminal. This forces lists to be "wide" and "shallow", simplifying list manipulation.

---

[6] "ϵ" is the empty or null string.

To allow easy creation and manipulation of expressions, all expressions are considered to be *lists* of operands, separated by operators. This avoids the proliferation of special non-terminals (<expression>, <factor>, <term>, ...) commonly used to force operator precedence. In Poe operator precedence is considered to be a *semantic* issue. Use of syntax to enforce operator precedence makes structure-oriented editing of expressions ungainly and difficult. One solution is to temporarily turn off syntactic and semantic checking and enter a special character-level mode to deal with expression editing. Poe's solution of treating expressions as flat lists allows the user to edit expressions flexibly at the token level while maintaining the benefit of immediate feedback if semantic errors arise inside the expression.

Incremental parsing and error repair

In many LBE's *incremental reparsing* is an important issue (see, e.g., [MS81]). That is, after editing operations, we must guarantee that a syntactically valid structure has been maintained. Poe's approach to this problem is to parse tokens only when they are *originally entered.* All editing operations involve structural units (i.e., parse trees). Insertion and copying of units is limited to contexts that admit the unit in question. That is, a subtree can be inserted or copied only under a non-terminal that matches the root of the subtree. This approach works well, except in the case where "unit productions" are involved. Unit productions allow the same structure to be rooted by different non-terminals, making identification of valid subtrees more difficult. For example, given a production

<Parameter expr> → <Expr>

a sub-tree rooted by "<Parameter expr>" may appear illegal if it is placed in a context expecting "<Expr>". In such cases, an LBE must examine not only the root of a subtree, but also all "trivial" subtrees.

Syntactic error repair is implemented using the FMQ LL(1) error repair algorithm. All symbols are given an insertion cost, and the FMQ algorithm computes the *locally least cost* insertion sequence that allows the next input symbol to be accepted as syntactically valid. Repairs can be controlled by adjusting the costs of particular insertions.

This technique works well for minor errors, but is less satisfactory for major errors. There are really two issues involved. First, the FMQ algorithm is *batch-oriented,* and its simple cost model is not entirely applicable in Poe's interactive environment. In particular, insertion costs do not distinguish between symbols that already have been entered, and those that are created as part of the repair process. For example, if an "If Then Else" construct has already been created, and the cursor is placed on the "If", entering a "Then" will invoke error repair, which will determine that insertion of "If <EXPR>" will allow the "Then" to be matched in a valid context. This "repair" should be deemed fairly cheap (and hence desirable) because its net effect is merely to move the cursor.

If a "Then" were entered with the cursor placed on a "<STMT>" non-terminal, then insertion of "If <EXPR>" will again allow the "Then" to be matched in a valid context. In this case, however, the screen is actually changed and new symbols are inserted (rather than matched) and therefore a higher cost ought to charged. Clearly, an extension of the FMQ model is required in this case.

A second problem with automatic error repair is the fact that the more complex (and costly) a repair is, the less likely it is to be acceptable to the user. In such cases, the user must often "undo" the "repair" and try again. A cost threshold probably ought to be established. Below this threshold, automatic repair would proceed. Above the threshold, the user would be engaged in a dialogue of possible choices, much as is done in the CAPS system [WDT76].

Semantic error checking

Semantic analysis in Poe is implemented using attribute grammars. As program fragments are entered, they are parsed and if necessary, repaired. Parse trees are built and decorated with attribute values. An incremental attribute evaluation mechanism is used ([JF82], [Joh83]). This mechanism is novel in that attributed graphs rather than attributed trees are supported. "Non-local" production instances [Joh83] are added to the parse tree as part of the attribute evaluation process, and these productions allow attribute information to flow directly to program structures that are structurally distant. For example, if the type of a variable is changed, all occurrences of the variable can be immediately re-examined to determine if they are semantically valid.

In the first implementation of Poe, the incremental attribute evaluator is driven by copy rules, which are evaluation functions that simply assign the value of one attribute to another attribute. After a sub-tree replacement takes place, the production instances immediately above and below the site of

the change are visited by the evaluator. When the evaluator visits a production instance, the attribute evaluation functions which are not copy rules are invoked. Then, the evaluator performs the copy rule evaluations. If it notices that it is about to over-write an attribute with a new value, then the production instance which receives the new value is visited.

In the first implementation, some important principles of incremental evaluation were recognized and included, but at the time a rigorous theory of incremental evaluation [Joh83, Rep82] had yet to be developed. Experience with the first implementation led us to conclude that evaluation functions should do no tree-walking, and should operate only on attributes in a single production instance. An attribute evaluator which is controlled by changing attribute values should manage the re-evaluation process. This permits the difficult task of managing the incremental re-evaluation process to be localized to a single table-driven routine, rather than being spread throughout the bodies of semantic routines. Further, as was mentioned above, it was realized that some syntactically unrelated nodes in a parse tree (such as definitions and uses of identifiers) must be bound together into non-local productions to make incremental evaluation in large programs feasible.

Also of value was the inclusion of "pointer-valued" attributes. Some attributes in the early version of Poe (as well as later versions) have as their values Pascal pointers to parse-tree nodes. The most important use of this technique is for attributes which indicate type information. The "type" attribute of a given identifier or expression is simply a pointer to the appropriate "<type>" non-terminal node in the parse tree[7]. Similarly, the "constituent" or "base" type attribute of a <type> node is a pointer to the appropriate <type> node of its subtree. This technique results in significant space savings, since parts of a parse tree are used for two related purposes which might otherwise require separate storage: They describe the context-free structure of the program, and also the value of an object in the domain of Pascal types.

The copy rule-driven incremental evaluation strategy of the early version of Poe had several defects; to name some of them, semantic routines have embedded in them tests to see if their desired attribute arguments have yet been evaluated, distinctions are made in the code of the evaluation functions as to whether an initial evaluation or an incremental evaluation is being performed, and only propagation of attributes whose evaluation rules are copy rules is table-driven.

An experimental version of Poe exists in which attribute evaluation functions are simple, direct mappings of input attributes to output attributes. Knowledge of the mechanics of attribute evaluation is localized to the attribute evaluator. Attribute evaluation is completely automated and is based on tables produced by an editor generator. The editor generator written in conjunction with Poe, called Poegen, processes context-free grammars augmented with attribute evaluation rules and produces tables which control Poe's parser and attribute evaluator.

In the experimental version, the system implementor is required to supply the following:

(1) an attribute grammar indicating, on a production-by-production basis, what the inputs and outputs of each evaluation function are;

(2) the evaluation functions;

(3) functions which test attribute values for equality.

From the first of the above items, the complex tasks of planning the attribute evaluation process on initial read-in and in response to incremental tree editing operations are performed in advance by Poegen. Poe takes the tables produced by Poegen, the evaluation functions, and the equality tests, and performs the desired semantic operations in real time as the user edits his program.

Much of Poe is table-driven. This includes parsing, error-repair and (in the experimental version) attribute evaluation. This suggests that creation of LBEs for other languages should be comparatively easy. Experience to date confirms this. For example, an LBE (without semantics) for the VAL data flow language [M82] was created in less than a week. Work on the "VOLE" (as its implementor terms it) is continuing, and a complete editor (with semantics) is expected within the next year. An experimental editor for the "ABE" data base language [K81] has also been built.

Efficiency Issues

For consistency, pre-defined types such as "integer" and "boolean" are represented internally as parse-tree fragments; thus all type attributes are pointers.

LBEs are designed to be highly interactive, and hence are very sensitive to load in time-shared environments. Experience with version 1 of Poe supports this observation. The HP 9800 version of Poe is much more "smooth" and responsive than the Vax version, even though the MC 68000 processor used in the HPs has perhaps an order of magnitude less "mip-power" than the current generation of VAX-11/780's. As a result, recent development work has assumed an environment at least as powerful as that provided by a typical MC 68000-based workstation.

The HP 9800 version of Poe contains some 27,000 source lines. Much of this represents monitoring and debugging code. The program requires about 270k bytes (plus the standard boot system), which is about the same as the system Pascal compiler. With effort this size could undoubtedly be reduced, although major reductions would probably be painful. Trends in workstations point toward ever more generous memory capacities, so significant efforts in this direction are probably misdirected.

Program read-in speed is about 300 lines/min, representing scanning, parsing, pretty-printing, semantic analysis, and construction of an internal tree-structured program representation. Read-in speed would be greatly enhanced if an "internal form" image of a program could be written. This involves writing a program tree in a form that doesn't use explicit pointers.

Program trees are rather space intensive, requiring (on average) *hundreds* of bytes per source line. This is a major concern in that the size of programs that can be edited is sensitive to how compactly program trees can be represented. Some space inefficiency results from the fact that Poe trees are not completely abstract (i.e., redundant nodes are stored to simplify program display). The quality of Pascal packing and heap routines has a very direct effect on program tree size. Alternatives to syntax-tree representations including linear structures and production based-structures are being experimented with. A sufficiently fast and compact structure is still an unresolved matter.

Other User Interface Issues

"Prettyprinting" is automatic (and inescapable) when programs are displayed. Prettyprinting information (indent, outdent, newline, etc.) decorates program trees, and controls the mapping of program trees to screen format. The creation of prettyprinting tables is interactive, allowing different formating conventions to be accommodated. Ways of allowing user selection of formating rules are under study, although the most interesting proposal to date suggests an automatic "analysis" program that examines programs adhering to a desired style, and which infers from the examples the necessary "prettyprinting rules".

The "undo" facility can be used to reverse the effect of any tree insertion, deletion, elision, or unelision. Poe saves a history of the last several user modifications[8], so that successive "undo" operations undo the effects of successively earlier modifications. There is a corresponding "Un-undo" (or "Redo") command which can be used to move back forward in time and re-instate operations which have been undone. (If the user performs several "undo" operations and then manually performs a tree modification, it is no longer possible to use the "Un-undo" feature to reverse the effects of the "undo" operations.) Internally, Poe maintains a stack of (sub-tree, parse-node pointer) ordered pairs. The sub-tree is a copy of a tree that was either inserted or deleted by the user, and the pointer refers to the node in the parse tree at which the change took place. Note that both insertions and deletions can be viewed as exchanges of a null (empty) sub-tree and a non-empty sub-tree. To undo an insertion, Poe simply takes the parse tree node pointed to by the current element of the history list and deletes its sub-tree. Similarly, to undo a deletion Poe inserts the sub-tree of the current history list element under the node pointed to by that element. If the user deletes a structure, and then deletes another structure which contains the site at which the previous deletion took place, then the pointer of the former history element will point to a node imbedded in the sub-tree of the latter history element. This situation is acceptable because of the strict "last-done first undone" protocol of the "undo" stack.

Ongoing Work

Poe-related research continues in a number of areas. Integration of program development and testing facilities is an important goal. Execution of programs and program fragments should be freely intermixed with program development and editing. This includes the ability to edit data as well as program text, and the ability to resume suspended executions. Questions of how to map the "state" of a suspended execution to a modified program text are

[8]Current versions of Poe save the last ten user interactions.

under study.

Structuring concepts embodied in current programming languages are oriented toward static program representations (e.g., program listings). Given the far more dynamic representations made possible by LBEs, redefinition and generalization of traditional program structures will be studied.

Completion of a version of Poegen that automatically creates linear-time incremental attribute evaluators for attribute grammars with non-local productions is expected shortly. This will aid in automating the production of LBEs for other languages, and will reduce attribute evaluation errors found in earlier versions of Poe.

The current version of Poe has no "context-search" command. This certainly is not because such a mechanism is unnecessary, but rather because search commands found in ordinary editors aren't really suitable. Poe is structure-oriented rather than character-oriented, and search commands ought to be in terms of that structure. Ways of compactly abbreviating structure (akin to the regular-expression notation used in Unix systems) need to be explored. An interesting observation is that Poe's error repair facility can be used to do restricted forms of context-search. That is, entering (e.g.) "Then" can mean create a new structure containing a "Then" *or* it can mean match an existing "Then", and place a cursor just beyond the matching symbol. Whether text-entry and context search can be unified is an interesting open question.

At present Poe operates entirely at the token level. This means that an individual token *can't* be edited; it can merely be created, moved or deleted. This makes modification of complex tokens (e.g., strings and comments) difficult and unpleasant. A "character level" is probably indicated, although it will have to be carefully controlled to avoid reparsing problems.

Alternate elision mechanisms need to be explored. For example, "first line" elision (in which the first line of a construct is used to elide it) has the potential to better convey the intent of the construct. Thus rather than being elided to <IF-THEN-{ELSE}...>, an IF-THEN-ELSE might be elided to

```
IF a=1 THEN ... ELSE ...;
```

Comments, when they are available might also be employed:

```
(* Test for completion *) IF a = 1 ...
```

The idea is to minimize the amount of space needed to represent a construct, while conveying as much information about it as possible. Elision needs a more formal and systematic treatment to reach this goal.

Ways of automatically or semi-automatically eliding structure will also be considered. At first glance, automatic elision might seem simple — elide structure that is distant from the current position of the cursor (obtaining a form of perspective). The problem with this is that as the cursor is moved, the display tends to oscillate wildly, making viewing and cursor synchronization difficult. An alternative might be to allow global commands such as "elide all procedure bodies" or "elide all structure at a nesting level of 3 or more".

As mentioned in the introduction, execution and program-testing capabilities will be incorporated into the Poe environment. A promising possibility is to use denotational semantic descriptions of run-time program behavior. Ideally, a system such as Poegen will take a formal description of the dynamic semantics of a programming language and produce interpretation and high-level debugging facilities that are available from within Poe and can be used on incomplete program fragments.

References

[AC81] Archer, James and Richard Conway, COPE: A Cooperative Programming Environment, Cornell University TR 81-459, June 1981.

[FMQ80] Fischer, C., D. Milton and S. Quiring, Efficient LL(1) Error Correction and Recovery Using Only Insertions, *Acta Informatica,* 13, 2, 141-154, 1980.

[Hab79] Habermann, A. N., The gandalf research project, *Carnegie-Mellon University Computer Science Research Review - 1979,* 28-35, 1979.

[JF82] Johnson, G. F. and C. N. Fischer, Non-syntactic attribute flow in language based editors, *Proc. 9th ACM Symp. Principles of Programming Languages,* 185-195, January 1982.

[Joh83] Johnson, G. F., An Approach to Incremental Semantics, PhD thesis, Univ of Wisconsin - Madison, August 1983.

[K81] Klug, Anthony, Abe – A Query Language for Constructing Aggregates-by-example, *Workshop on Statistical Database Management* (1981).

[LF79] LeBlanc, R .J. and C. N. Fischer, A Simple Separate Compilation Mechanism for Block-Structured Languages, *Sigplan Notices,* 14, 8, 139-143, 1979.

[M82] McGraw, James R., The VAL Language: Description and Analysis, *ACM Trans. Prog. Lang. and Sys.,* 4, 1, 44-82, January 1982.

[MS81] Morris, Joseph M. and Mayer D. Schwartz, The Design of a Language-Directed Editor for Block-Structured Languages, *SIGPLAN Notices,* 16, 6, 28-33, June, 1981.

[Rep82] Reps, Thomas, Generating Language-Based Environments, Cornell University TR 82-514, August 1982.

[SFGT81] Stenning, Vic, Terry Froggatt, Roger Gilbert, and Ellis Thomas, The Ada Environment: A Perspective, *Computer,* 14, 6, 26-36, June 1981.

[TR81] Teitelbaum, Tim and Thomas Reps, The Cornell program synthesizer: a syntax-directed programming environment, *Comm ACM,* 24, 9, 563-573, 1981.

[WDT76] Wilcox, T. R., A.M. Davis and M.H. Tindall, The design and implementation of a table-driven, interactive diagnostic programming system, *Comm ACM,* 19, 11, 609-616, 1976.