Languages and Interactive Software Development *

Susan L. Graham

Computer Science Division – EECS, University of California
Berkeley, CA 94720 USA

Abstract. Most software is developed using interactive computing systems and substantial compute-power. Considerable assistance can be given to the developer by providing language-based support that takes advantage of analysis of software artifacts and the languages in which they are written. In this paper, some of the technical challenges and new opportunities for realizing that support are discussed. Some language design issues that affect the implementation of language-based services are summarized. The paper concludes with some proposals for assisting user understanding of language documents.

1 Introduction

The development and maintenance of software has evolved from the days of punched cards, long turnaround, and – by today's standards – small, slow, computers, to a world in which developers work interactively, using visually and audibly rich workstations and having network access to information. As the technology has changed, languages, tools and development practices have followed an evolutionary path. Although researchers have proposed a variety of useful and visionary approaches to exploiting the benefits of interaction [5, 7, 13, 14, 15, 23, 27], common practice has lagged behind. One of the reasons is the effort and expense of building interactive services for each new language or environment.

Powerful interactive systems create both technical challenges and new opportunities for providing language-based services. In this paper we survey some of the technical issues that arise in providing language support in interactive environments. We also consider some of the ways in which the heritage of off-line batch processing still influences the design of languages and tools, and suggest some departures from past practice.

By the word *language*, we will mean formal languages, not natural, spoken languages. Developers use many such languages in their work. The most obvious

* This research was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense under Grant MDA972-92-J-1028, and by the National Science Foundation under Infrastructure Grant CDA-8722788. The content of the information does not necessarily reflect the position or the policy of the U.S. Government.

are programming languages, or possibly design or specification languages. In addition, most systems provide a variety of document description languages (for example, TeX), and a plethora of command languages for interactive shells, configuration management and build processes, information management and so forth.

The discussion that follows draws most of its examples from programming languages and uses a program as an example of a linguistic entity. Much of the discussion pertains equally well to other kinds of languages, and to components as well as complete entities. Indeed, it is for many of the other languages that language-based support is absent. We will sometimes use the term *language documents* to suggest the more general domain. In this context, one can think of tools such as the formatter of a LaTeX document or the `make` program applied to a `Makefile` as a kind of compiler or interpreter.

The potential for language-based support has been demonstrated in many single-language environments. Notable among them are LISP and Smalltalk systems. Our interest is in making the benefits more easily available for other languages. One of the ways to make language-based services more widely available is to develop language-based tools that are *description-driven*, that is, instantiated by specifying a particular language. That approach is facilitated by the use of formal declarative specifications as much as possible, to minimize recoding.

The Ensemble project at Berkeley is investigating the technology to provide interactive language-based services for both formal languages and multi-media natural language documents. Many of the ideas and observations in this paper are the outgrowth of that research effort and its predecessors, the Pan project [4] and the VorTeX project [6].

In the remainder of the paper, we first sketch some of the language-based services we have in mind. Next we describe some of the technical issues that arise in interactive language processing, followed by a discussion of the effect of certain language design choices on that processing. We highlight two issues that demand special attention in an interactive environment – the consequences of incomplete information, and the nature of document presentation. Finally, we consider the use of language services for user understanding of language documents.

2 Interactive Language-based Services

One of the important characteristics of an interactive system is the ability of a user to engage in a dialogue with the system. That property is exploited in a modest way by contemporary text-editors, such as Emacs [22], that incorporate services that check well-formedness properties of the document being edited and assist the user in discovering and correcting mistakes. Two familiar examples are checking natural language spellings and detecting unmatched bracketing characters such as parentheses. Using the Emacs extension language, it is possible to define language modes that extend checking to other syntactic forms, and also provide some assistance in introducing those forms.

The kinds of language-based services we have in mind are in keeping with the examples just given. The system "understands" the languages in which a document is written and can respond accordingly. A rich dialogue can ensue of the system knows not only syntactic properties of the language but also some semantic properties [2]. Even greater benefits can be achieved if the system can extend its services to documents that are compound, both in the sense of containing multiple languages, and in the sense of having structure and relationships that transcend a single file or storage unit.

Our emphasis in this paper is on services that require fine-grained structure and analysis. "In-the-large" services at the granularity of modules, chapters, functions, etc. are equally valuable, but involve a different set of issues and engineering decisions.

Consider the services of an interactive editor. A text editor deals with the language of text, consisting of characters, words and lines. (In some direct manipulation systems, that language is enriched to include fonts, sizes, colors, and other format-related attributes.) A program with a textual representation can be written and modified as a textual document. Among the other services offered by a text editor are search and navigation operations, such as "move to the next line", or "replace all occurrences of `foo` by `bar`".

If the program is regarded as a document in the programming language rather than the language of text, then similar services can be provided in that language, for example "move to the next variable", or "rename (*definition and uses of*) type `fooType` as `barType` within the innermost scope in which `fooType` is defined". Thus textual navigation and search/replace commands are extended to linguistic forms of those operations. Another example is to extend textual notions of cut/paste to structural or semantic operations.

In order to support structural and semantic versions of text operations, the system must maintain information about the syntactic and semantic structure of the document. Once linguistic information is available, new services can be provided. Familiar examples include highlighting or elision of structural components, formatting, class hierarchy navigation, and call-graph browsing.

The opportunity also exists to provide new operations based on information derived from analysis or user annotation. Van De Vanter [25] gives the following example. Many programs contain both mainstream problem-solving code and a possibly substantial amount of code intended to handle special cases, errors, and other infrequently occurring situations. If a developer is attempting to understand an unfamiliar program, then it is the mainstream code that is of interest initially. Understanding is enhanced if the developer can visually identify the mainstream code and downplay the rest. Identifying which code is mainstream might require the combined efforts of automated analysis and annotation by the author of the program, probably at some earlier time. If the identification of mainstream code is available, browsing and display techniques can be used to show it to the developer.

[2] In keeping with the use of the term in compilation, we use the term *semantic* to refer also to some properties that are syntactic in the traditional linguistic sense.

3 Interactive Language Processing

In order to provide interactive language-based services, a system builds a structural description of the program, augmented with annotations, links, and auxiliary information. Early systems, such as the Cornell Program Synthesizer [24], and Mentor [8] required that the system maintain a structurally well-formed program, by providing a *structure editor* interface that limits the user to structural modifications. Later *syntax understanding* systems support text editor interfaces [3]. Structure is inferred by analysis (namely, some form of parsing). Policies for analysis include analysis on user demand, periodic analysis (e.g. when the user pauses), continuous analysis, or analysis when some structural operation such as navigation is invoked. In many of these systems, structure also can be provided explicitly through the use of templates.

The reason that early systems require structural well-formedness is not only to eliminate the need for structural analysis, but, more importantly, to facilitate other structurally-based forms of analysis. For example, traditional kinds of static semantic checking might be carried out using some sort of attribute grammar technology on an abstract syntax tree. Even less formally specified analyses tend to be based on a structural representation.

Many interactive systems are based on the use of *incremental* analysis algorithms. An incremental algorithm is one that updates existing information by first determining what has been modified and then propagating the consequences of the modification. Historically the motivation for incremental algorithms as opposed to recomputation was performance – to reduce the number and extent of updates when local changes are made to large documents. As computing speeds have increased, the performance issue has declined in importance.

There are other important benefits of incremental algorithms. By determining what has changed, a system can provide a variety of change-related user feedback. (An example is highlighting on a screen.) More importantly, by determining what has *not* changed, a system can preserve information that cannot be reconstructed by analysis, such as user annotation, or other information provided by external agents. Incremental algorithms allow changes to be made in place, rather than the copying that would otherwise be required to capture changes by comparison.

Using incremental analysis instead of 'batch' analysis has two important characteristics. First, the processing order is largely temporal rather than structural. In other words, change-based analysis or services propagate from the changes outward rather than from the beginning of the document forward. Consequently, processing algorithms that depend on seeing one piece of information before another achieve "before-ness" differently in non-incremental and incremental settings. Furthermore, incremental algorithms need special mechanisms to handle non-adjacent textual ordering. Second, it is commonplace for processing to be done with incomplete information and incomplete documents, either because the program is only partially developed, or because conceptual changes are made in a sequence of steps.

[3] Later versions of the Cornell system support some text editing as well.

4 Language Issues

There are a variety of language design issues that complicate interactive language processing. Some of them are issues that cause difficulties for formal specification of languages. Often those same features complicate specifications used for compilation. An example is the need for unbounded lookahead to make certain parsing decisions. The more interesting issues in the context of this paper are the ones in which the temporal processing order comes into play.

In considering the material in this section, the knowledgeable reader might object that the language design issues are all problems that the language design community recognizes and can avoid. However, to be most useful, language-based services should be available for *all* languages. In fact, it is for badly designed languages that help is needed most! Additionally, designers of other kinds of languages seem to be making the familiar mistakes and then some.

4.1 Information Feedback

Conventional language analysis is decomposed into three stages

1. lexical transformation of a sequence of characters – screened or filtered to remove formatting information and commentary – into a sequence of tokens,
2. syntactic transformation of a token sequence into phrase structure,
3. "semantic" analysis to achieve bindings, name resolution, and attributions.

Although these stages can be composed into a single syntax-directed analysis in which lexical analysis and semantic analysis are embedded, the architecture used by many optimizing or retargetable compilers and by most interactive language processors is to construct an explicit structural representation first, and then to determine bindings and attributions by traversal of the structure, perhaps using some sort of attribute evaluation formalism. If the stages are independent, then there are known techniques to carry out each stage incrementally [2, 3, 18, 21]. However, if earlier analyses require feedback from later ones, then a multi-phase analysis becomes awkward.

Example: C/C++ Type Names

Consider the fragment

$a(*b);$

If `a` names a function, then the fragment denotes a call of function `a` with argument `*b`. However, if `a` denotes a C type, for instance in a context containing

typedef int $a;$

or a C++ class, then the fragment defines `b` to be a variable whose values are pointers to entities of the type or class denoted by `a`.

It is normally desirable to use different structural representations of the call `a(*b)` and the variable definition `a(*b)`. Yet the structural analysis of the fragment needs information about the binding of `a` that is determined by the later semantic analysis phase, which in turn requires a structural traversal. The need to know `typedef` bindings in order to determine structure also exists in C and C++ compilers. Since the textual occurrence of the `typedef` precedes uses of the defined identifier, maintaining binding information during parsing solves the problem, although at the possible cost of more complicated specification and additional mechanism.

In an interactive environment, in which attribution follows structural analysis in order to support incrementality, `typedef` bindings are necessarily a special case with a separate mechanism. Furthermore, if the contextual information is changed (for instance, the `typedef` is removed or its identifier spelling is changed) then not only must all fragments containing uses of `a` bound to the `typedef` be reparsed, but all attributions using attributes of `a` must be reevaluated. The reevaluation is initiated as a consequence of dependency information maintained by the incremental semantic evaluator. One way in which the reparsing can be triggered is by using a special token, say `TYPE-ID`, for identifiers designating types, and by replacing appropriately-bound occurrences of `ID` by `TYPE-ID` or vice-versa when type definitions are added or removed, in order to force a syntactic change event. Those replacements, in turn, require the existence of semantically analyzed bindings to reflect the scope rules of the language.

Example: User Operator Priority Settings

In several languages, notably PROLOG, ML, FIDIL [16], users can define a new infix operator and specify a parsing priority for it. The priority determines the structure of each expression in which the operator is used. The priority is normally associated with the operator during the attribution phase, and the operator definition usually depends on scope analysis. Thus many of the same issues arise as in the previous example. If there are a small number of possible priorities, as there are in FIDIL, then the token-based solution outlined in the earlier example can be used, by introducing a separate user-defined-operator token for each priority. However if, as is the case for PROLOG, and for some implementations of ML, there are a large (effectively unbounded) number of levels, then a complicated reparsing strategy may be required.

4.2 Contextually-determined Information

In some older languages, notably FORTRAN, PL/I, and some dialects of BASIC, keywords are not reserved, and the use of an identifier as a keyword is inferred from context. Although methods exist to do the appropriate analysis, they are not necessarily supported by description-driven tools.

The use of embedded sublanguages is also determined by context. One example is formatting specifications in output directives. Another is math mode in TeX. Typically, these sublanguages have their own syntactic and semantic

rules. Syntax analyzers in translators often handle sublanguages by maintaining a global state variable and switching among analyzers as the sublanguage boundary is crossed. Incremental analyzers must also be able to determine that state, but not by use of a global variable. Instead the sublanguage boundaries are part of the structure, either in the form of attributions, or in the form of links to separate structures. The treatment of sublanguages must be robust if the determining contextual cues are unavailable.

4.3 Formatting-related Syntax Rules

Some language designers introduce syntax rules – intended to provide readability or typing convenience for the user – that may complicate language specification or processing. The problems with FORTRAN whitespace insensitivity are well known to compiler writers.

Ends of lines are often used as delimiters, sometimes in complicated ways. FORTRAN, COBOL, and UNIX shell languages are examples of line-oriented languages in which escape symbols are required to prevent textual line breaks from being treated as logical end-of-statement delimiters. In Icon [12], a line break sometimes serves as a statement delimiter; but only if the last token before the line break and the first token after it are not part of the same construct.

In Haskell [17] indentation can be used to indicate nesting, in place of explicit bracketing tokens, thereby making some line breaks, white space, and column positions significant. In `make` [9], commands must be indented by at least one tab character (and not by the visually indistinguishable sequence of blank characters). In the former case, the reader is helped at the expense of the language tools. In the latter case, the opposite is true.

4.4 Preprocessors and Macros

Macros are a metalanguage that is often used for abbreviation or language extension. If a macro language and the language in which it is used have the same lexical and syntactic rules, as is the case in many LISP dialects, then the major complication in incremental analysis comes from the treatment of errors. Macro expansion and analysis of the expanded language can be intermixed, as long as the system retains enough information to replace the expansions when a macro definition is changed. That information is also needed to support selective viewing of macro expansions. Since the expansions are structurally well-formed, both replacement and selective viewing reuse mechanisms available for other purposes.

If the macro language transforms the text to which it is applied at the character level, or even at the token level, as is the case with macro processing embedded in preprocessors such as the C preprocessor (`cpp`), then the situation is more difficult. The nature of such a preprocessor is intrinsically a sequential rewriting of the text. Since expansion in different contexts may create different structure, an unexpanded macro call can cause syntactically invalid text in the language to which the macros are applied. Also, since expansion causes changes

in structure, selective viewing of expansions and macro redefinition become more complicated for the system to support.

Finally, if arbitrary file inclusion is incorporated, as it is in `cpp`, then the benefits of sharing copies of expansions, which are an important engineering consideration, may be difficult or impossible to achieve. Since multiple inclusions of the same file are used in the absence of linguistic support for interface modules, and those files are often large, the space required for multiple copies and the time for multiple analyses can be significant.

5 Incomplete Documents

The language design discussion illustrates the fact that language analysis often uses information that is available in a complete, well-formed document, but may be missing in a document that is being developed or modified. In a translator, if a document is incomplete or not well-formed then it is in error. Syntactic and semantic error processing mechanisms are invoked, and appropriate diagnostic information is generated for the developer.

However, from the developer's point of view, incompleteness or partial modification are different from errors. Suppose we refer to all these situations by the less-loaded term *anomalies*. Since anomalies are a normal occurrence, system services should be maintained in their presence. That has the following consequences for the design of an incremental system.

Partial structure and analysis must be available. Structure and attribution-based services such as navigation, display, and querying should continue to be available. Change-based analysis provides considerable help in this regard, since the structure and properties of unchanged components can be retained even if they are part of anomalous constructs.

Knowledge about anomalies is important to the user. Since the anomalies are part of the linguistic information about the document, linguistic services such as navigation and highlighting should apply to them as well.

The user should decide when to resolve anomalies. If an interactive system is to assist a developer, then the developer must be able to choose the order in which to work. That means that the services must not degrade if anomalies are unresolved. The requirement in early structure editors that syntax anomalies be absent was an important factor in their lack of wide-spread use.

It is because of the need to support anomalies, that some of the language design issues we have summarized are particularly problematical. If one is to provide language-based services using description-driven tools, then language features requiring special handling impose an additional barrier. Additionally, a system that is robust in the presence of anomalies, must provide linguistic services in the absence of linguistic information. For instance, the system must choose some way to format `a(*b)` for display even if its structure is unknown,

and must have some reasonable policy about communicating type anomalies that stem from a lack of binding information rather than a mistake on the part of the developer. Van De Vanter, Ballance, and Graham [26] provide further discussion of these issues.

6 Presentation and User Comprehension

An important part of the task of constructing and modifying language documents is human comprehension of both their form (i.e. the use of the language) and their content. The previous discussion has suggested some ways in which interactive tools can assist developers in using a language. Interactive systems can also assist developers in understanding the content of a language document. We will touch on two kinds of assistance – better readability and better association and preservation of auxiliary information. Space does not permit discussion of an important third kind of assistance, namely better understanding of dynamic behavior or of the output of language tools.

6.1 Presentation

The programming language research community has long realized that the same language can be represented in more than one form. The notion of abstract syntax goes back to the 1960's, along with the pejorative reference to concrete syntax as "syntactic sugar". Nevertheless, one of the characteristics of most language definitions is the inclusion of rather specific rules for their concrete syntax. The reason for the precision of the definitions, of course, is that they serve as input specifications for language-processing tools, notably compilers and interpreters. That precision was essential in the days of punched cards and paper tape.

The progression from keypunches to text editors to *language-sensitive* editors has done little to loosen the rules of concrete syntax. Syntax is usually expressed as a sequence of ASCII characters. Font shifts (for example, emboldened keywords) and colors are sometimes used, and layout styles are sometimes incorporated automatically. It is an easy matter to map those enhancements back to an ASCII character sequence.

In an interactive computing environment, the user prepares and modifies a language document on-line and also invokes a compiler, interpreter, or other language processor on-line. There is no reason that the form of the language read and written (and heard?) by the user need be the same as the input representation to a language tool, as long as an appropriate input representation can be generated automatically from the human-created version. The user should be able to use a representation that facilitates comprehension; not one that has been designed to ease translation. On the other hand, the input to the language translator need not be burdened with formatting-motivated syntax rules.

An interesting study by Baecker and Marcus [1] suggests some ways in which appearance can affect human understanding of language documents. They devised a variety of ways of presenting C programs to improve their readability.

An example appears in Figure 1. Many other researchers have proposed partic-

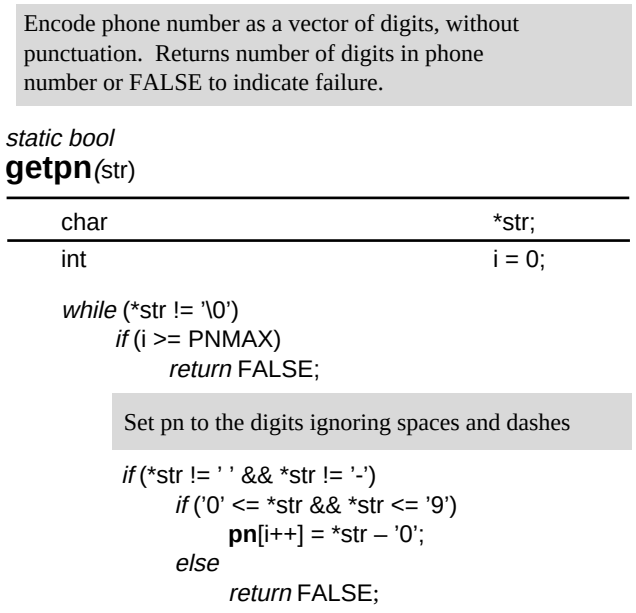

Fig. 1. A program presentation example from Baecker/Marcus [1, pg. 61]

ular stylistic choices of concrete syntax, formatting, and appearance to enhance understanding. Studies such as that of Oman and Cook [20] demonstrate that careful use of typographic effects can strongly influence how well programmers, either novice or expert, understand a program.

The work of Baecker and Marcus is intended primarily for display, not for interaction. By building interactive tools that treat appearance separately from structure and content, styles of presentation can be designed for comprehension and can be customized for user preferences. Concrete syntax can be defined to facilitate tool building and not be cluttered with readability considerations.

To support presentations such as the one in Figure 1, the system must provide high-quality typography and formatting, reordering of abstract syntax components, and the ability to map back and forth between the presentation and the components of the abstract syntax representation. In addition, the notion of ordering used for navigation must be carefully considered to avoid user confusion.

The Ensemble project at Berkeley is developing the technology to support such separation of appearance. The structure, the semantics, and one or more styles of appearance are formally specified for a language [4]. Appearance is specified by a *presentation schema*, which provide the rules that are applied incre-

[4] The conventional concrete syntax must also be specified, in order to pass language documents into and out of the Ensemble system.

mentally as the document changes. The first version of the presentation system is summarized in a conference paper [11]; a later version is described in Munson's dissertation [19].

6.2 Annotation

One common language feature intended for human comprehension is the comment. In most text-based languages, comments are text as well, and are separated from other constructs in the language by delimiters. Comments are normally ignored by language translators or interpreters. Some researchers have proposed systems of formal annotation as well, that are, in effect, an embedded sublanguage with formal properties.

If an interactive system supports separate mechanisms for presentation of language documents, and if it supports structural representations of language documents, then additional kinds of annotation become possible. A simple example is to associate comments, whether formal or in natural language, with semantically attributed structural components of the document, rather than with textual positions. These annotations can remain bound to the structural components (which might be important user abstractions) even if the textual document changes. Furthermore, it is possible to use the richness of multimedia to provide non-textual static or dynamic, or even audible comments, such as an animation of the use of a data abstraction. By associating computable predicates with the comments, mechanisms can be provided to discover and signal their possible lack of validity as the document changes. It is also possible to use increasingly common notions of hyperlinks to associate electronic information outside the document with its entities and structures.

In this scenario, a program in its entirety consists of much more than the instructions for an abstract execution engine. The input to a compiler or interpreter is obtained by extracting the relevant view in the appropriate representation. Interactive execution or debugging are easily incorporated in this point of view.

Annotations can be provided by tools as well as by people. The semantic attributes calculated by an incremental analyzer, or the attributes derived from data flow analysis constitute annotations. As another example, execution profiling data can be associated with components of a program and used both for improved compilation and for focusing the developer's attention.

An important property for a system that supports rich forms of annotation is flexible access to the information. It is useful to regard an annotated document as a semantically rich database, in which the information can be queried or viewed as desired. The technical challenge is to support that behavior even though the system representation of the information is not at all that of a database.

6.3 Information Retention

The final step in regarding a language document as a human-centered artifact in which comprehension is an important property is the retention of information

over time. One of the ways change-based analysis can be used is to preserve a language-based modification history, as opposed to the textual versioning systems in use today, and to retain the associated annotations. Some of the technology needed for that kind of history is a consequence of the mechanisms used for language-based *undo* services. Another often-suggested idea is to record, along with changes to a document, cognitive information such as a design rationale for the change. The impediments to realizing that idea are primarily nontechnical. The advantages of using a language-based approach lie in associating the rationale with the change itself and not just its natural-language description.

Acknowledgements

Many of the ideas expressed in this paper are the outgrowth of collaborations with research students and colleagues in the Pan project and the Ensemble project at Berkeley. Bruce Forstall's M.S. report [10] contains an extensive discussion of language specification issues from which some of the material in Section 4 is drawn. Michael Van De Vanter's dissertation [25] provides a wealth of insights about the ways in which an interactive system can assist user understanding of language documents, some of which are summarized in Section 2 and Section 6.

References

1. Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs.* Addison-Wesley, Reading, Massachusetts, 1990.
2. Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 185–198, Atlanta, Georgia, June 22–24, 1988. Appeared as SIGPLAN Notices, 23(7), July 1988.
3. Robert A. Ballance and Susan L. Graham. Incremental consistency maintenance for interactive applications. In K. Furukawa, editor, *Proc. Eighth International Conf. on Logic Programming*, pages 895–909. The MIT Press, Cambridge, Massachusetts and London, England, June 1991.
4. Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.
5. David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors. *Interactive Programming Environments.* McGraw-Hill, New York, 1984.
6. Pehong Chen, John L. Coker, Michael A. Harrison, Jeffrey W. McCarrell, and Steven J. Procter. The VorTeX document preparation environment. In *Proc. Second European Conf. on TeX for Scientific Documentation, Lecture Notes in Computer Science No. 236*, pages 32–54, Strasbourg, France, June 1986. Springer-Verlag.
7. Reidar Conradi, Tor M. Didriksen, and Dag Wanvik, editors. *Advanced Programming Environments.* Number 244 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1986.

8. Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The MENTOR experience. In David R. Barstow et al., editor, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, New York, 1984.
9. S. I. Feldman. *Make—A Program for Maintaining Computer Programs.* Bell Laboratories, Murray Hill, NJ, 1978. In the Unix programmer's manual, vol. 2.
10. Bruce T. Forstall. Programming language specification for editors. Master's report, Computer Science Division—EECS, University of California, Berkeley, November 1991.
11. Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 130–138. ACM Press, December 1992. *ACM Software Engineering News 17* (5), December 1992.
12. R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1983.
13. Peter Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984. *ACM SIGPLAN Notices, 19* (5), and Software Engineering Notes 9 (3), May 1984.
14. Peter Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1986. *ACM SIGPLAN Notices, 22* (1), January 1987.
15. Peter Henderson, editor. *ACM SIGSOFT '88: Third Symposium on Software Development Environments*, 1988. *ACM SIGPLAN Notices, 24* (2), Feb. 1989 and *Software Engineering Notes 13* (5), Nov. 1988.
16. Paul N. Hilfinger and Phillip Colella. FIDIL: A language for scientific programming. Technical report, Lawrence Livermore National Lab., Livermore, CA, January 1988.
17. Paul Hudak and Philip Wadler. *Report on the Functional Programming Language Haskell*, 1990.
18. Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *Conf. Record Ninth ACM Symposium on Principles of Programming Languages*, pages 196–206, 1982.
19. Ethan V. Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment.* Ph.d. dissertation, Computer Science Division – EECS, University of California, Berkeley, 1994. To appear.
20. Paul Oman and Curtis R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.
21. Thomas Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge, Massachusetts and London, England, 1984.
22. Richard M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In *Proceedings, ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147–156, Portland, Oregon, June 8-10, 1981. Published as SIGPLAN Notices 16(6), June 1981.
23. Richard N. Taylor, editor. *SIGSOFT '90 Proceedings of the Fourth Symposium on Software Development Environments*, Irvine, CA, December 3–5 1990. ACM SIGSOFT Software Engineering Notes, 15(6), December 1990.
24. Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

25. Michael L. Van De Vanter. *User Interaction in Language-Based Editing Systems.* Ph.d. dissertation, Computer Science Division – EECS, University of California, Berkeley, December 1992. Available as Technical Report No. UCB/CSD-93-726.
26. Michael L. Van De Vanter, Robert A. Ballance, and Susan L. Graham. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37(4):431–466, 1992.
27. Herbert Weber, editor. *SIGSOFT '92 Proceedings of the Fifth ACM Symposium on Software Development Environments*, Tyson's Corner, VA, December 9–11 1992. ACM SIGSOFT Software Engineering Notes, 17(5), December 1992.

This article was processed using the LaTeX macro package with LLNCS style