Syned -- A Language-Based Editor for an Interactive Programming Environment

E. R. Gansner, J. R. Horgan, D. J. Moore, P. T. Surko, D. E. Swartwout

Bell Laboratories

J. H. Reppy

Cornell University

ABSTRACT

Language-based editors (LBE's) for programming languages are central components of Interactive Programming Environments (IPE's). These editors incorporate knowledge about programming languages in order to make program construction less difficult. They may also serve as gateways to the IPE, structuring programs as objects amenable to manipulation and control by the IPE. This paper describes **Syned**, a prototype LBE which runs under the UNIX* operating system. **Syned** accepts the full C language, except for some pre-processor statements, and allows free and unrestricted intermixing of parsing and menu selection in the creation of programs. It offers a powerful editor control language and a highly flexible, extendable menu system. **Syned** can be configured to accept other languages in addition to C. This paper is primarily concerned with LBE's as program editors rather than as IPE input gateways. However, some aspects of **Syned** as an IPE gateway are discussed.

1. Introduction

Language-based editors (LBE's) for programming languages are central components of Interactive Programming Environments (IPE's). These editors [Do,Fe,Fi,No,Te] incorporate knowledge about the programming language in order to make program construction less difficult. They may also serve as gateways to the IPE, structuring programs as objects amenable to manipulation and control by the IPE. This paper describes our approach to language-based editing.

Syned is the editor component of an IPE we are developing. It is similar to those of the Gandalf [No,Fe], Cornell Program Synthesizer [Te], and Mentor [Do] projects, but contains several significant extensions. We are presently concerned with LBE's mainly as program editors rather than as IPE input gateways. However, some aspects of **Syned** as an IPE gateway will be discussed.

Because we wish to accommodate the needs of professional programmers at Bell Labs, we designed **Syned** to accept the full C language (as well as other languages) and allowed free and unrestricted intermixing of parsing and menu selection in the creation of programs.

2. Features

Syned is an LBE which supports programming in C as well as a set of facilities for creating program editing environments. **Syned** furnishes two modes of operation, tree and text, at all times. A high-level control language and an extendable menu system make it possible to tailor the editing environment to suit special needs.

2.1 Text/Tree Bimodality

An abstract syntax tree is maintained as the internal representation of programs constructed in the **Syned** environment. A display machine constructs the normal textual presentation on the screen. At any time in program construction, expansion of language nonterminals can be done using templates chosen

---

* UNIX is a trademark of Bell Laboratories

from menus. At high levels, users usually follow this approach. In addition, at any time, nonterminals can be replaced by text which is parsed into a subtree and inserted into the internal representation. Given this bimodality, the user may manipulate program subtrees (prune, graft, copy subtree to a buffer), or may manipulate blocks of text with a text editor and then have the block reparsed back into the tree. They may move the cursor in the tree (parent of the current node, child, left sibling, etc.) or in textual fashion (by line or syntactic unit, by scrolling, or by regular expression searching). The ability to construct programs either by tree or text building is, to our knowledge, a unique improvement in editors.

Our experience suggests that such duality overcomes much of the perceived rigidity of language-based editors [Wa]. Each entry method has advantages to users. Menus allow one to lay out the broad structure of a program quickly and clearly. Menus, offering only syntactically correct choices, are helpful if users are coding in a language which they seldom use. Choosing by menu also saves keystrokes when using a wordy language such as Ada. Parsing is useful when the user knows exactly what is wanted and how to express it correctly. The point at which the user decides to switch from menu selection to parsing is dependent on several things: the language in use, the particular user, and the level of experience. Beginning and experienced programmers have different needs, and simple languages require less parsing than rich but terse ones. Given the variety of needs, we feel an editor useful to professional programmers must offer parsing and menu entry at every nonterminal.

A typical **Syned** interaction to create a new program starts when the user lays out a basic control structure using menus, including personalized menus for laying out the user's own favorite structures. The remaining nonterminal placeholders are expanded using the text editor and parser. Sections of program are moved as subtrees, re-edited as text, and so on. The ability to switch modes easily seems to be a key to the usefulness of the editor.

While we have demonstrated the usefulness of bimodal editors, we do find that certain costs are incurred. Some work is involved in constructing the tables which instantiate **Syned** for a particular language (described below). However, this one-time investment is well rewarded. There is also greater resource consumption. In our experience using **Syned** under the UNIX operating system, processor time and memory usage are measurably greater than that of a screen text editor, but there is no noticeable response time difference. The bandwidth needed to keep the displayed version up to date makes the editor unattractive when a 300 baud line connects the user's crt to a time sharing server, but 1200 baud or greater is satisfactory to nearly everyone. **Syned** is designed to run in an average timesharing environment, but will certainly be even more pleasant to use in a modern setting providing, for example, high bandwidth to a large bitmap screen, and good local processing.

The way in which the bimodal editing is made possible is detailed in a later section. It involves the use of separate grammar tables for structure, menus, and parser, with uniform access to the tree building mechanisms.

2.2 Syned Control Language

Syned has a powerful command programming language. SCL (**Syned** Control Language) is a procedural language for manipulating abstract syntax trees. It has the repertoire of control structures of a modern structured language. SCL does not have the full range of data-structuring and type-defining devices; rather, because it is intended to manipulate a well-known data structure, SCL permits data types and structures specific to the purpose. The data types include the usual **boolean**, **integer**, and **string**, and the less usual types **node** and **menu**. The primitives of the language include all of the keyboard-callable editor operations, such as "copy this subtree into a buffer", plus such boolean operations as "is the cursor on a block node?", special tree manipulation operations such as "go to the root", and display manipulation operations such as "show the changes just made to the tree". Using this language we are writing language-dependent semantic routines such as type checking aids, and tailoring menus to user-preferred idioms. It is a very important tool for editor fashioning.

SCL routines are callable from menus or from the keyboard. Users may designate single keystrokes as names of such routines for convenience. Thus, for example, one may key "D" to initiate an interactive declaration procedure which checks to see if the cursor is resting on an identifier, then sequentially offers the user the choice of all syntactically valid scopes for the declaration beginning with the innermost. When

the user chooses a scope, the routine enters the inscription of the variable, offers a menu of type choices, and finally returns to the original position in the program.

2.3 Extendable Menus

Syned allows the tailoring of menus, both to describe the basic syntactic choices in the language and to offer application-specific extensions. One tailored facility allows the user to walk through a menu hierarchy representing commonly used UNIX functions. When the desired function is selected it is automatically declared, other statements are inserted in the program if needed, and each argument is declared as the user replaces formal placeholders with actual parameters. It is difficult to predict whether a proposed user interface feature will be useful, too sparse, or too intrusive. Using SCL one can quickly implement and modify these interface procedures.

Menus are defined in *menu tables*, which are not bound into **Syned** itself. In order to modify or extend the menu system, the user edits a menu file using a **Syned** grammar especially built for the purpose and then runs a binder program which integrates the menu table with the template grammar.

3. Architecture

Syned consists of four logically distinct "machines". The *tree machine* manipulates the abstract syntax trees. The *command machine* executes SCL programs, keyboard commands, and manages communication among the other machines. The *display machine* deparses trees to text and provides the text editing functions. The *parser* accepts text and generates strings of primitive tree building commands.

There are several tables and grammars which drive these machines. The *menu table* provides the menus to be displayed and the command strings which drive **Syned** to produce the results corresponding to the menu selection sequence. A *template grammar* represents the possible elementary subtrees used to build complex trees. A *parser grammar* is input to a parser generator in order to produce the parser.

3.1 Internal Machines

Syned's four internal machines are implemented as collections of related subroutines in a single UNIX process. The intermachine data transfer consists primarily of access to major data structures. Control coordination is done by the command machine.

3.1.1 Command Machine. The user communicates with the command machine. It causes direct execution of the primitives of the other machines, or obtains a list of commands to execute from the menu system, or executes an SCL routine which involves a complex use of the other machines. It relies upon a table of SCL routines and the *menu table* for descriptions of complex actions it is to perform.

3.1.2 Tree Machine. This machine builds and modifies the abstract syntax tree which is the principal internal representation of the user's program. It maintains the positioning of the tree cursor, which is the most important and widely relied-upon indicator of the state of **Syned**. These tasks rely upon the proper maintenance of a dozen pointers for each node of the abstract syntax tree. The tree machine consults the *template grammar* for knowledge as to what it may do in constructing and pruning subtrees. It is invoked by requests from the command machine.

3.1.3 Display Machine. The abstract syntax tree is mapped to text by this machine at certain times determined by the command sequences (e.g. after each keyboard command or after the completion of a phase of internal executions of SCL). It maintains an internal form of the text and determines how this is to be mapped to the screen in the face of scrolling, overlaying of menus, and other perturbations of the user's view. The *template grammar table* contains scripts which describe how the subtree under each node is to be displayed. The *menu table* provides all the information used to construct the displays of menus.

3.1.4 Parser. The parser maps program text to abstract syntax subtrees. Text to be parsed may be obtained from a file or from an ordinary text editor which may be invoked as an alternative to tree-mode editing. Parsing depends on the type of non-terminal to be expanded. For example, when the tree cursor is positioned on a <STATEMENT> node, statements and only statements may be parsed at that point. The parser refuses to construct a declaration under such a node, even if the declaration is properly formed. A **Syned** parser consists of an ad-hoc scanner and a YACC [Jo] grammar with actions for formulating

command strings. These strings are built up recursively from the parts obtained when reductions occur in the course of parsing. The strings are then used to drive the construction of abstract syntax trees. Many of the constructed strings contain calls to SCL routines which handle the context sensitive aspects of language translation.

Since the parser is capable of treating any non-terminal as if it were the start symbol, it can be used interactively as an incremental parser. A number of LBE's employ incremental parsers of a different sort from ours([Fi],[We]). Much of the effort in these editors is in the development of novel techniques to avoid unnecessary reparsing of program text. Each of these parsers represents considerable research and none relies on the established parser generator technology. The **Syned** implementation, on the other hand, extends and relies on an automatic parser generator, YACC. Parsers for new languages for **Syned** are therefore relatively easy to construct. We are in the process of automating the construction of these incremental parsers from YACC grammars. This automation has already been achieved for a simpler class of LBE's than Syned [Ki,Bo]. This method does not guarantee optimally small reparsings; however, as it relies upon established technology, it does not require parser handcrafting.

3.2 Tables and Grammars

One of the simplifying design ideas in **Syned** is the separation of *template grammar* and *menu tables*. Before we did this, we had very little flexibility in building menu systems. Now a two-step process is used. First, the two tables are built separately, with no externally imposed restrictions. Then the two are compiled. This two-step process yields a marked improvement in **Syned** loading time.

3.2.1 Template Grammar. In this table there is an entry to represent any nonterminal node type in a possible abstract syntax tree. With each nonterminal is associated the set of all possible replacements which constitute one-level subtrees to be hung under the original node. Thus under <EXPRESSION> there are replacement entries described by "<TERM> <BINOP> <TERM>" and "<UNOP> <TERM>", among others. Also associated with each entry is a script which describes how the subtree under the node is to be displayed. The script contains both the syntactic sugar of the language and a part governing the pretty-printing of the text on the screen.

This template grammar is strictly context-free with no empty productions. It must explicitly realize every valid syntactic form, (e.g. both "for(; x2 ; x3)" and "for(x1 ; x2 ; x3)" are separately represented). If the subtree "x1" in "for(x1 ; x2 ; x3)" is pruned, the action causes the correct new form "for(; x2 ; x3)" to be associated with the root node of the subtree. The internal grammar form of a string "for(x1 ; x2 ; x3)" is actually a template "<EXPRESSION> <EXPRESSION> <EXPRESSION>" indicating that the subtree consists of three sibling subtrees of type "<EXPRESSION>". The syntactic sugar "for(--;--;--)" resides in an associated display script for a simple pushdown deparsing machine. Although we describe it as "sugar", the display script can be an essential attribute of the node it describes. For instance, the grammar template for a conditional C expression "x1 ? x2 : x3" is identical to that of "for(x1 ; x2 ; x3)". Therefore, only the different display scripts differentiate these two constructs for the user.

3.2.2 Menu Table. In the standard menu table there is an entry for each nonterminal. The choices are not usually in strict correspondence with the entries in the template grammar. Each menu choice for a given nonterminal has associated with it a string of commands which, when executed, builds an appropriate subtree and/or performs other operations. Elaborate manipulations of the abstract syntax tree are sometimes needed in order to make a menu choice behave in a reasonable way.

The tailorable menu table is interlaced with the standard one. The menu corresponding to some nonterminals may have, in addition to the standard choices, an entry into an auxiliary menu hierarchy. For example, the menu hierarchy described above which offers UNIX system calls is accessible from <STATEMENT> nodes. Another menu set might offer a library function calls particular to the application being developed.

The set of choices available to the user through single-key-stroke actions yields, of course, syntactically correct language constructs. But they may or may not correspond to all possible valid choices in the language. Thus, the menus offer the user a different view of the language. In effect, they provide a chance to walk through a different set of productions than a parser for the language might require. The relationship of this "*menu grammar*", the user view of the grammar from the menu choices offered, to the standard

parsing grammar is an interesting topic which will be covered in more detail in section 4.

3.2.3 Table of SCL Procedures. This table consists of names and definitions of SCL routines. There are routines which are used by the parser, actions associated with menu choices for doing context sensitive manipulations, and other supplied routines useful for manipulations in the target language. The user may add his or her own procedures to the table. For instance, one can write SCL routines to build and insert often-used subtrees or to check for personal common mistakes. ("Show the places I have set a <CONDITION> node of the form '(a = b)', since I usually want '(a == b)'.")

3.2.4 Parser Grammar. A YACC grammar is used, extended as described earlier to make it multiple-entry. Its actions construct command strings and invocations of SCL routines. The command strings construct syntax subtrees which are syntactically correct but may be semantically incorrect. Subsequent SCL routines may make context sensitive changes to correct some semantic problems.

4. Relationship of the Menu and Parsing Grammars

The independence of the template grammar, which has all possible targets for the parser, and the menu grammar, which consists of all choices available from menus, allows complete freedom in tailoring the menu system to the needs of users. These needs are quite different from the technical demands of a YACC parse grammar. In fact, there is no reason to suppose that a parse grammar is appropriate for the presentation of a language to programmers. We are only in the initial stages of investigating the human factors of such presentations, but even now our menu grammars are distinctly different from the parse grammars.

The menu grammar may be ambiguous, with the user interactively resolving ambiguity. The constraints on menu-grammar design come mainly from the need for user-friendly menus. They must be neither too large and complicated, nor too small and numerous.

The parse grammar, of course, must define the full language. The menu grammar, on the other hand, may define a language which is perhaps both smaller and informationally richer than the full language. In the menu grammar, it may prove fruitful to allow only "reasonable" grammar constructions. Constructions which require opaque grammar rules are best left to parsing, as even language experts find long derivations in a grammar a tedious way to produce a simple piece of text. In this sense the menu language might be smaller than the parse language. It is also true that the menus may present a richer view of the language by making frequent and difficult language constructs immediately accessible to the user, e.g. the UNIX function menus. Also, one can easily imagine cases where it would be useful to offer only a subset of a language in a restricted set of menus. The combination of language extension and subsetting facilities possible in the menu system may allow editors to be constructed around programming methodologies, concerns for human factors, or desirable language subsets.

5. Central Role of Command Strings

Another simplifying concept in **Syned** is a tree-encoding form intermediate between text and tree to represent programs. This form consists of the sequence of primitive tree building and navigating commands called "*command strings*" throughout this paper. Use of this intermediate form allows uniform solutions of a variety of problems, and easy communication between the parts of **Syned**. It is essential to the tree/text bimodality of **Syned** that the parser and the template table present the same internal form to the tree machine. Command strings are used to manipulate subtrees, as output from the parser, to store the tree in a file, to log the transformation of a tree in a session, to represent subtrees held in buffers, and to handle the problem of undoing arbitrary commands.

The cost of reconstructing abstract syntax trees from tree-building command strings is proportional to the length of the representative command string. Therefore, minimization of space-time costs is the minimization of the length of these strings. The algorithm which generates the command string for a subtree creates a unique minimal form.

5.1 Tree Manipulation

One example of a common tree manipulation in **Syned** is the cutting and pasting of a subtree. This is achieved by producing a command string which when executed creates the subtree, stores it in a buffer, prunes the old subtree, and executes the command string from the buffer at the new desired node. SCL routines which do simple program transformations such as changing between "do-while" loops and "while-do" loops make use of this sort of manipulation.

5.2 Session Logging

Session logging has proved helpful in debugging the editor. We also expect such histories to prove useful in evaluating the worth and frequency of use of editor features. One may, for instance, begin to collect data on the relative merits of parsing versus menu entry [Me], and perhaps relate user preference to how menus are tailored, or assess the effect of the baud-rate of the users' screen connection to their host on the use of menus.

5.3 Undoing Commands

We feel that an "undo" command to back out editing transactions is necessary. Using session logs proved clumsy, and we have adopted a method of undoing transactions which required the definition of inverses for each primitive atomic tree manipulation action (prune, graft, and so on). A composite action is undone by composing the inverses of its constituents. This method relies upon representing simple actions, compound transactions, and subtrees as the command strings necessary to repeat the actions or create the subtrees.

5.4 Writing Trees to Files

It is desirable to store programs in a form which reduces the editor's startup time. The unique minimal command string of a program is an ideal format for the external storage of programs as structured entities. It compares favorably in size to the text of a program and allows the recreation of the abstract syntax tree for a program in minimal time.

6. The Interactive Programming Environment

Syned is intended to be the gateway to an Interactive Programming Environment. This environment will contain facilities for managing program modules as structured entities. The SCL language will be used to generate tree pattern-recognizing functions and other program-manipulating commands. Some features which should be available to the programmer using the IPE are: across-module type checking, across-module data flow analysis, an interpreter/debugger, help in generating the mechanisms for testing modules and doing system testing, and a project database management system. Central to this environment is the manipulation of programs as structured entities, in our case represented as **Syned** abstract syntax trees.

Acknowledgements

Many people have contributed suggestions, ideas, or implementations to the **Syned** project. The authors particularly appreciate contributions from J.M. Chambers, M.C. Golumbic, C.M.R. Kintala, and S.C. North, all of Bell Labs, Murray Hill.

References

[Bo] Bottos, B., "A Parser Generator for AGE", Memorandum for File, 1982.

[Do] Donzeau-Gouge, V., et al., "A Structure-Oriented Program Editor: a First Step Towards Computer-Assisted Programming", Proc. Inter. Computing Symp. Antibes, 1975.

[Fe] Feiler, P.H., R. Medina-Mora, "An Incremental Programming Environment", Carnegie Mellon University Computer Science Department Report, Dec. 1980.

[Fi] Fischer, C.N., G. Johnson and J. Mauney, "An Introduction to Release 1 of Editor Allan Poe", University of Wisconsin Technical Report 453, 1981.

[Jo] Johnson, S.C., "YACC: Yet Another Compiler-Compiler", TM 78-1273-4.

[Ki] Kintala, C.M.R., B. Bottos, "Syntax Directed Editors with Text-Oriented Features", TM 82-59545-6.

[Me] Medina-Mora, R., "Syntax Directed Editing: Towards Integrated Programming Environments", Carnegie Mellon University Thesis, March, 1982.

[No] Notkin, D.S., A.N. Habermann, "Software Development Environment Issues as Related to Ada," Carnegie Mellon University Computer Science Department Report.

[Te] Teitelbaum, T., T. Reps, "The Cornell Program Synthesizer, A Syntax Directed Programming Environment", CACM **24**, 9, 563-73, September 1981.

[Wa] Waters, R C., "Program Editors Should Not Abandon Text Oriented Commands", ACM SIGPLAN Notices, **17**, 7, July 1982.

[We] Wegman, M.N., "Parsing for Structural Editors", Twenty-first Annual IEEE Symposium on Foundations of Computer Science, October 1980, 320-327.

Syned -- A Language-Based Editor for an Interactive Programming Environment

E. R. Gansner, J. R. Horgan, D. J. Moore, P. T. Surko, D. E. Swartwout

Bell Laboratories

J. H. Reppy

Cornell University

ABSTRACT

Language-based editors (LBE's) for programming languages are central components of Interactive Programming Environments (IPE's). These editors incorporate knowledge about programming languages in order to make program construction less difficult. They may also serve as gateways to the IPE, structuring programs as objects amenable to manipulation and control by the IPE. This paper describes **Syned**, a prototype LBE which runs under the UNIX* operating system. **Syned** accepts the full C language, except for some pre-processor statements, and allows free and unrestricted intermixing of parsing and menu selection in the creation of programs. It offers a powerful editor control language and a highly flexible, extendable menu system. **Syned** can be configured to accept other languages in addition to C. This paper is primarily concerned with LBE's as program editors rather than as IPE input gateways. However, some aspects of **Syned** as an IPE gateway are discussed.

---

* UNIX is a trademark of Bell Laboratories