1983 Mid-Year Report

NASA Grant NAG 1-138

SAGA: A Project to Automate the Management of Software Production Systems

Principal Investigator
Roy H. Campbell

Research Assistants
Wayne Badger
Carol Beckman
George Beshers
David Hammerslag
Peter Kirslis
Paul Richards

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.

(NASA-CR-173206) SAGA: A PROJECT TO AUTOMATE THE MANAGEMENT OF SOFTWARE PRODUCTION SYSTEMS Progress Report (Illinois Univ., Urbana-Champaign.) 149 p
HC A07/MF a01 CSCL 09B G3/61

N84-16823
Unclas
15230

ABSTRACT

This report details work in progress on the SAGA project during the first half of 1983.

1. Summary

This report describes the current work in progress for the SAGA project. The highlights of this research are:

- A Parser Independent SAGA Editor.
- Design for the Screen Editing Facilities of the Editor.
- Delivery to NASA of Release 1 of Olorin, the SAGA parser generator.
- Personal Workstation Environment research.
- Release 1 of the SAGA Symbol Table Manager.
- Delta Generation in SAGA.
- Requirements for a Proof Management System.
- Documentation for and Testing of the Cyber Pascal Make Prototype.
- A Prototype Cyber-based Slicing Facility.
- A June 1984 Demonstration Plan.
- SAGA Utility Programs.
- Summary of UNIX[1] Software Engineering Support.
- Theorem Prover Review.

2. The Language-Oriented Editor

Several changes have been made to the editor during the past half-year. A new parser was implemented, replacing the original one that had been in use. The old parser was originally a batch parser that was modified to do incremental parsing. However, its general structure was such that error handling was difficult, and partial parses not possible in an interactive environment. The new parser repairs these deficiencies. When an error is detected, the parse is automatically suspended and the editor returns to command level. The user has the option of repairing the error immediately, or of moving elsewhere in the program and possibly making a change there.

Although a parse tree containing a syntax error can only be modified from its beginning up to the point of the error, this restriction may soon be removed. Appendix A contains further details.

---

[1]UNIX is a trademark of Bell Laboratories.

2.1. Parser Independence

The new parser is encapsulated within a module of the editor. The interface between this module and the rest of the editor is designed to be independent of the parser and the form of the tables that the parser uses. Parsers from a variety of different parser generating systems may now be used to build SAGA editors. Any parser generating system may be used if the resulting parser and its tables can support the functions required in the interface. This change gives us independence from the Mystro NQLALR parser-generator, and we are in the process of adapting SAGA's Olorin, and LEX and YACC of UNIX to use with the editor, both of which have full LALR(1) parsing capability.

The component of the editor that interprets the semantics of the program being edited has also been encapsulated into a module. A set of semantic evaluation procedure calls is provided at key points before, during, and after the parse to allow semantic evaluation to occur if desired. With this added ability, we can now start studying semantic analysis using Olorin.

The parser independent component of the editor that manipulates the parse tree forms an editor "back-end"; it can be used as a module in other SAGA tools. For example, it would allow a SAGA tool to manipulate the parse tree (perhaps to optimize code) without requiring that tool to use the editor command language.

Design and implementation of a screen editor user interface is now top priority and will be underway this next half-year. Additionally, a general and extensible command language is being designed to give the editor a full set of commands, replacing the basic command interpreter that was used during testing up to this point.

The new parser has been designed for an interactive environment, and so should be more reliable than the old one. The new command language will provide enough flexibility to let us begin using the editor in our own development work. We will soon be able to start extensive editor testing, so that we can begin taking the editor from an experimental tool to a usable one.

2.2. The Parser Modules

The Mystro-based parser module for the editor was the first to be finished and has been used to debug the parser-independent SAGA editor modules.

The Olorin-based parser module for the SAGA editor is complete and testing is in progress. Details of the status of Olorin can be found in Appendix B.

The standard UNIX tools, LEX and YACC, are being modified to be used to build a parser module. The inclusion of a LEX- and YACC-based parser for the SAGA editor allows the project to build editors for many of the languages used in UNIX (including C, Berkeley Pascal, Ratfor, and the UNIX Shell.)

LEX is a tool that generates lexical analyzers from a series of pairs: a regular expression that describes a member of a lexical class and an action to be performed whenever an element of that class is seen. It produces a function written in C that reads the standard input until a lexical entity is recognized and then performs the specified action.

The changes that are required to allow the LEX generated program to be integrated with the editor are minor. The major change makes the lexical analyzer expect input text to be passed as a parameter rather than reading it from the standard input. A few changes have been made to the program LEX itself. These changes were made to increase internal constants so that the large lexical specification for ADA[2] could be processed.

The program YACC reads a BNF-like grammar augmented with embedded actions and produces a C program that functions as a shift-reduce parser for the specified language. Since the editor itself maintains the LR stack, the bulk of the code that YACC generates can be removed, leaving only the LR tables and some code for performing look-ups. These code fragments are distributed into many small routines that are called from the parser interface.

---

[2]ADA is a trademark of the United States Department of Defense.

Most of the routines called for as part of the package "parsefns.p" have been written. No work has been done on the procedures necessary for the package "semanfns.p". It is unclear how difficult a task it will be to retrieve the actions from the YACC generated code so that they could be available to the editor routines. Automation of the production of LEX and YACC based SAGA editors should be possible by the end of the year. Further details are included in Appendix C.

3. Requirements for the Screen Editing Facilities of the Editor

A screen-oriented SAGA editor should be user friendly and provide several editing functions. Editing should be accomplished in a uniform screen-edit and command mode and should be simple. The performance of the editor should be fast, predictable, and reliable.

If desired, it should be possible to use the entire terminal screen or entire window within a terminal screen to display any portion of the file being edited. The user should be able to position the cursor within the screen to any point within the file of text and insert, replace or delete text directly at the position of the cursor position; any changes should be displayed as they are made.

The user should be able to select more complex editor commands by using the command mode of the editor. The command mode displays commands temporarily at the bottom of the screen. As such commands are executed, the screen will be updated immediately soon as possible to display the changed text. Edit commands will enable the user to move, copy, replace, insert or delete arbitrary fragments of text. These fragments can be selected by cursor positions, lines, syntactic constructions or semantic constructions within the text. Other commands will include scrolling or paging the text within the display and positioning the cursor by a search for a given string of characters, syntactic construction, or semantic construction. A detailed description of the requirements for the SAGA editor is in Appendix D.

4. Delivery to NASA of Release 1 of Olorin, the SAGA Parser Generator

A copy of Olorin was released to NASA in August. George Beshers traveled to NASA and installed the system on the ICASE VAX. Work has commenced on the new release of Olorin that will include an attribute grammar scheme for encoding semantic actions.

5. Personal Workstation Environment

We have continued to work on providing a suitable environment for SAGA on a personal workstation. The UNIX port to the Cadlinc workstation is undergoing revision as we gain experience with the multi-windowed display and mouse input device. Several small demonstration programs have been written to use these facilities. A brief description of the capabilities of the current system is in Appendix E.

Work is also continuing to provide ethernet support compatible with the current VAX local network with the eventual goal of using the Newcastle Connection to provide shared file/database access to the workstations. Moving SAGA to the workstation still depends on our finding a suitable Pascal compiler that can run on the Motorola MC68000 processor; our current expectation is to use the Path Pascal compiler for the MC68000, which is currently being debugged as part of the NASA Project NSG 1471.

6. Release 1 of the SAGA Symbol Table Manager

The first implementation of the prototype SAGA symbol table manager has been completed. During the Spring, the previously released functional specification was reviewed and changed to include new functions needed to begin implementing attribute evaluation during incremental re-parsing. The implementation of this specification has been completed, and some limited testing has been performed with the symbol table manager in isolation from the SAGA editor.

The additional functionality affected some basic concepts of the symbol table. It is now possible to insert both symbol *definitions* and *references* into a context. A *definition* is what is normally understood to be a symbol table entry. A *reference*

entry in a context indicates that a given symbol within that context may actually be defined in a different context. This distinction is needed to locate all the references to a particular symbol definition so that an attribute dependence graph can be constructed. (The reference mechanism is not a program cross reference; however, this could be easily added using a reference list attribute to the reference.) Note that a symbol can be both a reference and a definition simultaneously, in which case the reference refers to itself. The reference lists are maintained correctly, even during deletion from or insertion into contexts between the reference and definition.

A context "grafting" capability was also added to the basic symbol manager design. A *grafted* context is a special context that functions as follows: A graft is a context that, instead of containing symbol definitions and references, points to an existing context. If, during a search for a symbol, the graft is encountered, the symbol manager will look in the second context for that symbol. If the search fails in that context, the graft's parent context is searched next, not the parent of the context pointed to by the graft. This allows unnamed scope blocks to be created, such as during the processing of a Pascal "`with`" statement.

Current work on the symbol manager is directed toward integrating it with the SAGA editor. Internal routines that were implemented with simple algorithms (such as linear searching) are being replaced with more efficient ones. The symbol manager is being altered to use the paged memory system of the editor instead of keeping the entire symbol table in memory. More efficient mechanisms for defining attributes and referring to them are being investigated while minimizing the restrictions on dynamic attribute creation and deletion. A complete description for the symbol table manager is currently being written as a Masters Thesis.

7. Delta Generation in SAGA

A software development environment should keep a record of every version of the software as it is being developed and maintained. A new version of a program is created whenever that program is modified. Software engineering may often require

the construction of many different versions of the same program: accommodating hardware dependencies as the program is ported to different machines or allowing several software engineers to develop independent parts of the program concurrently. In particular, it must be possible to determine the differences between versions and thereby the individual modifications that have been made to the program. These changes provide management and maintenance information.

Modifications made to one version of a program can perhaps be applied to other versions of the same program. For example, once a version of an operating system has been ported to a new machine, it is very convenient if any subsequent modifications that are made to the original program can be applied to a version of the program that has been ported. One method of recording the differences between an earlier version of a program and its subsequent versions is to store the differences between that earlier version and the newer versions as updates or deltas. If the updates that are stored are applied to the earlier version to obtain the later version, the delta scheme is called a forward delta generation scheme whereas if the updates are applied to the later versions to obtain the earlier version, the delta scheme is called a backward delta generation scheme. In either case, the deltas are often generated by comparing the source of the new version of the program with the source of the old version. It may not be very easy to compute the exact deltas that distinguish the differences between versions.

Delta generation in SAGA is accomplished *accurately*. Two programs, one to find differences between versions of a program edited with the SAGA editor, and the other to reconstruct a parse tree from one tree and a set of differences, were written in the last year. The difference program finds contiguous sections of changed lexical tokens in the parse tree of an edited program. The insert and delete commands necessary to produce one lexical token list from the other are saved in a command file. This file may then be used to rebuild the lexical token list of the version. The editor can reconstruct the entire parse tree from this list.

The SAGA editor records the changes made to a program by setting the "modify bit" of terminal (leaf) nodes in the parse tree of the program as they are being edited. The difference program uses these modify bits as an indicator of which lexical tokens have been altered. The modified bit of a leaf node is set if the node was inserted or one of its neighboring nodes was inserted or deleted. Thus, if a sequence of leaf nodes representing lexical tokens has none of its modify bits set, then that sequence of leaf nodes and the lexical tokens it represents are identical in the old and new version.

The difference finder searches the parse tree for such sequences. A delta consists of the sequence of lexical tokens represented by leaf nodes between such unchanged sequences, between the beginning of the program and the first unchanged sequence and between the last unchanged sequence and the end of the program. The delta contains the nodes that must be inserted or deleted to get from one version to the other. The beginning and end of every delta is checked to determine if it can be reduced in size, with as many nodes matched as possible. This algorithm also removes any nodes whose modified bits were only set because a neighbor changed. For example, a change may have been to replace a string of tokens with an identical string of tokens. Here, the check will eliminate the change.

Since the difference finder depends on the modify bits, these must be reset before editing of a new version commences. A copy program resets the modify bits in the parse tree. The copy program is executed to produce a new version of the program. Accurate delta information can be obtained by making new versions after every editing session. Using this scheme, it is also feasible to produce delta information during an editing session. Alternatively, if the modify bits are preserved between editing sessions, the change information accumulates. Less accurate, but more condensed delta information, can be obtained by making a new version after multiple editing sessions. We await experimental evaluation of these schemes before recommending any particular version control and delta generation strategy.

The current scheme is implemented as a backward delta generation scheme. That is, a current working copy of the parse tree is kept and deltas that represent changes to restore that version to the earlier version are stored. The advantage of storing backward deltas is that the current version can always be accessed quickly and economically. However, we envisage the scheme being extended to include both backward and forward delta generation schemes. The difference program is described further in Appendix F.

7.1. The UNDO Operation

The purpose of the undo operation is to permit the selective removal of particular modifications that have been made to a program. It is intended to make the undo operation available in the editor in conjunction with a module that can display the differences between the current version of a program and other previous versions of the same program. The user will be able to select a particular difference and remove that difference from the current program.

7.2. Archiving Differences

The deltas created by the differencing software will be stored in a source control system. We have begun the design of software that will allow us to store these deltas under the control of RCS, a source control system.

8. Requirements for a Proof Management System

In a software development environment, it is often desirable, if not necessary, to do formal proofs. These proofs are almost always difficult. This difficulty manifests itself in two ways. The proofs cannot be given to a theorem prover unless the user is willing and able to carefully guide the theorem prover along what he hopes is the right path. The user cannot do the proof by hand since this method is too error prone and is not easily verifiable.

In an attempt to make proofs easier and verifiable, work has begun on a proof management system. This system will be composed of three parts:

- A Tree Editor,
- A Proof Checker, and
- A Theorem Prover.

Using these tools together a programmer will be able to:

- More easily construct a proof,
- Verify the validity of his inferences,
- Use the theorem prover to prove those propositions that it can, and
- Check his work incrementally as the proof is being written.

Appendix G contains further details.

9. Documentation for and Testing of the Cyber Pascal Make Prototype

The Cyber Pascal Make prototype has been tested and demonstrated to work. This prototype works in conjunction with the SAGA syntax-directed editor. While the user is editing the source, the editor is verifying syntactic correctness and Make is collecting information. On exiting the editor, the user invokes Make, which takes the information produced by the editor and writes a program from the source stored in the editor files. This program is the minimal one required to produce an object that reflects changes to the source. This program is compiled by available separate compilation facilities to produce an object code module. This object code module is then integrated with the object code created from the old version of the program to produce a new object code for the modified program.

Complete documentation for this facility is currently being produced as part of a Masters Thesis. This documentation will include details of interaction with the SAGA editor, editor-independent parts, and a user's guide.

10. Prototype, Cyber based, Slicing Utility

During the Spring semester, an experimental Pascal slicing utility was constructed from the SAGA tool base. The utility summarizes the updates that a portion of a large program makes to specified variables. The summary consists of a program (slice) which assigns the same values to the selected variables as the large program except that the slice only includes computations that are required for the derivation of

the values of those variables. The slicing utility, although rudimentary, suggests future SAGA tools. Appendix H contains further details.

11. June 1984 Demonstration Plan

The June demonstration will be based on a SAGA Pascal editor. The user interface will be based on a full-screen display and have primitive windowing capability. The windowing features will also be used to support help facilities. The editor will include an extensible command language with pattern matching capabilities similar to UNIX editors.

The editor commands will allow textual insertion, replacement, copying and deletion of individual characters and provide immediate visual feedback. (For example, the user can position the cursor anywhere on the screen and insert or replace characters at that point.) In addition, the editor will allow the insertion, replacement, copying and deletion of the lexical components of the program. This facility will be supported in a language and grammar independent manner by the use of regions that are under user control and can be modified to encapsulate program fragments. These regions will be used as arguments in the various editing commands. The contents of a region may be stored in a permanent file for repeated future use in editor commands.

Structured editing will be demonstrated using the editor. That is, the top-down refinement of a program by replacing place holders in standard statements commencing with the program declaration. The editor will also support some commands that use the semantics of the edited text. These will allow the detection of variables that are declared but not used or used and not declared and will also allow the user to find the point of declaration of variables and procedures.

The editor demonstration will show incremental re-parsing of the input text, error reporting of syntax and semantic errors, semantic-oriented program text formatting, and the ability to display, save, and input invalid programs as well as syntactically correct programs. An undo facility will be demonstrated that allows the user to

select a difference between an old version of a program and the current version of the program and restore that text of the older version in the current version. The editor will be demonstrated editing a large program (such as the symbol table manager, which consists of approximately 1200 lines of Pascal code).

If possible, by that time, we will also demonstrate the make facility (this requires porting the current version to UNIX and devising a scheme to allow appropriate separate compilation using the Berkeley Pascal compiler). Various display commands may be implemented in the demonstration editor including the ability to provide several windows containing program parts, differencing windows which allow a program version to be compared with a previous program version, and windows to supply user information such as help facilities or documentation.

We will also start integration of the symbol table manager into the SAGA editor. Addition of the symbol table to the editor will allow many of the static semantic errors (like mismatched types) to be detected during program preparation. In addition, the symbol table will permit cross referencing capability within the editor. Thus, the editor should be able to locate the declaration and subsequent uses of a variable. More details can be found in Appendix I.

12. SAGA Utilities

Three programs written for the Cyber were converted to run on the VAX under Berkeley UNIX. These three programs *check* a parse tree, *compact* a parse tree and its string table, and *compare* two parse trees to determine if they are the same.

12.1. The Consistency Checker

The consistency checker takes a subtree of the parse tree and examines the fields in the nodes. It tries to determine if the fields are consistent. The checking depends on information in the nodes including pointer information and will detect abnormal cycles in the parse tree. The consistency checker can be used as a separate program to check the entire parse tree between edit sessions. It has proved a valuable tool in

developing the editor and some of the other utilities such as the compactor.

12.2. The Compactor

The compactor takes the parse tree and string table for a program which has been edited and produces another tree and string table with any unused storage space removed. Unused storage space can be created as nodes are deleted by editing commands. No attempt is made within the editor to collect garbage produced by such editing operations as this is difficult to determine, and would slow the operation of the editor. Using this utility, space can be reclaimed without having to write out the program in source code and reparse it.

12.3. The Equality Tester

The equality tester was written mainly to see that the compactor works correctly. Nodes may be renumbered, but the tree structure must be the same for the two trees to be equal. The equality checker can also be used to see if the programs that find the difference between versions and rebuild parse trees from the differences are working correctly.

Besides being converted to Berkeley UNIX Pascal, the equality checker was changed so that it can handle large programs. The Cyber version keeps the map between numberings of the nodes in an array in memory. The UNIX version keeps the array in a file. Parts of the array are paged into memory by the same routines that page parse tree nodes.

13. Summary of UNIX Software Engineering Support

A brief summary on the use of UNIX to support Software Engineering is included in Appendix J.

14. Theorem Prover Review

During the last six months we have examined several forms of theorem provers as part of a study of the requirements for a proof management system within SAGA. The theorem provers studied are two interactive theorem provers: the UT interactive theorem prover and LCF, and two batch theorem provers: Robust and VERUS. Details of the study are included in Appendix K.

15. Conclusion

The early SAGA prototypes and designs have been considerably refined and enhanced in the current research period. A basis for a first usable version of the SAGA editor and other associated tools is now well advanced and we should have the basic components of system available for demonstration early in June. A diagram of the basic components is shown in Fig. 1.

We have developed several other prototype tools which will eventually be integrated into the SAGA system. Workstation support software for the SAGA system is being developed and several facilities are now available. Requirements for the proof management system are nearly complete and we hope to build a prototype shortly.

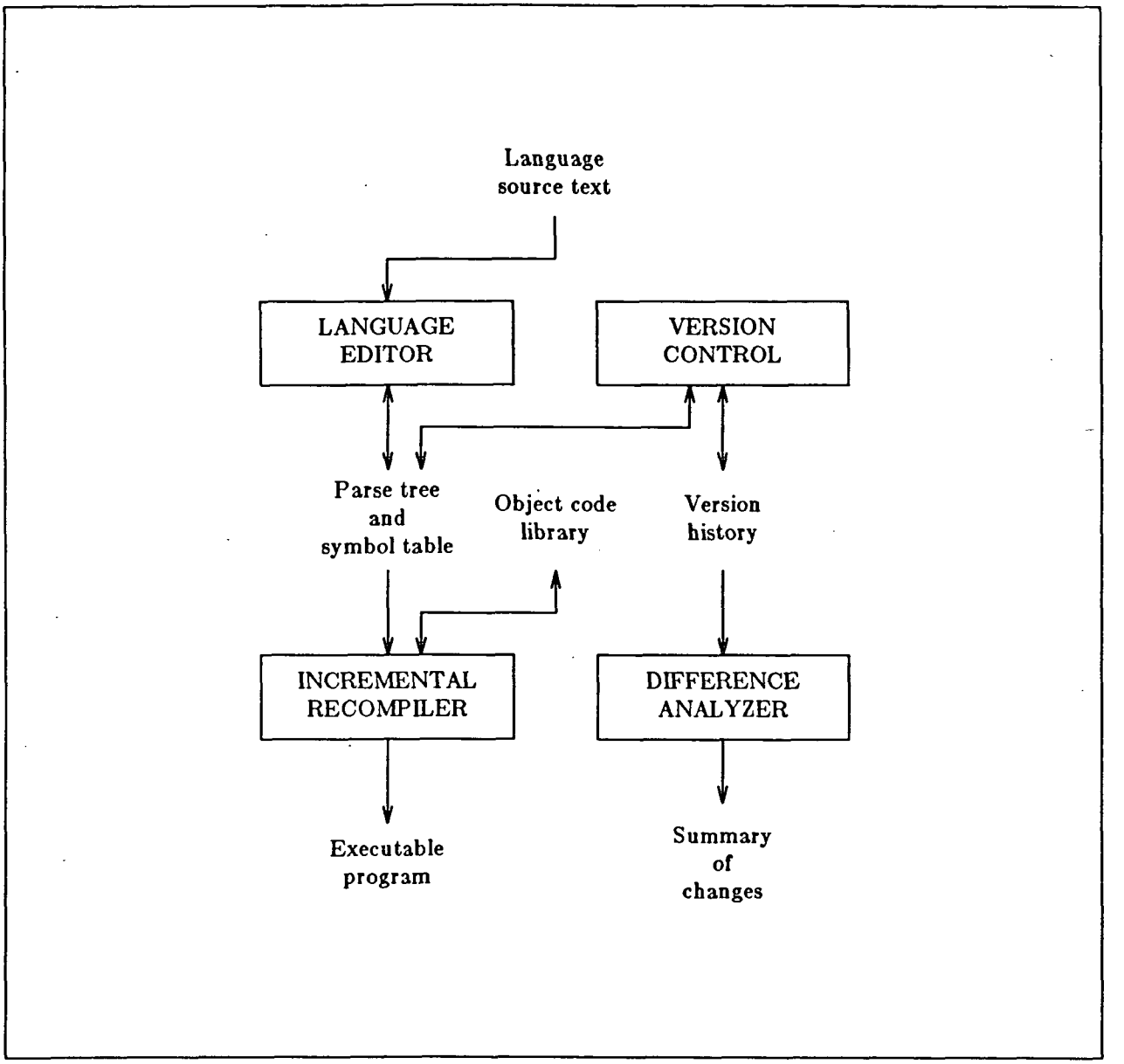


Figure 1: SAGA System Tool Interaction

The parse tree and symbol table, object code, and version history comprise the "SAGA information base". This figure shows the interaction of the various tools with the data. The arrows show the direction of data movement.

SAGA Mid-Year Report
Appendix A
The Language-Oriented Editor

Peter A. Kirslis

Department of Computer Science
University of Illinois
Urbana-Champaign
Illinois, 61801-2987

October, 1983

1. A Language-Oriented Editor

1.1. Review of Editor Capabilities

In SAGA, the language-oriented editor is used by the programmer for program development. The editor incorporates an LALR(1) parser augmented for the interactive environment with incremental parsing techniques [Ghezzi and Mandrioli, 80]. The user's program is stored internally as a parse tree and associated string table. The text string representation of the program is generated by routines which traverse a doubly-linked list of parse tree terminal nodes; no separate textual representation is kept. The user inputs his program from the keyboard in free format; no templates are used and no non-terminals appear on the screen. There is an editing cursor which moves through the program, and is always located between two terminals. The basic editing operations insert or delete a sequence of tokens beginning or ending at the cursor, or modify a sequence of tokens by copying them into a modify buffer, allowing textual modifications on this string of characters, and then tokenizing and parsing the result. All other editing operations are built up from some combination of these basic ones. At any time during the editing process, the user can ask the editor to print the set of legal tokens (the follow set) that can appear at this point in the parse; the user can also set an option which will cause the editor to automatically print the follow set whenever more input is needed from the terminal.

1.2. Incremental Reparsing

The incremental reparsing algorithm is reasonably efficient; each node in the parse tree contains all the information needed to begin a reparse from that point. There is no explicit parse stack; rather, there is a "left threading" of nodes in which each node points to the one that would be immediately below it in the stack. Each node also contains the state in which the parser was put after the node was shifted. With this available information, in order to begin a reparse at a given point, all that need be done is to find the most recently shifted node, set the stacktop variable to this

node, set the parser's state to the state found in the node, and the new parse is ready to begin. If the grammar in use contains empty productions (production rules with null right hand sides), and empty tokens are present in that part of the tree, then it may be necessary to traverse a few nodes in the tree in order to locate the most recently shifted token. When the new input is complete, a set of matching conditions determines where the newly built subtree can be grafted back into the original parse tree. The editor has been designed with an interface between the editor and parser that allows it to be used with any parser-generator program which can produce parse tables and access routines that can conform to the editor's LALR(1) style interface.

Advantages of using the editor are that the programmer gets immediate feedback about his errors, and that the parse trees generated are usable with other SAGA tools. Since the editor takes all input in textual form, it can be immediately used with pre-existing software; no conversion is needed. The editor's command language is presently an ad hoc implementation, but it does provide basic editor capabilities; we are in the process of designing a full command language whose elements will be orthogonal and extensible.

1.3. Error Handling

When a lexical error is encountered, the input string is marked at the point of the error, and the user is asked to modify the input line and try again. When a syntax error is encountered, the offending token is highlighted (or otherwise distinguished). The user has the option of repairing the syntax error immediately, or of scanning back through earlier portions of the program and possibly making a modification there (needed, for example, if a "begin" keyword was mistakenly omitted and its matching "end" just encountered).

At the moment we require the editing of any program with at least one syntax error to be limited to that portion of the program up to the first error. However, it is not clear that this is a necessary restriction. There are many cases in which a modification can be made to the program beyond the point of a syntax error, since the

incremental parsing techniques may keep the reparse sufficiently localized that the earlier section of the tree containing the error is never accessed. However, during a reparse, a reduction could be called for which refers to a syntactically incomplete section of tree because of the earlier error. If this case can always be detected in time to stop the parser, then the parse can be suspended at this point, and the editing limitation can be dropped.

1.4. Editor Generation

The editor has been designed to be reasonably language independent. To create an editor for a new language, a new grammar and associated semantic routines must be specified, but most of the editor's source code can be used as is. Figure 1 illustrates editor generation. We have built editors for the Pascal language and several abstract languages generated from grammars created by staff members at the University (and used only by them). An editor for the C language will soon be implemented, as will an editor for ADA®. The editor runs on a VAX® under Berkeley UNIX®, and an earlier version runs on a CDC Cyber 175.

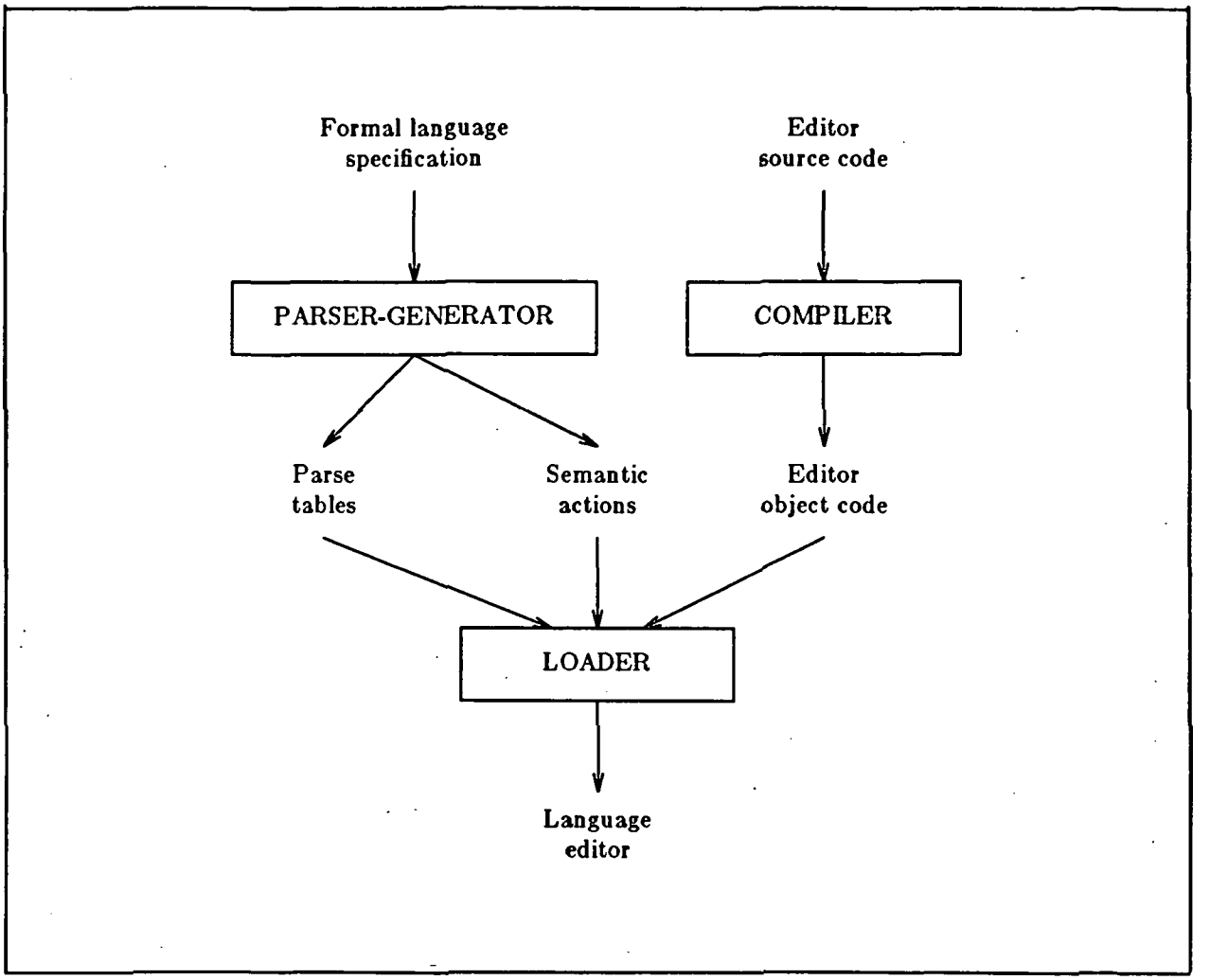


Figure 1: Editor Generation

References

[Ghezzi and Mandrioli, 80] Ghezzi, C. and D. Mandrioli, "Augmenting Parsers to Support Incrementality," Journal of the ACM, Vol. 27, No. 3, (July 1980).

SAGA Mid-Year Report
Appendix B
Olorin Update

George M. Beshers

Department of Computer Science
University of Illinois
Urbana-Champaign
Illinois, 61801-2987

October, 1983

1. Olorin Update

Since August, Olorin has progressed on several fronts. The input language for attribute grammars is almost complete and several of the implementation problems appear to be solved. The error correcting facility has been completed, tables can be produced and preliminary testing has been performed. Techniques to reduce the size of the parse tables are being developed. The interface between the current version of the Olorin system and the SAGA editor will be delivered with the new SAGA editor.

Much of the progress to develop an Olorin attribute grammars scheme in the first part of this year was a result of efforts to write attribute grammars for Pascal and for the Olorin input language. This resulted in a redesign of the grammar for attribute grammars to make the input more concise. Other work has addressed solutions to the selective update problem for large attributes. This work will be presented more completely in the year end report.

The implementation of the error correction facility is nearing completion. The mechanism being used was originally suggested by [Modry, 76]. Many of the improvements of [Schell, 79] are included. We expect that any single-token error would be corrected. Olorin now generates the additional tables needed, and they appear to be correct. The table reduction program has been extended to handle the additional information, and the error correcting parser is being coded.

The current Olorin tables waste space. This problem stems from the need to preallocate space to meet the needs of the largest language (ADA or HAL/S). To solve this problem, four additions have been made to the Berkeley Pascal Compiler. The compiler now recognizes the type **address**, the standard function **tsize**, and the standard procedures **allocate** and **deallocate**. The type **address** is a pointer to a variable of any type. The function **tsize** returns the size of a variable. The two new standard procedures, **allocate** and **deallocate**, take an address and a size in bytes, otherwise they act like **new** and **dispose**. These additions were suggested by Wirth's Modula II and conform to the specifications of the equivalent (same name) components in that

language. The advantage of these procedures is that now only the space actually used need be allocated. In particular, the error correcting information can be omitted from the editor, with considerable savings in space.

The error correcting version of Olorin should be finished by the year end report. The attribute grammar version should be done for the June demonstration.

References

Modry, John A., "Syntactic Error Recovery for LR Parsers", Technical Report UIUCDCS-R-76-833, University of Illinois, Department of Computer Science, (October 1976).

Schell, Richard M., Jr., "Methods for Constructing Parallel Compilers for use in a Multiprocessor Environment". Technical Report UIUCDCS-R-79-958, University of Illinois, Department of Computer Science, (October 1976).

SAGA Mid-Year Report

Appendix C

Using LEX and YACC with the SAGA Editor

David H. Hammerslag

Department of Computer Science

University of Illinois

Urbana-Champaign

Illinois, 61801-2987

October, 1983

1. INTRODUCTION

This paper documents the changes that were made to the source code of YACC, LEX, and the programs they generate, so that they can be used with the parser-generator independent interface to the SAGA editor as defined by Peter Kirslis. There are eight routines in the Editor/Parser Generator interface:

Routine	*Synopsis*
tokenize	Tokenize the characters in the line buffer and return a list of parse-tree nodes.
initparse	Do any work necessary before the parse begins for the first time.
legalterm	Given a state number, return a list of tokens that can legally occur.
legalnonterm	Given a state number, return a linked list of nonterminals that can legally occur.
termname	Given a token code, produce a printable representation of that token.
nontermname	Given a nonterminal code, produce a printable representation of that nonterminal.
ruleleftside	Given a rule number, return the code for the nonterminal on the left hand side of the rule.
rulelength	Given a rule number, return the number of symbols of in the right hand side of the rule.
parseaction	Given a state number and the code number of a grammar symbol, determine the appropriate parsing action to be taken.

Of these eight, all but tokenize are implemented by using pieces of the C code generated by YACC. Tokenize, the routine for lexical analysis, uses the LEX generated program, yylex, in its entirety. Because the interface is written in PASCAL and the code generated by LEX and YACC is in C, most of the routines in the interface are very small, and simply call a C routine that is actually a portion of the generated C code .

2. LEX

2.1. Changes to LEX Source

The changes to LEX source were made to increase internal constants so that the rather large lexical specification for ADA™ could be processed. The only changes occur in *sub2.c*, they are as follows:†

Original

```
--- 589,595 -----
    int s; {
        register int *p, i, j;
        int cnt, m;
!       int temp[300], k, neg[300], n;
        k = 0;
        n = 0;
        p = state[s];


--- 594,600 -----
        n = 0;
        p = state[s];
        cnt = *p++;
!       if(cnt > 300)
            error("Too many positions for one state - acompute");
        for(i=0;i<cnt;i++){
            if(name[*p] == FINAL)temp[k++] = left[*p];
```

Modified

```
*** 589,595
    int s; {
        register int *p, i, j;
        int cnt, m;
!       int temp[600], k, neg[600], n;
        k = 0;
        n = 0;
        p = state[s];


*** 594,600
        n = 0;
```

†These comparisons were generated by the UNIX utility "diff." See the UNIX Programmers Manual for an explanation of the notation in the margins.

```
        p = state[s];
        cnt = *p++;
!       if(cnt > 600)
              error("Too many positions for one state - acompute");
        for(i=0;i<cnt;i++){
              if(name[*p] == FINAL)temp[k++] = left[*p];
```

2.2. Changes to lex.yy.c

After the file *lex.yy.c* was generated by LEX, a number of changes were made to the C routines of *lex.yy.c*. The net effect of these changes is to enable yylex to deal with a buffer of text at a time.

2.2.1. Sagaio.c

The file *sagaio.c* consists of two routines, sagagetc and yywrap, and needs to be included, either textually, or by an include statement at the beginning of *lex.yy.c*. *Sagaio.c* contains:

```
sagagetc(buffer,pos)
int *pos;
char *buffer;
{
  char c;
# ifdef DEBUGIO
   printf("Calling sagagetc with pos = %ld ...",*pos);
# endif
  c=(buffer[(*pos)-1]);
# ifdef DEBUGIO
    printf("returning %c\n",c);
# endif
  (*pos)++;
  return(c);
}



yywrap()
{
 return(1);
}
```

2.2.2. Parameterization

The rest of the changes to *lex.yy.c* provide yylex the parameters necessary for buffered input. There are nine places where the code needs to be changed. As the line number of each change will vary with each different lexical specification used, the changes are shown here with some surrounding context and commentary.

```
# include "y.tab.h"
        .
        .
        .
# define output(c) putc(c,yyout)
/*
 * Give 'input' the necessary parameters */
 */
# define input(buffer,currpos) (((yytchar=yysptr>yysbuf?U(*--yysptr)\
     :sagagetc(buffer,currpos))==10?(yylineno++,yytchar):yytchar)\
     ==EOF?0:yytchar)
# define unput(c) {yytchar= (c);if(yytchar=='0)yylineno--;\
     *yysptr++=yytchar;}
# define yymore() (yymorfg=1)
extern struct yysvf yysvec[], *yybgin;
# define YYNEWLINE 10
/*
 * Here sagagetc and yywrap, which are the two parts of sagaio.c are
 * compiled separately and declared as external functions.
 */
extern sagagetc();
extern yywrap();
/*
 * yylex is given three parameters
 */
yylex(buffer,currpos,length)
char *buffer;
int *currpos;
int *length;
{
int nstr; extern int yyprevious;
/*
 * change the call to yylook
 */
while((nstr = yylook(buffer,currpos)) >= 0)
/*
```

```
 * A second statement is added to the body of the while loop
 * to return the length of the token found to the caller.
 */
{ *length=yyleng;
yyfussy: switch(nstr){
case 0:
if(yywrap()) return(0); break;
        .
        .
        .
case -1:
break;
default:
fprintf(yyout,"bad switch yylook %d",nstr);
}} return(0); }
/* end of yylex */
/*
 * yylook also needs two parameters
 */
yylook(buffer,currpos)
char *buffer;
int *currpos;
{
        .
        .
        .
                        if(yyz == 0)break;
                        if(yyz->yystoff == yycrank)break;
                        }
/*
 * modify the call to input
 */
                *yylastch++ = yych = input(buffer,currpos);
        tryagain:
# ifdef LEXDEBUG
        .
        .
        .
        if (yytext[0] == 0  /* && feof(yyin) */)
                {
                yysptr=yysbuf;
                return(0);
                }
/*
 * modify the call to input
```

```
*/
            yyprevious = yytext[0] = input(buffer,currpos);
            if (yyprevious>0)
                output(yyprevious);
        .
        .
        .
return(0);
}
        /* the following are only used in the lex library */
/*
 * Add two parameters to 'yyinput'
 */
yyinput(buffer,currpos)
char *buffer;
int *currpos;
{
/*
 * modify the call to input
 */
        return(input(buffer,currpos));
        }
yyoutput(c)
  int c; {
        output(c);
        }
yyunput(c)
   int c; {
        unput(c);
        }
```

2.3. Tokenize

The following function, 'tokenize', satisfies the requirements of the Editor/Parser Generator interface.

```
(*
 * 'tokenize' takes a buffer of characters and converts it into a
 * linked list of parse tree nodes, returning a pointer to the list
 * to the calling routine.  If a lexical error occurs, the variable
 * 'errorpos' is set to the position in the buffer of the first
 * character of the error token, and no list is returned.
 *)
```

```
function tokenize(*
        var inputbuf: charbuf;      /* buffer of chars to be tokenized */
          lastc:   cbufindex;      /* last char. in buffer. */
        var errorpos: integer      /* return: 0 or position of bad char */
      ): nodeindex*);                (* return: list of tokens or 0 *)

var
 list,t : nodeindex;
 startchar,currpos: integer;
 tokencode : integer;
 index,length : integer;

begin
 tokenize:=0;
 inputbuf[lastc+1]:= chr(10);  (* put in a newline for lex *)
 list:=0;
 currpos := 1;
 startchar := 1;
 length := 0;
 while ((startchar< lastc+1) and (tokencode <> TERROR)) do begin
      while ((inputbuf[currpos]=' ') and (currpos<lastc+1)) do
           currpos:= currpos+1;
      while ((inputbuf[startchar]=' ')  and (startchar <lastc +1)) do
           startchar:=startchar+1;
     if startchar < lastc +1 then begin
       tokencode:=yylex(inputbuf,currpos,length);
       if tokencode = TERROR then errorpos := startchar
              else begin
            index:=addstring(inputbuf,startchar,(startchar+length-1));
            t:= makenode(index,length,tokencode);
             chain(t,after,list);
              if list = 0 then tokenize:=t;
             list:=t;
           end; (* else *)

      startchar:=startchar+length;
     end; (* then *)

 end; (* while still more tokens in the buffer *)

 inputbuf[lastc+1]:=' ';  (* replace the blank for the editor *)

end; (* tokenize *)
```

2.4. Future work

Limited testing indicates that these modifications provide a suitable lexical analyzer for use with the Parser-Generator independent interface for the SAGA editor. The only work that needs to be done is to further modify the LEX source so that the necessary modifications to *lex.yy.c* are made automatically.

3. YACC

3.1. Changes to YACC Source

The changes to the YACC source occur in two files, *dextern* and *y1.c*.

3.1.1. dextern

The changes to *dextern* serve two purposes. The first change is to increase the number of allowable LALR(1) states for ADA, and the second is to define two files to be used for the generation of PASCAL code fragments for inclusion in the Editor/Parser Generator interface routines.

Original

```
--- 17,23 -----
  # ifdef HUGE
  # define ACTSIZE 12000
  # define MEMSIZE 12000
! # define NSTATES 750
  # define NTERMS 127
  # define NPROD 600
  # define NNONTERM 300


--- 120,125 -----
        /* I/O descriptors */
```

```
extern FILE * finput;           /* input file */
extern FILE * faction;          /* file for saving actions */
extern FILE *fdefine;           /* file for # defines */
extern FILE * ftable;           /* y.tab.c file */
```

Modified

```
*** 17,23
  # ifdef HUGE
  # define ACTSIZE 12000
  # define MEMSIZE 12000
! # define NSTATES 950   /* changed up from 750 for ada */
  # define NTERMS 127
  # define NPROD 600
  # define NNONTERM 300


*** 120,127
        /* I/O descriptors */

  extern FILE * finput;           /* input file */
- extern FILE * yynonterm;       /* file for nonterminals */
- extern FILE * yyterm;          /* file for terminals    */
  extern FILE * faction;       /* file for saving actions */
  extern FILE *fdefine;          /* file for # defines */
  extern FILE * ftable;          /* y.tab.c file */
```

3.1.2. y1.c

The change to *y1.c* is the addition of a call to a new procedure, writecase. This call occurs in the main line of YACC .

Original

```
--- 50,55 -----
        cpres(); /* make table of which productions yield a given nonterminal */
        cempty(); /* make a table of which nonterminals can match the empty string */
        cpfir(); /* make a table of firsts of nonterminals */
        stagen(); /* generate the states */
        output();  /* write the states and the tables */
        go2out();
```

Modified

```
*** 50,56
        cpres(); /* make table of which productions yield a given nonterminal */
        cempty(); /* make a table of which nonterminals can match the empty string */
        cpfir(); /* make a table of firsts of nonterminals */
-       writecase();
        stagen(); /* generate the states */
        output();  /* write the states and the tables */
        go2out();
```

3.1.3. wc.c

Wc.c is a new module added to the YACC source files. It consists of one function, writecase, which writes out the PASCAL procedure bodies for the interface procedures termname and nontermname.

```
# include "dextern"
writecase()
{
FILE *yynonterm;
FILE *yyterm;
int j,i,length;

    yynonterm = fopen("yynterm","w");
    fprintf(yynonterm,"begin\n\t if ((tokencode<1) or\
          (tokencode> %d ))\n",nnonter);
    fprintf(yynonterm,"\t\tthen writeln('bad token code = ',\
           tokencode:1);\n");
    fprintf(yynonterm,"\t\telse case tokencode of\n");

    NTLOOP(i){
          if( i == 0 ) continue;
```

```
                if (nontrst[i].name[0] != ' ')
                  {fprintf(yynonterm,"\t\t\t %d : begin\n",i);
                  length=strlen(nontrst[i].name);
                  for (j= 0 ;j<length;j++)
                    fprintf(yynonterm,"\t\t\t\t name[ %d ] := '%c';\
                    \n",j+1,nontrst[i].name[j]);
                  fprintf(yynonterm,"\t\t\t\t length := %d ;\n",length);
                  fprintf(yynonterm,"\t\t\t     end;\n");
                  }

                  else fprintf(yynonterm,"\t\t\t %d : writeln('%d is\
                    not a nonterminal, it is probably an action');\n",i,i);
                }

        fprintf(yynonterm,"end; (* procedure *)\n");
        fclose(yynonterm);

        yyterm = fopen("yyterm","w");
        fprintf(yyterm,"begin\n\t if ((tokencode<256 or (tokencode> %d ))\
              \n",ntokens+256);
        fprintf(yyterm,"\t\tthen writeln('bad token code = ',tokencode:1);\n");
        fprintf(yyterm,"\t\telse case tokencode of\n");

        TLOOP(i){
          if( i == 0 ) continue;
          fprintf(yyterm,"\t\t\t %d : begin\n",i+256);
          length=strlen(tokset[i].name);
          for (j= 0 ;j<length;j++)
          fprintf(yyterm,"\t\t\t\t name[ %d ] := '%c';\n",j+1,tokset[i].name[j]);
          fprintf(yyterm,"\t\t\t\t length := %d ;\n",length);
          fprintf(yyterm,"\t\t\t     end;\n");
          }

        fprintf(yyterm,"end; (* procedure *)\n");
        fclose(yyterm);
}
```

3.2. Changes to y.tab.c

The changes to the YACC generated file, *y.tab.c*, are much more drastic than the changes made to *lex.yy.c*. *Y.tab.c* was broken down into many small code fragments which were then turned into C functions to be called from the interface. There are four such files:

file	*synopsis*
yyaction.c	query the action table
yygoto.c	query the goto table
yyleft.c	return the left hand side of a rule
yyrlen.c	return the length of a rule

Also, in order to minimize the size of these files, the YACC produced tables are written out into individual files. There are eight files of tables: *yyact, yychk, yydef, yyexca, yypact, yypgo, yyr1,* and *yyr2.* These file names correspond to the names of the tables they contain.

3.2.1. yyaction.c

```
/* Modified to work with saga parsefns.
  Input params:
      state-----*int  the current state.
      token-----*int  the current token.
on action reduce ...returns 4 state = rule number.
on action shift  ...returns 2 state = new state.
on action accept ...returns 0.
on action error  ...returns 3.
*/
extern int yychar;
# define YYLAST 1893
# define YYFLAG -1000
# define YYACCEPT return(0)
# define YYABORT return(1)

/*      parser for YACC output      */

#ifdef YYDEBUG
int yydebug = 1; /* 1 for debugging */
#endif
int yychar = -1; /* current input token number */

yyaction(state,token)
int *state;
int *token;
```

```
{
# include "yyexca"
# include "yyact"
# include "yypact"
# include "yychk"
# include "yydef"

        register short yystate, yyn;
        register short *yyxi;

        yystate = *state;
        yychar = -1;

/* yynewstate (yystack all gone!) */
        yyn = yypact[yystate];

        if( yyn<= YYFLAG ) goto yydefault; /* simple state */

        if( yychar<0 ) if( (yychar=(*token))<0 ) yychar=0;
        if( (yyn += yychar)<0 || yyn >= YYLAST ) goto yydefault;

        if( yychk[ yyn=yyact[ yyn ] ] == yychar ){ /* valid shift */
                yychar = -1;
                yystate = yyn;
                *state = yystate;
                return(2);
                }

yydefault:
        /* default state action */

        if( (yyn=yydef[yystate]) == -2 ) {
                if( yychar<0 ) if( (yychar=(*token))<0 ) yychar = 0;
                /* look through exception table */

                for( yyxi=yyexca; (*yyxi!= (-1)) || (yyxi[1]!=yystate) ;\
                        yyxi += 2 ) ; /* VOID */

                while( *(yyxi+=2)
>= 0 ){
                        if( *yyxi == yychar ) break;
                        }
                if( (yyn = yyxi[1]) < 0 ) return(0);   /* accept */
                }
```

```
        if( yyn == 0 ) return(3);

        /* reduction by production yyn */

#ifdef YYDEBUG
                if( yydebug ) printf("reduce %d0,yyn);
#endif
                *state = yyn;
                return(4);
        }
```

3.2.2. yygoto

```
/* part of parse action
  given a rule number that a reduction was made by, and
  a state this returns the state to GOTO
*/
# define YYLAST 1893
yygoto(state,token)
int *state;
int *token;
{
# include "yyact"
# include "yypgo"
# include "yyr1"
# include "yychk"
int yyn;
int yyj;
 /* consult goto table to find next state */
 yyj = yypgo[*token] + (*state) + 1;
 if( yyj>=YYLAST || yychk[ (*state) = yyact[yyj] ] != -(*token))\
        (*state) = yyact[yypgo[*token]];

  return(*state);
}
```

3.2.3. yyrlen.c

```
yyrlen(rule)
int *rule;
{
# include "yyr2"
```

```
return(yyr2[*rule]);
}
```

3.2.4. yyleft.c

```
yyleft(rule)
int *rule;
{
# include "yyr1"
return(yyr1[*rule]);
}
```

3.3. Parsefns.p

The bodies of the functions termname and nontermname will be in the files *yyterm* and *yynonterm* which are automatically generated by the modified YACC. The user needs to textually insert those files into *parsefns.p*. The remainder of the functions included in *parsefns.p* are presented below. Note that at this time certain of the functions (legalterm and legalnonterm) have not been written.

```
function initparser(*: boolean*);
begin
 initialstate:=0;
 finalstate:= 1;
 maxrules := 414;
 maxstates := 809;
 maxtokens := 352;
 initparser:=true;
end;
```

```
procedure termname(*
            tokencode: tokenrange;       /* a terminal token code */
        var name:          charbuf;      /* return: a printable name */
        var length:         charbufindex /* return: name[1 .. length] */
    *);
    (* insert yyterm here *)
```

```
procedure nontermname(*
            tokencode: tokenrange;       /* a terminal token code */
          var name:          charbuf;       /* return: a printable name */
          var length:         charbufindex /* return: name[1 .. length] */
      *);
      (* insert yynterm here *)


function ruleleftside(*
            rulenumber: rulerange   /* a production rule number */
      ): tokenrange*);
begin
  ruleleftside:=yyleft(rulenumber);(* return: the token on the left side *)
end;



function rulelength(*
            rulenumber: rulerange;       /* a production rule number */
      ): integer*);                    (* return: the number of tokens on
                                    the right hand side of the rule *)
   begin (* rulelength *)
    rulelength:=yyrlen(rulenumber);
   end (* rulelength *);



procedure parseaction(*
            symbol:  tokenrange;    /* next terminal or nonterminal token */
            cstate:  staterange;    /* current parse state */
          var action:  actionkind;    /* the action to be taken */
          var result:  short*);       (* return: the new parse action *)
   var temp: integer;
   begin
    if ((symbol >= 0) and (symbol < 184))
      then begin
        action := SHIFT;
        result:= yygoto(cstate,symbol);
      end

      else if ((symbol >= 257) and (symbol <= 352))
        then begin
          temp := yyaction(cstate,symbol);
          case temp of
            0: begin
```

```
                    result:=0;
                   action:=ACCEPT;
                end;
              1: begin
                    result:=-1;
                    action:=ERROR;
                 end;
              2: begin
                    result:=cstate;
                    action:=SHIFT;
                 end;
              3: begin
                    result:=-1;
                    action:=ERROR;
                 end;
              4: begin
                    result:=cstate;
                    action:=REDUCE;
                 end;
           end; (* case *)
        end  (*then *)

        else begin
          action := BADSYMBOL;
             result:= -1;
         end;
    end;  (* parse action *)
```

3.4. Future Work

There are several things that need to be done to complete this work. The missing procedures, legalterm and legalnonterm, need to be written, a preprocessor needs to be written for YACC input, and much of the work needs to be automated.

3.4.1. legalterm and nonlegalterm

Earlier problems with these two procedures have been overcome, and the procedures are currently being implemented. They will not generate true follow sets, but rather a subset consisting of those symbols on which a valid shift or reduction is

known, that is the follow set minus those elements that are legal but are acted on only by default reductions. (The editor then builds the true follow set by combining this information with its parse stack context.)

3.4.2. YACC Preprocessor

Part of the interface calls for retrieving printable forms of terminals and nonterminals of the grammar. As noted above, changes have been made to YACC so that it will make printable forms of the grammar symbols. Because of the inner workings of LEX and YACC it is not possible to retain a nice form for the terminal symbols. Therefore a preprocessor needs to be written for YACC. This filter will take YACC definition lines of the form,

%token TPLUS(+) TSTAR(*) BEGIN

and strip off the parenthesised strings, while remembering that TPLUS is to be printed as "+" and TSTAR as "*." The input can then be submitted to YACC in the usual manner. Filtering the input will put one extra condition (aside from those imposed by YACC) on the user. For the filter to be effective, each terminal symbol must appear in a "%token" definition.

3.4.3. Automation

As was the case with changes to the code generated by LEX, most of the changes to the code generated by YACC were made by hand. Clearly most, if not all, of the required modifications could be and should be generated automatically. This includes the C routines and the separation of the tables into separate files. Further, many of the constants in the programs were hard-coded. Instead, they should be generated by YACC.

4. SUMMARY

Most of the routines called for by the Editor/Parser Generator interface have been written and given some non-rigorous testing. Some early problems with the interface have been cleared up, and the remaining work is in progress. This work includes: writing the interface routines legalterm and legalnonterm, writing the YACC preprocessor, and automating the generation process so that the step from YACC and LEX input to a working editor needs as little user intervention as possible.

SAGA Mid-year Report

Appendix D

Language-Oriented Editor Requirements

Peter A. Kirslis

George M. Beshers

Department of Computer Science

University of Illinois

Urbana-Champaign

Illinois, 61801-2987

October, 1983

Language-Oriented Editor Requirements

An editor should provide a comfortable environment in which to edit a language. It should allow easy specification of each section to be edited, and provide a syntax that is uniform for all commands. A language-oriented editor must check the syntax of the language as it is being edited, and also perform limited semantic checking. The languages that can be edited should not be limited only to programming languages, but any language that can be specified with an LALR(1) grammar; thus, the editor is more widely applicable, and allows users to edit a number of languages while only learning one set of basic commands.

1. User Interface

The user interface determines how the user converses with the editor. It should be clear and consistent. There should be a small set of basic commands that provide enough power so that a new user can start editing quickly. Additionally, there should be advanced commands that can be learned later to perform more complex editing operations. On-line help should be available to provide immediate answers to any difficulties the user may encounter.

An extensible command set should be considered, in which the user can define new commands as sequences of already existing commands. A command macro facility with parameter passing is fairly easy to implement, and provides a degree of customization to the user. Another advantage is that more complex commands can be built on an experimental basis to test their usefulness.

Since the editor is maintaining a parse tree during the editing session, basic tree manipulation commands can and should be provided; likewise, these can be included in user-defined command macros to allow easy creation and testing of experimental structure-oriented commands.

The editor should not replace commands found in text editors, but should augment those commands with structure-oriented ones.

The command set should be orthogonal and extensible, and its syntax should be uniform. Orthogonality means that all commands complement one another, and that arguments can be specified in the same way regardless of the command. Extensibility permits commands to be defined for each language, improving the editor's usefulness with other languages. Language-independent commands must be the same among languages, and language-dependent commands must conform in syntax to the language-independent ones. All the above must be supported by adequate help facilities.

A higher-level user interface allows additional editor commands to be added which specify operations more abstractly and at a level closer to the problem; e.g. commands allowing reference to structural blocks in the language being edited in addition to characters and lines.

2. Screen Editing

A screen editor provides a much more suitable user interface for development than a line editor does. Changes made to the text are immediately displayed on the screen. The editor reduces the user's need to keep a paper copy handy, since he no longer has to mark it up to keep track of the changes just made.

A multiple window display should be considered, since being able to display two separate sections of a program at once (e.g., declarations and statements) often improves editing speed and reduces errors, since the additional information displayed on the screen need not be remembered. These windows should be creatable, modifyable, and removable independently of one another. If the windows may be overlapped, then several contexts can be kept handy while only looking at one or two at a time. If possible, full use should be made of all the lines on the screen even when in multiple window mode. Some workstations also provide a mouse for graphical input; if one is available, an editor should try to make use of it to move the cursor about a window in which an editing command can act. An example of mouse use is marking a beginning and ending line on the screen and then pressing a delete function key to remove those

lines from the file (and the display).

A screen editor can also display language errors more effectively than a non-screen editor; the display can separately mark lexical, syntactic, and semantic errors, and separate windows can be used to display more lengthy error messages.

3. Language Independence

The editor should be as language independent as possible, with an automated editor generation facility in which the input data file contains all the language dependent information. This way, the bulk of the editor code need not be altered when switching to a new language.

4. Parser

The parser and parser generating tools should be correct (recognize the specified language, and no extraneous sentences.) Syntactic and semantic checking with immediate error notification must be provided. A data structure should exist that contains all state information about a parse that is underway. This structure permits partial parses per call, and also allows the parsing of different parse trees. Nodes should be re-used in the new tree whenever they are unchanged, to minimize copying and re-linking.

A parsing method must be chosen: a recursive descent LL(1) parser or a table-driven LALR(1) parser. The former method can be implemented by menus of templates, from which the user chooses the constructs to be added; this method prevents syntax errors, but at the expense of flexibility. The latter method allows free-form input of data from the terminal, and is much more flexible, but at the expense of more complex code needed to handle syntax errors. The SAGA editor uses LALR(1) parsing.

Multi-line comments of unlimited length should be allowed.. (They can be implemented as a sequence of bounded one line strings.) There should also be a way to search for sections of comments and modify them easily.

5. Semantics

The semantic evaluation mechanism must be sufficiently flexible to support arbitrary structures on each tree node. The semantic implementation should permit the user to code simple attributes directly, and guarantee their maintenance within the parse tree. It should also permit the user to code large structured attributes which are selectively updated as the tree is modified.

6. Performance

We envision the SAGA editor user using a workstation with a great deal of dedicated processing power. The hardware must have 32 bit addressing, a megabyte of main memory, and large amounts of disk space. The processor itself must be as fast as an M68000 or VAX 750.

The screen management routines should perform selective update, only rewriting those portions which have actually changed since the display was first put on the terminal screen.

Incremental re-parsing techniques should be used so that a minimum of work needs to be done to the parse tree when a change is made. The parser should operate in O(nodes added + nodes deleted) time. It should not be necessary to re-parse the program from the beginning up to the change, or to re-parse every token all the way to the end. Searching the symbol table should take O((nesting level) * log(number of symbols)) time.

The semantic evaluator should not do unnecessary re-evaluation. In particular, small changes should not cause widespread propagation if such propagation is not inherent in the language definition. An important example is the adding or deleting of a statement in a procedure; this should not cause the re-evaluation of most of the attributes in the remainder of the procedure.

The responsiveness of the SAGA editor should be fast enough to make SAGA a viable alternative to conventional software tools. Users should not give up and return

to simple textual software tools because the editor keeps them waiting too long.

7. Modularity

The implementation of the editor should be as modular as possible so that individual components can be more easily debugged, and so that modules can be re-used as parts of other programs.

The program/data interface should be uniform so that the data produced by the editor can be used by the remaining SAGA tools. The data side of the interface contains the parse tree, string/symbol table, object library of compiled routines from the parse tree, source history, and any other data structures needed to be kept between program sessions. A uniform way of accessing data of different types (structure) is desirable, although we may need a set of routines for each type of data.

8. Conclusion

The requirements discussed in this appendix provide guidelines along which to develop a language-oriented editor. Attention to them should hopefully aid the design of such editors.

SAGA Mid-Year Report

Appendix E

The CADLINC Workstation Window System

Paul Richards

Department of Computer Science

University of Illinois

Urbana-Champaign

Illinois, 61801

October, 1983

THE CADLINC WORKSTATION WINDOW SYSTEM

Personal workstations provide fast, economic computer support for many activities including software production. A key concern in providing support for such activities is an effective user interface. This user interface may be computationally expensive; however, the availability of cheap processing power makes it practical to use such an approach in a workstation. This document describes components of a user interface for the SAGA software development project. The components include a window manager, pointing device for use in menu selection, and Ethernet support.

1. Workstation Description

The CADLINC is a MC68000-based workstation with 750kb of memory, an 80Mb hard disk, a 1024×800 bit-mapped raster display, a keyboard with attached mouse and a 10 MHz Ethernet controller. Two serial ports provide serial communication with the departmental VAXes for UUCP. The hardware provides adequate processing power to use the workstation as a stand-alone programming development system.

Version 7 UNIX® has been ported to the CADLINC, with some additional extensions taken from the Berkeley VAX® implementation of UNIX. These extensions include a file system that performs operations in the "correct" order so that a crash in the middle of a file operation will not irrevokably damage the file system, and an improved terminal device driver that makes it easier to manage multiple processes from a single terminal. Some low level system support for the 10 MHz Ethernet has been developed, with current work attempting to provide the protocol support needed to communicate with the departmental VAX network.

In addition to these extensions, a windowing display system has been written to support the use of the raster display as the console terminal. This windowing software will now be described in further detail.

2. The Multi-Window Display System

The windowing system is an adaptation of software developed at MIT for another workstation project called "NUnix". The MIT software supplemented the standard Version 7 terminal device driver with a window device driver that provided "virtual" terminal support on a bit-mapped screen. Additional support was provided for an optional keyboard that could be associated with any of the virtual displays. The code was organized in such a way that it was relatively easy to implement the core routines that needed to drive the CADLINC hardware, and interface it with our implementation of UNIX.

To a user process, these "windows" look exactly like standard terminal devices, including most of the standard UNIX device control (ioctl) functions. The initial state of the system after startup uses the entire display as the system console. As such, the window system is indistinguishable from a standard terminal connected with the processor. The driver for the window system can emulate the Heathkit/Zenith H-19 terminal and control codes, so that programs such as the *vi* text editor that use cursor motion can still be used with the console.

However, by issuing the appropriate device control function calls on the console, additional *sub-windows* can be created. A *sub-window* is rectangular area on the bit-mapped display that behaves as another terminal device, independently of what activity occurs on the remainder of the display. Textual output written to that device produces a display of characters on the rectangular region. The boundaries of the region are treated as the edges of the terminal. If a line of text is written that is too long for the area reserved to the sub-window, the characters beyond the right edge of the window are not displayed. If several lines of text are written in the area, filling the entire region, additional output can cause the region to scroll the previous text upward or wrap around the sub-window and display the new text at the top line, depending on options selected for the window.

Several sub-windows can co-exist simultaneously on the raster display, with the window device driver sorting out requests for each display. When a sub-window is created, a UNIX "special device file" is created for that window. By referring to this file, processes other than the window's creator can access the window. Using this capability, a process can open several windows and initiate other processes that are attached to the windows. The MIT window manager performs exactly this function; some illustrations of typical use of the window manager are described later.

Up to this point, only output operations on sub-windows have been described. The windowing system also supports sharing a keyboard among the current windows. One window is selected as the current "keyboard" window, which means that everything entered at the keyboard is put into the input queue for that particular window. Processes that request input from windows that are not the keyboard window are suspended until that window becomes the keyboard window, and input data is entered at the keyboard. Any process can request that a particular window become the current keyboard window; hence, the processes that make such requests should have some mutual agreement as how to share this device. The answer may be to use a manager process such as the window manager to handle this facet of the windowing system.

An additional capability of the windowing system is that it allows the sub-windows to overlap, with windows partially or completely obscuring other sub-windows. The window device driver knows which windows are fully exposed and which are not. The handling of output to a covered window depends upon an option selected for that window. One choice is to have the process suspended when it attempts output to a covered window, until the window is uncovered (presumably by the action of some other process). A second option is to have the window device driver reserve enough memory to store the entire window's text, in an area called the window's "save buffer." All output written to such a window is simultaneously stored in the save buffer. If the window is covered and then re-exposed, the save buffer is used to reconstruct the text on the window as it appeared before being covered. If,

while the window is covered, some text is written on that window, the process is not suspended. Instead, the output from the program changes the save buffer in the same manner in which it would have made changes to the window. Then, when the window is exposed, the new text is displayed on the window.

A process can request that the device driver notify it when the state of a window that the process has open changes. This notification is accomplished via the UNIX *signal* mechanism, with a new type of signal *SIGWIND*. A process receives a *SIGWIND* interrupt whenever one of the windows becomes covered or uncovered because of action of this process or another. In this way, a program that uses a window for output can be told when it needs to refresh the window, e.g. re-display whatever was there before the window was covered.

A small collection of graphics primitives are also available for use with the windowed system. A line drawing primitive (with clipping at window borders), and a version of the BitBlt raster operation are available. Currently, the raster display memory is accessible to every process' virtual address space, and the graphics routines directly manipulate this memory. However, because of unanticipated interference between several processes (and the kernel) simultaneously accessing this memory, such operations will have to be moved into the device driver. This is currently under development.

3. The Mouse Pointing Device

The NUnix windowing system also implemented a device driver for a mouse pointing device. However, most of the code that came with the windowing system was machine dependent on the NUnix machine. A new implementation for the CADLINC was written that fulfilled the needs of the MIT code.

The mouse is an integral part of the CADLINC system keyboard. This keyboard produces output for both keystrokes and mouse data. The serial line device driver must distinguish which source generated data that came from the keyboard system and route it to the appropriate higher level routines. The mouse generates data on one

of two conditions: either the mouse has been moved since the last data was generated, or one of the buttons on the mouse has changed position (either depressed or released). An additional UNIX signal *(SIGMOUS)* is available to notify processes that new data is available from the mouse.

The mouse device driver keeps a "current *x*" and a "current *y*" location for the mouse, and the state information for each of the three buttons on the mouse. This data is retrievable by either a UNIX *read* operation on the mouse device, or an *ioctl* operation on an open window. If the mouse is opened as a file, the process opening the mouse is given exclusive access to it, which means that no other process can read the mouse. However, if the mouse is accessed via the window system, then all processes that are using the current keyboard window are given access to the mouse data, and others (accessing windows that are not the keyboard window) are suspended until their window becomes the keyboard window.

It is customary with mouse systems to use an additional cursor on the screen to show the current location of the mouse. Under the current system, it is the user processes' responsibility to display and maintain this cursor; the kernel only maintains the mouse's position. This seems to be the biggest shortcoming of the current implementation, since the cursor tends to jump and skip if the processing load on the workstation is high. Alternative methods of managing the mouse cursor are under investigation.

4. An Example Session with the Window Manager

At the end of this report are several examples of the use of the window manager. This section provides a narrative of what is being done with each example.

Figure 1 This is how the display looks when running as a standard terminal. The reverse video stripe at the top is called the window's *label*, and indicates the device name for that window. At this point we are about to invoke the MIT window manager "wmgr."

Figure 2 This is the initial window manager display. The top of the screen is a *menu*, which is a collection of options that can be selected by moving the

mouse to position the cursor over a selection, and depressing a button on the mouse. The cursor at this time is the inverted circle over the *create* option, which is what we are about to do.

Figure 3 By depressing one of the mouse buttons, the *create* option has been selected. The menu item is highlighted by reverse video, and the manager prompts the user for the name of the new window (upper left corner).

Figure 4 Now the window manager wants the user to select which options will be selected for this window. The choices are selected again by moving the mouse to position the cursor over an item and depressing the left button, which toggles the option state. This window will have the following options: emulate an H19 terminal, scroll from the bottom instead of wraping around, and allocate a save buffer for this window. Depressing the right button ends the option selection.

Figure 5 This menu is used to set the physical dimensions of the window. There are two ways to accomplish this: the absolute pixel dimensions of the window can be entered (which is what the *Create-Window* device control function requires from the user process), or the mouse can be used to point to where the window should go. The mouse is handled by the window manager, which tracks the mouse location and retrieves the x and y locations of the corners of the window. In this menu, we have selected the mouse to point to the location of the new window. The cross-hair cursor shows where the upper left hand corner of the window is to be placed.

Figure 6 Once the upper left hand corner has been fixed (by depressing and releasing a mouse button), moving the mouse will "open" the window, which means that the dashed lines indicate where the window will reside.

Figure 7 Now the window's location and dimensions have been selected, the window manager provides one other menu: the program menu. This menu allows the user to create the window and leave it alone, or to create it and start a program running with the window as the process's controlling terminal. In this menu, the user has selected to start a *csh* process running in the window.

Figure 8 Now the window manager has issued the actual *create-window* call to the window driver, and has started the *csh* process going. The label "**Editor**" is what we originally had specified in Figure 3. However, the current keyboard window is still the window manager's control menu. In order to use the new window, it must be selected as the keyboard window, as the user is about to do in this frame.

Figure 9 Asking to change the keyboard window brings up the list of available windows that are eligable to become keyboard windows. Here the user selects the **Editor** window.

Figure 10 Having attached the keyboard to the window, the user can start to interact with processes attached to it. He is about to start a *vi* editing session in the **Editor** window.

Figure 11 Here the user has entered a simple program using *vi*. The *curses* and *termcap* terminal control libraries have been modified to query the window device driver for the dimensions of the window at terminal initialzation time, so *vi* knows the dimensions of the window.

Figure 12 Now that the program has been entered, the user wants to create a second window to do the compilation and testing. Here he is selecting the location of the second window.

Figure 13 Having started a second *csh* in the new window, the user tries to compile and load the program. In this case there is an undefined symbol in his program, which indicates that the name of the arc-cosine routine is not "arccos."

Figure 14 The user intends to look for the correct name of the subroutine. To do this without disturbing the error message (so he can go back to it quickly), he opens another window.

Figure 15 In order to find the correct name of the routine, the user searches through the header file for the math library. He could have just as easily started the *man* program to bring up an online UNIX Programmers Manual page, or in the case of SAGA, opened another edit window to look at the specifications for the math library.

Figure 16 The user has changed his program, and switches back to the **Editor** window. Setting the keyboard to the **Compile** window causes it to overlay the **WhatzItCalled** window. The text is restored from the save buffer. Eventually the program is loaded with the correct library and is executed.

It was not necessary for the user to re-start programs in order to locate the information he needed at any point. Because the windows retain their information while they are shuffled (like pieces of paper), the time to switch contexts is significantly reduced. The remaining examples show other facets of the window system:

Figure 17 Here another window is opened to demonstrate the graphics library. The program *sun* is a UNIX *plot* filter that knows about the window in which it is run and scales the display accordingly, as can be seen in the next figure.

Figure 18 The display of graphics data displayed by the program *sun*.

Figure 19 In this figure, a covered window is exposed and selected as the keyboard. A directory listing of the window device directory shows the special files

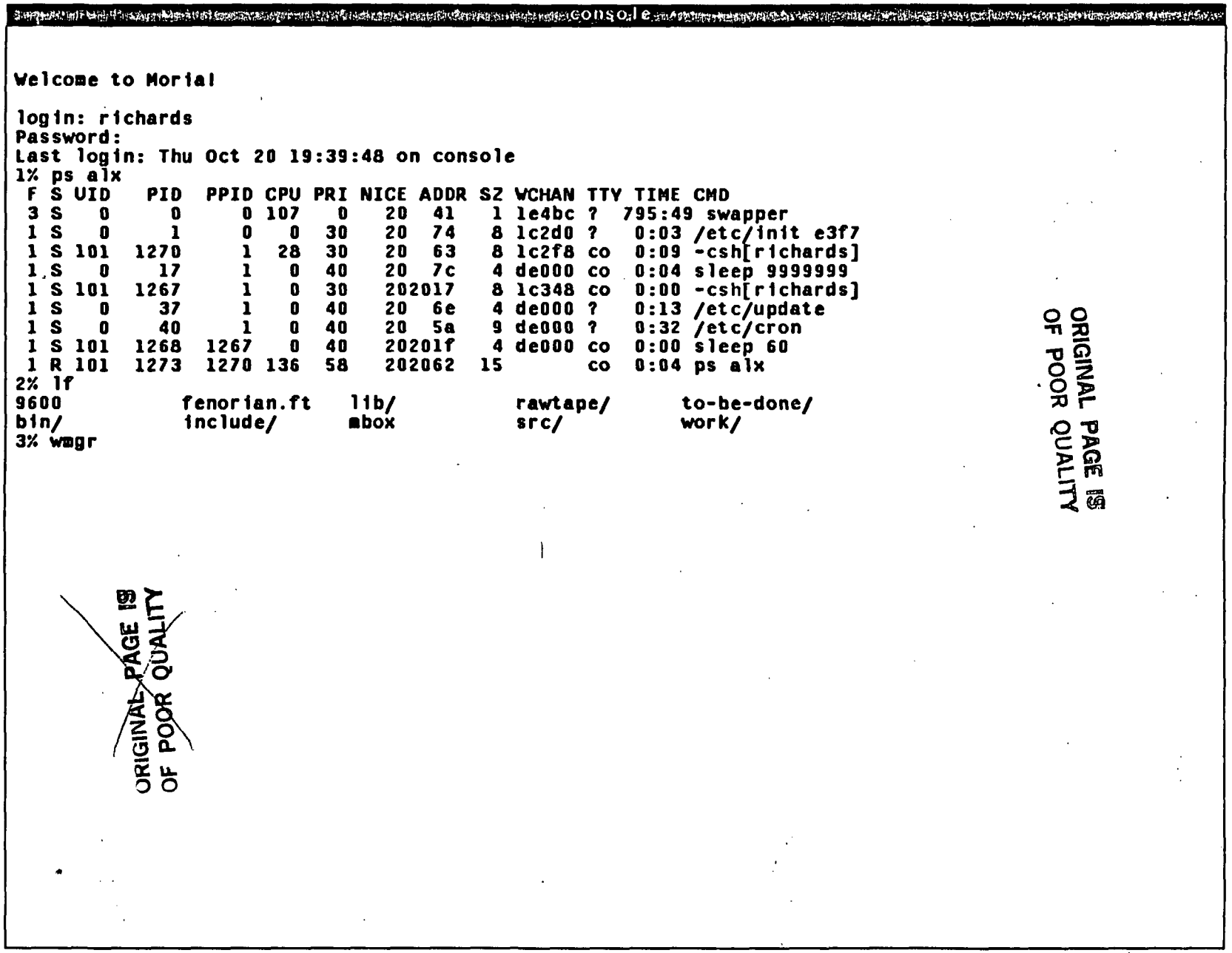
associated with the different windows currently available.

Figure 20 Accessing one window from a process not initially bound to that window is relatively easy as can be seen by the *echo* command in the **Compile** window accessing the **Graphics** window.

5. Future Directions

There are two components of the window system that are under improvement. The first is to improve the reliability, further reduce the differences between standard terminals and the window system, and provide improvements in the graphics part of the window system. Some race conditions have been found in the UNIX interface to the terminal handler. New algorithms for saving and restoring the hidden part of covered windows have been published and should be included.

The other area of improvement is to produce a programming interface to the window system that can be mapped to systems other than the CADLINC. The SAGA editor is not going to be designed to run only under this window system – it also must run with standard terminals. The document on the Maryland Window System [Torek, 1983] describes such a general interface. We are considering using this interface or a derivative of it for use within the SAGA environment.



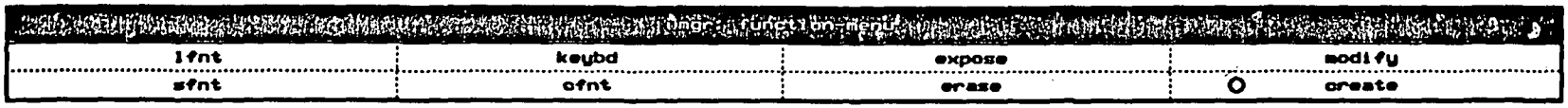


Figure 2

Figure 3

Figure 4

img: parameter menu				
nrow=0	color	ulcx=0	ulcy=0	[mouse]
ncol=0	mode=STORE	width=0	height=0	QUIT

Figure 5

umgr: parameter-menu				
nrow=13	color	ulcx=0	ulcy=361	mouse
ncol=51	mode=STORE	width=539	height=250	QUIT

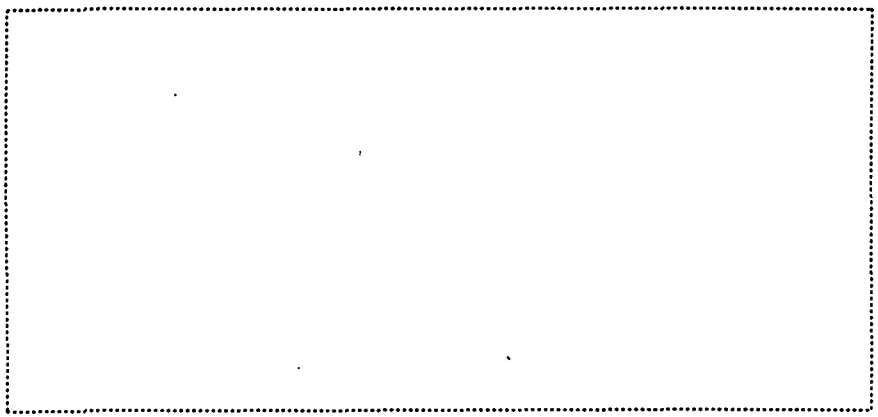
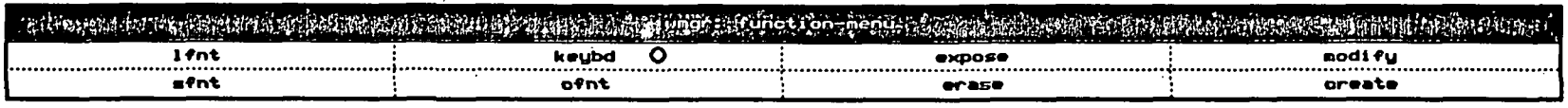


Figure 6

umgr:: program menu			
/bin/sh	/bin/ee	arglist=	[create]
/oda/bin/draw	/bin/csh	[create-expose]	QUIT

Figure 7



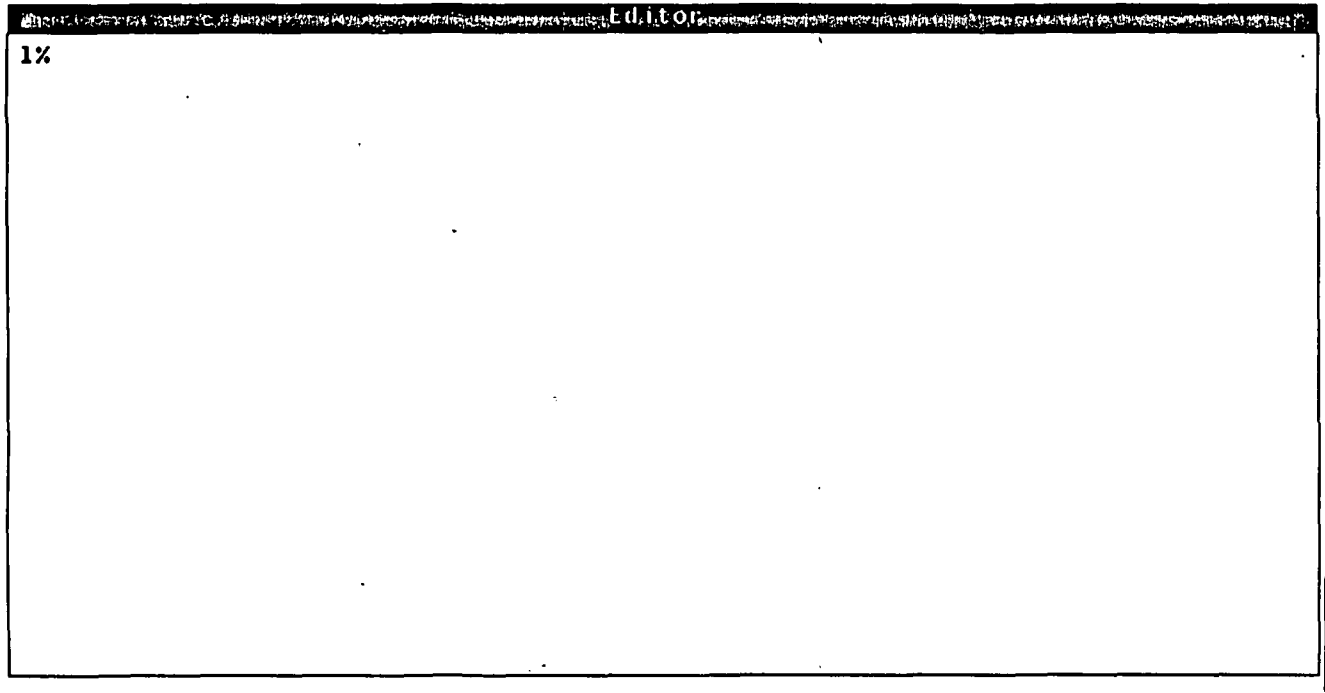
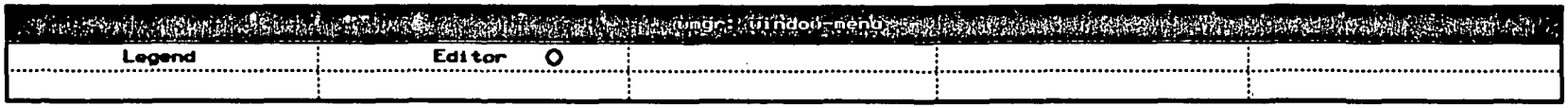


Figure 8



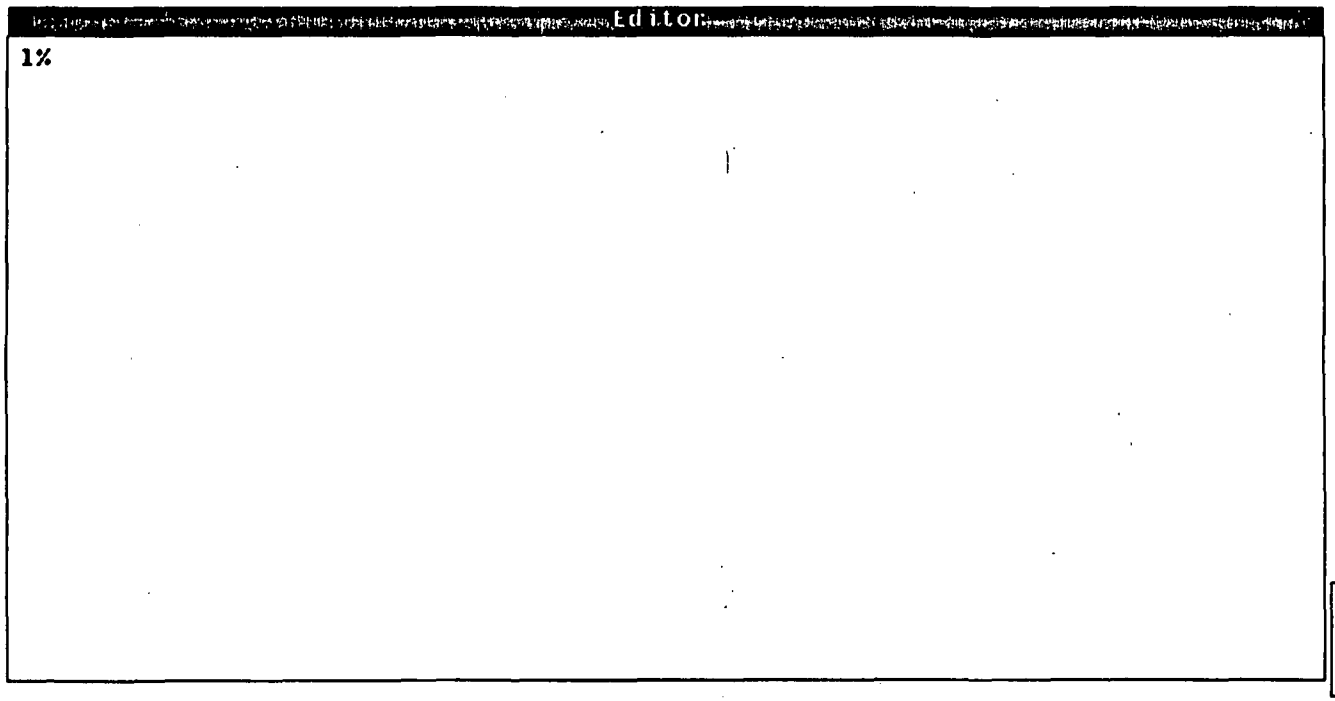
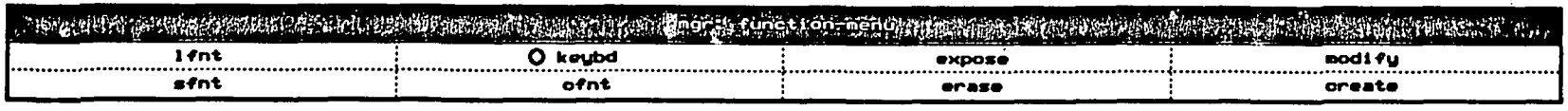


Figure 9



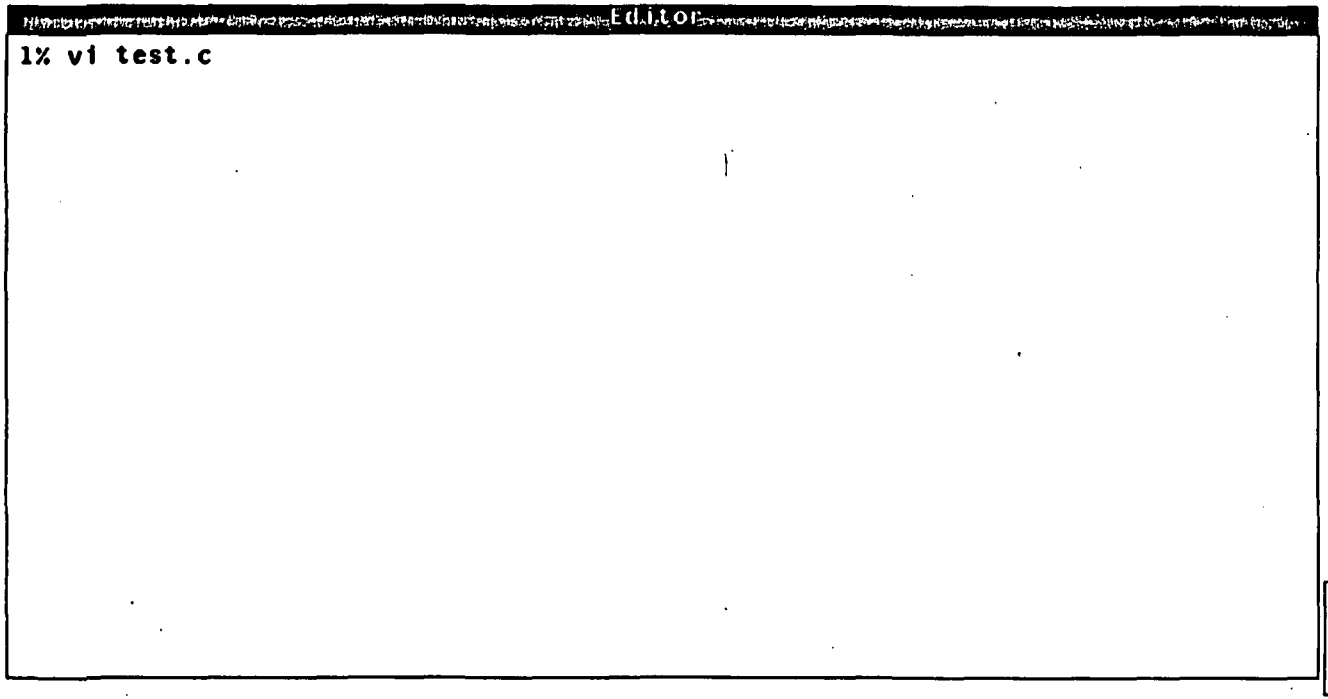
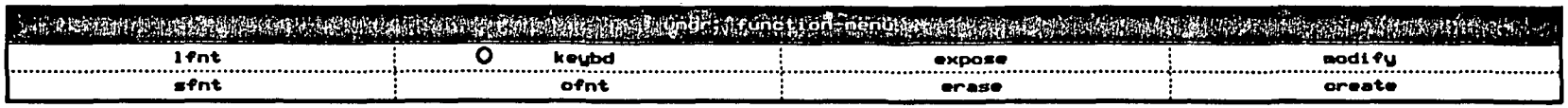


Figure 10



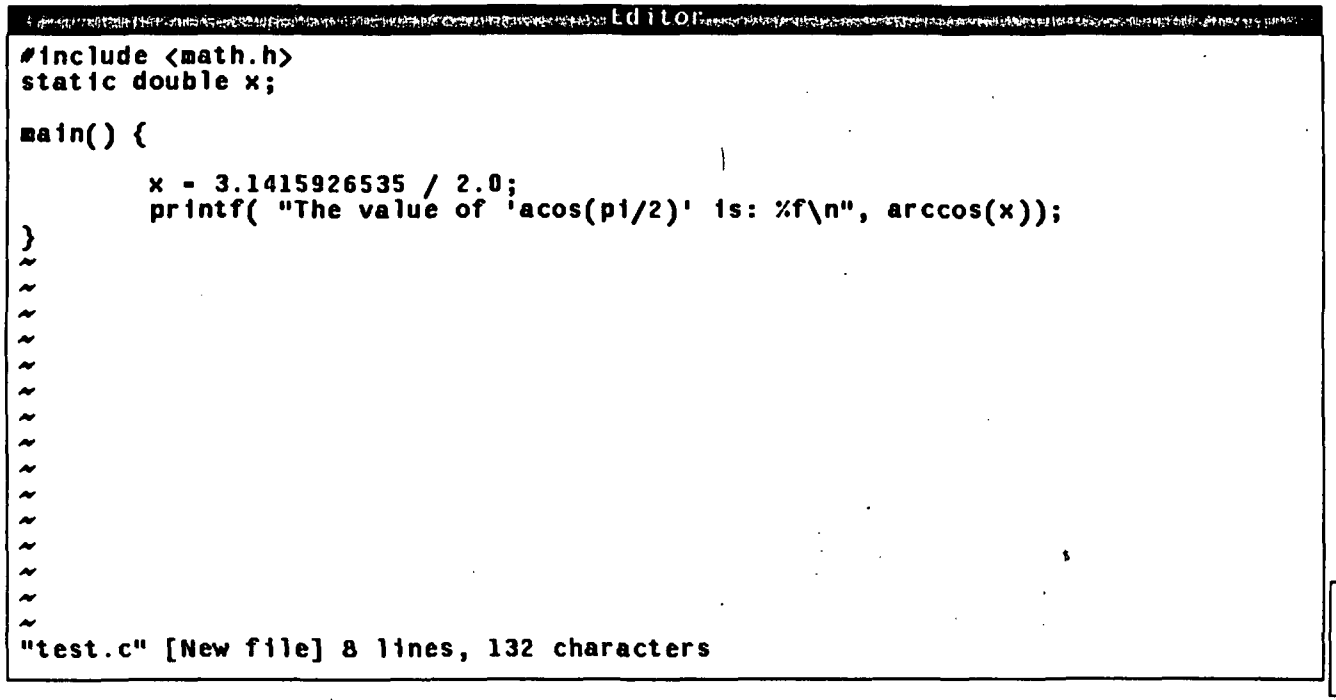
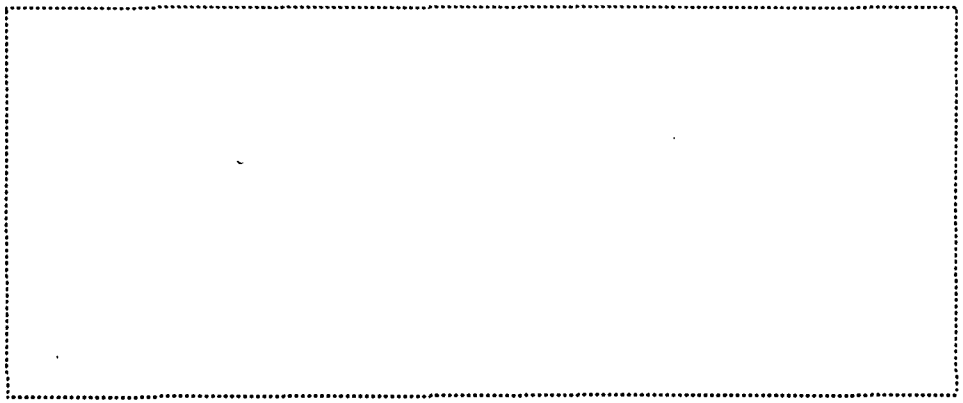


Figure 11

umgn: parameter-menu

nrow=12	color	ulcx=21	ulcy=98	[mouse]
ncol=57	mode=STORE	width=593	height=242	QUIT



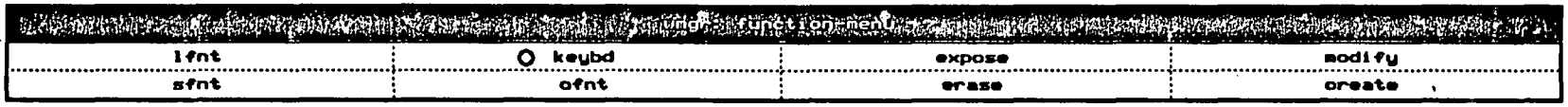
Editor

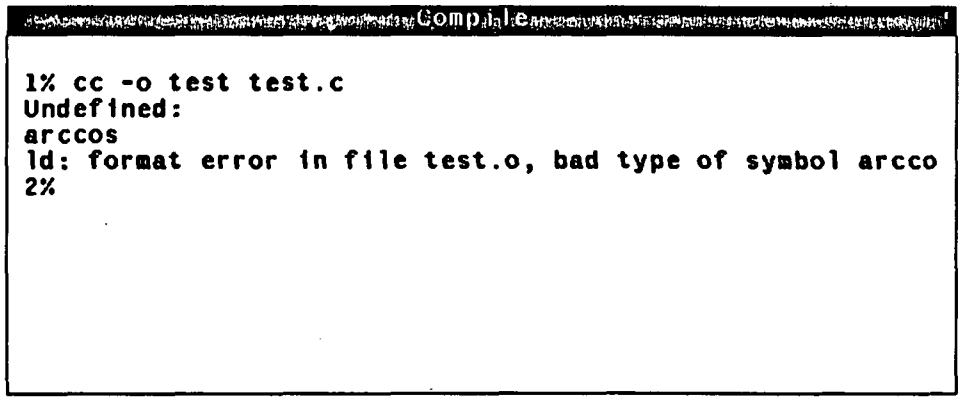
```
#include <math.h>
static double x;

main() {

        x = 3.1415926535 / 2.0;
        printf( "The value of 'acos(pi/2)' is: %f\n", arccos(x));
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"test.c" [New file] 8 lines, 132 characters
```

Figure 12





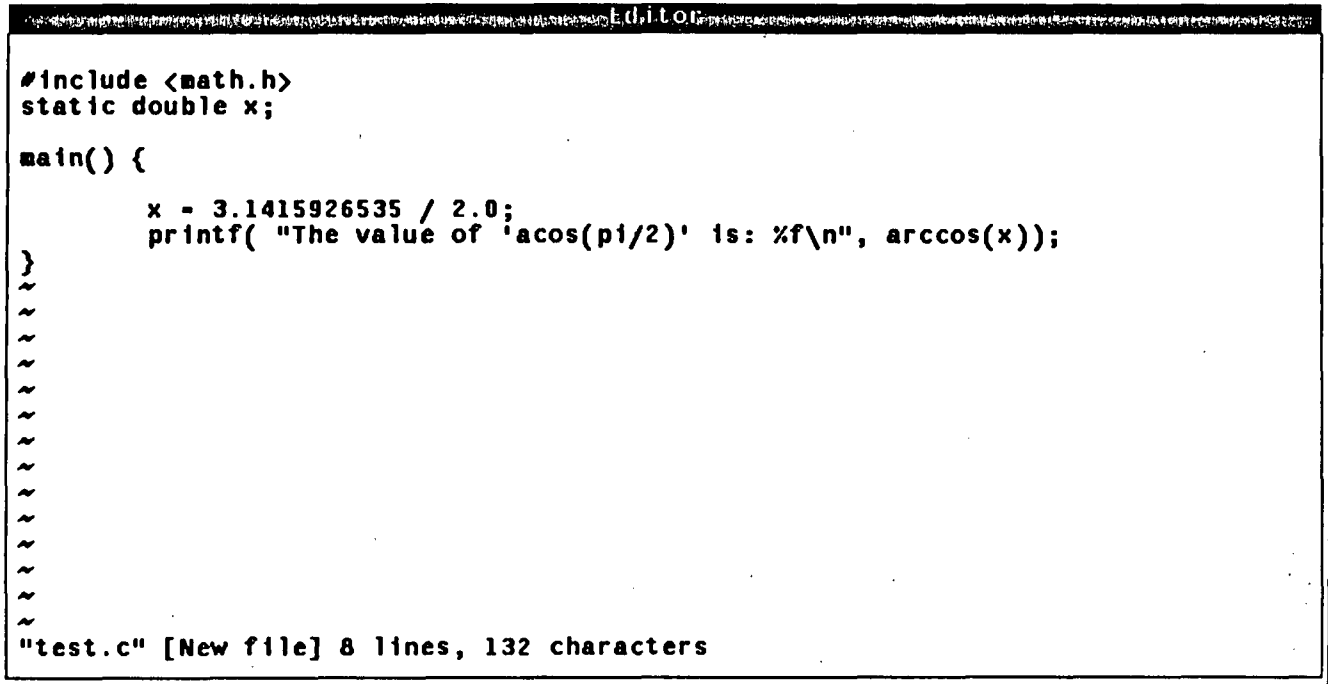


Figure 13

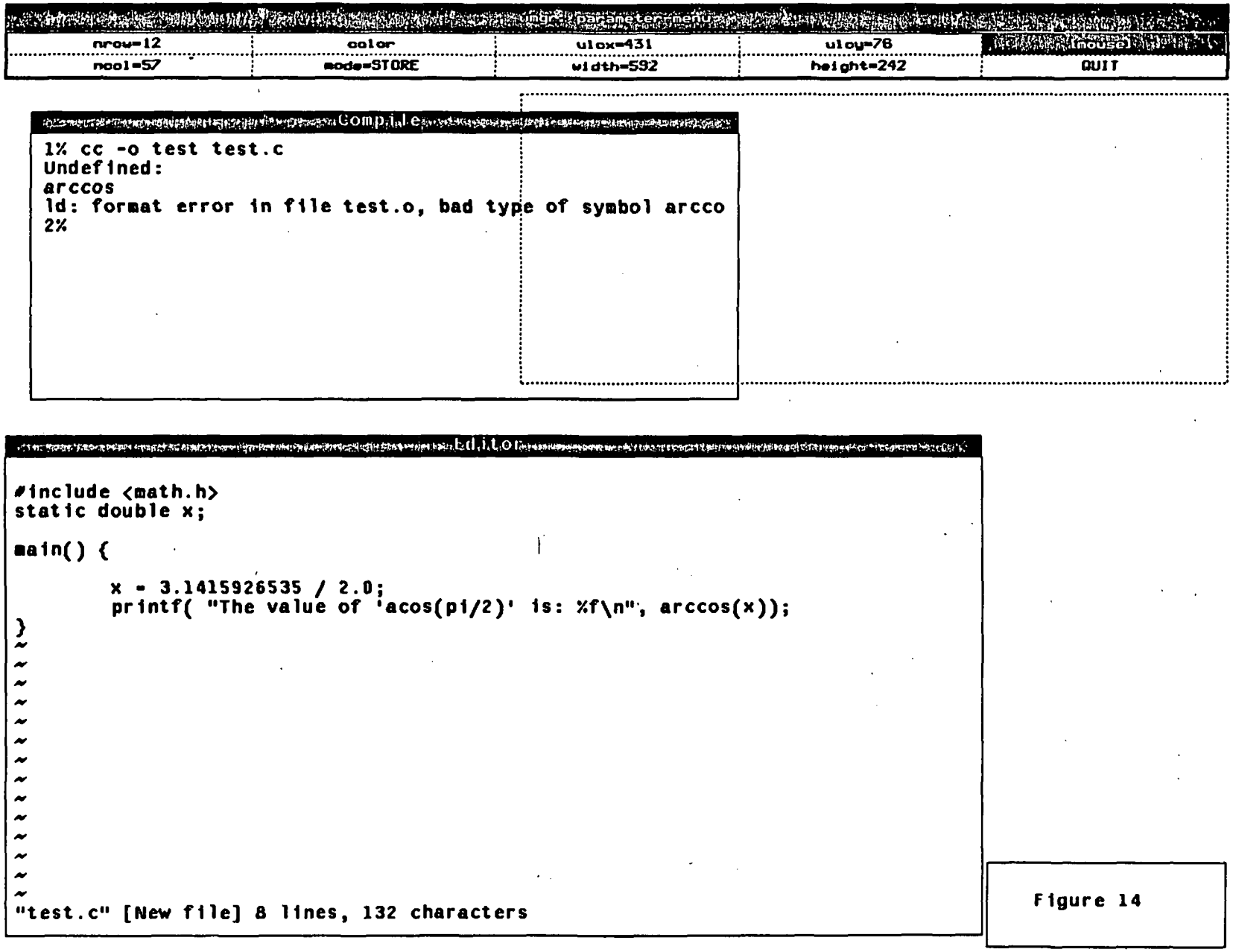


Figure 14

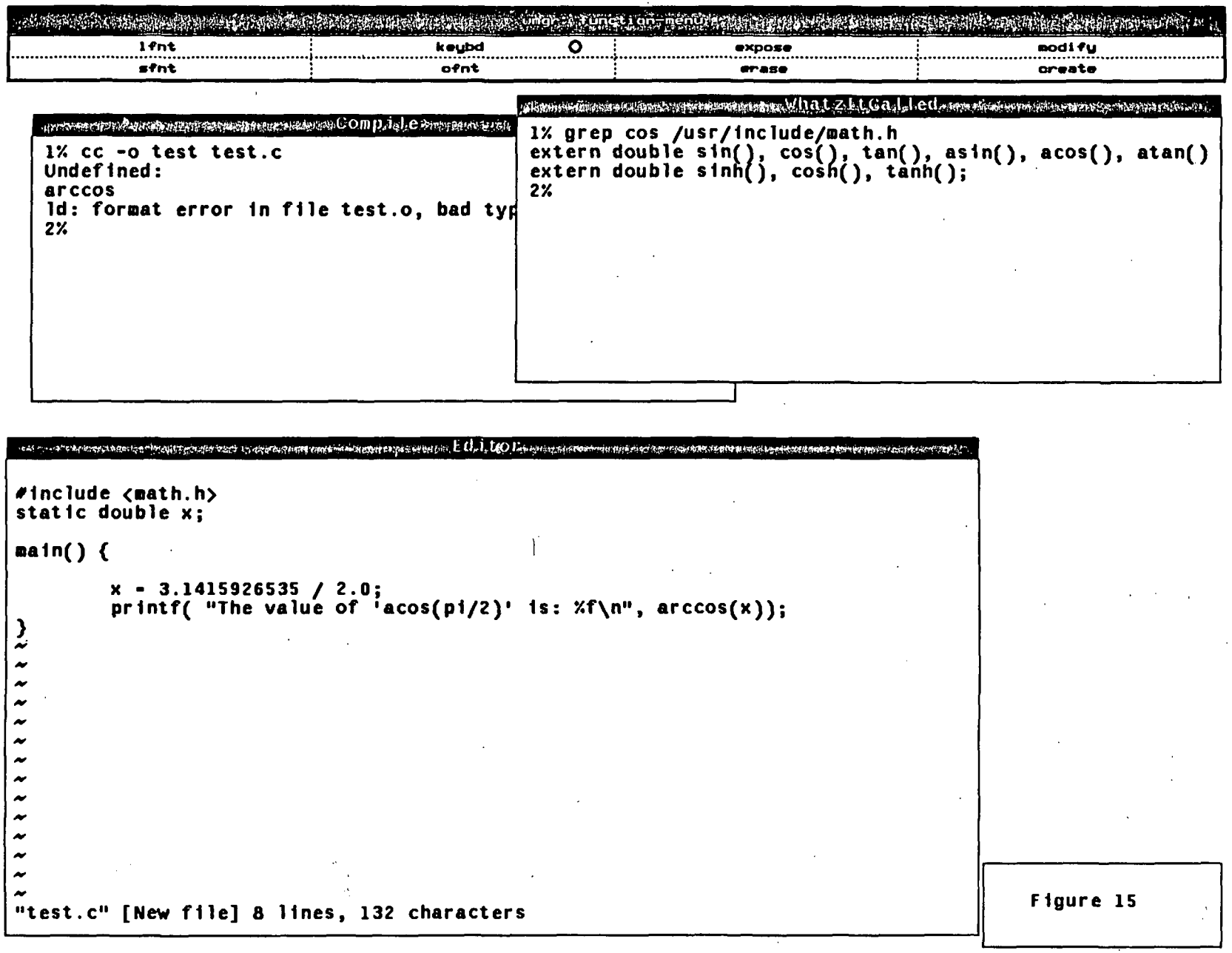


Figure 15

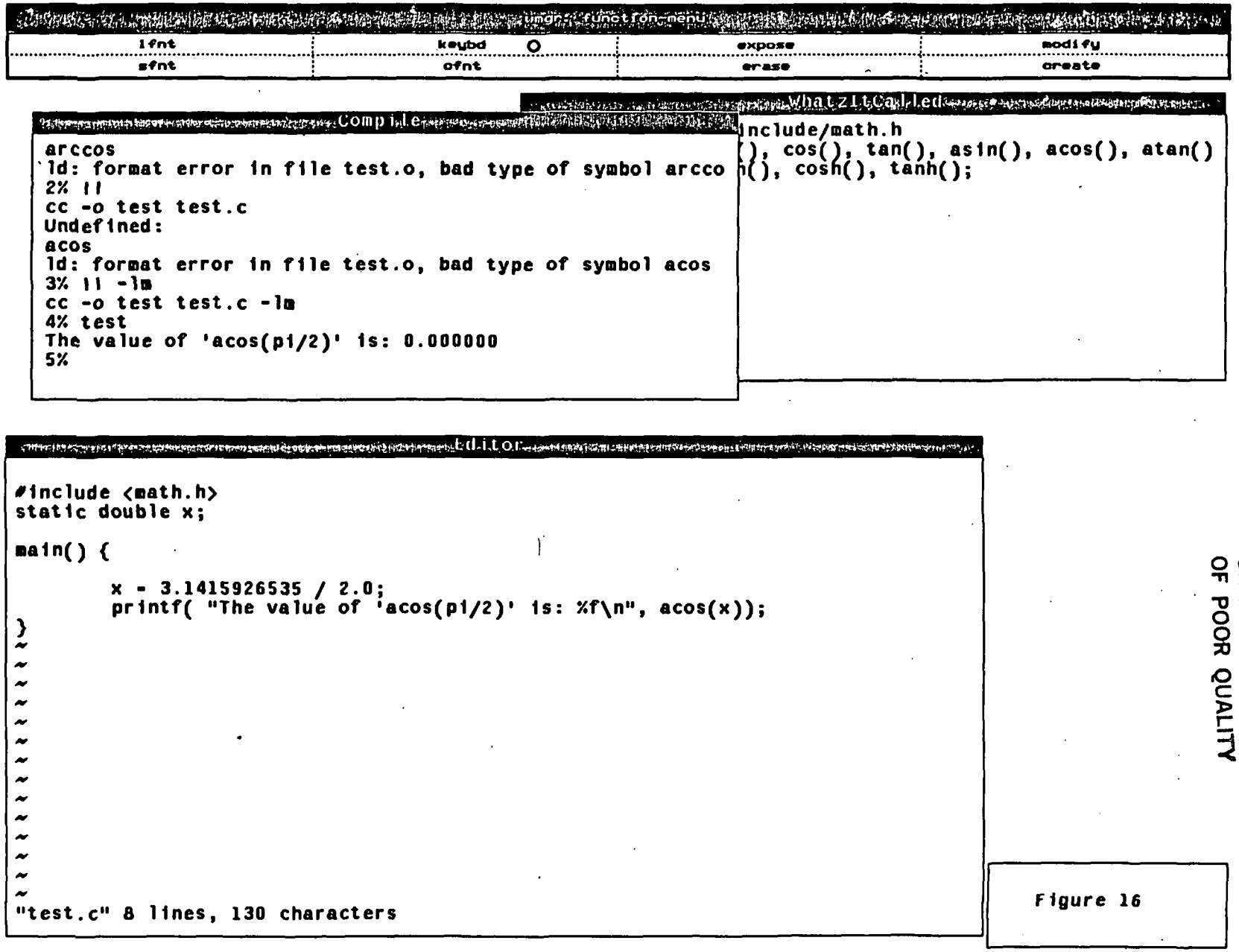


Figure 16

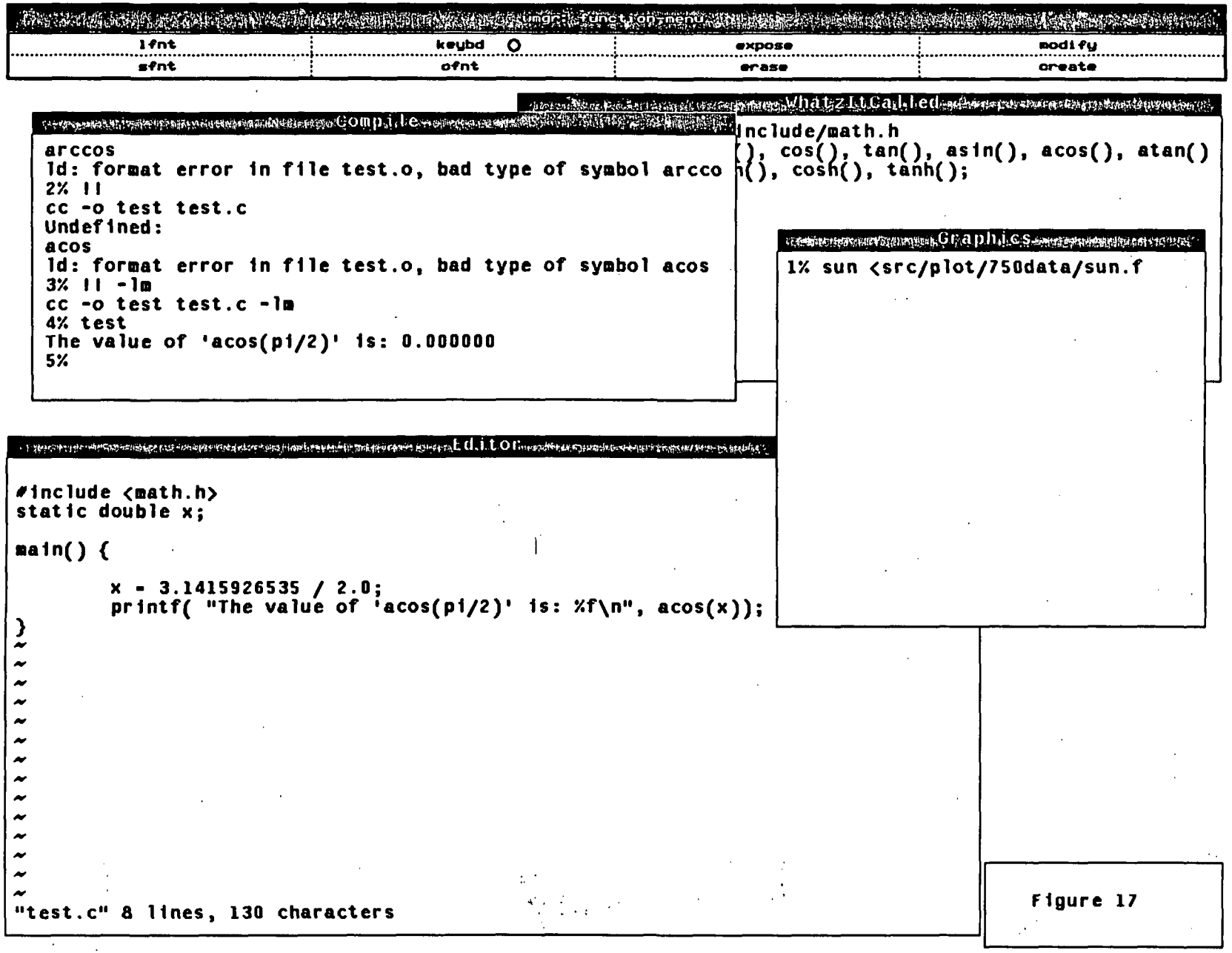


Figure 17

ORIGINAL PAGE IS
OF POOR QUALITY

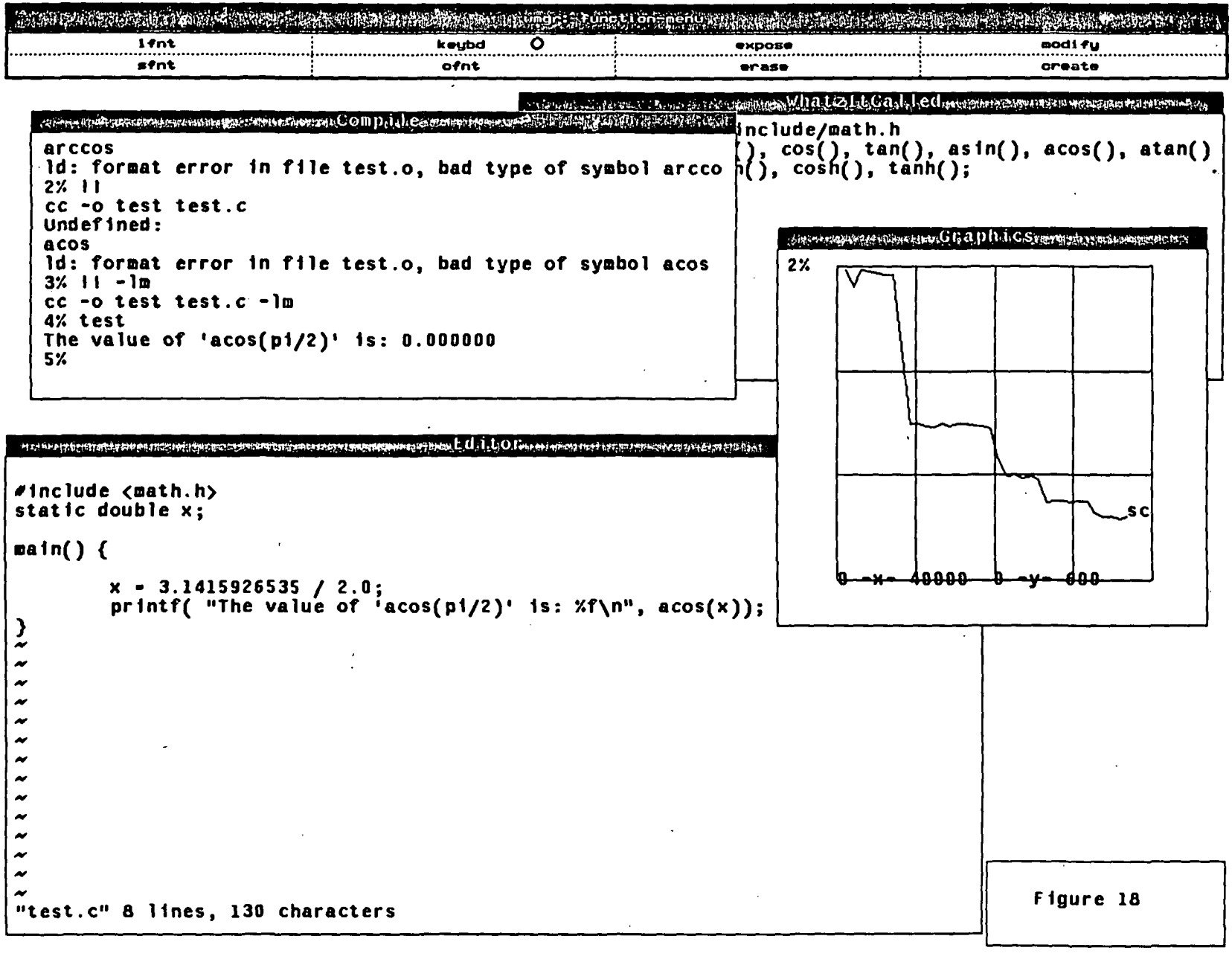


Figure 18

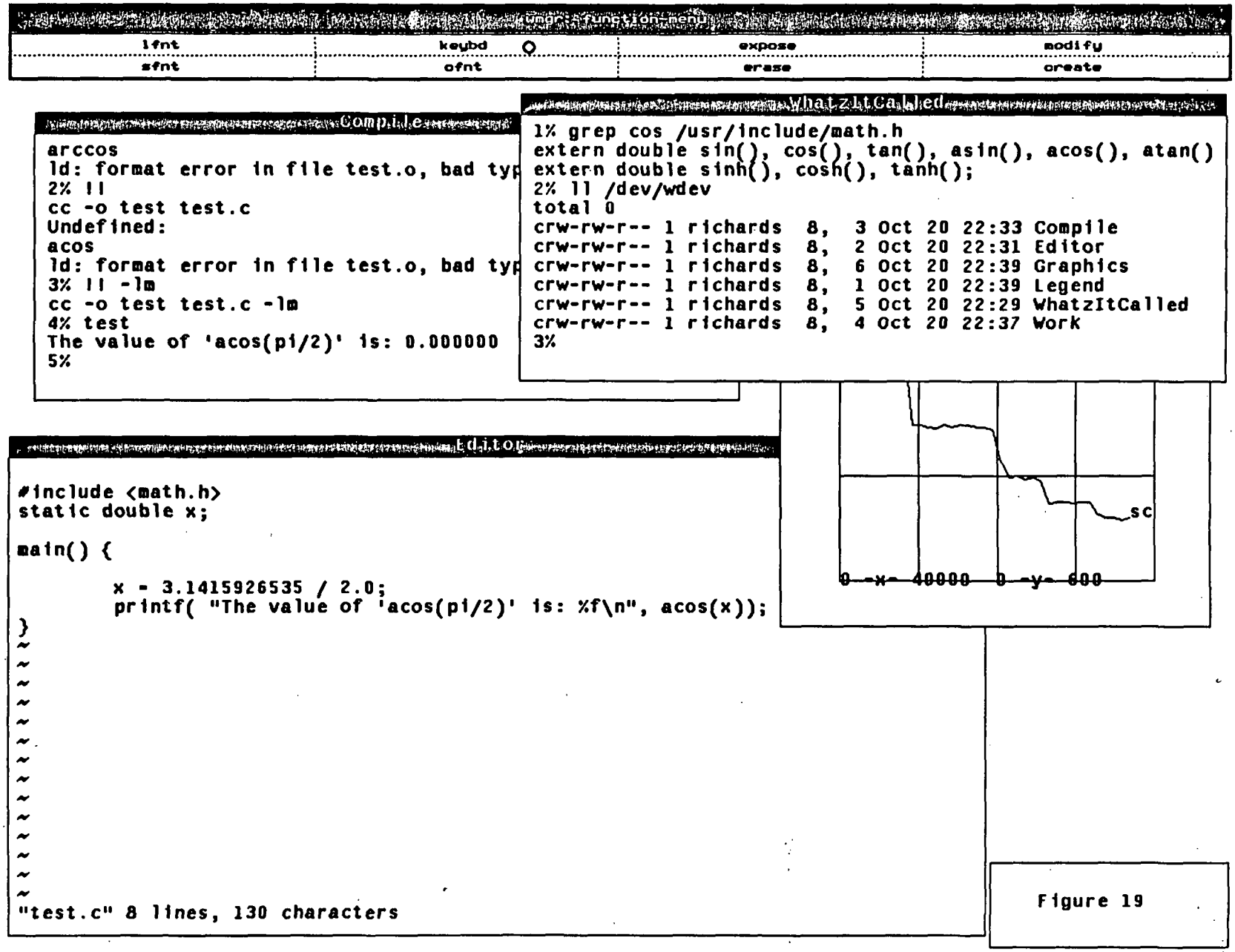


Figure 19

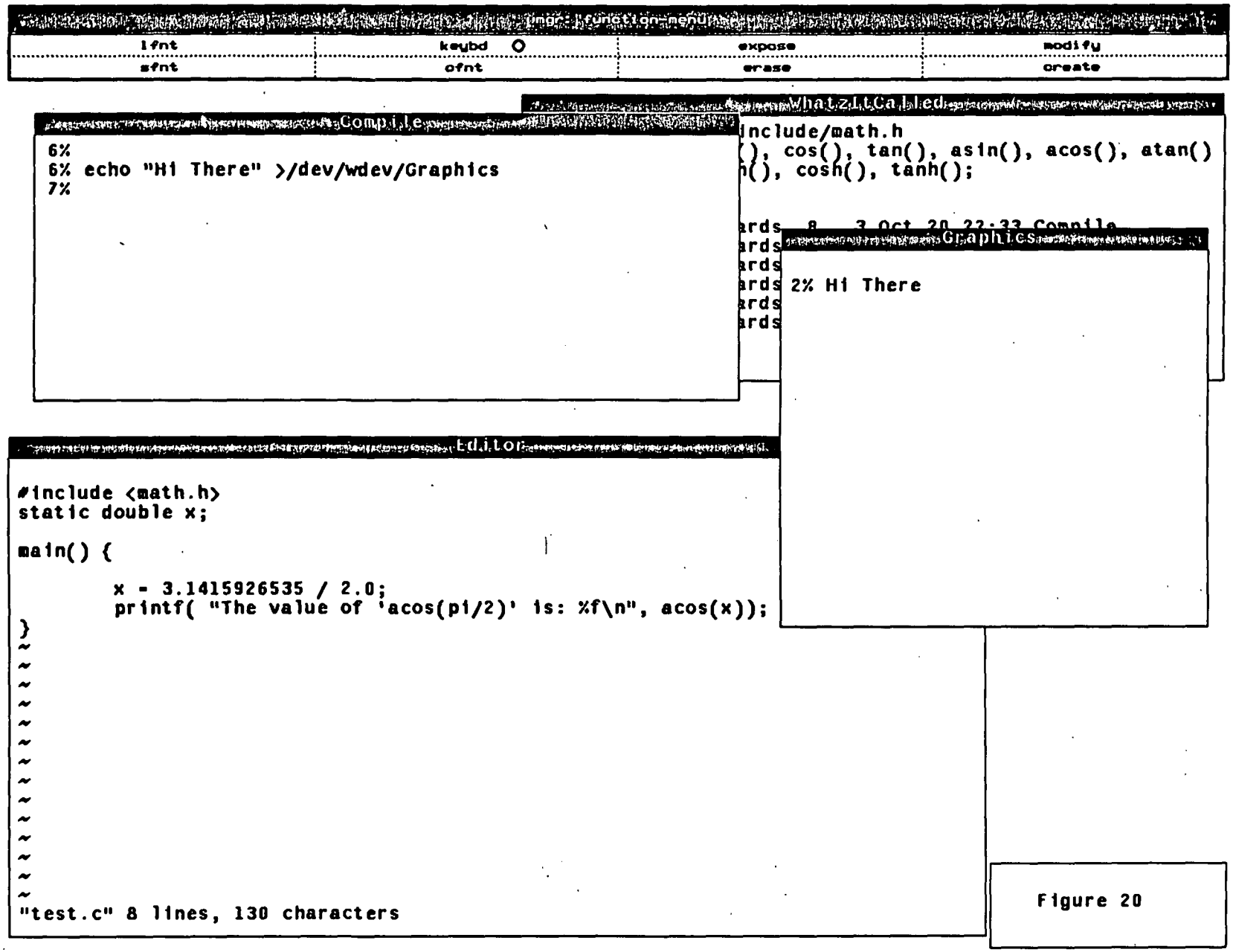


Figure 20

SAGA Mid-Year Report

Appendix F

Differences and Possibilities for Diff and Undo Commands

Carol S. Beckman-Davies

Department of Computer Science

University of Illinois

Urbana-Champaign

Illinois, 61801-2987

October, 1983

1. Finding Differences and Rebuilding Terminal Lists

Two programs, one to find differences between versions of a program edited with the SAGA editor, and the other to reconstruct a parse tree from one tree and the differences, were written. The program finding differences finds contiguous sections of changed tokens. The insert and delete commands necessary to produce one token list from the other are saved in a file. This file may then be used to rebuild the terminal list. The editor can take the terminal list and build the entire parse tree.

The difference program uses a bit in each terminal node which indicates whether it or its neighbors have changed. This modified bit is set if the node was inserted or one of its neighboring nodes was inserted or deleted. Thus, a sequence of terminals which do not have their modified bits set must occur as a sequence in the original version. The difference finder searches for this sequence. When it finds the sequence, the program brackets a change using the last unchanged section (or the beginning of the program) and the section just found. The bracketed sections, the change section from the old program and the one from the new, contain the nodes that must be inserted or deleted to get from one version to the other. The beginnings and ends of the change sections are checked to see if they match. As many nodes as possible are matched. This takes out of the difference the nodes whose modified bits were set only because a neighbor changed. It might also catch some changes that a user makes and later undoes, but not all.

Since the difference finder depends on the modified bits, they must be set correctly. This can be accomplished by copying the parse table and string table files for the existing version, clearing the modified bits, and editing this tree. The modified bits are preserved between editing sessions; therefore, the change information will accumulate. Consequently, the difference between a working version and one with modifications, where the modifications could have been entered in several sessions, can be found. The parse tree must be preserved between editing sessions for the accumulation to occur.

The differences between the two versions, and a string file to hold the tokens to insert, are kept. Backward differences are kept. That is, the current version is kept in its entirety. The older version is obtained from the current version and the difference.

A problem with using the parser on a list of terminal nodes exists. The program which reconstructs the terminal list reconstructs it exactly. When a list includes an empty token, the parser deletes the empty token that was there and inserts another. This wastes one node for every empty token in the program and sets the modified bits in the empty token and the adjoining two nodes. This can be overcome by clearing the modified bits. The compacter can reclaim the nodes, probably after editing when compaction would be advisable anyway. Having the problem handled in this way is probably easier than changing the parser.

2. Diff Command

Work on putting a **diff** command into the editor has begun. The first version will be a simple command based on the difference finder which reports contiguous differences. The simple difference command will display the old and new lines in the program. The old and new blocks for a changed section will be displayed together. A range for the difference can be specified in the current tree being edited. Only differences within the range will be displayed. The user will also be able to specify the number of lines of context to print around the difference.

Plans to improve the diff command and add an undo command based on the differences are being developed. Several possibilities for these are under consideration.

Three issues need to be considered for the diff command: how to specify a range for the diff, what to show, and how to show it. Some possibilities for specifying the section of the parse tree for the diff are specifying a range with pointers and specifying a subtree. The simple diff command will allow the user to specify a range on which to find differences with two editor pointers. Differences before the range will be found to match the section of code in the old version which corresponds to the new section of interest. One way to avoid finding differences which do not interest the user would be

to have the user give the starting location in the old program which corresponds to the beginning of the new section. Or it might be possible to keep links between the two versions. The diff command could find the link before the section of interest and closest to it, and start finding differences from those points.

Specifying a subtree as the section in which to find the difference is another possibility. The type of subtree allowed could be restricted to "special" subtrees, such as procedures, declarations, and statements. Links between the trees could be kept for this type of difference as well. With special subtree differences, other methods to match sections to be compared could probably be developed. Allowing several methods for selecting sections to compare gives the most flexibility.

Once a difference is found, what should be shown to the user must be decided. One possibility is to compress details in large differences. For example, if a diff on the entire program is done, the differences could be reported for large sections, such as saying, "procedure initialize has changed." If a diff is done on a procedure, changes could be reported by declaration sections and statements, for example. The subtrees on which to report should depend on the grammar.

If differences are compressed, the user will sometimes want to see the actual code. When that kind of difference is requested, giving the user context around the change would be useful. The simple version of the diff command will let the user specify the number of lines of context to show around the change.

Another possibility for displaying differences is to break them up according to the parse tree. The simple version of the diff command will display all changes in one area together. The changes might be at the end of one procedure and beginning of another, or the end of a declaration section and beginning of the executable statements, or otherwise unrelated. Perhaps a more informative method of display would show the change to the first section, and then the change to the contiguous but unrelated section, instead of showing them together. Division of differences should probably be made on the basis of special subtrees, such as procedures or statements. A potential

problem with using the tree structure is matching the old and new section. A change which affects the special subtrees could be broken up; however, deciding which parts of the old and new versions go together would be difficult or impossible.

Besides breaking apart contiguous differences, joining distant differences might also be informative. Changes that are related might be far apart. For example, an inserted **repeat...until** around an existing block of statements might be more informatively displayed together. Displaying contiguous differences would show first the insertion of the **repeat**, then any changes to the statement block, then the insertion of **until**. This idea has some difficulties. First, some way to correlate the differences must be found. Correlations can be obtained from: the grammar, if that is possible, user input for each language or each edit session, the way in which commands are entered (e.g., all changes between valid parse trees), or some other information. Another question is how the user can know where the other parts of the change are in relation to the rest of the code. The ordered display showing contiguous differences gives the user an idea of where all the changes in the code are. If the new **until** clause is shown when the new **repeat** is found, the user has fewer clues on the location of the **until** clause.

Other amenities related to the actual display of the differences can be added. These would generally not affect nor be affected by decisions on what to display. Some possible improvements to the display could make it more informative. A special character could be printed at the beginning of lines with differences when context is given. The old and new sections could be shown in separate windows. Some better indication of where the difference is, e.g., line numbers or the name of the procedure containing the difference, could be given. These features should not affect the rest of the diff command.

3. Undo Command

Another command which can eventually be added to the editor is **undo**. The first decision to make for undo is whether diff and undo should be related. If they are

unrelated, the only possibility for a basis for an undo seems to be the commands the user enters. The commands can be undone in two ways: in a stack-like manner, undoing the last command first, or in any order. Anticipating the effect of a stack-like undo would be fairly easy for the user. A stack-like undo is very inflexible, however. If a user deletes ten lines, then inserts ten lines, and realizes he deleted the wrong ten lines, a stack-like undo does not help. Whether he uses the undo or not, he must retype ten lines. A stack-like undo is simple, but of limited help.

A more flexible undo would allow undoing any command. This method has many severe problems. First, the user must select a command to undo. Displaying the list of commands that may be undone is, by itself, of little help. The user would generally not be able to tell which command he wanted to undo. Before a command could be undone, the location of the effect must be found. Since sections of code could be inserted and deleted, any reference used to record the location of the command when it was done cannot be used to undo it without translating the location somehow. Such translation would probably be messy or expensive. Further, the meaning of an undo is not always clear. The user might have deleted lines in a section of code that an undo should delete. He might have inserted lines in a block an undo should delete. Undo would be rife with special cases. The user would have many rules concerning undo to remember. Seeing the effect of an undo would be difficult without performing the work necessary to actually do it. An undo able to undo arbitrary commands would be inefficient and hard to use.

The clear choice for a flexible undo which is relatively easy to use is one related to diff. The user could request a difference and select a difference to be undone from the ones displayed. Undo would have to follow a diff command which saved information for it or execute a diff of its own. How the differences are displayed would have little effect on undo. What is displayed is important. A difference displayed as one unit by diff should probably be treated as a unit by undo.

An undo which could reverse the effect of just the last command is also useful. That undo would be unrelated to diff and the undo command discussed here. It could be implemented completely separately from these, if it is decided that such a command is beneficial.

Some principles should guide selection of features for undo. Undo should not require much effort from the user. Undo should try to avoid introducing syntax errors into the program. Having undo *never* cause errors would be difficult and more effort than it would save. Finally, allowing the user to undo part of a difference would be convenient. The user might not agree with the division upon which diff decides.

Many possible schemes for an undo command have been proposed but many of them create a dilemma. If the undo only permits a change in one location at a time, the introduction of syntax errors is guaranteed. Deleting or inserting a **begin...end** or **repeat...until** requires changes in more than one location in the parse tree. Allowing more than one change in the parse tree necessitates a change in the incremental reparsing procedures. Plans for changing the parser are being developed, but it will be some time before changes are implemented and stable enough to use. Until the parser changes, many of these suggestions will be difficult to implement.

The first undo scheme requires the user to select one or more differences from a list of displayed differences as arguments to the undo command. This scheme would be straight-forward to implement and should provide the user a simple selection procedure. The scheme encourages the user to select a set of syntactically related differences as arguments to the undo in order to avoid introducing syntax errors. However, this scheme does not allow the user to select part of a difference to be undone.

Another possibility for selecting differences to undo uses movable pointers. Diff could set "pointers" around the changed sections at the beginning and end of the sections in the new and old versions, four pointers for each difference. The user could then move the pointers, that is, change the boundaries of the sections. When the user

has the sections selected to his satisfaction, the section in the new program would be deleted and replaced by the section from the old version. If the user can select more than one difference to undo, this command is quite flexible. It can avoid syntax errors in the same manner as the first. The user can contract or expand the sections to change. The user would, however, have to do more work with this type of command. An added advantage of this type of undo is that it could be generalized to allow replacement of arbitrary sections of the new version with arbitrary sections of the old. A movable difference pointer undo is flexible but at the price of more work on the user's part.

A third possible scheme avoids the user having to select multiple differences. The difference provided as an argument to the undo command selects a subtree within which the undo command will operate. Undo would use the first token of the difference to select a subtree which is the largest subtree that has that token as its leftmost descendent. The first difference in the tree is undone. If it exists, the next difference in the subtree is found. The user is queried about whether to undo that difference. This continues until the last difference in the subtree is reached. Alternately, if the user does not undo a difference, the largest subtree that has the first token of that difference as leftmost descendent could be found and all differences in that subtree skipped. As one example of how this might work, consider a statement block around which a user has put an **if cond then begin ... end**. The user decides to undo the change. He selects the difference containing the **if** to be undone. This selects the subtree, say **statement**, to which the **if** reduces. Undo removes the **if cond then** and asks whether to delete **begin**. The user responds yes, so undo deletes it, goes to the first change in the block statement, and asks the user if he wants to undo that change. He responds no, so undo skips all the subtree for that difference. Assuming that the skipping includes all changes in the statement list, undo asks whether to delete **end**. The user responds yes. The difference is undone without introducing syntax errors. The hope is that any change which must be undone with another will be contained in the subtree that is selected, thus avoiding syntax errors.

Selecting the largest subtree with the token as the leftmost descendent will not prevent syntax errors if the user selects some difference other than the first one of a set that must be undone together. The selected subtree will not contain all the differences that must be undone. It is also not certain that all the differences that must be undone together will be in the selected subtree even if the first difference is the one used to select the tree. Syntax errors may still be introduced. Skipping subtrees decreases the amount of work the user must do. The user might still have to answer too many queries to make one change. The user is dependent on the division of differences which diff gives. He cannot undo any finer differences. This method has a premise which seems sound but is unproven, and it may require too much work from the user.

Probably the most severe problem with this method is the difficulty in explaining it. A user who does not know much about parse trees or the grammar being used would have difficulty predicting the effect of an undo. The user could select a subtree in which to work with a tree oriented command instead of having undo select one based on a token. This would make the scope of the undo clearer to the user, even for one who understood parse trees well. Tree-dependent actions within the subtree would still be confusing.

A final possibility for undo is a syntax-directed command. The user selects a difference to undo. It is undone and reparsed. If no syntax error occurs, the command is done. If one does occur, undo looks for a difference to undo that will correct the syntax error. The desired difference must occur before the point at which the error occurred. If more than one change can be undone to correct the error, undo could ask the user which to undo. Even if only one change will correct the error, asking whether to undo it would be a good safeguard. This type of undo does not require much extra effort on the user's part but avoids syntax errors. It does not allow flexibility on how much of a difference to undo. This method would take much more processing time. Discovering whether undoing a change will correct an error would take time and would have to be done for several changes. Once the correction is found, the reparsing must be done a second time. This method will work with the reparsing method as it now

stands. It only changes the tree in one spot at a time, which is all the incremental parser can handle.

4. Summary

Several possibilities for an undo exist, each with advantages and disadvantages. Perhaps the best method will be a combination of several suggestions. What is best will also depend on the diff command. If diff breaks up contiguous differences well, being able to undo only part of a change becomes less important. If diff relates changes that must be undone together to avoid introducing syntax errors, the few times that an undo causes an error can be tolerated. A method weak in avoiding errors could be used.

Related diff and undo commands seem to be a good idea for an addition to the SAGA editor. Having undo related to diff improves the flexibility for the user. It also complicates the design of the commands since features of one will affect the other. A harmonious combination of diff and undo would make editing easier for a user.

SAGA Mid-Year Report
Appendix G
Proof Management System

David H. Hammerslag

Department of Computer Science
University of Illinois
Urbana-Champaign
Illinois, 61801-2987

October, 1983

1. INTRODUCTION

Program verification often requires the proofs of large, cumbersome theorems. Such proofs, when done by hand, are time consuming and error prone. An alternative is to let a theorem prover attempt the proof. However, most theorem provers cannot prove large involved propositions without user assistance. When the prover fails, the user is left with little understanding of why the proof failed; was the proposition false, or was the theorem prover simply unable to prove it? To aid in such proofs a proof management system is being developed. This system will be used to aid the programmer in doing formal, first order, natural deduction proofs, including the proofs arising from program verification. The system will consist of three modules: a tree editor, a proof checker, and a theorem prover.

2. THE TREE EDITOR

The user invokes the tree editor to create nodes, link them together into trees, and edit the text in a node. Tree editor commands will be provided that are the tree analogues to common line editor commands. Among such commands will be:

- Begin a new tree
- Add a child
- Delete a tree
- Move a tree
- Match a tree pattern (matching both tree properties and the text in the node)

The user must also be able to edit the block of text in the node. Therefore a text editing facility is also needed to do insertions, deletions, movement, and substitution according to pattern match.

The tree editor must also be programmable, able to perform tree I/O and able to interface with other modules, either user or system defined. The programmability could simply consist of the ability to store a sequence of operations to be reapplied at any node. For tree level I/O, a representation of the tree data type, preferably human readable, needs to be developed. To allow the editor to interface with outside

programs, i.e. a pretty printer or, in this case, a proof checker, the structure of the tree data type must be available to "the outside world."

3. THE PROOF CHECKER

The proof checker will allow the user to verify that the tree that he has built in the tree editor constitutes a valid proof. A set of inference rules will be known by the proof checker and presumably the inference rules referred to in the tree nodes by the user will be among those rules that the proof checker knows about. When the proof checker has determined what rule of inference was used it can recursively work its way down the tree. As it verifies the validity of nodes it will use a protected field to mark the node as being verified. The proof checker will allow quick confirmation of the validity of the proof.

There are still many issues unresolved concerning the proof checker. One of these issues is what the proof checker should do when it comes upon a node that cannot be verified. What can be done depends on how the proof tree is constructed. If all of the nodes are of the same type, namely assertions, then when the proof checker comes upon a node that cannot be verified, it should still continue down the tree to the lowest level possible, verifying as much of the tree as possible. If a node's children are not necessarily all of the same type, then it would not be possible for the proof checker to go any deeper into the tree. In either case, the proof checker will mark as verified only those nodes whose inference rule has been verified. In addition to helping the user find his errors, this will also make subsequent rechecks of the same tree much faster.

4. THE THEOREM PROVER

In this system, the theorem prover will be treated as a black-box. Once invoked, it will run with little or no guidance from the user. Little if any information, aside from whether the proposition was true or not, will be returned to the user. It is not anticipated that a prover will be written, but rather that an existing theorem prover

can be used. Clearly an interface between the tree editor and the theorem prover will have to be written. Additionally, the theorem prover will have to be able to either mark nodes itself, or pass sufficient information back to the editor so that the editor can mark the appropriate nodes.

5. THE TREE NODE

Although the three modules will be distinct, they will interact. The point of that interaction will be through (proof) tree nodes. In addition to pointers to other nodes, the node will consist of a block of arbitrary text. To increase the generality of the editor the meaning of the text in the node will be interpreted by convention. A convention for proof tree nodes might be:

(<LABEL>;<INFERENCE_RULE>;<ANTECEDENTS>;<FORMULA>).

Then a node:

(MAIN THEOREM ; Modus Ponens ; (g(x),(p->q)); f->p)

would be interpreted as representing a proposed theorem, f->p, referred to as "MAIN THEOREM" which was deduced from its children via modus ponens, and has antecedents "g(x)" and "p->q". Additionally, each node will have at least one field which is protected from the user; these fields may contain system information such as whether or not the proposition represented by the node has already been proven.

6. TOOL INTERACTION

Interaction of the theorem prover, the proof checker, and the tree editor will allow the user to write proofs in a convenient environment. Ideally the user will spend most of his time in the tree editor, building proof trees. During an editing session the user will be able to invoke the proof checker or the theorem prover at any arbitrary node. If the proof checker is called, it will attempt to verify the proof of the current proposition. If the theorem prover is called, it will try, under constraints of time and/or depth of search, to prove the proposition designated by the current node. Both of these constraints will be viewed as parameters that may be adjusted by the user. If

the proposition is proved, either by the proof checker or the theorem prover, the protected field will be set to signify that the proposition is true.

In order to speed up proof checking, the protected information in the node will be used to keep the proof checker from rechecking a proposition if it has been previously examined by the proof checker, verified, and had none of its children modified. At this point some of the generality has been taken away from the tree editor; it must now keep track of which nodes have been changed and must reset the protected field for ancestors of the modified nodes. The loss of generality in the editor is offset by the time gained in proof checking, where trees will not have to be rechecked.

7. DISCUSSION

There are already systems in existence that could be considered to be proof management systems. For example Abraham's Proofchecker FOL and AFFIRM [Ger79a, ISI79] all, in one manner or another, encompass the notion of making formal proofs easier. One of the important differences between existing systems and our proposed system is the separation of the three components in our system; the editor will have only limited knowledge of the proof checker, and no knowledge of the logic used by the proof checker. This gives the user great flexibility in how the system is used. We believe that this flexibility constitutes an advantage.

One manifestation of this flexibility is the ability to construct alternate proof trees. For example, if the proposition being proven was "(A OR B)," it could be to the users advantage to be able to have proofs of both A and B under construction at the same time. In our system, because the editor has no knowledge of the logic being employed, the user could certainly have two, or any number of children, attached to the "(A OR B)" node. Then, when the user is satisfied that one of these children is the root of a proof, either by inspection or by using the proof checker on that subtree, he could delete the extra children before the proof checker is asked to verify the "(A OR B)" node. As most other systems require that any tree constructed be a valid proof

tree, this sort of flexibility by the user would not be allowed.

Another advantage of our system becomes apparent when programming the editor. Most systems require that any programming be proven correct in one manner or another (see [Boy81] for a discussion of some of the strategies used). A user defined rule that is correct only some of the time could not be employed under such a system. In our system the user may program the editor to build any sort of tree he desires, whether it constitutes a valid proof tree or not. Therefore the user is free to program the partially correct rule and use it at his discretion. Any misuse of the rule will be caught by the proof checker. It is up to the user, who presumably wrote the rule, to determine when it should be used, however if he is in doubt, the proof checker could always be invoked immediately after the rule was used.

8. CONCLUSION

Through the interaction of the tree editor, the proof checker, and the theorem prover, the task of writing, understanding and "debugging" complex proofs should be greatly simplified. By using the theorem prover the user can be freed from the tedium of doing proofs in all of their rigorous detail, and by using the proof checker as the proof is being written, the user can detect errors in his reasoning earlier, where they are more easily correctable.

References

[Boy81]
R. S. Boyer and J. Strother Moore, "Metafuctions: proving them correct and using them efficiently as new proof procedures," pp. 103-184 in *The Correctness Problem in Computer Science*, ed. Robert S. Boyer; J. Strother Moore, Academic Press, London (1981).

[Ger79a]
S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile, "An Overview of Affirm: A Specification and Verification System," USC Information Sciences Institute, Marina del Ray, Ca (November 1979).

[ISI79]
ISI Research Staff, "Program Verification: Annual Report, Fall 1979," USC Information Sciences Institute, Marina del Ray, Ca. (1979).

SAGA Mid-Year Report

Appendix H

A Program Slicer for Pascal 6000

F. M. Hardin

Department of Computer Science

University of Illinois

Urbana-Champaign

Illinois, 61801

October, 1983

A PROGRAM SLICER FOR PASCAL 6000

BY

FRED MATTHEW HARDIN

B. S., Texas A&M University, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1983

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Professor Mark A. Ardis for all his guidance and help; I also thank Pete Kirslis and Wayne Badger for graciously answering all my questions about SAGA.

Finally, I would like to give special thanks to my family for all their support and advice throughout my years in college.

TABLE OF CONTENTS

1. INTRODUCTION

There are many tools in existence used to aid programmers in debugging their programs. A recent development in this field has been the program slicer [2, 3, 4,], a tool that attempts to produce a minimal form of a program that produces a given behavior.

This thesis examines the implementation of a program slicer for a syntax-directed CDC PASCAL 6000 editor implemented on the SAGA system, a software development system developed at the University of Illinois [1].

As a means of introduction to the actual implementation, a brief examination of the SAGA PASCAL editor will be followed by a general discussion of program slicing. The methods of the implementation will then be discussed along with certain problems encountered during the work.

2. SAGA PASCAL EDITOR

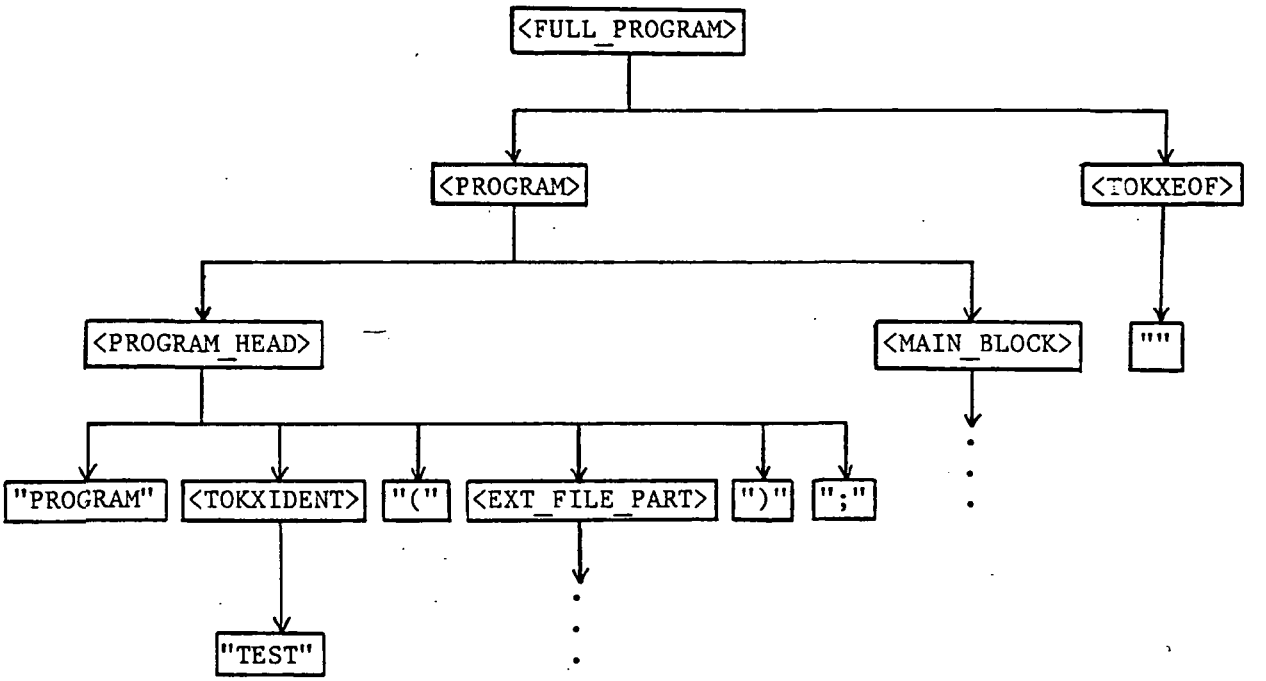
The PASCAL 6000 editor of the SAGA system is a syntax-directed editor that uses a grammar to do an LALR parse and produces a parse tree and string table for the file currently begin edited.

The parse tree produced by the editor contains, in tree form, the list of production rules used to decompose the program. Each node of the parse tree is either a terminal or nonterminal. A nonterminal node represents the right hand side of a production rule with the son node(s) used to represent the left hand side. A terminal node has no son and points to the string table entry that holds the characters of the particular terminal. In Figure 1, a graphical description of the parse tree for the beginning of a typical program is shown.

The string table produced by the editor is a long array of characters used to hold all the words and characters needed to print out the program. All the keywords, variable names, procedure names, and special characters ("+", "-", "*", "/", "=", etc.) are held in this array. The individual elements of this array are accessed by giving the starting point of the string in the array and the length (in characters) of the string.

Example of Parse Tree Structure

Parse Tree:



Production Rules Used:

```
[  1]   <FULL_PROGRAM>    -->  <PROGRAM> <TOKXEOF>
[  2]   <PROGRAM>         -->  <PROGRAM_HEAD> <MAIN_BLOCK>
[  6]   <PROGRAM_HEAD>    -->  PROGRAM <TOKXIDENT> ( <EXT_FILE_PART> ) ;
[269]   <TOKXIDENT>       -->  TOKXIDENT
[273]   <TOKXEOF>         -->  TOKXEOF
```

Figure 1

3. PROGRAM SLICING

3.1. Purpose

Program slicing is an attempt to reduce a program to a minimal form that produces a given behavior. This behavior is represented by a slicing criterion that has the form of variables at a given point in the program. An example of a slicing criterion would be "variables {x y z} at line 10." The part of the program retained after slicing is called the slice; statements of the retained program are referred to as being in or included in the slice.

A slice includes all statements of the original program that can possibly affect the values of the variables listed in the slicing criterion immediately after execution of the line listed in the slicing criterion. In other words, the slice is the answer to the following question: if the program were to stop execution immediately after the listed line, what statements in the program could have possibly affected the values of the listed variables at this point?

Figure 2 shows an example program and some slicing criteria with their resulting slices.

Program slices can have many practical uses. A slice can be very useful as a debugging aid in that it can remove all the non-essential statements in a program when the cause of a bug is trying to be determined [4]. Slices can also be useful during program maintenance by allowing a person who did not write the code to quickly determine which

Examples of Program Slices

Full Program:

```
 1     program test(input, output);
 2     begin
 3        sum := 0;
 4        diff := 0;
 5        read(a, b);
 6        read(c);
 7        if (c = 0) then
 8           sum := a + b
 9        else
10           diff := a - b;
11        write('sum =', sum, 'diff = ', diff);
12     end. (* program test *)
```

Slice of {c} at Line 8:

```
 1     program test(input, output);
 2     begin
 6        read(c);
12     end. (* program test *)
```

Slice of {diff} at Line 8:

```
 1     program test(input, output);
 2     begin
 4        diff := 0;
 5        read(a, b);
 6        read(c);
 7        if (c = 0) then
 9        else
10           diff := a - b;
12     end. (* program test *)
```

Figure 2

statements are important to a particular behavior of the program to be modified [4]. The last major use of a program slice is to give a measure of the sequential nature of the program. If two slices of the same

program have few statements in common, then the slices represent parallel threads in the program that could possibly be executed by parallel processors to speed up the execution of the program as a whole [3].

3.2. Methodology

The basic methodology of slicing involves the removal from the program of lines that have no effect on the program's behavior as specified by the slicing criterion. A converse approach to the problem is to start with nothing and decide which lines from the program should be included in the slice. The latter approach lends itself well to the slicing of a block-structured language like PASCAL.

In many block-structured languages, there are two basic types of executable statements, assignment statements and control statements. In general, an assignment statement is included in the slice if it can affect, either directly or through a chain of other assignment statements, the variables in the slicing criterion.

For example, if the assignment statements

```
b := c;    a := b;
```

have been executed, then the value of "a" after execution of the second statement depends on the value of "b" prior to the second statement. Likewise, the value of "b" prior to the second statement depends on the value of "c" prior to the first statement. Thus, the value of "c" prior to both statements affects the value of "a" after the execution of both statements. By following the reverse control flow of the program from

the statement listed in the slicing criterion, a set of variables of interest (VOI) can be calculated for each statement in the program. Therefore, an assignment statement will be included in the slice if it affects the value of any variable in the current VOI.

Since the flow of a program is crucial to the order in which assignment statements are executed, any control statement is included in the slice if it can choose either to execute or not execute an assignment statement that has already been included.

4. METHOD OF IMPLEMENTATION

4.1. General Algorithm

The initial step for the slicing algorithm is to build a statement tree from the structural information of the program held in the parse tree and string table. To determine the statements of a program from the parse tree, it is necessary to recognize which production rules signify the beginning of a new statement or output line. A new output line is necessitated by certain keywords in the language such as else or end. A new node in the statement tree is created for each new statement or output line found in the parse tree. Figure 3 shows the list of production rules that cause the creation of a new node in the statement tree.

The program statement will always be the root of the statement tree, and statements of the program are added as sons of the root as they appear in the program. Simple statements (assignment, procedure call, and null statements) will always be leaf nodes without any sons, and compound statements (all others) will be parent nodes with at least one son. Figure 4 shows an example of a program and its statement tree.

The next step of the algorithm is to determine in which of the sons of the program statement does the statement listed in the slicing criterion fall. The slicer then slices from right to left all the sons of the program statement beginning at the son in which the statement listed in the slicing criterion falls.

Production Rules that Create Statement Tree Nodes

Rule #	Production Rule		
[2]	<PROGRAM>	-->	<PROGRAM_HEAD> <MAIN_BLOCK>
[24]	<DECL_ELEMENT>	-->	<LABEL_DECL>
[25]	<DECL_ELEMENT>	-->	<CNST_DEF_PART>
[26]	<DECL_ELEMENT>	-->	<TYPE_DEF_PART>
[27]	<DECL_ELEMENT>	-->	<VAR_DECL_PART>
[28]	<DECL_ELEMENT>	-->	<PROC_DECL>
[29]	<DECL_ELEMENT>	-->	<FCN_DECL>
[118]	<VALUE_INIT_PT>	-->	<VALUE_SYMBOL> <VAL_INIT_LIST>
[162]	<BEGIN_SYMBOL>	-->	BEGIN
[163]	<END_SYMBOL>	-->	END
[231]	<MATCH_STMT>	-->	<ASSIGN_STMT>
[232]	<MATCH_STMT>	-->	<PROC_STMT>
[233]	<MATCH_STMT>	-->	GOTO <LABEL>
[234]	<MATCH_STMT>	-->	BEGIN <STMT_LIST> <END_SYM_BRK>
[235]	<MATCH_STMT>	-->	<WHILE_STMT>
[236]	<MATCH_STMT>	-->	<REPEAT_STMT>
[237]	<MATCH_STMT>	-->	<FOR_STMT>
[238]	<MATCH_STMT>	-->	<WITH_STMT>
[239]	<MATCH_STMT>	-->	<CASE_STMT>
[240]	<MATCH_STMT>	-->	<EMPTY_STMT>
[241]	<MATCH_STMT>	-->	<TOKXPLACE>
[242]	<MATCH_STMT>	-->	IF <EXPRESSION> THEN <MATCH_STMT> <ELSE> <MATCH_STMT>
[244]	<UNMATCH_STMT>	-->	IF <EXPRESSION> THEN <STATEMENT>
[245]	<UNMATCH_STMT>	-->	IF <EXPRESSION> THEN <MATCH_STMT> <ELSE> <UNMATCH_STMT>
[246]	<ELSE>	-->	ELSE
[249]	<CASE_END>	-->	END
[250]	<OTHER_SYMBOL>	-->	OTHERWISE
[251]	<END_SYM_BRK>	-->	END
[254]	<CASE_ELEMENT>	-->	<CASE_LBL_LIST> : <STATEMENT>
[258]	<UNTIL_SYMBOL>	-->	UNTIL

Figure 3

Example of Statement Tree Structure

Full program:

```
1     program test(input, output);
2     begin
3        read(a, b, d, e, f);
4        if a=b then
5           c := d
6        else
7           c := e;
8        writeln(c, f);
9     end.
```

Statement Tree:

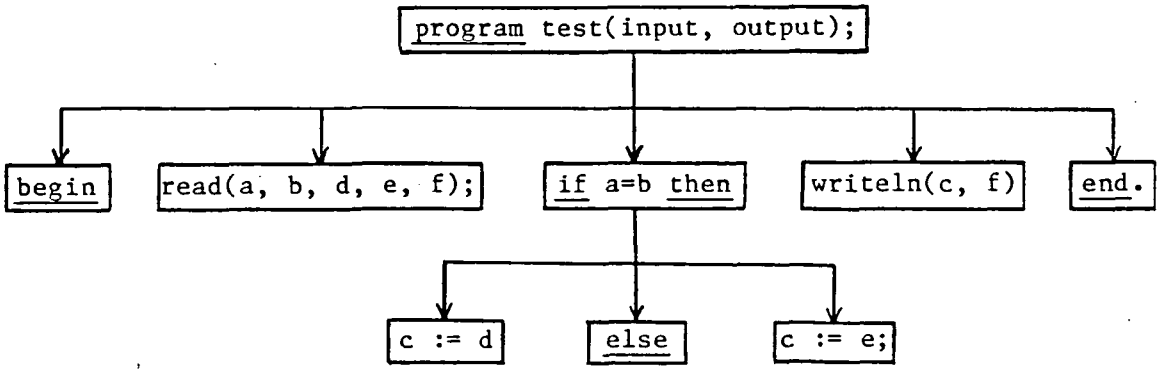


Figure 4

For simple statements, the one statement is sliced and is included in or excluded from the slice accordingly. For compound statements, the general rule is to slice all the sons of the compound statement from rightmost son to leftmost son. If any of the sons have been included in the slice, then the parent compound statement is also included in the

slice.

At the beginning of the slicing procedure, the variables listed in the slicing criterion make up the initial value of the VOI.

4.2. Null Statements

Since null statements contain no variables, they are never included in a slice, and they have no effect on the VOI.

4.3. Assignment Statements

An assignment statement is included in a slice if the target variable on the left hand side of the assignment is currently in the VOI. If the statement is included, then the left hand side variable is removed from the VOI, and all other variables in the statement are added to the VOI. Note that these other variables included all array indices and pointer variables used on the left hand side of the equation as well as all variables used on the right hand side.

For example, if the current VOI is {a, b}, then none of the assignment statements,

```
c := a + b;      h[a] := c;      a↑ := b;
```

would be included. Note that even though the variable "a" is on the left hand side of two of the statements, neither are included because "a" is not being altered by either statement. Given the same current VOI, then the statement,

```
a[x] := c[d] + e↑ - f.g - 1;
```

would be included, and the VOI would be set to {b, c, d, e, f.g, x}.

4.4. Procedure Call Statements

Since the SAGA editor does not contain any scoping and declaration information normally held in a symbol table structure, it is not feasible for the slicer to locate a procedure, substitute parameters, and slice the statements that make up the procedure. In addition, it would be impossible for the slicer to do anything about previously compiled procedures held in a library. Thus, it was decided that all procedure calls would automatically be included in a slice and that all variables in the parameter list would be added into the current VOI.

The only exceptions to this are the procedures write and writeln; these are never included in a slice. It is possible to single out write and writeln because the two never affect the value of any variables, and the PASCAL 6000 grammar separates them from other procedure calls, thus allowing for their unique handling. Note that write and writeln can technically change the value of a variable that happens be a file, but without the symbol table information mentioned above this special case cannot be detected, so it is ignored.

4.5. If Statements

An if-then or if-then-else statement is sliced by first slicing both the then statement and the else statement. If either statement is included, then the if statement is included in the slice, and all variables in the condition expression are added to the VOI. Otherwise, the

if statement is not included in the slice, and the VOI is not altered.

To make the slice obtained from an if-then-else statement syntactically correct, it is sometimes necessary to add punctuation to an else statement that was not included in the slice. In the program slice of Figure 5 , if a semicolon is not added to the end of line 3, the slice will not be syntactically correct. To prevent this problem, it is necessary for the slicer to add a semicolon to an else clause that has none of its statements included, and whose last statement originally had a semicolon. Note that if a semicolon is added to all else clauses that had no statements included, then line 8 of the example slice would make the program syntactically incorrect.

4.6. With Statements

A with statement is sliced by first slicing the statement enclosed by the with. If that statement is included in the slice, then the with statement is included in the slice, and all array indices and pointer variables in the record expression are added to the VOI. Note that the record name is not added, since the VOI is a set of variables, not a set of record names.

For example, if the with statement,

```
with a[c].b do ...
```

is included in a slice, then only the variable c is added to the VOI, because a[...].b is a record name and not a variable. Note that the record name is a prefix of variable names that may be included in the

Punctuation Addition

```
 1      if a = 0 then
 2         b := 1
 3      else
 4         d := 3;
 5      if c = 0 then
 6         if f = 0 then
 7            c := b
 8         else
 9            d := 1
10      else
11         b := b + 3;
```

Slice for criterion of {b, c} at line 11:

```
 1      if a = 0 then
 2         b := 1
 3      else ;          (* semicolon added to original line *)
 5      if c = 0 then
 6         if f = 0 then
 7            c := b
 8         else         (* no semicolon added *)
10      else
11         b := b + 3;
```

Figure 5

VOI, even though the record name itself is not.

4.7. *Case Statements*

A case statement is sliced by first slicing all statements in each individual case element of the statement. If any statement from a case element is included then the case statement is included in the slice, and all variables in the case expression are added to the VOI.

4.8. Begin Statements

A begin statement is sliced by first slicing all statements in the begin-end block. If any statements in the block are included, then the begin statement is included; otherwise the begin statement is not included, and the VOI remains unaltered.

4.9. Loop Statements

The processing of loops presents a special problem for the slicing algorithm since it is impossible to determine how many times a given loop would execute. Figure 6 shows an example where a slice depends on how many times a while loop executes. If the while loop of Figure 6 executes zero times, then the value of "a" after the end of the loop is 0, and there would be no statements from the loop body included in the slice. If the loop executes exactly one time, then the value of "a" after the end of the loop is 1, and the only statement from the loop body to be included in the slice would be statement 7. By the same logic, it can be seen that the final value of "a" and the statements from the loop body to be included in the slice depend on the number of times the loop is executed. Unfortunately, there is no way for the slicer to determine from the code how many times a given loop will execute. Thus, the slicer must make the worst case assumption that a loop will execute enough times so that all possible statements from the loop body will be included in the slice.

Slicing of Loops

```
 1      a := 0;
 2      b := 1;
 3      c := 2;
 4      d := 3;
 5      while (not done) do
 6         begin
 7            a := b;
 8            b := c;
 9            c := d
10         end; (* while loop *)

            pre-loop VOI = {a}
```

Figure 6

By this assumption, the slicer must slice each loop body it encounters over and over until the VOI from the previous pass is not changed after the current pass. Before each pass is made, the VOI that existed before the previous pass is added to current VOI to save the intermediate results of each pass. This is done to account for the possibility that a given loop will execute less than the number of passes necessary to obtain the final slice. When the VOI remains unchanged after any pass, then all statements to be included from the loop body will have been recognized by the slicer. Note that the intermediate slices obtained after each pass are monotonically increasing in size, so the process of slicing the loop over and over must terminate.

Another problem in slicing loops involves the interpretation of having an initial slicing criterion that starts the slice in the middle

of a loop. Should one assume that the loop executes once, twice, or many times before it stops at the statement designated by the slicing criterion? The answer to this is to again assume that loops will execute a large enough number of times so that all statements of interest in the loop body will be included in the slice. The practical result of using this assumption is that there will be no discernible difference between starting the slice in the middle of a loop versus starting the slice at the end of the loop. If the user wanted a slice that assumed that a loop executed a specific number of times, then the user's only recourses are to use the slicer results and walk through these by hand to get the desired results, or to modify the program and reslice.

4.9.1. For Statements

A for statement is sliced by first slicing the body of the loop. If any statements in the loop body are included, then the for statement will be included in the slice, and successive passes of slicing the loop body are then made to calculate all statements in the loop body to be included. After all passes have been made, the implied assignment statement of the for statement is handled by removing the left hand side variable from the final VOI and adding all other variables from the for statement to the final VOI. For example, if the final VOI after all passes is {a, b}, then the statement,

```
for a := (z + 1) to (x * y + 2) do ...
```

will result in the VOI being changed to {b, x, y, z}. Finally, the original pre-loop VOI is added to the final VOI to account for the possi-

bility that the loop may execute zero times.

4.9.2. While Statements

A while statement is sliced by first slicing the body of the loop. If any statements in the loop body are included, then the while statement will be included in the slice, and all variables in the while expression are added to the VOI obtained from the first slice of the loop body. Successive passes of slicing the loop body are then made to calculate all statements in the loop body to be included. In addition, the original pre-loop VOI is added to the VOI obtained after all passes have been made to account for the possibility that the loop may execute zero times.

4.9.3. Repeat Statements

A repeat statement is sliced by first slicing all statements in the loop body. If any statement is included from the loop body, then the repeat statement is included in the slice, and all variables from the until expression are added to the VOI obtained from the first slice of the loop body. Successive passes of slicing the loop body are then made to calculate all statements to include from the loop body. Note that unlike while and for statements, the pre-loop VOI is not added to the final VOI obtained after all passes have been made on the loop body since the loop body in a repeat statement must execute at least one time.

5. IRREGULARITIES OF IMPLEMENTATION

5.1. With Statements

As mentioned above, the SAGA editor does not now contain variable and scoping information normally held in a symbol table structure. Without this information it is impossible for the slicer to determine the correct context for a with statement; the slicer has no information on how to append variable names to a record name given in a with statement. Since a context cannot be determined, the slicer ignores the record name in a with statement, and treats each variable name in any statement inside the with statement as it appears in the original program. For example, in Figure 7 the slicer would interpret each variable in the assignment statement as it appears regardless of the presence of the with statement. The variable "a" would be seen as "a", not "rec.a", and so on.

A user could possibly get around this problem by including both the short and full names of a variable if he wished to use that variable in the slicing criterion. However, one must be careful with this approach because the slicer will not equate the long and short names of the same variable, possibly leading to an incorrect slice being produced.

5.2. Goto Statements

Because of the extreme difficulty in tracing the reverse control flow of a program that has goto statements in it, it was decided that goto statements would be completely ignored if encountered by the

With Statements

```
var
   rec : record
      a : integer;
      b : real
   end;

         .
         .
         .

with rec do
   a := b + c;
```

Figure 7

slicer. The practical effect of this is that any program that contains a goto statement will probably not slice correctly.

6. CONCLUSION

The PASCAL 6000 slicer described above has been implemented to run with the SAGA syntax-directed editor. The slicer is currently running on a CDC Cyber 175 machine.

This thesis has shown that a useful debugging tool, the program slicer, can be implemented fairly easily. Namely, a tool that can remove program statements that are not important to the current program behavior being studied by a programmer. This tool can also be used by programmers trying to maintain unfamiliar programs.

The few problems with designing a program slicer were overcome without too much difficulty. The problem of slicing loops was solved in a reasonably efficient manner. The problems encountered in slicing with statements are due to the unavailability of symbol table information. With this information available the problem would not exist. The problems of goto statements could be solved by using a slightly more complex flow analysis scheme analogous to dataflow analysis. Finally, true dataflow analysis could be used to solve the problems of procedure call statements. Time limitations were the main reason these were not done in the slicer implementation.

There are several methods by which slicers could be implemented. The slicer could easily be implemented as part of a compiler since the compiler already computes all the necessary parse tree and symbol table information needed by a slicer. Since symbolic debugging tools usually

contain dataflow analysis routines, it should not be too difficult to implement a program slicer with this kind of tool. Overall, the program slicer is a potentially useful debugging and maintenance tool that has been shown to be reasonably simple to design and implement.

APPENDIX -- PASCAL 6000 GRAMMAR

Rule #	Production Rule		
[1]	<FULL_PROGRAM>	-->	<PROGRAM> <TOKXEOF>
[2]	<PROGRAM>	-->	<PROGRAM_HEAD> <MAIN_BLOCK>
[3]	<PROGRAM>	-->	<DECLS_WTH_VAL>
[4]	<PROGRAM>	-->	
[5]	<PROGRAM_HEAD>	-->	PROGRAM <TOKXIDENT> ;
[6]	<PROGRAM_HEAD>	-->	PROGRAM <TOKXIDENT> (<EXT_FILE_PART>) ;
[7]	<EXT_FILE_PART>	-->	<EXTERNAL_FILE>
[8]	<EXT_FILE_PART>	-->	<EXT_FILE_PART> , <EXTERNAL_FILE>
[9]	<EXTERNAL_FILE>	-->	<TOKXIDENT> <FILE_CHAR>
[10]	<FILE_CHAR>	-->	/
[11]	<FILE_CHAR>	-->	+
[12]	<FILE_CHAR>	-->	/+
[13]	<FILE_CHAR>	-->	
[14]	<MAIN_BLOCK>	-->	<DECLS_WTH_VAL> <STMT_PART>
[15]	<MAIN_BLOCK>	-->	<STMT_PART>
[16]	<BLOCK>	-->	<DECLARATIONS> <STMT_PART>
[17]	<BLOCK>	-->	<STMT_PART>
[18]	<DECLS_WTH_VAL>	-->	<DECL_ELEMENT>
[19]	<DECLS_WTH_VAL>	-->	<VALUE_INIT_PT>
[20]	<DECLS_WTH_VAL>	-->	<DECLS_WTH_VAL> <DECL_ELEMENT>
[21]	<DECLS_WTH_VAL>	-->	<DECLS_WTH_VAL> <VALUE_INIT_PT>
[22]	<DECLARATIONS>	-->	<DECL_ELEMENT>
[23]	<DECLARATIONS>	-->	<DECLARATIONS> <DECL_ELEMENT>
[24]	<DECL_ELEMENT>	-->	<LABEL_DECL>
[25]	<DECL_ELEMENT>	-->	<CNST_DEF_PART>
[26]	<DECL_ELEMENT>	-->	<TYPE_DEF_PART>
[27]	<DECL_ELEMENT>	-->	<VAR_DECL_PART>
[28]	<DECL_ELEMENT>	-->	<PROC_DECL>
[29]	<DECL_ELEMENT>	-->	<FCN_DECL>
[30]	<LABEL_DECL>	-->	<LABEL_SYMBOL> <LABEL_PART> ;
[31]	<LABEL_SYMBOL>	-->	<LABEL>
[32]	<LABEL_PART>	-->	<LABEL>
[33]	<LABEL_PART>	-->	<LABEL_PART> , <LABEL>
[34]	<LABEL>	-->	<TOKXCONST>
[35]	<CNST_DEF_PART>	-->	<CONST_SYMBOL> <CONST_DEF>
[36]	<CONST_SYMBOL>	-->	CONST
[37]	<CONST_DEF>	-->	<CONST_NAMER> <CONSTANT> ;
[38]	<CONST_DEF>	-->	<CONST_NAMER> <TOKXIDENT> ;
[39]	<CONST_DEF>	-->	<TOKXPLACE> ;
[40]	<CONST_DEF>	-->	<CONST_DEF> <CONST_NAMER> <CONSTANT> ;
[41]	<CONST_DEF>	-->	<CONST_DEF> <CONST_NAMER> <TOKXIDENT> ;
[42]	<CONST_DEF>	-->	<CONST_DEF> <TOKXPLACE> ;
[43]	<CONST_NAMER>	-->	<TOKXIDENT> =

```
[ 44]   <CONSTANT>        -->  <UNSIGNED_NUM>
[ 45]   <CONSTANT>        -->  <TOKXSTRING>
[ 46]   <CONSTANT>        -->  <SIGN> <UNSIGNED_NUM>
[ 47]   <CONSTANT>        -->  <SIGN> <TOKXIDENT>
[ 48]   <SIGN>            -->  +
[ 49]   <SIGN>            -->  -
[ 50]   <UNSIGNED_NUM>    -->  <TOKXCONST>
[ 51]   <TYPE_DEF_PART>   -->  <TYPE_SYMBOL> <TYPE_DEF>
[ 52]   <TYPE_SYMBOL>     -->  TYPE
[ 53]   <TYPE_DEF>        -->  <TOKXPLACE> ;
[ 54]   <TYPE_DEF>        -->  <TYPE_NAMER> <TYPE> ;
[ 55]   <TYPE_DEF>        -->  <TYPE_NAMER> <TOKXIDENT> ;
[ 56]   <TYPE_DEF>        -->  <TYPE_DEF> <TYPE_NAMER> <TYPE> ;
[ 57]   <TYPE_DEF>        -->  <TYPE_DEF> <TYPE_NAMER> <TOKXIDENT> ;
[ 58]   <TYPE_DEF>        -->  <TYPE_DEF> <TOKXPLACE> ;
[ 59]   <TYPE_NAMER>      -->  <TOKXIDENT> =
[ 60]   <TYPE>            -->  <SIMPLE_TYPE>
[ 61]   <TYPE>            -->  <STRUCT_TYPE>
[ 62]   <TYPE>            -->  PACKED <STRUCT_TYPE>
[ 63]   <TYPE>            -->  <POINT_TYPE>
[ 64]   <SIMPLE_TYPE>     -->  ( <SCALAR_TYPE> )
[ 65]   <SIMPLE_TYPE>     -->  <CONSTANT> .. <CONSTANT>
[ 66]   <SIMPLE_TYPE>     -->  <TOKXIDENT> .. <TOKXIDENT>
[ 67]   <SIMPLE_TYPE>     -->  <CONSTANT> .. <TOKXIDENT>
[ 68]   <SIMPLE_TYPE>     -->  <TOKXIDENT> .. <CONSTANT>
[ 69]   <SCALAR_TYPE>     -->  <TOKXIDENT>
[ 70]   <SCALAR_TYPE>     -->  <SCALAR_TYPE> , <TOKXIDENT>
[ 71]   <STRUCT_TYPE>     -->  <ARRAY_TYPE>
[ 72]   <STRUCT_TYPE>     -->  <RECORD_TYPE>
[ 73]   <STRUCT_TYPE>     -->  <SET_TYPE>
[ 74]   <STRUCT_TYPE>     -->  <FILE_TYPE>
[ 75]   <ARRAY_TYPE>      -->  ARRAY [ <INDEX_LIST> ] OF <TYPE>
[ 76]   <ARRAY_TYPE>      -->  ARRAY [ <INDEX_LIST> ] OF <TOKXIDENT>
[ 77]   <INDEX_LIST>      -->  <INDEX_ELT>
[ 78]   <INDEX_LIST>      -->  <INDEX_LIST> , <INDEX_ELT>
[ 79]   <INDEX_ELT>       -->  <SIMPLE_TYPE>
[ 80]   <INDEX_ELT>       -->  <TOKXIDENT>
[ 81]   <RECORD_TYPE>     -->  RECORD <FIELD_LIST> <RECORD_END>
[ 82]   <RECORD_END>      -->  END
[ 83]   <FIELD_LIST>      -->  <FIXED_PART>
[ 84]   <FIELD_LIST>      -->  <FIXED_PART> ; <VARIANT_PART>
[ 85]   <FIELD_LIST>      -->  <VARIANT_PART>
[ 86]   <FIXED_PART>      -->  <RECORD_SECT>
[ 87]   <FIXED_PART>      -->  <FIXED_PART> ; <RECORD_SECT>
[ 88]   <RECORD_SECT>     -->  <VARIABLE_LIST> : <TYPE>
[ 89]   <RECORD_SECT>     -->  <VARIABLE_LIST> : <TOKXIDENT>
[ 90]   <RECORD_SECT>     -->
[ 91]   <VARIABLE_LIST>   -->  <TOKXIDENT>
[ 92]   <VARIABLE_LIST>   -->  <VARIABLE_LIST> , <TOKXIDENT>
[ 93]   <VARIANT_PART>    -->  CASE <TAG> OF <VARIANT_LIST>
```

```
[ 94]   <TAG>              -->  <TOKXIDENT> : <TOKXIDENT>
[ 95]   <TAG>              -->  <TOKXIDENT>
[ 96]   <VARIANT_LIST>     -->  <VARIANT>
[ 97]   <VARIANT_LIST>     -->  <VARIANT_LIST> ; <VARIANT>
[ 98]   <VARIANT>          -->  <CASE_LBL_LIST> : <FLD_LST_PART>
[ 99]   <VARIANT>          -->
[100]   <FLD_LST_PART>     -->  ( <FIELD_LIST> )
[101]   <CASE_LBL_LIST>    -->  <CASE_LABEL>
[102]   <CASE_LBL_LIST>    -->  <CASE_LBL_LIST> , <CASE_LABEL>
[103]   <CASE_LABEL>       -->  <CONSTANT>
[104]   <CASE_LABEL>       -->  <TOKXIDENT>
[105]   <SET_TYPE>         -->  SET OF <SIMPLE_TYPE>
[106]   <SET_TYPE>         -->  SET OF <TOKXIDENT>
[107]   <FILE_TYPE>        -->  FILE OF <TYPE>
[108]   <FILE_TYPE>        -->  FILE OF <TOKXIDENT>
[109]   <POINT_TYPE>       -->  ↑  <TOKXIDENT>
[110]   <VAR_DECL_PART>    -->  <VAR_SYMBOL> <VAR_DECL_LIST>
[111]   <VAR_SYMBOL>       -->  VAR
[112]   <VAR_DECL_LIST>    -->  <VARIABLE_LIST> : <TYPE> ;
[113]   <VAR_DECL_LIST>    -->  <VARIABLE_LIST> : <TOKXIDENT> ;
[114]   <VAR_DECL_LIST>    -->  <VAR_DECL_LIST> <VARIABLE_LIST> :
                                <TYPE> ;
[115]   <VAR_DECL_LIST>    -->  <VAR_DECL_LIST> <VARIABLE_LIST> :
                                <TOKXIDENT> ;
[116]   <VAR_DECL_LIST>    -->  <TOKXPLACE> ;
[117]   <VAR_DECL_LIST>    -->  <VAR_DECL_LIST> <TOKXPLACE> ;
[118]   <VALUE_INIT_PT>    -->  <VALUE_SYMBOL> <VAL_INIT_LIST>
[119]   <VALUE_SYMBOL>     -->  VALUE
[120]   <VAL_INIT_LIST>    -->  <VALUE_NAMER> <VALUE_SPEC> ;
[121]   <VAL_INIT_LIST>    -->  <VAL_INIT_LIST> <VALUE_NAMER>
                                <VALUE_SPEC> ;
[122]   <VALUE_NAMER>      -->  <TOKXIDENT> =
[123]   <VALUE_SPEC>       -->  [ <SET_VAL_LIST> ]
[124]   <VALUE_SPEC>       -->  [ ]
[125]   <VALUE_SPEC>       -->  NIL
[126]   <VALUE_SPEC>       -->  <TOKXIDENT>
[127]   <VALUE_SPEC>       -->  <CONSTANT>
[128]   <VALUE_SPEC>       -->  ( <VALUE_LIST> )
[129]   <VALUE_SPEC>       -->  <TOKXIDENT> ( <VALUE_LIST> )
[130]   <SET_VAL_LIST>     -->  <SET_VAL_ELMTS>
[131]   <SET_VAL_LIST>     -->  <SET_VAL_LIST> , <SET_VAL_ELMTS>
[132]   <SET_VAL_ELMTS>    -->  <CONSTANT>
[133]   <SET_VAL_ELMTS>    -->  <TOKXIDENT>
[134]   <SET_VAL_ELMTS>    -->  <CONSTANT> .. <CONSTANT>
[135]   <SET_VAL_ELMTS>    -->  <TOKXIDENT> .. <TOKXIDENT>
[136]   <VALUE_LIST>       -->  <CONSTANT> OF <VALUE_SPEC>
[137]   <VALUE_LIST>       -->  <TOKXIDENT> OF <VALUE_SPEC>
[138]   <VALUE_LIST>       -->  <VALUE_SPEC>
[139]   <VALUE_LIST>       -->  <VALUE_LIST> , <CONSTANT> OF
                                <VALUE_SPEC>
```

```
[140]   <VALUE_LIST>      -->  <VALUE_LIST> , <TOKXIDENT> OF
                               <VALUE_SPEC>
[141]   <VALUE_LIST>      -->  <VALUE_LIST> , <VALUE_SPEC>
[142]   <VALUE_LIST>      -->
[143]   <PROC_DECL>       -->  <PROC_HEADING> ; <PROC_FCN_FOLL> ;
[144]   <FCN_DECL>        -->  <FCN_HEADING> ; <PROC_FCN_FOLL> ;
[145]   <PROC_FCN_FOLL>   -->  <BLOCK>
[146]   <PROC_FCN_FOLL>   -->  FORWARD
[147]   <PROC_FCN_FOLL>   -->  EXTERN
[148]   <PROC_FCN_FOLL>   -->  FORTRAN
[149]   <PROC_HEADING>    -->  PROCEDURE <TOKXIDENT> <PARM_LIST>
[150]   <FCN_HEADING>     -->  FUNCTION <TOKXIDENT> <PARM_LIST> :
                               <TOKXIDENT>
[151]   <PARM_LIST>       -->  ( <FRML_PARM_LST> )
[152]   <PARM_LIST>       -->
[153]   <FRML_PARM_LST>   -->  <FRML_PARM_SCT>
[154]   <FRML_PARM_LST>   -->  <FRML_PARM_LST> ; <FRML_PARM_SCT>
[155]   <FRML_PARM_SCT>   -->  VAR <PARM_GROUP>
[156]   <FRML_PARM_SCT>   -->  <PARM_GROUP>
[157]   <FRML_PARM_SCT>   -->  <PROC_HEADING>
[158]   <FRML_PARM_SCT>   -->  <FCN_HEADING>
[159]   <PARM_GROUP>      -->  <VARIABLE_LIST> : <TOKXIDENT>
[160]   <PARM_GROUP>      -->  <VARIABLE_LIST> : DYNAMIC <TOKXIDENT>
[161]   <STMT_PART>       -->  <BEGIN_SYMBOL> <BLOCK_BODY> <END_SYMBOL>
[162]   <BEGIN_SYMBOL>    -->  BEGIN
[163]   <END_SYMBOL>      -->  END
[164]   <BLOCK_BODY>      -->  <STMT_LIST>
[165]   <STMT_LIST>       -->  <STATEMENT>
[166]   <STMT_LIST>       -->  <STMT_LIST> ; <STATEMENT>
[167]   <STATEMENT>       -->  <MATCH_STMT>
[168]   <STATEMENT>       -->  <UNMATCH_STMT>
[169]   <STATEMENT>       -->  <LABEL> : <MATCH_STMT>
[170]   <STATEMENT>       -->  <LABEL> : <UNMATCH_STMT>
[171]   <ASSIGN_STMT>     -->  <VARIABLE> := <EXPRESSION>
[172]   <ASSIGN_STMT>     -->  <TOKXIDENT> := <EXPRESSION>
[173]   <VARIABLE>        -->  <VARIABLE> [ <EXPRESS_LIST> ]
[174]   <VARIABLE>        -->  <TOKXIDENT> [ <EXPRESS_LIST> ]
[175]   <VARIABLE>        -->  <VARIABLE> . <TOKXIDENT>
[176]   <VARIABLE>        -->  <TOKXIDENT> . <TOKXIDENT>
[177]   <VARIABLE>        -->  <VARIABLE> ↑
[178]   <VARIABLE>        -->  <TOKXIDENT> ↑
[179]   <EXPRESS_LIST>    -->  <EXPRESSION>
[180]   <EXPRESS_LIST>    -->  <EXPRESS_LIST> , <EXPRESSION>
[181]   <EXPRESSION>      -->  <SIMPLE_EXPRES>
[182]   <EXPRESSION>      -->  <SIMPLE_EXPRES> <REL_OP> <SIMPLE_EXPRES>
[183]   <EXPRESSION>      -->  <TOKXPLACE>
[184]   <REL_OP>          -->  =
[185]   <REL_OP>          -->  <>
[186]   <REL_OP>          -->  <
[187]   <REL_OP>          -->  <=
```

```
[188]   <REL_OP>          -->  >=
[189]   <REL_OP>          -->  >
[190]   <REL_OP>          -->  IN
[191]   <SIMPLE_EXPRES>   -->  <TOKXIDENT>
[192]   <SIMPLE_EXPRES>   -->  <FACTOR>
[193]   <SIMPLE_EXPRES>   -->  <TOKXSTRING>
[194]   <SIMPLE_EXPRES>   -->  <SIGN> <FACTOR>
[195]   <SIMPLE_EXPRES>   -->  <SIGN> <TOKXIDENT>
[196]   <SIMPLE_EXPRES>   -->  <SIMPLE_EXPRES> <ARITH_OP> <FACTOR>
[197]   <SIMPLE_EXPRES>   -->  <SIMPLE_EXPRES> <ARITH_OP> <TOKXIDENT>
[198]   <ARITH_OP>        -->  +
[199]   <ARITH_OP>        -->  -
[200]   <ARITH_OP>        -->  OR
[201]   <ARITH_OP>        -->  *
[202]   <ARITH_OP>        -->  /
[203]   <ARITH_OP>        -->  DIV
[204]   <ARITH_OP>        -->  MOD
[205]   <ARITH_OP>        -->  AND
[206]   <FACTOR>          -->  <VARIABLE>
[207]   <FACTOR>          -->  <UNSIGNED_NUM>
[208]   <FACTOR>          -->  NIL
[209]   <FACTOR>          -->  ( <EXPRESSION> )
[210]   <FACTOR>          -->  [ <ELEMENT_LIST> ]
[211]   <FACTOR>          -->  [ ]
[212]   <FACTOR>          -->  <TOKXIDENT> ( <ACT_PARM_LIST> )
[213]   <FACTOR>          -->  NOT <FACTOR>
[214]   <FACTOR>          -->  NOT <TOKXIDENT>
[215]   <ACT_PARM_LIST>   -->  <EXPRESSION>
[216]   <ACT_PARM_LIST>   -->  <ACT_PARM_LIST> , <EXPRESSION>
[217]   <ELEMENT_LIST>    -->  <ELEMENT>
[218]   <ELEMENT_LIST>    -->  <ELEMENT_LIST> , <ELEMENT>
[219]   <ELEMENT>         -->  <EXPRESSION>
[220]   <ELEMENT>         -->  <EXPRESSION> .. <EXPRESSION>
[221]   <PROC_STMT>       -->  <TOKXIDENT> ( <ACT_PARM_LIST> )
[222]   <PROC_STMT>       -->  <TOKXIDENT>
[223]   <PROC_STMT>       -->  WRITE ( <SPECIAL_PARMS> )
[224]   <PROC_STMT>       -->  WRITELN ( <SPECIAL_PARMS> )
[225]   <PROC_STMT>       -->  WRITELN
[226]   <SPECIAL_PARMS>   -->  <SIMPLE_EXPRES> <FIELD_WIDTH>
[227]   <SPECIAL_PARMS>   -->  <SPECIAL_PARMS> , <SIMPLE_EXPRES>
                               <FIELD_WIDTH>
[228]   <FIELD_WIDTH>     -->  : <SIMPLE_EXPRES> : <SIMPLE_EXPRES>
[229]   <FIELD_WIDTH>     -->  : <SIMPLE_EXPRES>
[230]   <FIELD_WIDTH>     -->
[231]   <MATCH_STMT>      -->  <ASSIGN_STMT>
[232]   <MATCH_STMT>      -->  <PROC_STMT>
[233]   <MATCH_STMT>      -->  GOTO <LABEL>
[234]   <MATCH_STMT>      -->  BEGIN <STMT_LIST> <END_SYM_BRK>
[235]   <MATCH_STMT>      -->  <WHILE_STMT>
[236]   <MATCH_STMT>      -->  <REPEAT_STMT>
```

```
[237]   <MATCH_STMT>      -->  <FOR_STMT>
[238]   <MATCH_STMT>      -->  <WITH_STMT>
[239]   <MATCH_STMT>      -->  <CASE_STMT>
[240]   <MATCH_STMT>      -->  <EMPTY_STMT>
[241]   <MATCH_STMT>      -->  <TOKXPLACE>
[242]   <MATCH_STMT>      -->  IF <EXPRESSION> THEN <MATCH_STMT> <ELSE>
                               <MATCH_STMT>
[243]   <EMPTY_STMT>      -->
[244]   <UNMATCH_STMT>    -->  IF <EXPRESSION> THEN <STATEMENT>
[245]   <UNMATCH_STMT>    -->  IF <EXPRESSION> THEN <MATCH_STMT> <ELSE>
                               <UNMATCH_STMT>
[246]   <ELSE>            -->  ELSE
[247]   <CASE_STMT>       -->  CASE <EXPRESSION> OF <CASE_LIST>
                               <CASE_END>
[248]   <CASE_STMT>       -->  CASE <EXPRESSION> OF <CASE_LIST>
                               <OTHER_SYMBOL> <STMT_LIST> <CASE_END>
[249]   <CASE_END>        -->  END
[250]   <OTHER_SYMBOL>    -->  OTHERWISE
[251]   <END_SYM_BRK>     -->  END
[252]   <CASE_LIST>       -->  <CASE_ELEMENT>
[253]   <CASE_LIST>       -->  <CASE_LIST> ; <CASE_ELEMENT>
[254]   <CASE_ELEMENT>    -->  <CASE_LBL_LIST> : <STATEMENT>
[255]   <CASE_ELEMENT>    -->
[256]   <WHILE_STMT>      -->  WHILE <EXPRESSION> DO <STATEMENT> OD
[257]   <REPEAT_STMT>     -->  REPEAT <STMT_LIST> <UNTIL_SYMBOL>
                               <EXPRESSION>
[258]   <UNTIL_SYMBOL>    -->  UNTIL
[259]   <FOR_STMT>        -->  FOR <FOR_LIST> DO <STATEMENT> OD
[260]   <FOR_LIST>        -->  <TOKXIDENT> := <EXPRESSION> TO
                               <EXPRESSION>
[261]   <FOR_LIST>        -->  <TOKXIDENT> := <EXPRESSION> DOWNTO
                               <EXPRESSION>
[262]   <FOR_LIST>        -->  <TOKXPLACE>
[263]   <WITH_STMT>       -->  WITH <RCD_VAR_LIST> DO <STATEMENT> OD
[264]   <RCD_VAR_LIST>    -->  <VARIABLE>
[265]   <RCD_VAR_LIST>    -->  <TOKXIDENT>
[266]   <RCD_VAR_LIST>    -->  <RCD_VAR_LIST> , <VARIABLE>
[267]   <RCD_VAR_LIST>    -->  <RCD_VAR_LIST> , <TOKXIDENT>
[268]   <RCD_VAR_LIST>    -->  <TOKXPLACE>
[269]   <TOKXIDENT>       -->  TOKXIDENT
[270]   <TOKXSTRING>      -->  TOKXSTRING
[271]   <TOKXCONST>       -->  TOKXCONST
[272]   <TOKXPLACE>       -->  TOKXPLACE
[273]   <TOKXEOF>         -->  TOKXEOF
```

REFERENCES

[1] Campbell, R. H., and Richards, P. G., "SAGA, A System to Automate the Management of Software Production", Proceedings of the 1981 National Computer Conference, Chicago, May 1981.

[2] Weiser, Mark, "Deducing Sequential from Parallel Behavior", Submitted for Publication, 1982.

[3] Weiser, Mark, "Program Slicing", Proceedings of the 5th International Conference on Software Engineering, March 1981.

[4] Weiser, Mark, "The Slicing Abstraction in Software Production and Maintenance", Unpublished Paper, January, 1979.

SAGA Mid-Year Report
Appendix I
June Demonstration Plan

Roy H. Campbell
George M. Beshers

Department of Computer Science
University of Illinois
Urbana-Champaign
Illinois, 61801-2987

October, 1983

JUNE DEMONSTRATION PLAN

1. OVERVIEW

The June demonstration will be designed to exhibit the SAGA editor as the primary user interface to the SAGA system. The principal points to be covered are the reliability, language independence, user interface, and semantic checking ability of the editor. To establish user confidence the editor must be reliable; that is, the results of any editing operation must be predictable and must conform to the specification of the editing commands. For ease of use, each phase and facility of the SAGA system must be independent of the language being edited. For example, error messages detected by the language-specific components of the SAGA system must be reported in an accurate and appropriate but language-independent manner. The user interface must support textual as well as structured and language-oriented editing commands; the same editor interface should allow the editing of English text as well as software specifications and programs. The editor should detect as many language-dependent errors as possible so that the user spends less time in a compile-edit cycle.

2. DEMONSTRATION PLAN

The goal of the June Demonstration will be to show that the SAGA editor can support software development activity. The demonstration will show that the editor:

- is reliable. It will perform editing commands according to their specifications.
- is reasonably fast. Although no attempt will be made to optimize the editor, the demonstration will show that the editor commands have a response time which is reasonable for an interactive software development tool.
- has a uniform response time.
- is predictable. The result of performing various editor commands in combination should be obvious from the function of those commands.

- is documented. Full documentation of the commands as they exist for the demonstration will be available both in the form of a manual and in the form of on-line help.
- is crash resistant. Edits will be performed on a temporary copy of a SAGA files. The permanent copy will only be updated on a successful conclusion of the editing session. This permanent file will be used for back-up if the system crashes during an editing session (provided, of course, that the system does not destroy the permanent file's contents).

Various tests will be shown to demonstrate the various editing facilities. These tests include:

- a Pascal, ADA, C, and text editor.
- positioning of the cursor.
- reading a large (1000 line) program into the editor from a text file.
- interactively reading and correcting a program read in from a text file.
- modifying a large (1000 line) program and demonstrating the incremental reparse using the debug features.
- screen-editing a Pascal program. Characters will be inserted, deleted, and replaced using cursor positioning and direct data entry at the cursor position.
- command-editing a Pascal program. Lexical components of the Pascal program will be inserted, deleted, replaced, and copied. Commands that use line, syntactic and semantic properties of the program to select components of the program for modification will be demonstrated. Use of these commands will not require knowledge of the formal specifications of the language (that is, the productions of the grammar and semantic attributes of the productions).
- searching a Pascal program. The cursor will be positioned by a search command.
- verification of the editor using an editor test suite. A test suite of commands will be read and executed by the editor command interpreter to verify the correct implementation of the various editing commands.
- verification of the file system interface by executing repeated modifications to a file.
- verification of the correct parsing of a program being edited. Various incorrect Pascal programs will be read by the editor and it will correctly determine the lexical, syntactic, and some semantic errors

within those programs. The Pascal verification suite will be used as a basis for these tests.

- correct suspension of the incremental parser when an error is detected. The editor should suspend attempting to parse an incorrect program and should return to screen-edit and command mode.
- non-formatting capability. The editor should read program text and write program text without reformatting that text.
- formatting capability. The editor should reformat programs that are entered. This reformatting should provide suitable display of comments.
- simple text editing. The editor will be demonstrated editing simple text such as might be found in formal documentation of a program.
- structured editing. The editor will be used with a set of user-defined command extensions that enable the user to develop his program by top-down development.

In addition to these basic facilities, the editor *may* have the additional features, if time permits, which will allow a demonstration of the following.

- Display of differences between two versions.
- Application of the undo command to one or more differences.
- Use of a Berkeley Pascal Make facility to reduce compilation time.

3. DETAILS OF FEATURES TO BE DEMONSTRATED

3.1. Reliability and Robustness

Reliability is perhaps the single most important requirement in an editor. For the June demonstration, we plan to have a verification suite for the editor. The suite will include: all the editor primitive commands, a comprehensive set of command sequences, and a few examples of command procedures. Reliability will also be enhanced by a set of built in consistency checks, the frequency of which can be controlled by the user.

The June demonstration will include a simple back-up facility for SAGA edited files. This back-up scheme will consist of automatically copying a file to be edited into temporary storage. This temporary file will be edited by the user. At the end of the

editing session, the temporary file will be copied back into permanent storage. If a system crash occurs during editing, any changes that have been made to the file will be lost. We are currently considering more sophisticated recovery schemes which will allow crash recovery capability during editing. One proposal is to maintain a session file (trace of user activity), and a backup copy of the parse file. Recovery could also be integrated with the diff and undo schema. Another alternative is to supply additional information in the parse file to facilitate recovery. We have not actually made a design decision on this problem and it will not form part of the June Demonstration.

3.2. Language Independence

Language independence will be demonstrated by having a few different editors available for demonstration. In this way we will demonstrate the flexibility of the editor generation tools and of the editor's command language. Examples of languages for which we believe the SAGA editor can be very useful are Pascal, Ada, C, Euclid, CLU: that is, languages with formal definitions, and static scoping. Thus textual editing should be the same regardless of language. Structural editing should use the same commands, but on different types of sub-trees. The semantic errors and other information should be displayed in a consistent fashion.

3.3. User Interface

Because software developers spend a great deal of time editing, the user interface to the SAGA editor is very important. As soon as full-screen editors became available to the SAGA project team, everybody started using them. We believe from our own experience, therefore, that full-screen editing is more desirable than line editing.

The editor used at the demonstration will have a new user-extensible command language which has textual and tree manipulation primitives and cursor positioning commands. The ability to be able to define new commands, possibly as macros, is an important mechanism for allowing the user or group of users to devise appropriate editing facilities which they consider user friendly and appropriate.

3.3.1. Full Screen Display

After some discussion within the SAGA group, work is underway to give the SAGA editor a full screen display. We believe that the advantages of having a full screen editor are worth the development effort and the CPU overhead. The requirements for the display are similar to requirements for the displays of other full-screen editors. Thus, we plan to use either the *curses* package which comes with the Berkeley 4.1b UNIX® system or the Maryland window package (which already supports windows).

3.3.2. Command Language

The command language is currently in the design phase of development. The goal with the language is to produce a powerful and extensible tool for editing both the text and the associated parse trees. The language must permit the user to examine the results of the semantics associated with his program (particularly the symbol table error messages), in whichever language he is working. Finally the command set must be devised as a language, not just a set of commands, which a user can learn and *remember* without constant reference to the on-line help facilities or manuals.

The ability of the SAGA editor to do textual manipulation of a program should be comparable to other full screen editors. To test this criterion a the following basic text grammar will be used to generate a SAGA editor for simple text. (Note: LETTER, DIGIT, EOLN, EOF are considered to be predefined.)

```
%terminal
word =
        (LETTER (LETTER | DIGIT)*)
      | DIGIT+
      | '!' | '"' | '#' | ' | '%' | '&' | '\'' ...  ;
            (* All valid ASCII punctuation *)
eoftoken = EOF  ;
space = ' ' | '\09'  ;
eolntoken = EOLN  ;
%parser_skip space eolntoken  ;
```

```
                (* White space should not be part of the grammar *)
%grammar
file
        : word_list eoftoken  ;

word_list
        : word_list word
        |  ;
```

In Olorin, end-of-line (EOLN) is the ASCII character, but end-of-file (EOF) is the physical end of the file. Thus, this grammar takes zero or more words where a word is a normal identifier, an integer, or a punctuation mark. This simple grammar will be used to demonstrate the application of the editor to English text.

3.3.3. Formatting and Comments

The SAGA editor is currently too restrictive when formatting the program text. In future versions of the editor, the amount of automatic formatting will be at the discretion of the user. This includes the possibility of preserving the input format as would be done with a conventional editor.

At the time of the demonstration, comments will be handled correctly. The user will be able to do textual editing within comments. Multiple line comments will be allowed.

3.4. Symbol Table and other Semantics

The preliminary work for building a parser generator to support attribute grammars is over half done. These attribute grammars will interface with the symbol table to provide some semantic checking by the SAGA editor. Errors which the SAGA editor for Pascal will catch include:

Undeclared Identifiers,
Identifiers used improperly, i.e., type where constant is required,
Type checking of expressions,
Type checking of parameter lists.

Thus most compile time errors will be caught by the SAGA editor. Examples of errors

that will probably not be caught are:

goto into structured statements,
memory allocation overflow,
any run-time check.

Any semantic evaluation which is performed by the editor at demonstration time will not be optimized. We will restrict the semantics for the Pascal SAGA editor so that most evaluations do not propagate any great distance through the evaluation tree.

4. SUMMARY

In June, we hope to provide a convincing demonstration of the SAGA editor. The demonstration will show the features of the editor that are reliable, and several new features which may be less dependable. The demonstration will include examples that show how the editor supports software engineering.

SAGA Mid-Year Report

Appendix J

Summary of UNIX Software Engineering Support

Roy H. Campbell

Department of Computer Science
University of Illinois
Urbana-Champaign
Illinois, 61801-2987

October, 1983

1. UNIX as an Environment for Software Engineering

UNIX is used by many organizations as the operating system environment within which to support software engineering. In this short paper, we outline some of the major reasons why UNIX has become adopted for such use. The reasons we provide are not comprehensive, but they summarize the benefits that have been experienced from using UNIX to support a wide range of research and software development activities within the Department of Computer Science at the University of Illinois. The experience represents accumulated knowledge based on the use of nine networked UNIX systems by at least 300 users (faculty, research assistants, and secretaries).

It is sometimes not obvious to newcomers to UNIX why UNIX users are so pleased with the operating system. Certainly, UNIX is a system that must be used to be appreciated. Many of the facilities and capabilities of the system do not exist in other operating systems; or if they do exist, they are often not sufficiently well-integrated with other system provisions to allow the power of UNIX. This power is not obvious to the casual observer of the UNIX system.

UNIX is particularly useful to the software engineer because it provides many basic tools with which software can be developed, organized, and maintained. In particular, the UNIX environment encourages the user to prototype (that is, construct an example implementation from the various software components available on UNIX), build tools using standard utilities (for example: AWK - a powerful pattern matching editor; YACC - a parser generator; LEX - a lexical analyzer generator) and tailor the UNIX environment to support a software development project.

UNIX is based on a hierarchical organization of two fundamental system concepts: system resources (files, directories, devices, data, and programs) and processes. The hierarchies provide locality of reference and a scheme for structuring user programs and data.

The file hierarchy permits the creation of an unlimited number of directories and files of arbitrary sizes allowing users to freely exploit the file system itself as a means to

organize software modules and systems. For example, a user can create a separate directory for each set of related files with which he is working. A program also can be put in its own directory, and split into small files that can be separately compiled. Because small files of a few words are economical, UNIX encourages the user to develop personalized libraries of useful job control procedures. Such libraries are very important for the construction of software engineering environments as they allow the system to be tailored to providing better support. The file system also supports dating user files with the last access time and modification time. These times may be accessed easily through UNIX utilities and are exploited extensively in source and object control software (Source Code Control System, Make).

The process structure of UNIX permits users to create background jobs as well as foreground jobs – all with the same operating system commands. Neither the job control language nor the actual programs being run need to be changed. UNIX also provides convenient and powerful ways to control background jobs. The progress of background jobs can be examined and the jobs may be terminated, suspended or moved to lower priorities in the system. The process hierarchy allows a collection of concurrent processes to be terminated by the user if the user so desires.

A great strength of the UNIX system is the uniform way in which it supports file and device access. All such access is organized like an indexed file in which the record is a byte. Operations on files and devices include read, write, and seek a record. Devices which do not support a particular operation (a line printer will not support a read or seek) return standardized error codes. Thus, all I/O from a program is uniform, no matter to which device the I/O is directed. This permits simple redirection of I/O from one program to a file, a device, or to the I/O of another program (a UNIX filter).

Communication between processes is simple and understandable. Each process may communicate with other processes through pipes. A pipe is a specialized form of I/O which is buffered in memory and provides a stream of bytes from the output of one process to the input of another. Alternatively, processes may also transmit brief

messages through signals, a form of software interrupt. These two forms of communication permit the construction of concurrent pipelines of processes.

Processes are very economical on UNIX. User processes tend to be small and implement a particular function. Because of the pipeline ability, it is customary to design a software system as a collection of small interacting processes. These processes may be reused in different ways by reconnecting their pipes in a different manner. This encourages the development of many small, but useful, software modules.

UNIX I/O to terminals is organized as an ASCII byte stream. UNIX has very good I/O control facilities built into its kernel allowing many different terminals to be connected to a system. The I/O also provides a very useful downloading capability. A system running UNIX is particularly easy to interface to other, small systems (like microprocessor systems on a board) and the downloading facility permits fast development of software for that small system. The I/O support has also encouraged the use of dial-in lines and the networking of UNIX systems.

UNIX is written in C, a reasonably high-level programming language. C programs are small and modular. The language encourages separate compilation. The separate compilation and linking facilities are a major reason why UNIX can be run in a self-supporting manner on small machines. C supports the user making a small change in one software module of a large system by allowing that module to be recompiled and relinked into the system very quickly. This is very important for prototyping and the construction of particular tools from existing software modules.

UNIX has a very powerful command language supported by a command interpreter called the shell. The shell permits the creation of other shells recursively. Each shell provides an environment for executing shell commands. The ease with which new instances of the shell can be created allows easy construction of different environments. For example, the UNIX editor may be invoked within a shell, and within the editor another UNIX shell might be instantiated. The first editor may edit a program listing and the second shell may be used to invoke a dynamic debugging facility for that

program.

Perhaps the greatest contribution UNIX can make to providing a software engineering environment comes from the wealth of portable software that has been written for UNIX. Most of this software is written in a modular fashion, with separate compilation a concern, as collections of small processes in C. We list some of the programs we have found very useful:

nroff A text formatting program. Important because it encourages good clear documentation of software in manual and technical report styles. Most of the documentation for UNIX is prepared for processing by this program.

troff A type setting program. Various forms of this program allow most type setting equipment to be used for documentation.

man A program which retrieves manual pages for on-line documentation of UNIX and user facilities.

grep A program for quick searches of text files for particular patterns of characters.

style A program which checks the style of English prose.

diction A similar program to style.

spell A program which checks the spellings and endings of words. Spell has a 30,000 word dictionary.

sccs A program to manage source programs and text. Source programs can be logged out of a library. A record is kept of which user has logged out which programs. Locks may be placed on particular programs to prevent them from being logged out and changed. Sccs supports version control and records differences between versions in the form of forward differences.

rcs A similar program to sccs which supports version control and keeps backward differences between versions.

sdb A symbolic debugging system for C.

make A program that can be used to determine which modules out of a large collection of modules require reprocessing (recompiling) because of changes made to those modules since the last execution of make.

mail A program to transmit mail from one user to another.

notes A program which supports a distributed (networked) discussion forum between users. Notes have been used for documenting bugs and fixes, designs, programs, hardware, projects, agendas and minutes, etc..

vi Screen Editor.

emacs	Screen Editor with programmable interface.
ed	Simple line editor.
ex	Line editor.
xed	Line editor with interactive cursor/character commands.
sort	Will sort on any field.
awk	String processing language.
uucp	A facility to transfer files between UNIX systems and execute remote commands on one UNIX from another. Uucp may be used over any asynchronous line that can be used to transmit bytes.

SAGA Mid-Year Report

Appendix K

Theorem Prover Reviews

David H. Hammerslag

Department of Computer Science

University of Illinois

Urbana-Champaign

Illinois, 61801-2987

October, 1983

1. INTRODUCTION

As preliminary research for the proof management system, several theorem provers have been informally investigated. This is a subjective survey of some these theorem provers. Note that, unfortunately, some of the theorem provers discussed have not been used by the author; the only exposure to them has been from the literature.

With the proof management system in mind, several aspects of the individual provers were considered. One of the most important criterion was the speed of the prover. A prover that is to be useful in a proof management system must respond quickly. The "intelligence" of the prover is a factor that, while important, is often at odds with the speed criterion. We would like a prover to be clever enough to prove nontrivial propositions. The ease of use of the prover was also examined. Ease of use has two aspects: the ease with which a user can use the prover by itself, and the ease with which it could be utilized in the proof management system. A final criterion was the amount of feedback that the theorem prover returns or could return to the user or the proof management system in the event of a failed proof. Four theorem provers will be discussed. They fall into two broad categories: those which interactively accept and respond to user input, and those which take a proposition and attempt to prove it in a batch mode.

2. Interactive Theorem Provers

To use an interactive prover, the user types input, either propositions to be proven or directives, to the system and then carries on a dialogue with the system until the proposition is proved. Two of the interactive theorem provers investigated are: the UT interactive theorem prover [Ble83], and Edinburgh LCF [Gor79].

2.1. The UT Theorem Prover

The UT interactive theorem prover is part of a theorem proving system. It attempts to prove propositions given to it by using a system of natural deduction.

Because it uses a natural deduction system, the steps that the prover makes are more likely to be understood by the user than are the steps used by a refutation style prover. When the prover cannot prove a proposition, the user can ask the theorem prover where it failed, then attempt to guide the prover over that spot via directives and proof steps.

As it is already part of a system, this prover is not likely to be usefully adapted to the proof management system. It is however, useful and interesting to compare the UT system with the proposed management system. Briefly, the prover does not allow the user to arbitrarily step around in the proof tree, nor does it allow the construction of arbitrary proof trees. Both of these features will be available in the proof management system. On the other hand, the UT theorem prover offers more sophisticated proof techniques such as subgoaling and rewriting; it is unlikely that these features will be directly accessible to the user of the proof management system.

2.2. LCF

Edinburgh LCF (Logic for Computable Functions) is an extension of the LCF system originally written at Stanford University. It can be used as either an interactive proof checker (in the spirit of Stanford LCF), or as a programmable theorem prover. In its latter use, LCF is rather difficult for the novice to use. LCF has its own language, ML (Meta Language), for communicating with the system. ML is a very strongly typed language, and one of the types is theorem. Using ML, a user can write procedures which map forms or theorems to theorems. The user defines these "tacticals" which are applied, under the user's control, to the proposition to be proved. In this use, and as an interactive proof checker, the LCF user is hampered by the difficult syntax and strong typing of ML. Due to its close ties with its user defined theories and ML, it is unlikely that LCF could be used in another system.

Comparing LCF with the proof management system is a difficult task. In LCF the user is really trying to program a theorem prover with his own heuristics, while in the proof management system the user is viewing the theorem prover as a decision

procedure. When LCF is being used simply as a proof checker, it is similar to the proof management system without the flexibility gained by generalized tree editing.

3. Batch Theorem Provers

A batch theorem prover is what one is likely to think of when one thinks of a theorem prover; i.e., the user presents a proposition to the theorem prover without the ability to guide the prover while the proof is underway. Of the batch theorem provers researched, two are surveyed here: the Robust theorem prover [Gre83], and VERUS [Com83].

3.1. Robust

The Robust theorem prover is being developed at the University of Illinois. This prover uses a variation of the resolution method, with controlled forward and backward chaining. Employing this method gives the prover two advantages: it is able to prove theorems of moderate difficulty, and it is very fast.

Because this is a resolution style prover, no useful information is available to the user if a proof fails. However, as the prover becomes more powerful, this becomes less of a concern. Since this prover is still being developed, it is awkward as a stand alone prover (all of its input must be in clause form). On the plus side, because the prover is still under development and is being developed locally, interfacing the prover to the proof management system would be greatly simplified.

3.2. VERUS

The VERUS theorem prover uses the Hintikka tableau method of theorem proving and therefore is very fast. It was written with a specific purpose in mind: proving theorems about state machine specifications. VERUS is easy to use as a stand alone prover, as this is its intended use. It is really two programs, a parser which takes a high level, structured proof outline and produces input to the prover, and the prover itself. Due to this two phase approach, it would probably be relatively easy to use VERUS in a proof management system.

There were also negative aspects to VERUS. Due to the fact that the tableau method is used, the prover must be supplied with constants for instantiation. While this could be done by the management system, it would complicate the interface. VERUS does not appear to be very intelligent. For proofs of any consequence, a substantial user-supplied proof outline is required. Further, VERUS performs especially poorly outside of state machine specification proofs. Finally, because VERUS was written by a private corporation, it would be difficult to get the source code, and therefore difficult to work with.

References

[Ble83]
W. W. Bledsoe, "The UT Interactive Prover," The University of Texas, Austin, Austin (April 1983).

[Com83]
Compion Corp., "VERUS User's Guide," Compion Corporation, Urbana, Ill (January 1983).

[Gor79]
Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth, *Edinburgh LCF,* Springer-Verlag Lecture Notes in Computer Science, No. 78, New York (1979).

[Gre83]
Steven Greenbaum, Paul O'Rorke, and David Plaisted, "Comparisons of Abstraction and a Robust Resolution Strategy," Department of Computer Science, University of Illinios, Urbana, Illinios (September 22, 1983).