01 Dec 1993

A Syntax-Directed Editor for Borland's Turbo Pascal

John Gatewood Ham

Thomas J. Sager

A Syntax-Directed Editor for Borland's Turbo Pascal

J. Ham* and T. Sager

CSC-93-34

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401

*This report is substantially the M. S. thesis of the first author, completed December 1993

ABSTRACT

This study details the design and implementation of the LSD program, a syntax-directed editor for use in editing the source code for Borland's Turbo Pascal. LSD is a dual-mode editor which allows both traditional text editing and also grammar-based editing. LSD promotes better programming for novice users by allowing the user to edit the program with a graphical representation of a parse tree. A list of syntactically correct choices is displayed at each point where a choice must be made in the structure of the program. Since only these choices are available, no syntax errors are possible. For more advanced users, the ability to take an existing program and display the syntax tree will help show the underlying structure of the Pascal language grammar.

ACKNOWLEDGEMENTS

I would like to express my appreciation to my committee members, Drs. Thomas Sager, George Zobrist, and Wayne Bledsoe, for their patience.

I would also like to thank Chuck Kincy, John Stone, Dan Miner, Charles Hamilton, Adam Lewis, Paul Belk, Yi-Wan Wang, Hong-Him Lim, Chun-Kee Ng, Pei-Yu Li, Mihai Sirbu, Eugene Ham, Martina Schollmeyer, Larry Reeves, Jui-Lin Lu, Andrew Townley, and Charles Ham for their help during this work.

The following commercial products which I had to purchase myself allowed me to finish this thesis:

- QEDIT Advanced Editor
- ISPF/PDN Editor
- Abraxas PCYACC and PCLEX Tools
- IBM C/SET++, IBM Developer's Toolkit 2.1
- IBM OS/2 2.1 Operating System
- TLIB 5.0 by Burton Systems
- OPUS Make

Most importantly, my deepest gratitude to IBM for building an operating system and compiler up to the job.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

A. PROGRAMMING LANGUAGES

A programming language is a means of recording instructions for a computer in an abstract manner. Instructions in a computer language are understood and manipulated by humans more easily than the binary instructions the hardware executes. A program is a sequence of instructions written in a computer language. Computer languages, like human languages, can be described by grammars. Several pieces which together form a grammar are terminals, non-terminals, productions, and a start symbol. A terminal corresponds to a word or punctuation in the English grammar. A terminal is atomic and thus cannot be decomposed into any smaller unit. A production, or rule, corresponds to the familiar sentence structure discussed in English grammar textbooks. A sentence can be made up of a *Subject* and a *Predicate* followed by a period. For a computer grammar, this would be written:

$$\textit{Sentence} \longrightarrow \textit{Subject Predicate} .$$

To show more rules, or productions, this example must be extended. A *Subject* can be a noun:

$$\textit{Subject} \longrightarrow \textit{Article Noun}$$

An *Article* can be A or The. A *Noun* can be many things, including Dog, Cat, and Child. A *Predicate* is too abstract to be just a single word, so it is too early to tell what it might be. The potential *Noun* and *Article* values could be written:

Article ⟶ THE
Article ⟶ A
Noun ⟶ DOG
Noun ⟶ CAT
Noun ⟶ CHILD
Noun ⟶ HOUSE
Noun ⟶ YARD

A *Predicate* can be a *Verb* followed by an *Adverb* and an *Object*, or it can be just a *Verb*. The associated productions would be:

Predicate ⟶ *Verb Adverb Object*
Predicate ⟶ *Verb*

Both *Verb* and *Adverb* are simple enough to allow enumerating.

Verb ⟶ RUNS
Verb ⟶ EATS
Verb ⟶ SLEEPS
Adverb ⟶ QUICKLY
Adverb ⟶ SLOPPILY
Adverb ⟶ SOUNDLY

An *Object* is not always present. There are two ways to handle this in a grammar. One is to have a separate production for each case:

Predicate ⟶ *Verb Adverb*
Predicate ⟶ *Verb Adverb Object*

Another way to deal with this is to have an optional item in the production. An example would be:

Predicate ⟶ *Adverb Object*
Object ⟶ ϵ
Object ⟶ *Preposition Article Noun*
Preposition ⟶ IN
Preposition ⟶ ON

A production with an ϵ after the arrow is called an epsilon production. Some references use λ instead of ϵ. There is no semantic difference. The ϵ means no non-terminals or terminals are on the right-hand side of the production.

Some observations can be made about the structure of the productions at this point. The arrow is a separator between two pieces of the production, called the left-hand side and the right-hand side. The left-hand side in the examples given has always had only one member. A closer look at the right hand side is now in order. Notice that the all capitalized members are words in English that are not to be further substituted. These are terminals. The members in mixed case which are to be further substituted are non-terminals. The left-hand side is always a single non-terminal. The original production had a left-hand side of *Sentence*. This is the start production, since it is always used first.

An example of a sentence produced using these productions will make this clear.

Sentence ⟶ *Subject Predicate* .
Subject ⟶ *Article Noun*
Article ⟶ A
Noun ⟶ DOG
Predicate ⟶ *Verb Adverb Object*
Verb ⟶ SLEEPS
Adverb ⟶ SOUNDLY
Object ⟶ *Preposition Article Noun*
Preposition ⟶ IN
Article ⟶ THE
Noun ⟶ YARD

At this point no further substitutions can be made. If the terminals from the right-hand sides are read starting at the first production a sentence is indeed formed:

A DOG SLEEPS SOUNDLY IN THE YARD .

Thus the sentence can be produced, or derived, from the start production. While this is interesting, it does not solve any

typical problems. What is needed is not a generator of sentences, but instead an analyzer. For instance, is the following a valid sentence?

SOUNDLY THE DOG . YARD IN A SLEEPS

It has the same words as the sentence generated earlier. Something more than an examination of which terminals are entered is necessary to decide if the sentence is valid according to the grammar. Fortunately, the answer lies in using the grammar techniques previously discussed. If the substitution method is reversed, the sentence can be examined:

Adverb ⟶ SOUNDLY

Sentence must start with a *Subject*, and *Subject* must be a *Noun*. *Adverb* cannot be used here. This series of terminals is not valid because no sequence of productions beginning with the start production *Sentence* leads to this series of terminals. What about the first sentence examined?

A DOG SLEEPS SOUNDLY IN THE YARD

Article ⟶ A
Noun ⟶ DOG
Subject ⟶ *Article Noun*
Verb ⟶ SLEEPS
Adverb ⟶ SOUNDLY
Preposition ⟶ IN
Article ⟶ THE
Noun ⟶ YARD
Object ⟶ *Preposition Article Noun*
Predicate ⟶ *Verb Adverb Object*
Sentence ⟶ *Subject Predicate*

Admittedly, some yet to be disclosed method was used to pick the correct series of productions. The example shows such an appropriate series of productions exists. This is enough to be certain the series of terminals is a valid sentence according to the grammar. Finally, it should be noted that the left-hand side of each production always has one item, and it is a non-terminal. This is not accidental. Productions can be created which have multiple members on the left-hand side, and these left-hand sides may include terminals. These substitutions are much more complicated and are not necessary for reasonable computing languages. A grammar containing exactly one non-terminal on the left hand side of all productions is known as context-free. Any other grammar is context-sensitive or unrestricted. The LSD program is based on a context-free grammar.

Suppose there is a stream of terminals which when analyzed is determined to be valid according to the specified grammar. Is it a valid program? This question is much harder than merely determining if the stream of terminals is valid according to the grammar. By determining that the stream of tokens can be generated by the grammar one can know that the series of tokens is syntactically correct. However, it may not be semantically correct. For instance, with the example grammar, the following sentence can be generated:

THE HOUSE EATS SOUNDLY IN A DOG .

The syntax is sound, but it is obvious that the sentence is meaningless. This clearly demonstrates that while a program may be syntactically correct it may not produce the desired results. It

is known that simply calculating whether an arbitrary program will halt is impossible.[1] Determining whether the program will produce the desired result is even more complicated; therefore, there is no way to check this automatically. The LSD editor is designed to eliminate syntax errors, but the semantic accuracy must be determined by the user.

B. TEXT EDITING

Since the advent of computer terminals, programs have been entered using text editors. A text editor is a program which allows the creation of character streams. A character stream is an ordered collection of characters, just as the sentences discussed in section one are collections of ordered terminals. A text file is a stored character stream. Text editors range from primitive to complex, but all have certain basic operations in common. A character may be inserted or deleted at an arbitrary point in the text stream. The resulting stream may be stored. Most editors also allow the user to view the text stream on the screen and have a logical position indicator within the text stream called a cursor. The insertions and deletions mentioned take place at the cursor.

The majority of text editors today view the text as being made up of more complex units than merely characters although they still allow manipulation of the individual characters. Lines are the most common level of abstraction supported by the text editor. A line is defined as being a character stream terminated by some special sequence. A text file can then be interpreted as a stream of lines as well as a stream of characters.

Beyond these simple operations a text editor can provide almost limitless flexibility in editing. The editing is based on the fundamental atomic units in the character stream, the characters themselves, and the abstractions such as lines which the editor supports. No notion of a grammar is present.

Compilers are translation programs which convert programs specified with character stream input to programs a machine can run. All syntax checking of the information entered with a text editor must be performed by the compiler after editing is complete. This slows down the work of an inexperienced programmer because a misspelled word will not be detected until the program is compiled. Then the editor must be re-entered, the error corrected, and the compilation re-tried. This cycle must be repeated until a syntactically correct program is created. After a syntactically correct program is created, the programmer may finally start checking the program for correct semantics. The time spent generating a syntactically correct program can be considerable, and it postpones the time when a program may be tested for semantic compliance to its original specifications.

Since compilers take text (character streams) as input, text editing is the traditional way to create the compiler input. Text editing works well for experienced programmers who no longer make many syntax errors, but it is a poor system for beginners. A syntax-directed editor such as LSD is an ideal editor for novice programmers.

C. SCANNING AND PARSING

1. Scanning. Once entered, several actions are required to analyze a character stream to determine if it contains a syntactically correct program. First, the character stream must be converted to a stream of terminals which can be analyzed as was discussed in section one. The process of converting a stream of characters into a stream of terminals (often called tokens in this context) is called lexical analysis, or scanning. Many scanning methods exist, but the method of choice is to use regular expressions for recognizing the terminals in the character stream. This will be easier to explain with the help of some examples.

```
WORD             = (LETTER)+
UNSIGNED_INTEGER = (DIGIT)+
SIGNED_INTEGER   = [+|-](UNSIGNED_INTEGER)
PERIOD           = .
ELLIPSIS         = ..
REAL_NUMBER      = ((SIGNED_INTEGER)(PERIOD)(UNSIGNED_INTEGER)) |
                   ([+|-](PERIOD)(UNSIGNED_INTEGER))
```

Again, the left-hand side, right-hand side pattern is evident, with an equals as the divider. The items on the left-hand side correspond to terminals, or tokens, which can be used for the grammatical analysis. The right-hand sides are regular expressions made up of characters (or sets of characters) and operations upon them. A set of parentheses around an expression indicates the enclosed regular expression is treated as a unit. The '|' symbol means 'or', so that 'STUFF = A|B' means that 'STUFF' can be either 'A' or 'B'. Brackets around an expression mean the enclosed regular expression is optional. A '+' means that the preceding regular expression is repeated one or more times. A '*' means that the preceding item is repeated zero or more times. A '*' can be

replaced with a combination of '[', ']' , and '+'. 'J*' can be written as '[J+]', but the '*' is a more convenient notation. Using regular expressions, every acceptable terminal can be specified.

Tools exist for the software developer to generate automatically source code for translating character streams to token streams. These tools require that the tokens be specified with regular expressions. The most prominent tool for generating scanners is called lex. Lex was originally written on a UNIX system and generates the C language source for a scanner. The tool used for generating the scanner of the LSD editor was Abraxas' PC-lex, a more powerful descendant of lex.

2. Parsing. The next step in analyzing a program is to see if the terminal stream is valid according to the grammar. This step is called parsing. The function which accepts a terminal stream and returns TRUE if the token stream is valid according to the grammar is called a parser. Fortunately, tools exist for the developer which accept a grammar specification as input and output source code for a parser. The most prominent tool for generating parsers is called yacc. Yacc was originally written, like lex, on a UNIX system. Yacc generates the C language source for a parser. The tool used for generating the parser of the LSD editor was Abraxas' PC-yacc, a more powerful descendent of yacc.

3. Reasons For Using Tools. The lex and yacc tools are beneficial to the developer for several reasons. First, they allow the developer to concentrate on the abstract regular expressions and grammar, which are central to the problem being solved, and leave the generation of the scanner and parser to a program. Careless

mistakes are prevented because the generators have been extensively tested. Another benefit of yacc is that changes to the grammar productions or terminals can be made easily. If a programmer hand codes the parser and scanner, any changes must be done carefully by hand to insure that nothing is broken, and that no harmful side-effects result. A whole battery of testing must be done to insure that the scanner and parser are working correctly. With lex and yacc, however, a change in the specifications and a regeneration of the scanner and parser can be done simply. Since the code is generated by the tools, no fear of incorrect function exists, and no extensive testing is necessary to find bugs in the code. Bugs in the logic of the grammar and the regular expressions used to denote the acceptable terminals may still exist, but this is true whether the tools are used or not. Lex and yacc allow rapid updates of the grammar, either to correct it or enhance it, without the worries associated with hand-coded scanners and parsers. Because the output of these tools is C code, the software must be written in the C language or a superset of it. The language C++ is a superset of C. Since C++ is adequate for this project, having the scanner and parser as C code is not a handicap.

D. TYPICAL PROGRAMMING ERRORS

There are three errors which plague a person typing in code as a character stream. First, a required terminal may be omitted or spuriously inserted. The bane of programmers in Pascal and C, and especially of programmers who use both, is the semicolon. For example, programs written in the C language grammar require a semicolon in places where the Pascal language grammar forbids them.

With a text editor, a semicolon is treated as a character just like any other. The user of the text editor will not discover the incorrect or missing semicolon until editing is complete, and the compilation has failed.

Second, a terminal may very well be misspelled. Terminals which are words that have special meanings are called keywords. If misspelled, these are no longer the same terminals. Suppose there exists a production which specifies:

$$A \longrightarrow \text{aterm } B \text{ cterm}$$

Instead of aterm the character sequence taerm is entered with the text editor in the stream:

some stuff taerm something else cterm

The parser will be unable to detect that taerm something else cterm should be derivable from A. The compiler will catch the error only after editing is complete. A final type of error is more subtle but still fairly common. Often in a grammar a production of the form:

$$A \longrightarrow (\ B \ )$$

will be present, where the parentheses are delimiters of whatever B is. If a programmer forgets either the '(' or the ')' in the middle of a complex character stream, then a mismatch of the parentheses and a corresponding misinterpretation of the total character stream will occur.

E. SYNTAX-DIRECTED EDITING

With a syntax-directed editor like LSD, all three of the common errors mentioned in section D can be prevented. Since the program is entered by replacing non-terminals using lists of productions, there is no way to leave out or add an extra terminal. It is also impossible to misspell a terminal which is a keyword. Forgetting one of a pair of delimiters is actually a special case of the first problem and is thus prevented. By preventing these errors from ever entering the terminal stream, the extra compilation attempts associated with these errors can be avoided. In addition, the programmer will not need to memorize the exact productions in the grammar. Whenever a choice of productions is necessary, a list will be displayed with all valid choices.

How exactly does syntax-directed editing work? In general, the method used is to begin with a copy of the grammar's start production (call it p_1), then for each right-hand side non-terminal an appropriate production (call it p_2) is chosen which has that non-terminal as the left-hand side. The non-terminal on the right-hand side of p_1 is replaced by the right-hand side of p_2. This is continued recursively until no non-terminals remain on the right-hand side of p_1. All that will be left on the right-hand side of p_1 will be a stream of terminals, a syntactically correct program according to the grammar. This correct sequence of terminals is then converted into a stream of characters acceptable to the compiler.

LSD follows the general method, but instead of substituting in the right-hand sides and generating a modified p_1 to view, a

structure is built documenting the sequence of substitutions used. The structure used is a tree.

F. TREES

A little background on trees is necessary to understand the terminology used later. A graph is a collection of nodes and edges. An edge is a connection between two nodes. An edge in a graph is described by a pair of nodes: the two nodes are connected by the edge. An edge connecting nodes a and b is written either as (a,b) or (b, a). A directed edge is described by an ordered pair of nodes, where the first node is the start of the edge and the second node is the end of the edge. A directed edge from a to b would be written (a,b). A directed edge from b to a would be written (b,a). A directed edge from a to b may also be written a→b. A graph composed of nodes and directed edges is a directed graph.

An sequence of edges in a directed graph such that the ending node of any edge in the set is the starting node of the next edge in the set, if one exists, is called a path. This is shown in figure 1. It can be seen that the following edges exist in the

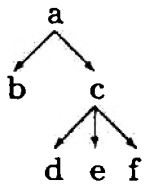


Figure 1. Example Graph

graph: a→b, a→c, c→d, c→e, and c→f. By definition, any edge by itself is a path. In this graph there are also longer paths,

namely (a→c, c→d), (a→c, c→e), and (a→c, c→f). If a node from the graph is included in the path more than once, the path is said to have a cycle. For instance, the graph in figure 2 has a cycle. The cycle in this graph is (a→c, c→d, d→b, b→a). A directed

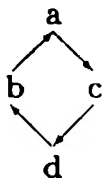


Figure 2. Graph With Cycle

graph which contains no cycles is termed an acyclic directed graph. A node in a directed graph may have edges which start or end upon it. The indegree of a node is the number of edges which end on that node. The outdegree of a node is the number of edges which start on that node. A tree is a directed acyclic graph in which every node has an indegree of one, except for one node called the root. The root has an indegree of zero. A node in a tree with an outdegree of zero is called a leaf.

Figure 3 contains a simple tree. The node a is the root.

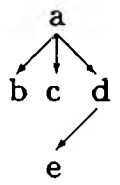


Figure 3. Example Tree

The indegree of b, c, d, and e is one, while the indegree of a is zero. The outdegree of a is 3, the outdegree of b, c and e are

zero, and the outdegree of d is 1. The degree of a tree is defined as being equal to the highest outdegree of any node in the tree. The tree in figure 3 is a ternary tree, or tree of degree 3. Unary trees have degree 1, binary trees have degree 2, ternary trees have degree 3, and for trees with a degree n greater than 3 the term n-ary is used. Thus a tree with degree 5 would be a 5-ary tree.

A results of the definition of a tree is that for every node in the tree except the root, some edge exists which ends on that node. All nodes in a tree will be connected by directed edges. For a pair of nodes connected by a directed edge, the node on which the edge starts is said to be the parent of the node upon which the edge ends, and the ending node is the child of the starting node. A subtree is a portion of the tree which is itself a tree starting at a particular node and containing all nodes beneath that node. The subtree at node d includes the nodes d and e. A given tree may be represented in a variety of semantically equivalent ways. For instance, all of the trees in figure 4 are equivalent.

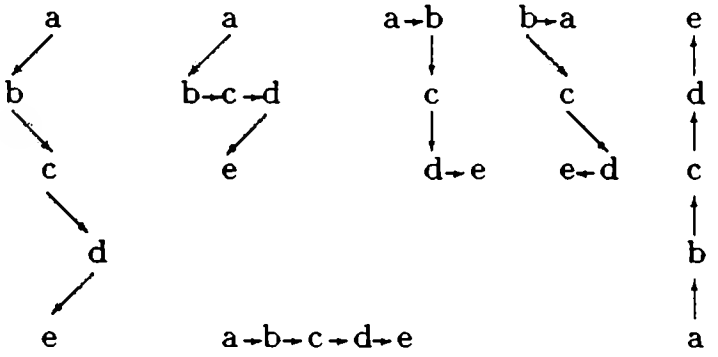


Figure 4. Equivalent Trees

One practical consideration when storing trees in a computer is the degree of the tree. A tree which has a high degree, but

with only a few nodes of this degree, will waste space. To store a tree node, a list of items which represent the links to the child nodes is necessary. The more items in this list, the more space will be taken. If this space is usually empty, then it is being wasted. The routines to build and to traverse the tree will be more complex and slower in a tree of high degree than in a tree of lower degree due to the processing of the additional links.

Fortunately, any n-ary tree can be represented as a binary tree.[2] The method used is to take an n-ary tree and convert it to a binary tree which represents the same idea logically, but not physically. The best way to examine this is with an example. In figure 5, the tree on the left is the original ternary tree; the tree on the right is the resulting binary tree. The resulting

Figure 5. Ternary Tree and resulting Binary Tree

binary tree is drawn in the peculiar fashion to show the logical meaning of the tree. Each left node is indeed a lower node on the original tree. Each right node is a node that was a child of the same parent node. Nodes B, c, D have the same parent A in graph 1 of figure 5, but D is the right child of C which is the right child of B in graph 2 of figure 5. This method of interpretation allows storing an n-ary tree as a binary tree, thus simplifying the routines to manipulate the tree and saving space.

G. LSD'S TREES

Trees are used in LSD to model the sequence of productions chosen while editing. Given a sequence of productions:

$$A \rightarrow x\ Y\ z$$
$$Y \rightarrow Y\ t$$
$$Y \rightarrow t$$

how can it be represented with a tree? Productions are modeled by making the left-hand side the parent of the elements of the right-hand side. For example, the first production $A \rightarrow x\ Y\ z$ would be the subtree shown in figure 6.

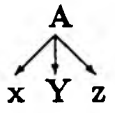


Figure 6. Ternary Tree for production: $A \rightarrow x\ Y\ z$

Without changing its meaning, this ternary tree can be re-drawn as the binary tree shown in figure 7. This second form is much easier

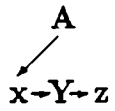
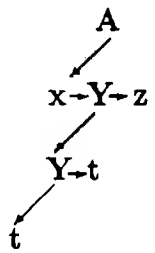


Figure 7. Binary Tree for production: $A \rightarrow x\ Y\ z$

to store and use as discussed. When a subtree is displayed on the screen, an n-ary tree can be used. This is accomplished by converting a copy of the binary tree back to its n-ary original form and using this copy. The binary form is better for the programmer than the n-ary form, since the concept of having two separate sides of the production is implied by this picture. Furthermore, if an interpretation is attached to this type of

subtree which states that A has been replaced in the new level (which is lower in the picture) with the right-hand side x Y z, then it becomes clear that displaying a series of productions will be possible using this notation. The direction of the arrows in the subtree preserves the order of the items in the production. A tree showing the series of productions shown earlier can now be drawn as in figure 8. By interpreting the subtree with the method



This is the original non-terminal.

This is the first substitution.

This is the second substitution.

This is the final substitution.

Figure 8. Tree Showing Productions

just described, the productions can be read starting at the top of the graph. The productions read match the productions intended.

The series of substitutions in a syntax-directed editor is designed to produce a stream of terminals which represent a valid program. The graph can provide this function if it is interpreted correctly. Several observations are necessary at this point. First, all leaves are terminals, and all other nodes are non-terminals. Second, within any right-hand side of a production, the order of the terminals is left to right as desired. Third, any non-terminal node is the head of a subtree. Fourth, any such subtree can be examined recursively using these four observations to provide the terminals which go in the subtree's place in the terminal stream.

Examining the previous example tree, starting at the second level (the first right-hand side in the tree), the first member of the right-hand side is the terminal x. The x is the first terminal in the terminal stream. The next member of the production is Y, a non-terminal, so its production (the right-hand side of the production Y→Y t) must be examined. The first member of the right-hand side on the second substitution level is the non-terminal Y, so its production must be examined. The first member of the third substitution level is the terminal t. This t is the second terminal in the terminal stream. No more members exist in this production, thus we return to the previous production and examine the next member. The next member of the production now being examined (the second substitution) is the terminal t. This t is the third terminal in the terminal stream. No more members exist in this production, so we return to the first substitution level. The terminal z becomes the last member of the token stream giving us the stream (x, t, t, z). This process of examining the nodes is a called a preorder traversal.

A method for constructing a tree corresponding to a series of productions has been demonstrated. Also, a method for creating an appropriate token stream from one of these trees has been demonstrated. These methods form the basis of the syntax-directed editing capability of LSD.

H. LSD'S TREE EDITING

With a tree representation of the sequence of substitutions, it is possible to review the substitutions and even alter them. A subtree may be deleted or moved. If a subtree is deleted, the node

which started the subtree is left. All nodes below this remaining node are deleted, and no edge goes out to a lower logical level any more. If a node is a non-terminal, and it has no outgoing edge to a lower level (which means that no right-hand side has been chosen), an insertion may be made of an appropriate right-hand side. Then this lower level may be added to the graph. Figure 9 is an example of an insertion.

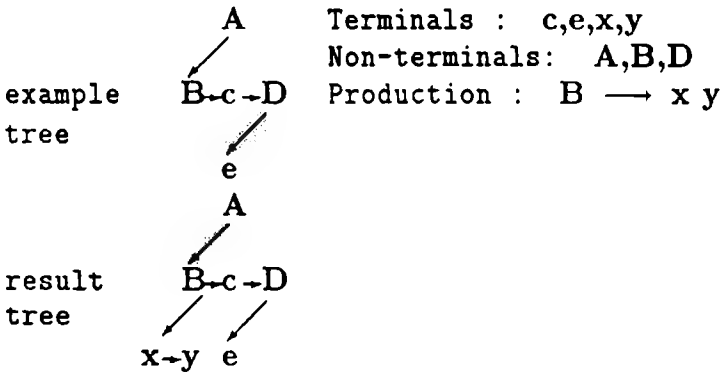


Figure 9. Tree before and after insertion

To show more editing, suppose that in addition to D ⟶ e, a production D ⟶ B also exists in the grammar. Then a possible series of actions would be:

1. A deletion at node D.
2. An insertion using the D ⟶ B production.
3. A move of the right-hand side beneath the original B to the new B. This is done by deleting the right-hand side beneath the original B, and then inserting this deleted subtree beneath the new B.

The trees corresponding to this sequence are shown in figure 10.

Two types of non-terminal nodes exist. The non-terminal nodes which have a right-hand side attached below them are expanded, and those which have no right-hand side are unexpanded. If a tree has

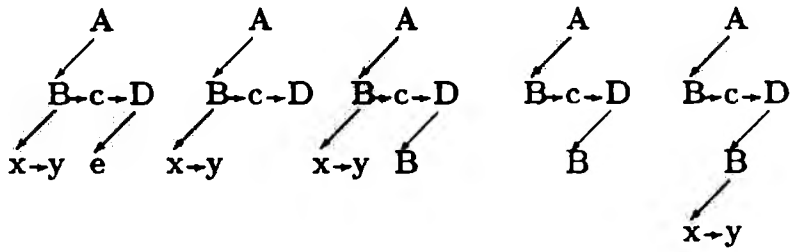


Figure 10. Series of trees during editing

no unexpanded non-terminals, then it represents a syntactically correct program according to the grammar. With a tree containing no unexpanded non-terminals, the traversal method described previously may be used to generate the stream of terminals. Since the expansions are done with appropriate productions in the grammar, no incorrect expansion is possible. This prevents the introduction of syntax errors. Instead of always substituting the left-most non-terminal on the right-hand side of the current p_1, the programmer may choose to move anywhere in the tree. Work may continue from the new location. This allows the programmer the ability to enter a program in a more flexible order than strictly left to right.

LSD provides the basic operations of text editors mentioned in section 2. A non-terminal may be replaced by an appropriate production, which corresponds to insertion. A subtree may be removed corresponding to deletion. A tree may be traversed to produce a stream of terminals corresponding to saving the stream of characters in a text editor. The concept of being at some location in the tree and editing relative to this position is analogous to the cursor-relative insertions and deletions in a text editor. If the characters corresponding to each token can be generated from

the token, a character stream can be generated which is suitable input for a compiler. LSD also allows moving subtrees which would correspond to moving groups of lines in a text editor. LSD provides these features without allowing syntax errors. If a programmer is likely to make syntax errors, LSD is a better choice for entering programs than a text editor.

II. ANATOMY OF LSD

The LSD program has a complex design because of its many functions. This section describes the structure of LSD and its components. First, the design goals for LSD will be discussed. Next, the constraints placed on LSD by the decisions made in the design phase are examined. Then the various pieces of LSD which correspond to modules of the program, their associated responsibilities, and their function are discussed. Later, the relationship between these modules is shown. Finally, the way these modules together form LSD is presented.

A. DESIGN CONSTRAINTS AND METHODOLOGY OF LSD

A syntax-directed editor was desired for teaching new programmers how to program without the burden of memorizing the syntax of a programming language. Since this editor was to run on IBM PC clones, and the complexity of the program and size of the data to be manipulated were too great for MS-DOS, IBM OS/2 2.1 was chosen as the target operating system for LSD. To use the lex and yacc development tools, the language in which LSD would be implemented would have to be C or C++.

A module is a portion of a program that performs one specific task or set of tasks. One of the most important design considerations for LSD was to provide modularity, since a modular design meant the program's pieces could be built and tested independently. Essentially, a module is a specialized service provider for the entire program. A module often depends upon other modules for help performing its task. For instance, a string module will have the responsibility of providing comparison, copy,

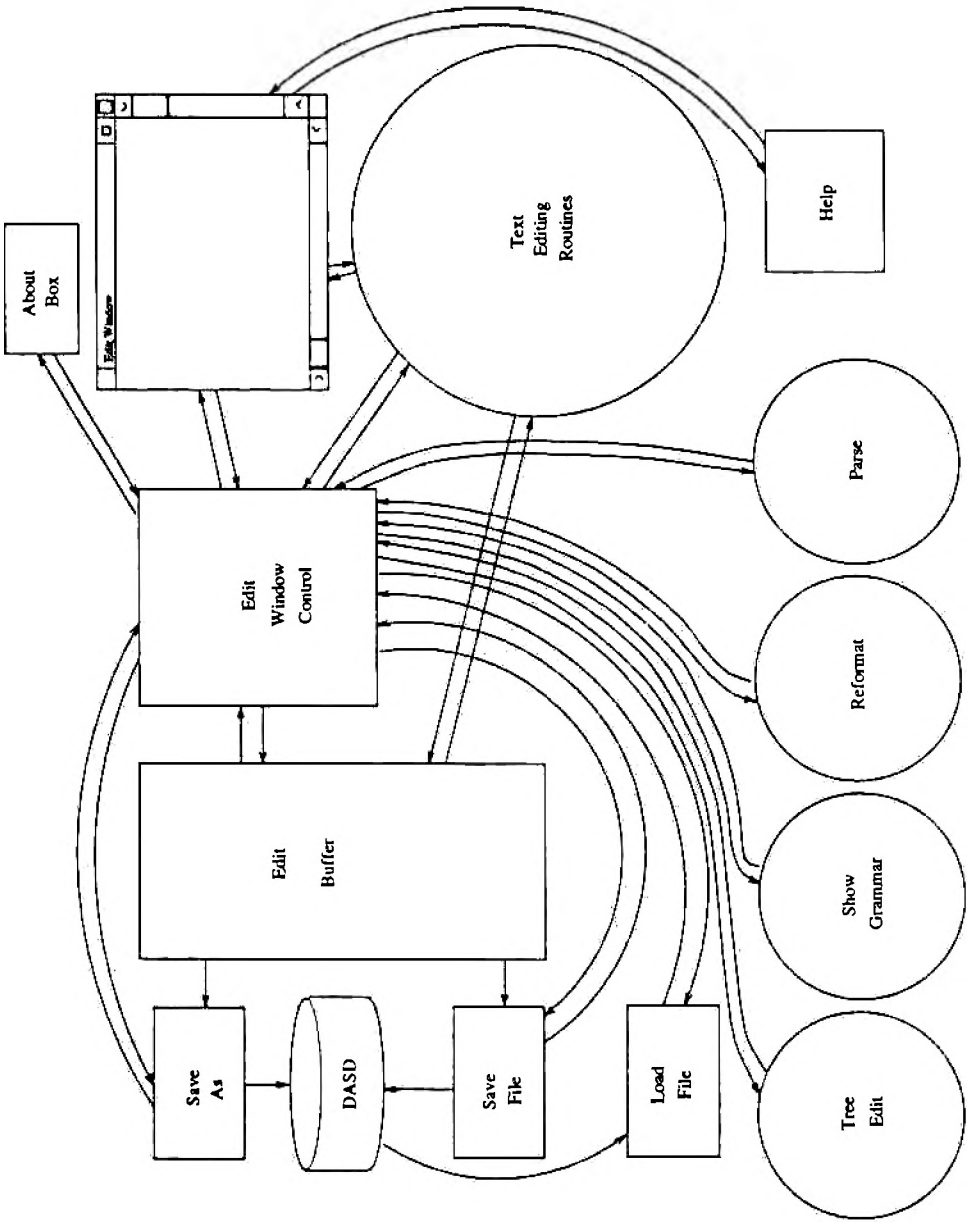


Figure 11. Edit Code Map

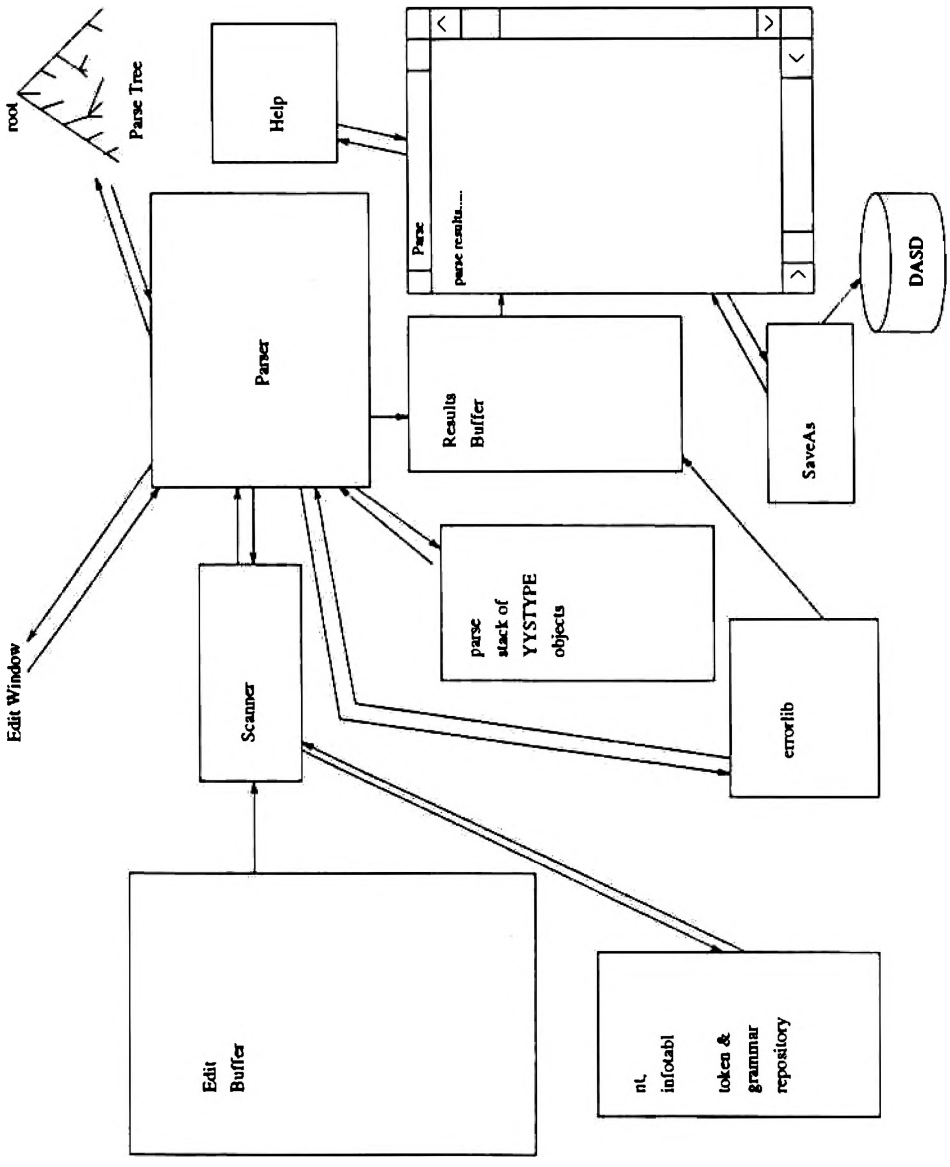


Figure 12. Parse Code Map

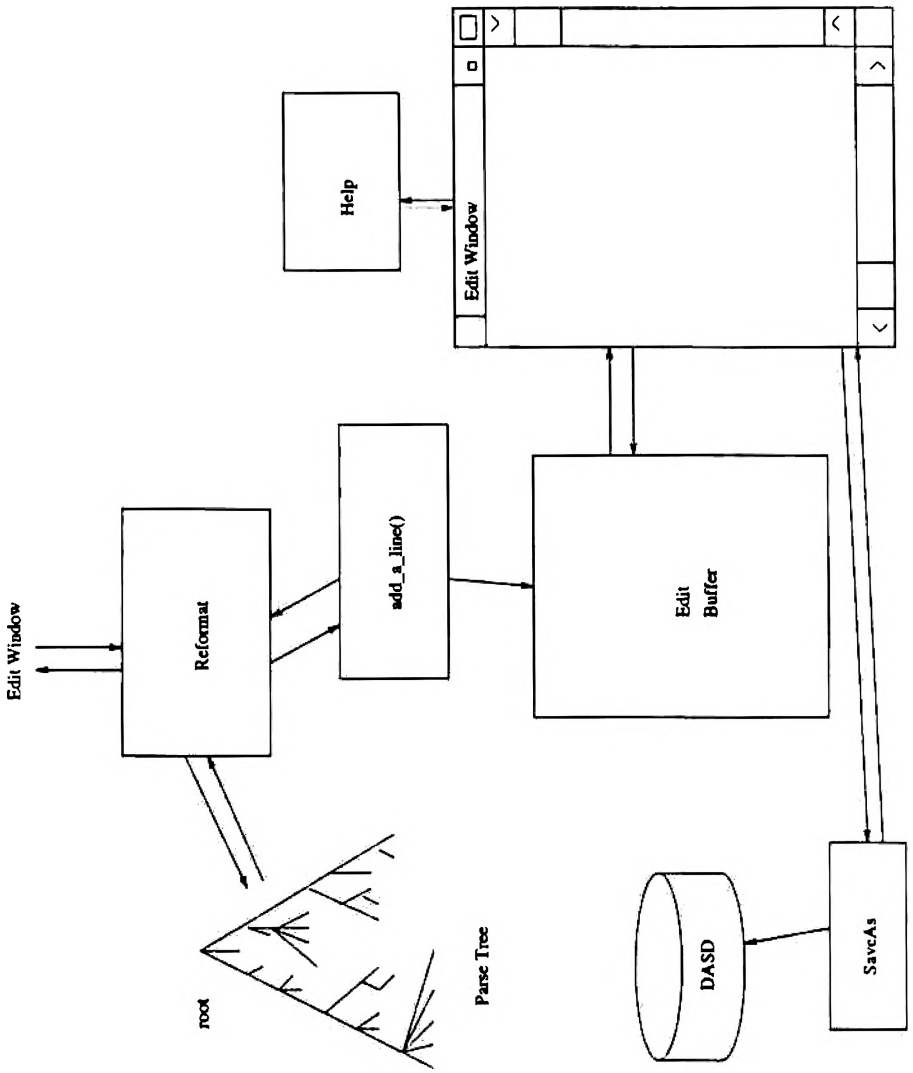


Figure 13. Reformat Code Map

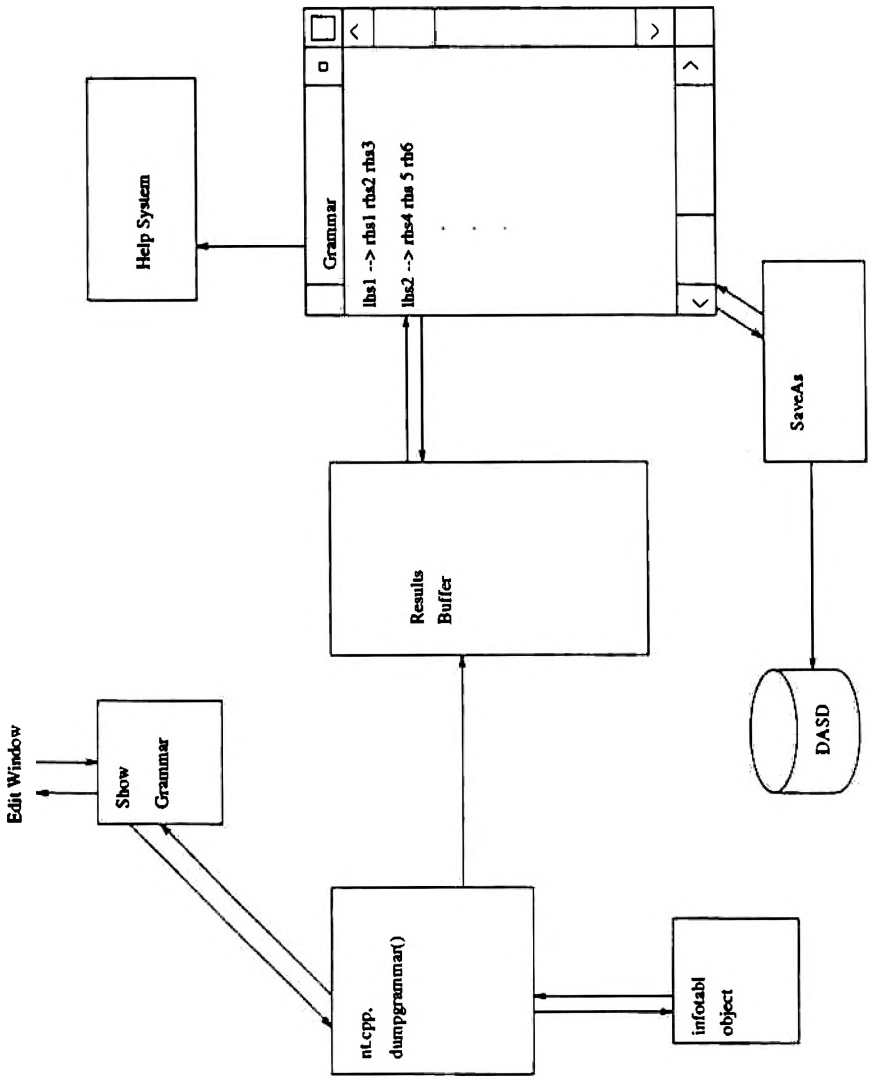


Figure 14. Showgrammar Code Map

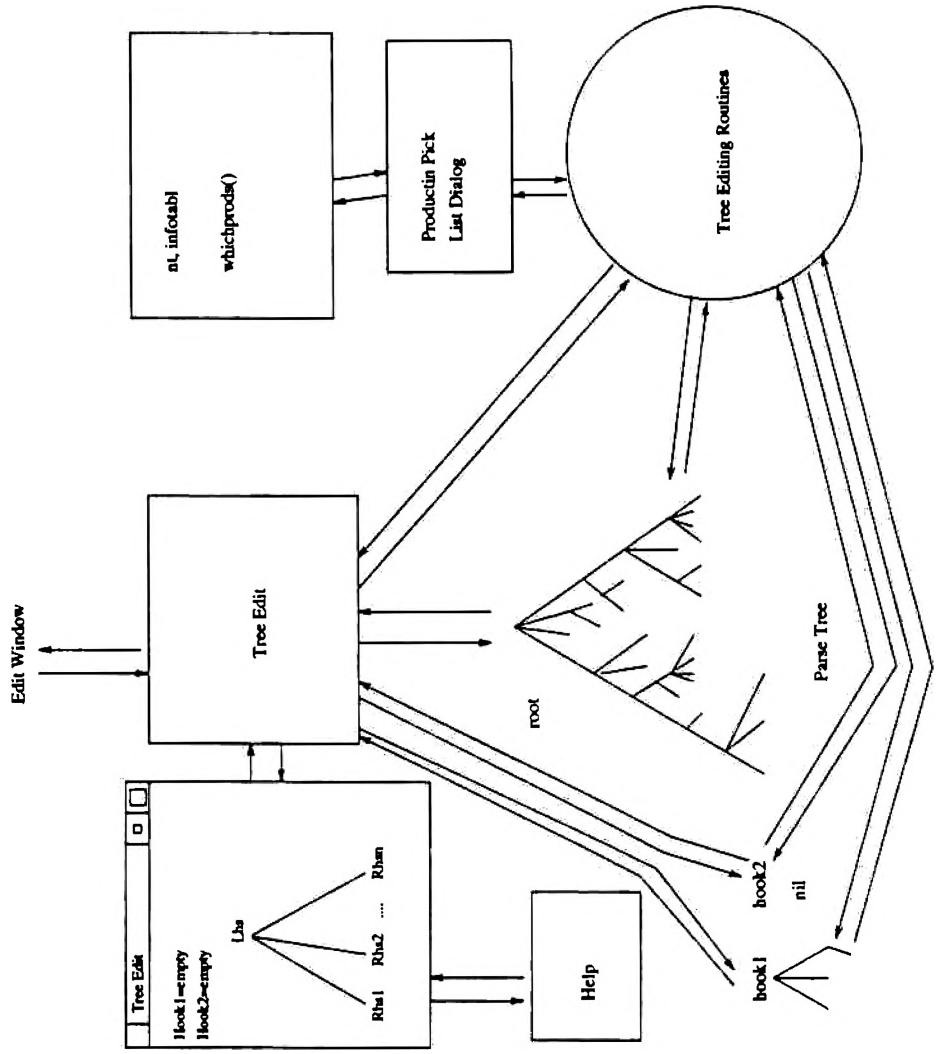


Figure 15. Treeedit Code Map

creation, and destruction of strings. A text editing module will be a higher level module which uses string modules to do its string manipulation as part of its text editing task.

The division of labor achieved through modular construction eases the testing burden because any change in one module does not change any other module. A module can be perfected without damaging any other module. Once all modules are complete, they can be integrated to form LSD. The use of modules allows the problem of creating a large project like LSD to be addressed with a divide and conquer scheme. Instead of making one monolithic program to do everything, several specialized modules are created. Each module performs a specific task. As a last step these modules are brought together by a control program which coordinates the various modules' activities.

To allow development using language-supported objects, the C++ language was chosen. Objects are simply modules of the program which have both methods (or functions) and data, and which can be dynamically created and destroyed. Some of the data may be made private to the module. This means that only by using the methods of the object can the object's data be manipulated. This is called encapsulation and allows the programmer to separate pieces of the program easily. This separation is accomplished by preventing any accidental manipulation of this private data, since only through a valid call to a method of the object can any private data of the object be used.

Objects were used to provide modularity on a small level. On a larger scale, individual source files were used as modules. A

breakdown of the LSD program's function was done according to which functions had to be supported. This design methodology is known as functional decomposition of the program into source file modules. LSD includes a text editing module, a parse tree definition module, a parse tree editing module, and many utility modules. Functional decomposition was followed by an object-oriented encapsulation of the remaining information within each file module. When two or more copies of a module were needed, an object was used since multiple objects with the same definition can be created.

B. SOURCE FILE MODULES

The LSD editing system has two editing modes, one for text editing and one for tree editing. Each mode has its associated source files, and there are also files shared by the two modes. First the text editing mode and its files are discussed. Then the files used by both the text and the syntax-directed editing portions are examined. Finally, the syntax-directed editing files are explained.

1. The Parse Module. When a user wishes to check the source program entered using the text editor, the Parse menu bar option is selected to call the parser into action. The parser has two jobs to perform. First, it checks the syntax of the program to make sure the program is valid according to the grammar. Second, the parser builds a parse tree for use by the parse tree editor. The parser used by LSD was created using the Abraxas PC-Yacc parser generator. The PC-Yacc parser generator generates C code for an LALR(1) shift-reduce parser. For this thesis the primary focus is upon editing, not parsing. The generated parser is treated as a

black box. The internal function is known to be correct, the input is a stream of tokens from the lex generated scanner, and the output is a yes or no indicating whether the input was valid according to the grammar. If the input was valid, a parse tree is generated as well. The Abraxas PC-lex scanner generator was used in a similar manner to provide the translation from the character stream to the token stream required by the yacc parser.

Understanding the function of the parser generated for LSD requires some background information on shift-reduce parsing. A shift-reduce parser has a parse stack on which to place tokens, a list of grammar productions, and a routine to fetch more tokens. When the parser starts, it fetches a token and places it on the parse stack. Next, the parser will examine the production lists. The parser compares the right hand sides of all productions having only one item to the item on the parse stack. If any of these right-hand sides match, the token on the stack is removed, and, instead, the left-hand side of the production in question is placed on the stack. This step is called a reduction because the group of items on the stack are replaced by one. Then the parser re-checks the grammar productions to see if the item on the stack is the right-hand side of any production again. If there is a right-hand side in the grammar with the same item as the item on the top of the parse stack, then the stack item is replaced as before. This process repeats until no matching grammar production is found. Then the parser fetches another token and places it on the stack. This step of fetching another token is called shifting. The shift and reduce steps led to the name shift-reduce parser.

The parser then continues examining the grammar for a right-hand side which matches what is on top of the stack. If a match is found, a reduction occurs, and the search for matches continues. Otherwise a shift occurs, and the search for matches re-starts.

What happens when there is a choice of shifting or reducing? This could happen with the example grammar and input string shown in figure 16. Two different sequences of shifts and reductions are

$$S \longrightarrow a$$
$$S \longrightarrow \mathrm{a}\ A$$
$$A \longrightarrow \mathrm{b}\ A$$
$$A \longrightarrow \mathrm{c}$$

Sample Input="abc$"

Figure 16. Shift-Reduce Example Grammar

examined to show how using one set of productions and input two different results can be obtained. In case 1, as shown in figure 17, a was reduced on the second step to S using the $S \longrightarrow$a production. This leads to an error condition. In case 2, shown in

stack	input	action	production
	abc$		
a	bc$	shift	
S		reduce	$S \longrightarrow$a
Sb	c$	shift	
Sbc	$	shift	
Sbc$		shift	
result:	Error!		

Figure 17. Shift-Reduce Example Case 1

figure 18, choosing to shift instead of reduce on the second step the results in an accept condition.

stack	input	action	production
	abc$		
a	bc$	shift	
ab	c$	shift	
abc	$	shift	
abA	$	reduce	$A \longrightarrow c$
aA	$	reduce	$A \longrightarrow b\ A$
S	$	reduce	$S \longrightarrow a\ A$
S$		shift	
result:	Accept!		

Figure 18. Shift-Reduce Example Case 2

How can an algorithm determine whether to shift or reduce in this situation? This is a critical question because it has been shown that making the wrong choice can lead to an error condition even with an input which is correct according to the grammar. One method is to "look ahead" and see what the next token will be. This means that after a had been shifted onto the stack, the parser would "look ahead" in the input and see the token b. The parser can then realize that it must choose to shift. Why? If the parser reduces using the $S \longrightarrow a$ production the only valid symbol which can come next is the $ symbol.

This process continues until one of three conditions occurs:

1. A match is made with the start symbol of the grammar, and no more terminals exist to shift except a lone $ symbol.
2. A match is made with the start symbol of the grammar, but more unshifted terminals besides $ exist.
3. Input is exhausted and condition 1 is not met.

The first case is indicative of a program which is valid according to the grammar, and the other two cases mean that an error has occurred.

Different methods of examining the grammar for appropriate right-hand sides exist. These methods distinguish the various types of shift-reduce parsers. For efficiency reasons, LSD uses an LALR(1) method which performs the parse in a logically equivalent but different manner than that just discussed. The reader is referred to chapter 4 of the "dragon book"[3] for a more detailed explanation of the exact method used by yacc generated parsers.

Shift-reduce parsing is a type of parsing known as bottom-up parsing. The name bottom-up stems from the way the parse is done. The first matches made are of terminals which are leaves of a parse tree. The information left on the stack is one step closer to the root of the parse tree. This reduction process continues until only the root of the tree is left. The parse tree is constructed from the leaves to the root, or bottom-up, since the leaves are traditionally drawn below the root, contrary to the names leaf and root.

The file parser.y is the input to the parser generator. The yacc parser generator input consists of a definition of the terminals, the non-terminals, the productions of the grammar, and semantic routines. Each item on the stack corresponds to either a non-terminal or a terminal. The stack items may contain additional information called attributes. Attached to each production is some C language code, called a semantic routine. This semantic routine is to be performed when a reduction occurs for the production.

Since the stack is changed after the semantic routine is executed, the semantic routine may access the elements on the stack which correspond to the right-hand side members of the production. This information, along with whatever else is done in the semantic routine, is used to create information for the new left-hand side item placed on the stack as a result of the reduction. Building attributes of the non-terminal being placed on the stack is called synthesizing an attribute. Synthesized attributes allow information to be passed from the leaves up through the parse tree as it is constructed. With yacc parsers, no explicit tree is automatically constructed, but the sequence of reductions is performed, and the synthesized attributes can flow up by their placement on the stack. Each reduction can preserve information from items on the stack so the information can be passed up to the next reduction.

In LSD the semantic routines attached to the productions in the yacc input are used to create a parse tree made of objects. The objects used are homogeneous. Fields within each object hold whatever information is unique to that object. These objects form the nodes of the parse tree. The section "LSD's Trees" in the introduction explains the structure of the parse tree. The use of a bottom-up parser in LSD means that the tree must be built from the leaves to the root instead of from the root to the leaves. How is this done? The nodes are constructed within the semantic routines. One attribute in the parse stack element being created is a pointer to a node. This attribute is updated to point to the new tree node. Because every parse stack element may have one of

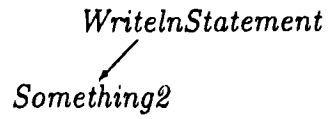


Figure 19. WritelnStatement Tree

LSD's parse trees are built from the leaves to the nodes. This is accomplished by building the tree that represents a node when that node is created during the reduction by the parser. The semantic routine that accompanies the reduction is specified in the parser.y yacc input file. This routine will first create a TREETHING object to store the node's information in. Then text showing the production matched is added to the list of text to display in the parse results window. Next any compiler directives from the terminals in the production are added to the node's compiler directive list. Any information from terminals not inferrable from the grammar is stored in the TREETHING object. Then the trees representing the nodes from the non-terminals in the right-hand side of the production being matched are added to the linked list of right-hand side items in the TREETHING object. This step effectively builds the subtree of the TREETHING object for the production being matched.

At this point the TREETHING object holding the subtree representing the production is complete. The YYSTYPE stack object pushed onto the stack to represent the left-hand side of the production which was matched and is being reduced has a pointer to hold the address of this newly created TREETHING object. When this non-terminal is a member of the right-hand side of some production

these nodes, the attributes can be passed up to the next reduction, or level of the tree. This allows some level X to get to the nodes created at level X+1 by examining the parse stack elements' pointers to node values. This information is used to make the branch headed by the node X. Then the level X-1 will have the branch headed by X, and all the other branches from level X, to build the branch at level X-1. This process continues until the root is reached, at which point the entire tree is finished and attached to the root.

The objects used for the parse stack are YYSTYPE objects and are defined in the yystype.hpp file. The methods for the objects in this file are in yystype.cpp. The parse tree node objects are TREETHING objects and they are also defined in the yystype.hpp and yystype.cpp files.

LSD's parse trees only store information which is not available in the grammar. For instance, given a production:

WritelnStatement ⟶ **WRITELN** (*Something2*) .

the right-hand side which is stored in the parse tree will have only one item, namely *Something2*. The three terminal items, 'WRITELN', '(', and ')', are inferrable from the grammar and thus do not need to be saved. This reduces the memory requirements and the time to traverse the tree for source regeneration. As discussed in the introduction to this thesis, LSD's trees are stored using linked lists to hold pointers to the right-hand side elements of the production. Since only the non-terminals and non-inferrable terminals are stored, the above production would be stored as in figure 19.

p, it will be used to build the subtree representing the left-hand side non-terminal of the production p.

An example should make the bottom-up tree building process clear. The parsing of a simple writeln statement will be used to show the steps. In figures 20 and 21 the leftmost column has the source line with a special caret symbol to show how much of the input source has been processed. The next column shows the stack contents, with the top of stack being towards the top of the page. Trees under construction are shown to the right of the stack. An arrow points from any stack element to its associated tree if the stack element has one. Finally, the grammar statements used for any reductions are shown in the rightmost column. The grammar being used is the actual grammar for LSD which is in the appendix. For the three non-terminals which are not really stored in the tree, the examples show where they would go logically and enclose them in dashed outline boxes and connect them with dotted lines instead of solid lines. The first three items seen in the input are placed on the stack since no reductions can be made. This leaves the caret marker just before the closing parenthesis. This right parenthesis is the next token to be read, and is thus the lookahead. When the parser sees this lookahead, it is able to know that a reduction is possible using the grammar production:

$$\textit{UnsignedConstant} \longrightarrow \textbf{QSTRING}$$

The reduction will have a tree associated with it. The tree is shown in figure 20. The dotted line from the *UnsignedConstant* on the stack shows that there is a pointer in the stack element to the

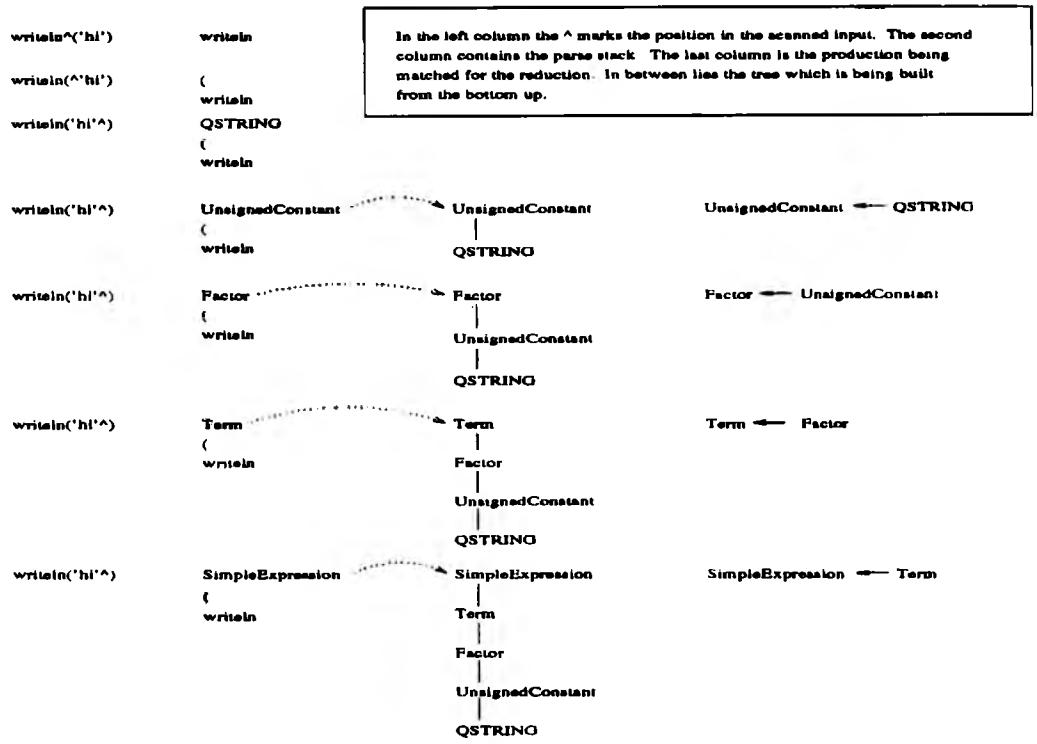


Figure 20. Writeln Example Part 1

tree. Every non-terminal in the stack will have an associated tree.

In the next step, the reduction

$$Factor \longrightarrow UnsignedConstant$$

is used. Again, this is because with a lookahead of ')' the parser knows a reduction must occur. The *UnsignedConstant* element on the stack is popped. The tree associated with *UnsignedConstant* is then placed into the new node created for *Factor*. Finally the new tree is added to the *Factor* stack element and the element is pushed onto the stack.

Similar actions take place for the reductions for the productions:

Term ⟶ *Factor*
SimpleExpression ⟶ *Term*
Expression ⟶ *SimpleExpression*

However, the action appears different for the

Moduhfires ⟶ *Epsilon*

production shown in figure 21. What really happens is that a second tree is constructed to represent the new production since it is unrelated to the first. This poses no problem since the stack element for *Moduhfires* is able to point to the new tree. Since *Expression* is not popped the tree associated with it is still pointed to by the stack element for *Expression*. The next reduction by

Something2 ⟶ *Expression Moduhfires*

pops both the *Expression* and *Moduhfires* off of the parse stack (since they are the right-hand side of the production) and it attaches the two trees associated with them to the new node created for *Something2* and then pushes the *Something2* stack element with its pointer to the new tree.

Finally a reduction by the production

WritelnStatement ⟶ WRITELN (*Something2*)

occurs. Notice the dashed boxes around 'WRITELN', '(', and ')'. These right-hand side items are inferrable from the grammar and so they do not really need to be stored in the tree. Their logical position is shown, and when the source is regenerated grammar stored in the mygrammar[] array. This is an array of productions and it includes all the terminals which are not really stored in the tree.

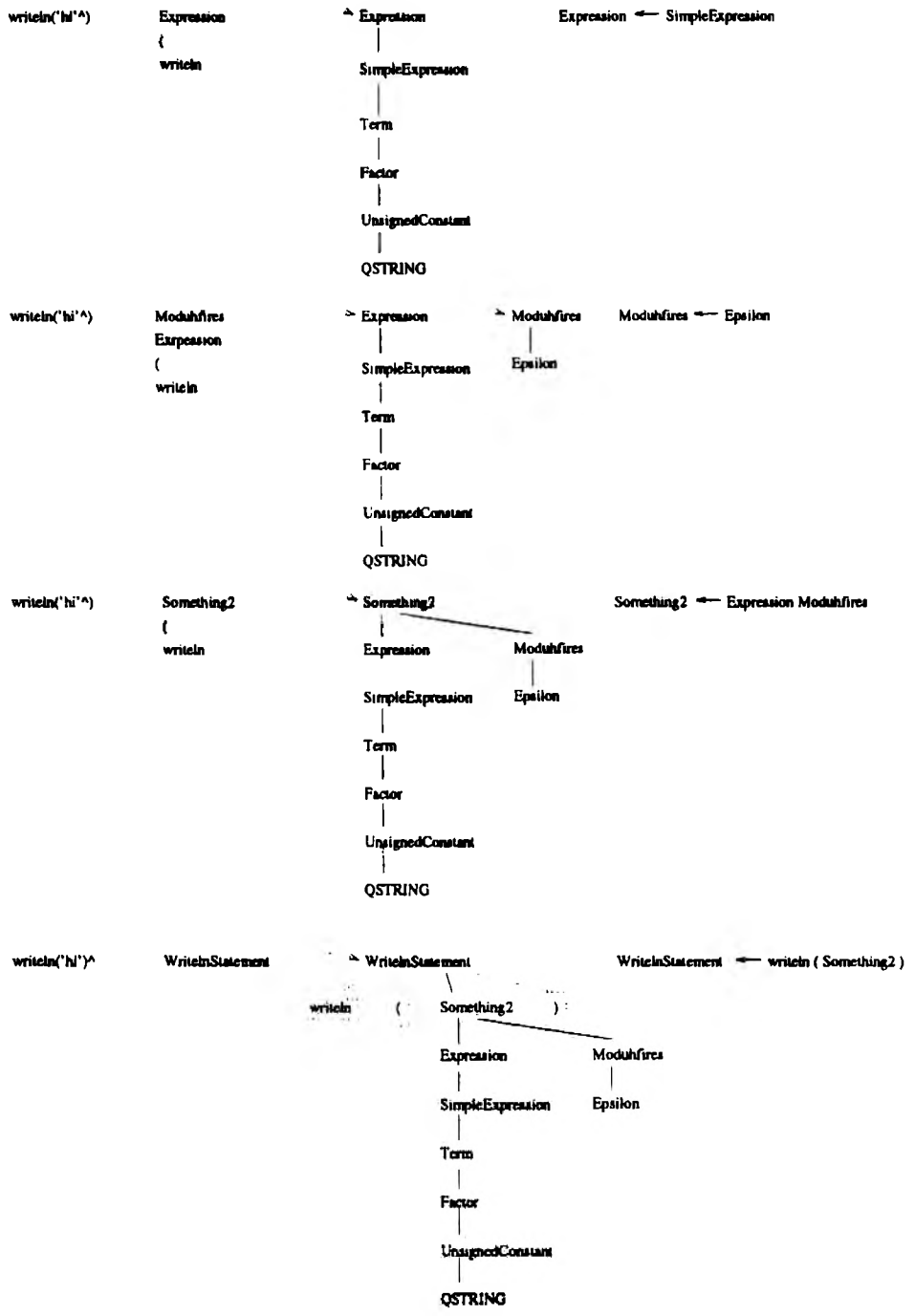


Figure 21. Writeln Example Part 2

2. The nt.cpp Module. LSD's grammar information exists in two places in the source code. The parser.y yacc input file has a complete copy of the grammar in terms of logical symbols listed in the %token <> and %type <> statements. The %token <> symbols are exported to yytab.hpp as part of the yacc processing. The %type <> statements are for non-terminals of the grammar that is listed in the parser.y file. This copy of the grammar is not usable by the rest of LSD however.

The nt.cpp file has another copy of the grammar stored in the mygrammar[] array. This is an array of production objects. The production objects are defined in the productn.cpp and productn.hpp files. The production object has a field tag to store the unique production tag. This tag is used to differentiate between two productions with the same left hand side non-terminal. The production object also contains a field lhs to store the left hand side non-terminal number. There is also an array rhs[] to hold the right hand side item numbers. Negative numbers are non-terminals, positive numbers are terminals. The kount field holds the number of right hand side elements. Two additional arrays, tabinfo[] and rhscode[] exist to hold formatting information used when source is created from a tree.

What integers are used for the lhs field and the rhs[] array? Since the non-terminals from parser.y are not usable, a separate array slugarray[] of character pointers is used to hold the string names of the non-terminals. Whenever any part of LSD needs to work with the grammar the string names of non-terminals are used instead of integers so that the code is easier to read. Two functions

exist to work with slugarray[]. The ntname() function converts an integer (which would be stored in the mygrammar[] production objects) to a string name. The ntnum() function converts a string name to a number.

The terminals used for the lhs field and the rhs[] array of the production objects are the same used by the parser.y file. However, just as the non-terminals are referred to by name for the sake of maintainability and readability, the terminals are also referred to by name. The keywords_instance[] array of keyword objects is used to hold this information. The keyword objects are described in the keyword.cpp and keyword.hpp files. They have two fields, one for the name and one for the integer value. The keyword_init() function is called to allocate space for the array and load it when LSD starts, and when LSD finishes the keyword_done() function is called to free the memory. Two functions are used to access this array. The keyword() function converts a string to the integer value used by the grammar, and the figureitoutlater() function converts an integer value to the string used by the rest of the LSD program.

One additional lookup array exists to mirror the slugarray[] array. The badwords[] array holds the string names of the terminals which are not supported by LSD but which are in the Borland Turbo Pascal language. This array is accessed using the isbadword() function which returns 1 if the string passed in is in the badwords[] array, and 0 if the string is not.

The structure used to hold the grammar, the array of production objects called mygrammar[], has been discussed.

However, it remains to be seen how mygrammar[] is loaded. The fire_it_up() function is called to load the grammar when LSD is started. The fire_it_up() function is a series of calls to the addprod() function. The addprod takes the tag, the left hand side string name, an integer value for the count of right hand side items, and a comma delimited list of the right hand side items' string names. These names are the terminals in keywords_instance[] and the non-terminals in slugarray[].

The addprod() function uses the ability of C++ to have functions with variable numbers of parameters so that one function can be used for all productions. If the calls to addprod() are carefully examined you will notice that some of the terminals and non-terminals have special prefix characters. These consist of the tabbing codes and the codes to determine when a newline should occur when source is created from a parse tree. This information is used in the emit() method of the TREETHING objects and is the source of the information for the tabinfo[] and rhscode[] arrays in the production objects.

The fire_it_up() function allows the grammar to be loaded quickly and yet it is still readable for easy maintenance. Since everything is done with the names of the terminals and non-terminals, a change of a name is easily accomplished. Changes of the grammar which are necessary when parser.y is changed are also easy to make.

The nt.cpp file contains other functions used by LSD to interface with the grammar system. The dump_grammar2() function is used by lsdmt.cpp to create the list of grammar productions used

for the grammar window. The whichprods() function is used by graftree.cpp to generate the pick list of productions for node insertions of non-terminals. This function fills the gramchoice[] array with all the productions with a given non-terminal for the left hand side. The number of productions available is stored in gramcount. This function is called in the pdialog_func() dialog function in the graftree.cpp file.

3. Other Files.

a. The string Object. The string object is described in string.hpp and string.cpp. This object is used to store the text of the lines of text for LSD. The string object has a maximum and a length as well as the actual text. All the data is private and methods are used to manipulate the data.

b. The aline Object. The aline object is described in aline.hpp and aline.cpp. This object is used to store one line of text for LSD. The aline object has a string object, a next pointer and a previous pointer. A linked list of aline objects is used to hold the text buffers of LSD for the text editing, the grammar window, and the results window. A text buffer in LSD is a linked list of aline objects.

c. The fileio Module. The fileio module has two functions for accessing files. The load_file() function is used to read a file on disk into memory as a linked list of aline objects. The unload_fileas() function is used to write a file on disk from a linked list of aline objects as a text file.

d. The comment_item Object. The comment_item object is described in comment.hpp and comment.cpp. This object is used to store the

text of the compiler directives for LSD. The comment_item object has a character array to store the text and a next pointer. Each TREETHING object may have a linked list of comment_item objects for the compiler directives which occurred. The comment_item objects are built in the parser.y yacc routine yyparse(). Each production which has a terminal checks to see if the cd field of the YYSTYPE object is NULL. If it is not NULL, then a comment_item object is created to hold the information from the cd field and the newly created comment_item object is added to the linked list of comment_item objects headed by the compiler_directive field of the TREETHING object which corresponds to the current production being matched.

e. The utils Module. The utils.hpp and utils.cpp files contain the information for the utils module of LSD. The routines are bomb(), upshift(), genid(), genstamp(), get_default_font() and fixcolors(). The bomb() function takes an error message and then puts up a message box with that error and then halts the program. The upshift() function will change all the letters in a character array to upper case. This is used before comparisons so that case-insensitive equivalence can be done. The genid() function will change the first letter in a character array to a uppercase letter and all the rest of the letters will be lower case. This is used by the emit() method of the TREETHING object during recreation of source from a tree. The genstamp() function is used by the emit_prep() function in the yystype.cpp file to add a date and time stamp to regenerated source files. The fixcolors() function is used in lsdmt.cpp and graftree.cpp to force a white background for

the windows. The get_default_font() function is used by lsdmt.cpp and rwindow.cpp to query PM to find a default fixed-pitch font for use in the windows of LSD.

f. The Treething Object. The TREETHING object is described in the yystype.hpp and yystype.cpp files. The TREETHING object has six pointers to other TREETHING objects. The rnext and rprev pointers are the recovery chain pointers so that a delete of the TREETHING objects allocated can be done even if the tree is not complete and no common root exists for the allocated subtrees. This is used by the yypunt() function in the yaccpar.sk file used by yacc to create the parser. The next and prev pointers point to the next and previous elements in the production of which the TREETHING object having these pointers is in the right hand side. The rhs pointer points to the linked list of TREETHING objects representing the right-hand side of the TREETHING object. The lhs pointer is used to point to the left hand side TREETHING for this TREETHING object.

The idname field of the TREETHING object is used to hold the text of a terminal which is not inferrable from the grammar, such as an identifier. One constraint is that no production may have two such fields, but this was not a problem for the grammar LSD uses. The newprod field of the TREETHING object is used to store the unique tag for the production which is used to differentiate productions in the mygrammar[] array in the nt.cpp file. The expanded field of the TREETHING object holds TROO if the right hand side exists, and FALLS if it does not. The TROO and FALLS constants are defined in the constants.hpp file. The expanded field is used by the graftree.hpp file during creation of new nodes

on the tree since for a new production which has non-terminals on the right-hand side no right-hand side can exist at the time the TREETHING node is being created, and some way to insure that no attempt to use the rhs pointer must exist to prevent unmitigated disaster and destruction as the program wantonly steps all over memory in an attempt to walk a tree that is not there.

The emit() method of the TREETHING object is used to recreate source from a tree. The whoami() method is used by parser.y to put the matched statements into the results window of the lsdmt.cpp file. The fixit() routine is used by the yypunt() function in the yaccpar.sk file to NULL out the pointers to other tree elements before deletion through the recover chain. This prevents the destructor from trying to recurse through the tree (which probably is not there since the deallocation will be in allocation order, not tree order). The other methods of TREETHING are self-explanatory and are used to access the fields of the TREETHING object.

g. The YYSTYPE Object. The YYSTYPE objects are described in the yystype.hpp and yystype.cpp files. They are used by the scanner to transmit the information about tokens to the parser. The parse stack used by LSD is actually a stack of YYSTYPE objects. This is not how yacc usually works. Normally yacc uses a stack of integers, but simply knowing which token has been seen is not enough for the LSD program. Using the YYSTYPE objects allows including all necessary information in a neat package.

The YYSTYPE object has lineno and colno fields to hold the line and column numbers upon which the token starts. This

information is used in the errorlib.cpp file when an error occurs to let the user know where the error is in the text buffer. The tokenum field is used to hold the integer value of the token. It will be one of the constants from the yytab.hpp file created by the yacc processing of parser.y and is one of the constants described on one of the %token <> lines.

The cd field was described earlier and holds the compiler directive which immediately preceded the token if there was one. The toketext field is used to hold the actual string which comprises the token from the text buffer.

The anode field is used by the parser.y file to pass the synthetic attribute of the subtree for a given non-terminal up the tree during the parse so that the tree can be built bottom-up. See the APPROPRIATE SECTION. The whichone field is used by the scanner.l file to signal whether the real, integer, cmd, or strang field is appropriate. These fields are used to hold the values of literals which are not inferrable from the grammar.

h. The infotable Object. The infotable object is described in the infotabl.hpp and infotabl.cpp files. The infotable object is used by graftree.cpp to hold the information for the currently displayed production's subtree elements. The name field stores the terminal or non-terminal name. The width field stores the length of the name field. The number field holds the terminal or non-terminal number. A negative number means this is a non-terminal number. The location[] array holds the four corners of the bounding box for the text on the window in window coordinates. The line[] array holds the (x1,y1,x2,y2) coordinates of the line from the left hand

side element to this element. It is not used for the left hand side element. The xpanded field is TROO if the right hand side of this item exists and FALLS if the right hand side of this item does not. It is used for non-terminals and parallels the expanded field of the TREETHING object. The node pointer is a pointer to the node this infotable object is describing. The up pointer is a pointer to the left hand side node. The reset() method is used to clear out the information in an infotable object so the object can be re-used.

i. The graftree Module. The graftree module is responsible for the tree editing portion of LSD. The gedit() function is called by lsdmt.cpp in response to the LSD_TREE_EDIT message. The gedit() function creates a standard frame window using window_func3() as the message handler. The window_func3() function performs the usual PM overhead work for choosing fonts and setting colors and by calling the PM_prep() function. Next the showit() function is called to draw the current subtree, which will be the one off of the root. As the last step of initializing the window messages are sent to activate the appropriate menu choices.

The window_func3() takes care of keyboard, menu, and mouse input as described in the user manual. The interesting parts are the actual insertion, deletion, hook activities, and traversing of nodes. The treedelete() function is called to delete a subtree. The treeinsert() function is called to insert a subtree. The move_to_hook1() and move_from_hook1() functions are used for accessing hook1. The move_to_hook2() and move_from_hook2() are used for accessing hook2. To traverse the tree the treeright(),

treeleft(), treeup(), and treedown() functions are used. These functions all act on the current node which is the node corresponding to the element of the it[] array of infotable objects at position whichhot. The current node in the tree is pointed to by current_node.

For the traversal options that remain within a production, the delete option, and the move to hook options, only whichhot is changed and then showit() is called to redraw the screen. For the other options the it[] array is recalculated and showit() is called to draw the tree. For window_func3(), PM_prep() is called for preparing to redraw, and one of the functions called is showit_prep(). The showit_prep() function will recalculate the it[] array using the node passed in as the left hand side position in the tree. The hook, deletion, and insertion functions modify it[] directly.

The treeinsert() function will call an application modal dialog box which has the pdialog_func() window procedure. The which_prods() function from the nt.cpp file is called to fill the gramchoice[] array and the WM_INITDLG message handling will take this information and display the productions available in the dialog list box. The user can either choose a production or cancel. If cancel is chosen the treeinsert() function does nothing more. Otherwise the insertion() function is called to perform the actual tree modification creating a new right hand side according to the production chosen. The current_node is set to point to the new left hand side and the tree is redrawn.

The treedelete() function will call the deletion() function to perform the actual tree modification deleting the appropriate right hand side's subtree.

The push() and pop() routines are used to store the right hand side position for productions which are higher up in the tree than the current production. This allows LSD to return the user to the appropriate right hand side item when the treeup() function is called to go up one level of the tree.

Several other functions exist in graftree.cpp to perform common tasks. These include do_non_terminal() and do_terminal() which fill the name field of the it[] array elements. The mouse_adjust() function is used to decide whether to move the whichhot position in response to a mouse click or not. The moveme() function changes the whichhot and redraws only the two items' changed boxes. This is much faster than calling showit() to redraw the whole production. The showanode() function is used to draw one element of the it[] array.

j. The rwindow Module. The rwindow.cpp and rwindow.hpp files contain the functions used for the results window that are not particular to either the parse results or the grammar display. These include the window procedure window_func2(), its support functions, and the functions for creating the buffer to be displayed in the results window.

The support functions are local to this module and include all functions which are used to scroll the window. These functions and their associated tasks are shown in table I. The functions for creating the buffer to displayed are externally visible and they

Table I. Support functions for rwindow module

Name	Task
top2()	Move to top of buffer
bottom2()	Move to bottom of buffer
home2()	Move to column 0 of buffer
end2()	Move to column after last letter of buffer
redo_hscroll_bar2()	Adjust horizontal scroll bar
redo_vscroll_bar2()	Adjust vertical scroll bar
redo_screen2()	Redraw window

are used by nt.cpp and parser.y files to create the grammar display and parse results respectively. These functions and their associated task are shown in table II. The rwindow module uses an

Table II. Buffer functions for rwindow module

Name	Task
nuke_status_lines()	Eliminate buffer
init_status_lines()	Initialize buffer
add_status_line()	Add 1 line to buffer in first-in, first-added order
add_sorted_status_line()	Add 1 line to buffer in ascending alphabetical order

identical text buffer to the one used by the lsdmt module for holding the pascal source. The text buffer is a linked list of aline objects with pfirstline as the pointer to the head of the list. The only tricky code in the window_func() window procedure is for managing the separate thread for the case when a parse is being done. The overall structure is described in the OS/2 Information section.

k. The parseit Module. The parseit.cpp and parseit.hpp files contain the information about the parseit() function. The

parseit() function is called by the window_func() window procedure in lsdmt.cpp to parse the source in the text editing buffer. The parseit() function creates a results window after setting the showparse flag to 1. This lets the window_func2() window procedure in rwindow.cpp know it should be a parse results window.

l. The sgrammar Module. The sgrammar.cpp and sgrammar.hpp files contain the information about the sgrammar() function. The sgrammar() function is called by the window_func() window procedure in lsdmt.cpp to show the grammar used by LSD. The sgrammar() function creates a results window after setting the showparse flag to 0. This lets the window_func2() window procedure in rwindow.cpp know it should be a grammar results window.

m. The lsdmt Module. The lsdmt.cpp and lsdmt.hpp files contain the code for the main() function of LSD. The lsdmt module contains the code for the text editing portion of LSD as well. The charts in table III and table IV show the functions in the lsdmt.cpp file and their associated tasks. The window_func() window procedure of the hand_frame window is the main event handling routine of LSD. This frame window is the text editing window. The parse window, the grammar window, and the tree edit window are all invoked by window_func() and upon completion of any of these three windows control returns to the window_func() function.

C. OS/2 INFORMATION

IBM's OS/2 2.1 operating system provides many of the services needed for LSD. These include windowing routines, timer services, thread routines, semaphores, and the message handling system. Since these capabilities are part of the operating system, I was

Table III. Functions in the lsdm module, part 1

Name	T	
main()	M	ogram
start_busy_mode()	Ch	icon,m itle bar
	ar	e edit ow read only
end_busy_mode()	Un	art_bu e()
recalc()	Re	late w limits
title_bar()	Dr	it win itle bar
move_cursor()	Mo	sor an date title bar
redo_hscroll_bar()	Up	orizon scroll bar
redo_vscroll_bar()	Up	ertica roll bar
redo_screen()	Re	dit fra window
window_func()	Wi	roced or edit window
jump_dialog_func()	Di	indow pr edure for
	use	ump to e dialog
about_dialog_func()	Dia	ndow pr edure for
	use	out dia g
nuke_file()	Elim	edit bu fer
new_file()	Init	e edit b ffer
bs()	Proc	ackspace key
delchar()	Proc	elete ke
left_arrow()	Proce	eft arro key
right_arrow()	Proce	ght arrow key
up_arrow()	Proce	arrow key
down_arrow()	Proces	wn arrow key

Table IV. Functions in the lsdmt module, part 2

Name	Task
page_up()	Process page up key
page_down()	Process page down key
hscroll()	Do horizontal scrolling for edit window
atextletter()	Process letter key
home()	Process home key
end()	Process end key
enter()	Process enter key
do_the_parse()	Function to do the thread for parsing the text buffer to create the parse tree
do_the_regen()	Function do do the thread for regenerating the text buffer from the parse tree
add_a_line()	Function to add one line of text to the edit buffer
delete_line()	Function to delete one line of text from the edit buffer
insert_line()	Function to insert one line of text into the edit buffer
bottom()	Move to bottom of buffer
top()	Move to top of buffer
AddAboutToSystemMenu()	Add choice to system menu for LSD to show the about box

able to concentrate on the algorithms necessary to solve the problem instead of spending a great deal of time with the interface. With multi-threading I was able to improve LSD's responsiveness to the user.

OS/2 Presentation Manager (PM) is an event driven message based system. When an event occurs, such as a key pressed on the keyboard, or a move of the mouse, this causes PM to generate a message and route it to whichever window is appropriate. If no activity occurs to generate messages for a particular window, the thread of the process which is the creator of the window will not be sent any messages and will be idle.

Each window in PM has an associated window procedure. This window procedure is the part of the code that receives the messages from PM about the activities in that window. A user's application is built as a message handler and the operating system routes all messages which the application should get to the window procedure. The window procedure then does whatever is necessary to process the messages.

LSD uses a frame window like the one shown in figure 22. In a frame window, the client area is the responsibility of the the window procedure. The other areas such as the scroll bars and the title bar are handled by PM. The information which results from events in those areas of the frame window is passed to the window handling procedure in the form of messages. The window procedure can also send messages to those windows. For instance, when the user moves the tab in the scroll bar, this causes PM to send a message to the window procedure with information about the new

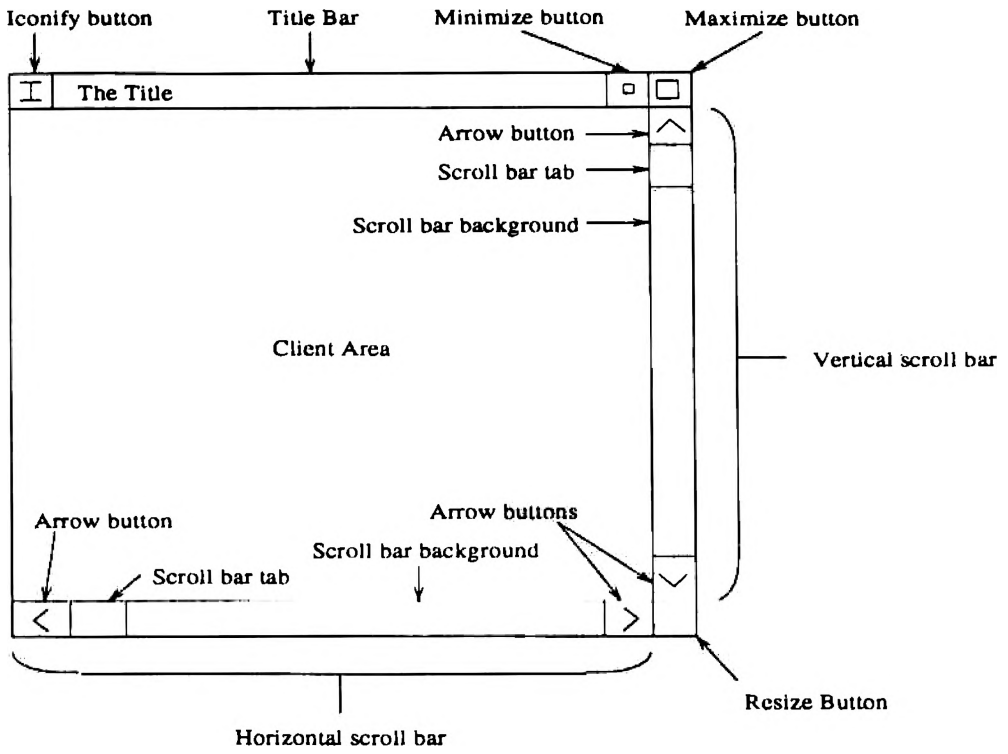


Figure 22. Frame Window

location of the the scroll bar tab. If the application redraws the window it can send a message to the frame window to change the title bar or reposition the scroll bar tab.

The window procedure has roughly 0.1 seconds to process a message and return. If the window procedure takes longer than this then PM may decide that the application is dead (not responding to messages) and terminate the application. This is done in order to protect the system message queue. If an application fails to process messages then the process message queue could fill to capacity. This means that PM cannot put any more messages for the application into the application message queue. This would in turn cause the system message queue to fill to capacity and crash PM. How can an application perform any message response that takes

longer than 0.1 seconds? OS/2 provides a solution to this problem by providing support for threads.

A thread is a path of execution through an address space. Traditional operating systems support multiple processes running, with each process having exactly one thread. A multi-threaded system supports more than one thread of execution running in a process' address space. OS/2 is multi-threaded and supports the creation of multi-threaded applications. All of the process' global data will be visible by all the threads within the process. This means there could be a problem if two threads are trying to change global data simultaneously. Successful multi-threaded programming requires the programmer to insure that resource access is synchronized between threads as part of the program design.

If a message will take more time to handle than the 0.1 seconds, then an application should start another thread to do the work and then the thread which handles the message can return. This poses an interesting problem, however. What if the thread started makes changes that requires updating the window? The window procedure needs to know when the thread is done and then it can do the screen update. Another solution is to have the thread do the screen update, but then the window procedure will still need to know this is occurring so that it does not try to update the screen at the same time.

Semaphores can be used to synchronize the access to a shared resource, or to signal that some event has happened or needs to happen. OS/2 provides operating system support for semaphores.

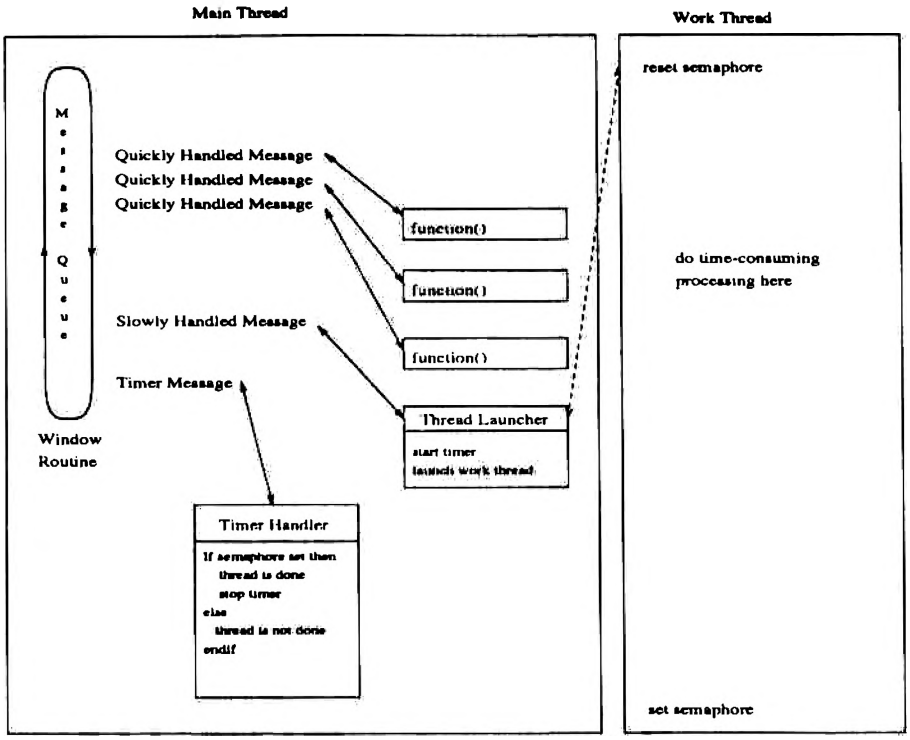


Figure 23. Asynchronous Work Thread Usage

This, combined with the timer services also provided by OS/2, allows easy synchronization of threads.

For the LSD program, semaphores are used to signal the window procedure that a thread has completed its processing. The general structure of the code required to do an asynchronous work thread is show in figure 23. The OS/2 Presentation Manager (PM) turns all keyboard and mouse actions into messages and routes them to the proper windows. PM will also route messages for an application. All messages for an application go into that process's own message queue to be handled by the associated window procedure. An application may have more than one window procedure, each with its own message queue, but this discussion only covers what happens when there is a single application message queue. Messages get put

into the queue by PM and the priority assigned to different types of messages will determine what the next message to be seen by the window procedure will be. If no messages exist in the queue, the window procedure is idle. When messages exist, the window procedure will be given messages one at a time to process, and the window procedure will have 0.1 seconds to finish and return control to the message dispatcher.

In figure 23 three types of messages are shown: Quickly Handled Messages, Slowly Handled Messages, and Timer Messages. For actions which take less than 0.1 seconds, the message is a Quickly Handled Message and this will result in a function call, which will perform the action and return and the window procedure will return control to the message dispatcher. For actions which may take more than 0.1 seconds, a thread launcher function is called. The thread launcher will start a timer and then start the work thread. The timer function tells PM to send a special timer message to the window procedure every so many milliseconds. When the timer message arrives, the Timer handler code is executed. This will check to see if the semaphore is set. If the semaphore is set, then the work thread is done. The timer handler will then turn off the timer and return control to PM. If the semaphore is not set, then nothing is done and control is returned to PM. The semaphore is used to signal the main thread's window procedure that the work thread is finished. When the work thread starts it resets the semaphore and does its work and only sets the semaphore as the last step.

Just as PM supports multiple message queues per process, it also supports multiple timers and multiple distinct semaphores. LSD uses two different semaphores and two different timers. One semaphore/timer combination is for the parse thread, and the other semaphore/timer combination is used for the reformat.

III. USER MANUAL FOR LSD

A. INTRODUCTION

To start LSD, simply type LSDMT. When the program starts an information box will be displayed. Click on the enter or press the enter key. Now the text editing window will appear. It will have a title bar which says (0,0,1)"New File" Insert Mode. The numbers are row, column, and line count for the file. The rows and columns start on zero, not one. There will also be two scroll bars, one for horizontal movement and another for vertical movement.

Initially the vertical movement scroll bar will not function since all the lines of the new file will fit in the window. A menu bar is just below the title bar. The menu bar has choices for File, Edit Text, Tree Edit, Reformat, Parse, Grammar, Font, and Help. Some of these items are highlighted and some are not. The highlighted menu items are currently available, and the non-highlighted menu items cannot be used. The appearance of the highlighting will depend upon the desktop configuration. Only five of the choices are initially available, File, Edit Text, Grammar, Font, and Help.

To select a menu bar item the user may either move the mouse pointer over the item and click the mouse's left button, or you may depress the alt key and the underlined letter of the menu choice. To load a file into LSD for processing, use the File menu bar choice. This will bring down a pull down menu and the user can again make choices either by using the mouse or the alt-key combinations. If a menu item has one letter underlined, this is an accelerator key. This allows the user to type in that letter while

depressing the alt key to select that menu choice. This is known as an alt-key combination.

Initially there exists a choice of Open, SaveAs, or Quit. Close and Save make no sense if no file is loaded, so they are unavailable. Once the user chooses Open, a file dialog box appears. The user can either type in the filename in the Open filename area and then click on the Ok (or hit return), or the user can search around the disk using the advanced file dialog features. The file dialog is a standard feature of OS/2 2.1 and is described in the OS/2 manuals.

Once a file has been loaded, the Parse menu bar option becomes available. Also, now that the file is loaded, you may edit the text. The text editing commands will be discussed in a later section. To process the text and generate a parse tree, choose the Parse option. This will start the parse engine. When the source which is in the text editing buffer has been parsed, the results are shown in a parse window on the screen.

If an error occurs during the parse, the title bar of the parse window will be 'Bad Parse', otherwise the title bar will be 'Good Parse'. The scroll bars (or the arrow keys) may be used to scroll the text in the parse window so that the entire file may be seen. The text displayed in the parse window is a list of the productions matched during the parse. If the parse was bad, the error message is displayed after the last matched production.

There is a menu bar on the parse window which will be discussed in the parse window section. The important choice now is the Quit option, which can be selected with the mouse, the alt-Q

key, or the ESC key. This will close the window and return to the text editing window. If the parse was bad, the source needs to be fixed using the text editing window.

Once a good parse occurs, the last two text editing window menu bar choices become available. These are Tree Edit and Reformat. The Reformat option will generate new source text from the parse tree created during the parse step. This text will replace the text in the editing window. The file is unchanged, however. To save the text in the edit window at any time, the user can choose the File option, then the Save option of the pull down menu. To save it to a new file, the user can choose the SaveAs option instead of the Save option.

The Tree Edit option is the most complicated, but also the most powerful. It will start the tree edit window which allows direct editing of the parse tree which was created during the parse step. The tree editor is discussed in detail in a later section, but some of its features will be described now.

The tree editor shows one production in the tree, with the left-hand side at the top of the window and the right-hand side elements across the bottom of the window. These items are either terminals or non-terminals (the left-hand side is obviously always a non-terminal). The current item is highlighted and has white text on a black background. To change the current item the user can use the mouse and position the cursor over the item to make the current item and then click the left button once. All editing in the tree edit window occurs on the current item.

If the current item is not the root of the tree, the user can double-click on the left-hand hand side item. This will move the current item up one level of the tree and the production which the left-hand side is in will be shown. The current item will be the right-hand side element which is really the left-hand side item from the production being viewed before the double-click.

If the current item is not a terminal and is a right-hand side item, then a double-click with the left button will go down one level of three. The production which has that non-terminal as the left-hand side will be shown. The current item will be the same item, displayed as the left-hand side of the production.

Using these two techniques, the entire tree can be traversed and the user will see how every production is connected. Editing the tree, as opposed to browsing it, is covered in the Tree Editing section of the paper. To exit the tree editor, choose the Quit menu bar option or press the ESC key.

To exit LSD the user needs to insure that only the text editing window is open. Then the File option is chosen from the menu bar, and the Quit option from that pull down menu should be selected.

A normal sequence of operations using LSD would be to:

1. Load in a file.
2. Parse it.
3. Use the tree editor to modify it.
4. Use the reformat option to create the new source.
5. Use the text editor to add any comments to the source.
6. Save the new source into the old file with the save option.

To make sure no syntax errors were introduced during the text edit, a final parse may be done.

LSD can guarantee a syntactically correct program, but the semantics are still entirely in the hands of the user. While LSD can make sure every **BEGIN** has a corresponding **END**, and that **VAR** occurs in the right place, it cannot check to make sure a variable has been declared or the types of an expression are correct.

B. TEXT EDITOR DETAILS

The text editor works on an edit buffer. The edit buffer is a copy of the file opened with the **Open** pull down menu option of the **File** option of the menu bar. Any changes made are made to this edit buffer, not the file. In order to make the changes permanent the edit buffer must be saved back to the file using the **Save** menu option of the **File** option of the menu bar. The edit buffer can potentially hold many more lines of text than can be displayed in the text edit window. In order to allow the user to work on the entire file, some way of changing which lines are being displayed must be provided. In LSD the text in the window can be scrolled to show different portions of the file by using cursor movement, which is described next.

1. Cursor Movement. Within the edit window is a cursor. The cursor is a black square which shows the current text entry position. This is where any newly typed text will go. Scrolling in the window to see different areas of the edit buffer is accomplished by moving the cursor. The cursor is moved around in the edit buffer and the window text is scrolled by forcing the cursor to remain within the window. Any time a movement of the cursor will result in a new cursor position which is within the

text edit buffer but outside of the window occurs, the text in the window is scrolled to adjust the displayed text. The text shown in the window is moved so that the current cursor position is moved in the edit buffer but not in the window. Instead, the part of the buffer which includes the cursor position in the edit buffer is now within the window. This will be explained in detail next.

a. Vertical Cursor Movements. To move the cursor vertically, the arrow keys are used. If the cursor is on the top line in the window and you press the up arrow key with more lines existing before the line displayed, then the text in the window will move down, and the top line will be the line immediately before the old top line. If the cursor is at the bottom line in the window and the down arrow key is pressed with more lines existing after that line in the edit buffer, then the second line from the top becomes the top line. The text in the window is scrolled upward and the line immediately after the old last line in the window becomes the new last line in the window.

For faster movement, the PGUP and PGDN keys are used to move a window-full at a time upward and downward respectively. If a move up is attempted and the cursor is at the top of the buffer, or a move down is attempted and the cursor is at the bottom of the buffer, then nothing happens.

On the vertical scroll bar, there are arrows at either end. These arrows correspond to the up and down arrows on the keyboard. The user can click on the arrows with the mouse and get the same results as by pressing the corresponding arrow keys. Within the scroll bar is a "tab", a grey rectangle which is highlighted. This

item can also be used to scroll the window. If the user puts the mouse cursor over this tab, then depresses the left mouse button, then moves the mouse forward or backward while holding down the left button, the text is scrolled in the direction opposite the direction the mouse is moved. If the entire file fits in the window then the vertical scroll bar is inactive.

If the user clicks in the scroll bar below the tab, a PGDN is simulated. If the user clicks in the scroll bar above the tab a PGUP key is simulated. The tab size will vary with the amount of the file displayed. The more of the file which is displayed in the window, the larger the tab will be. A small, square tab means a file much larger than the text editing window is in the text edit buffer.

There are two more cursor movement key combinations. ctl-PGUP will move the cursor to the top of the edit buffer and ctl-PGDN will move the cursor to the bottom of the edit buffer. These cursor movements are also available as the Top of File and Bottom of File options on the Edit Text menu bar.

b. Horizontal Cursor Movement. To move the cursor horizontally, the arrow keys are used. If the cursor is on the rightmost column in the window and the right arrow key is pressed while the cursor is not on column 127, then the text in the window will move left, and the rightmost column will be the column immediately after the old rightmost column. If the cursor is at the leftmost column in the window and the left arrow key is pressed while the cursor is not on column zero, then the text in the window will move right,

and the leftmost column will be the column immediately before the old leftmost column.

For faster movement, the END and HOME keys are used to move to the end of the line and column zero respectively. If an attempt is made to move right and the cursor is on column 127 of the buffer, or an attempt is made to move left and the cursor is on column zero of the buffer, then nothing happens. On the horizontal scroll bar, there are arrows at either end. These arrows correspond to the left and right arrows on the keyboard. Clicking with the mouse on these arrows produces the same results as pressing the corresponding arrow keys.

Within the scroll bar is a "tab", a grey rectangle which is highlighted. This item can also be used to scroll the window. If the mouse cursor is placed over this tab and the left mouse button is depressed, then the mouse is moved left or right while the left button is held down, the text is scrolled in the direction opposite the direction the mouse is moved. If the user clicks in the scroll bar to the left of the tab, the cursor is moved 8 columns leftward, and if the user clicks the scroll bar to the right of the tab, the cursor is moved 8 columns rightward. The tab size will vary with the amount of the file displayed. The more of the file which is displayed in the window, the larger the tab. A small, square tab means a file much wider than the text editing window is in the text edit buffer.

2. Modes. The text editing window is modal. The three modes are insert, overwrite, and busy. The current mode is always displayed

in the title bar of the text editing window on the far right. Each of these modes will be described in detail.

In the insert mode, if a letter is typed then the letter is inserted at the cursor position, moving any letters that were after the cursor's position to the right. If a return is typed then the line the cursor is on is split at the current cursor position. If the cursor is past the end of the text on the line then a blank line is inserted. Either way, the cursor is moved down to the new line.

In overwrite mode any letter typed replaces whatever letter was under the cursor. If the cursor is beyond the end of the text on the line then the space between the last letter on the line and the current cursor position is filled with spaces and the new letter is inserted at the current cursor position.

All letters in either mode that you type are placed at the current cursor position, and the cursor position is advanced one column. To toggle between insert and overwrite mode, use the INS key. The text editor is initially started in insert mode.

The third mode, busy, is a special mode which is activated whenever the LSD program is busy doing something which will not allow the user to edit text. During busy mode, the cursor movement keys and the scroll bars are available, but no editing functions are available. Busy mode is entered whenever a parse, a reformat, a grammar viewing, or a tree edit is occurring. When one of those activities is done, the mode will return to insert or overwrite, whichever was active when busy mode was initiated.

3. Inserting a Line. To insert a line, use ctl-N. This will add a blank line at the cursor position. This can also be done with the Insert Line option on the pull down menu of the Edit Text menu bar. This command does not work in busy mode.

4. Deleting a line. To delete a line, use ctl-Y. This will delete the line at the current cursor position. This can also be done with the Delete Line option on the pull down menu of the Edit Text menu bar. This command does not work in busy mode.

5. Edit Window Keys. The edit window keys are listed in table V. The items marked with an asterisk are unavailable in busy mode. The ctrl- prefix means to press the ctrl key, so ctrl-PGDN means to press the ctrl key and then press the pgdn key. The alt- prefix means to press the alt key, so alt-P means to press the alt key and then press the P key.

Table V. Edit Window Keys

Key	Action
F1	Get Help
ctrl-Y	Delete line *
ctrl-N	Insert line *
ctrl-J	Move cursor To line
ctrl-PGUP	Move cursor to top of buffer
ctrl-PGDN	Move cursor to bottom of buffer
PGUP	Move cursor up 1 windowfull
PGDN	Move cursor down 1 windowfull
↑	Move cursor up
↓	Move cursor down
→	Move cursor right
←	Move cursor left
Del	Delete character *
Ins	Toggle insert mode *
Home	Move cursor to column 0
End	Move cursor to end of line

C. PARSE WINDOW

The parse window appears as the result of a parse request. The parse window will show a record of the productions matched during the parse. If the parse has an error, then the error message will be displayed after the last successfully matched production.

1. Parse Window Usage. The parse window is similar to the text editing window. The scroll bars and menu function identically, as do the up and down arrow keys. The left and right arrow keys cause a scroll to the right and left one column respectively. The HOME, END, PGUP, PGDN, ctl-PGUP and ctl-PGDN keys also work in the same way as they do in the edit window. Since the parse window does not allow editing, the INS, DEL, ctl-Y, ctl-N, and the regular letter keys do not do anything. The return (or enter) key will advance the cursor one line and move the cursor to column zero.

The parse window menu has three options. Help and Quit work the same as the corresponding text edit menu choices. The SaveAs menu choice brings up the save file dialog and allows the user to save the contents of the parse window into a file.

The parse window is initially displayed as if the user had done a ctl-PGDN so that the end of the parse results are shown. This is so that if there is an error it is displayed immediately. If the parse is a good parse, the title bar will say 'Good Parse'. If the parse is bad, the title bar will say 'Bad Parse'. While the parse window is open, the edit window is in busy mode.

Table VI. Parse Window Keys

Key	Action
F1	Get Help
ctrl-J	Scroll window to Line
ctrl-PGUP	Scroll window to top of buffer
ctrl-PGDN	Scroll window to bottom of buffer
PGUP	Scroll window up 1 page
PGDN	Scroll window down 1 page
↑	Scroll window up 1 line
↓	Scroll window down 1 line
→	Scroll window right 1 column
←	Scroll window left 1 column
Home	Scroll window to column 0
End	Scroll window to last column of current line

2. Parse Window Keys. The parse window keys are listed in table VI.

D. GRAMMAR WINDOW

The grammar window appears as a result of the grammar text editing window menu choice. The grammar window behaves identically to the parse window except that it starts at the beginning of the information initially instead of at the end. The text displayed is an alphabetical list of all the productions in the grammar.

1. Grammar Window Usage. The SaveAs option is particularly useful to create a file containing the grammar of LSD. This file can then be printed or saved. If there is a parse error but the user is unsure what the grammar requires, the grammar window will show the necessary information. The text editor is in busy mode while the grammar window is open.

2. Grammar Window Keys. The grammar window keys are listed in table VII.

Table VII. Grammar Window Keys

Key	Action
F1	Get Help
ctrl-J	Scroll window to Line
ctrl-PGUP	Scroll window to top of buffer
ctrl-PGDN	Scroll window to bottom of buffer
PGUP	Scroll window up 1 page
PGDN	Scroll window down 1 page
↑	Scroll window up 1 line
↓	Scroll window down 1 line
→	Scroll window right 1 column
←	Scroll window left 1 column
Home	Scroll window to column 0
End	Scroll window to last column of current line

E. TREE EDITING

Tree editing consists of several categories of activity, including tree navigation, subtree deletion, subtree insertion, and subtree moving. Navigation is accomplished in one of two ways, either using the mouse or the cursor keys. The current node is highlighted by having white letters on a black background. The text editor is in busy mode while the tree edit window is open.

1. Tree Navigation. The current node is changed in one of two ways, either by using the mouse or using the arrow keys. With the arrow keys, the user presses the key which corresponds to the direction to move the current position in the parse tree. If a direction does not make sense nothing happens.

For instance, if the cursor is on the left-hand side node and the right or left arrow keys are pressed nothing happens. If the up arrow key is pressed instead, then the right-hand side item of the production which contains that non-terminal becomes the current

item. If the down arrow key is used, the left-most item on the right-hand side of the production corresponding to the production of which this non-terminal is the left-hand side becomes the current item.

If the current item is on a right-hand side item of the production (displayed at the bottom of the window) and the up arrow key is used, the left-hand side of the production (at the top of the window) becomes the current item. If the left arrow key is used and an item exists before the current item in the right-hand side of the production, then this item becomes the new current item. If the right arrow key is used and an item exists after the current item in the right-hand side of the production, then this item becomes the new current item. If the down arrow key is used, the production which has this non-terminal as a left-hand side is displayed, and the left-hand side of that production becomes the current item. If the current item is a terminal then nothing happens.

If the mouse is used to traverse the tree, then the mouse cursor is moved over the item to make current and the left mouse button is clicked. To simulate the actions of the up arrow on the left-hand side item, or the down-arrow on a right-hand side non-terminal, the left mouse button is double-clicked instead of single-clicked. If the mouse button is clicked outside of the text of an item nothing will happen.

2. Subtree Deletion. If the current item is a non-terminal on the right-hand side of the production (which will be at the bottom of the window), then the subtree having the current node as the root

may be deleted. This can be done with the keyboard by pressing the DEL key, or the mouse may be used. To use the mouse, the cursor is moved over the item and the right mouse button is double-clicked. Once a subtree is deleted, the non-terminal has a new state known as unexpanded. An unexpanded non-terminal is marked on the window by putting it between a '<' and a '>'. An unexpanded non-terminal joe would be represented by <joe>. Delete does nothing on an unexpanded non-terminal or a terminal. This is also available as the Delete menu option.

3. Subtree Insertion. If the current item is an unexpanded non-terminal on the right-hand side of the production (which will be at the bottom of the window and have the special '<' and '>' symbols bracketing it), then an insertion of a new subtree having the non-terminal as the left-hand side may occur. This insertion may be done with the keyboard by pressing the INS key, or the mouse by moving the cursor over the item and double-clicking the left button.

This will bring up a dialog box with a list of valid productions which may be used to expand this non-terminal. If there is only one valid production no dialog box is displayed. Instead, the system chooses the production automatically. Once the production has been chosen, the current node is the newly expanded non-terminal and its subtree is shown. Any non-terminals in the new subtree are marked as unexpanded. Note: the meaning of a double-click of the left mouse button on a non-terminal in the right-hand side of a production depends upon the state of the non-terminal. Either way, the non-terminal's subtree will be

shown. The difference is whether this will be an existing or a new subtree. This is also available as the Insert menu option.

4. Subtree Moving. While editing with the tree editor there will be times when the user wants to move a subtree. The user may have spent a long time building a FOR statement only to discover that a WHILE statement was required. Rather than throw away the statement executed by the FOR statement, this subtree can be saved on a hook and then reinserted into the WHILE statement's subtree.

The user can move the statement subtree from the FOR statement onto one of two hooks. Hooks are places serving as temporary root nodes for trees being moved. LSD contains two hooks in the event two subtrees need to be exchanged. When the user needs to expand the statement executed by the WHILE statement, the subtree on the hook can be inserted into the parse tree to fill in the statement node.

The roots of the subtrees on two hooks are shown in the upper left hand corner of the tree edit window. The non-terminal of the subtree which is on a hook is shown or the word 'empty' appears.

To move a subtree to a hook click on the Hook option of the menu bar and a pull down menu will appear. There are four choices: Move to hook1, Move from hook1, Move to hook2, and Move from hook2. If hook1 is full, then the Move to hook1 option is deactivated and the Move from hook1 option is activated. If hook1 is empty, the Move to hook1 option is activated and the Move from hook1 option is deactivated. The menu choices for hook2 work in the same fashion.

Table VIII. Tree Edit Window Keys

Key	Action
F1	Get Help
F3	Move current item to hook1
F4	Move current item from hook1
F5	Move current item to hook2
F6	Move current item from hook2
Insert	Expand current item
Delete	Unexpand current item
↑	Move current item up
↓	Move current item down
→	Move current item right
←	Move current item left
Escape	Exit Tree Edit

When doing a move to hook, it does not matter what non-terminal is being moved. If a move from a hook is being done, the non-terminal acting as the root of the subtree on the hook (displayed in the upper left-hand corner of the tree edit window) must match the non-terminal which is the current item. Also, the current item must be an unexpanded non-terminal.

5. Tree Edit Window Keys. The tree edit window keys are listed in table VIII.

BIBLIOGRAPHY

[1]Sara Baase, Computer Algorithms: Introduction To Design and Analysis, 2nd Edition, (New York: Addison-Wesley), 1988, p. 2

[2]Robert L. Kruse, Data Structures and Program Design, (Englewood Cliffs, NJ: Prentice-Hall), 1984, p. 210

[3]Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, (Reading, Massachusetts: Addison-Wesley), 1988, pp. 159-266

VITA

John Gatewood Ham was born on June 10, 1964, in Florence, Alabama. After receiving his primary and secondary education in Paducah, Kentucky, public schools, he studied at the University of the South, in Sewanee, Tennessee. He received a Bachelor of Science degree in Mathematics from the University of the South, in Sewanee, Tennessee, in May 1986.

Following his graduation, Mr. Ham was employed by Norpax Security Printers as a COBOL programmer from 1987 to 1990. During that time he rewrote the personal computer check order entry system and the coupon payment book system.

Mr. Ham began graduate studies at the University of Missouri-Rolla in August 1990 to earn a master's degree in computer science. He has been enrolled continuously from August 1990 to the present.